Reactive Reflection in an FRP Language for Small-Scale Embedded Systems

Takuo Watanabe
Department of Computer Science
Tokyo Institute of Technology
takuo@acm.org

Abstract

This paper introduces a reflective functional reactive programming language designed for resource-constrained embedded systems. Using the reflection mechanism provided by the language, a program module can observe or modify its execution process via time-varying values that are connected to the internal of the metalevel of the module. Thus reflective operations are also reactive and described in a declarative manner. An example shows how the mechanism can realize an adaptive runtime that reduces the power consumption of a small robot.

CCS Concepts • **Software and its engineering → Functional languages**; *Data flow languages*; • **Computer systems organization →** *Embedded software*;

Keywords Functional Reactive Programming, Reflection, Embedded Systems

ACM Reference Format:
Takuo Watanabe. 2017. Reactive Reflection in an FRP Language for Small-Scale Embedded Systems. Presented at *Workshop on Meta-Programming Techniques and Reflection (Meta'17)*. 5 pages.

1 Introduction

Functional Reactive Programming (FRP)[1–4] is a programming paradigm for reactive systems based on the functional (declarative) abstractions of time-varying values and sequences of events. FRP has been actively studied and recognized to be promising for various kinds of reactive systems including robots[3, 4]. This suggests that FRP can be useful for other embedded systems in general. However, with a few exceptions, the majority of the FRP (especially pure-FRP[1]) systems developed so far are Haskell-based, and therefore they require substantial runtime resources. Hence, it is virtually impossible to run such FRP systems on resource-constrained platforms.

We designed and developed a pure-FRP language Emfrp for small-scale embedded systems[5]. The term small-scale here indicates that the target platforms of this language are not powerful enough to run conventional operating systems such as Linux. An Emfrp program can be represented as a DAG whose nodes and edges respectively correspond to time-varying values and their dependencies. The DAG is constructed at compile-time and never change at runtime. Although this static construction guarantees the predictability of the amount of the runtime memory, it loses the flexibility of realizing adaptive behaviors at runtime.

To provide a certain degree of flexibility and adaptability to the statically designed runtime system of the language, we designed a reflection mechanism for Emfrp and discuss its use in advance of actual implementation[6]. The proposed mechanism can provide a high-level and controlled access to the internal of the language runtime via time-varying values. The distinctive characteristic of our approach is that the reflective operations are also reactive.

This work-in-progress paper presents our current prototype implementation of Xfrp, a reflective extension of Emfrp, with an example use of its reflection mechanism.

[1] FRP based on purely functional languages

Meta'17, October 22, 2017, Vancouver, Canada

2 Overview of Xfrp

Xfrp is a reflective extension of Emfrp[5], a purely functional reactive programming language designed for resource-constrained embedded systems. This section presents the basic (non-reflective) features of the language with an example followed by the execution model of the language.

2.1 Basics

An Xfrp program consists of one or more *modules*. Listing 1 is an example Xfrp module for a simple robot controller[2]. It runs on Pololu Zumo 32U4 Robot[3] (Figure 1), a small (about 10cm × 10cm) tracked robot having two motors with rotation encoders, an accelerometer and a gyroscope. It is solely controlled by an on-board ATmega32U4 (8-bit AVR microcontroller with 32KB flash memory and 2.5KB RAM).

The controller reads data from an inertial sensor (gyroscope) to detect when the robot is being rotated. It controls the pair of motors to cancel the rotation. As a result, the robot keeps its direction.

[2] This example is adapted from an existing example for Pololu Zumo 32U4 Robot. https://github.com/pololu/zumo-32u4-arduino-library
[3] https://www.pololu.com/category/170/zumo-32u4-robot

```
1  module RotResist
2  in  gyroZ  : Int,     # gyroscope (z-axis)
3      t      : Int = 0  # current time (usec)
4  out motorL : Int,     # left motor
5      motorR : Int      # right motor
6  use Std
7
8  # This function is used to constrain the speed of
9  # the motors to be between -maxSpeed and maxSpeed
10 const maxSpeed = 400
11 fun motorSpeed(s) = min(max(s, -maxSpeed), maxSpeed)
12
13 # PD-control parameters
14 const kp = 11930465 / 1000
15 const kd = 8
16
17 # time difference
18 node dt = t - t@last
19
20 # Calculates the angle to turn from the gyroscope
21 # data and dt
22 node angle = gyroZ * dt * 14680064 / 17578125
23
24 # Calculates the turning speed using a simple
25 # PD-control method.
26 node turn = motorSpeed(-angle / kp - gyroZ / kd)
27
28 # Controls the motors
29 node motorL = -turn
30 node motorR = turn
```

Listing 1. Rotation Resistant Robot Controller

Figure 1. Zumo 32U4

A module definition contains a single module header followed by one or more type, constant, function or node definitions used in the module. In Listing 1, the module header (lines 1–6) defines the module name (`RotResist`), then declares two input nodes (`gyroZ` and `t`) and two output nodes (`motorL` and `motorR`), and specifies the library module (`Std`) used in this module.

The rest of the module (lines 8–30) consists of the definitions of three constants (`maxSpeed`, `kp` and `ka`), one function (`motorSpeed`) and five nodes (`dt`, `angle`, `turn`, `motorL`, and `motorR`). A node definition looks like

`node` n `=` e or **`node init`**`[`c`]` n `=` e

where n is the node name and e is an expression that describes the (time-varying) value of the node. The optional **`init`**`[`c`]` specifies the constant c as the initial value of the node. Note that if e contains another node name m, we say that n refers to m and hence n depends on m. While the value of m changes over time, the value of n varies also.

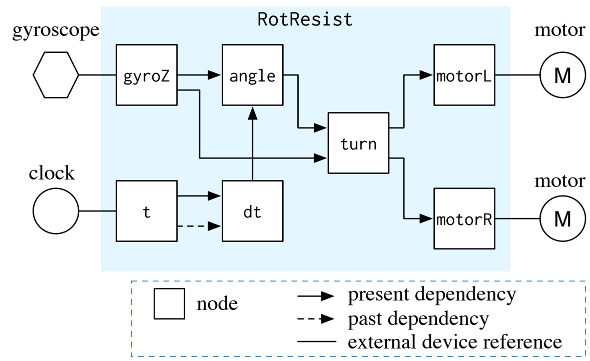

Figure 2. Graph Representation of Listing 1

Xfrp supports three kinds of nodes: *input*, *output* and *internal*. Each input or output node has a connection to an external device (or a system entity), while an internal node has no such connection. In the example, `gyroZ` and `t` are input nodes connected to the gyroscope and system clock, respectively. Their values represent the current motion data and time. The internal nodes `dt` (line 18), `angle` (line 22) and `turn` (line 26) respectively express the time difference (elapsed time from the last *iteration*), the angle of the current turn, and the speed of the motor.

The definition of `dt` has an expression `t@last`, which refers to the value of `t` at the "previous moment" — the value evaluated in the previous *iteration* (See Section 2.2).

2.2 Execution Model

An Xfrp module can be represented as a directed graph whose nodes and edges correspond to time-varying values and their dependencies respectively. Figure 2 shows the graph representation of Listing 1, which consists of seven nodes and eight edges.

We categorize the edges (dependencies) into two kinds: *past* and *present*. A past edge from node m to n means that n has m`@last` in its definition. A present edge from node m to n, in contrast, means that n directly refers to m. In Figure 2, the dotted arrow line from `t` to `dt` is the past edge. All other edges are present.

By removing the past edges from the graph representation of an arbitrary Xfrp program, we should obtain a directed-acyclic graph (DAG). The topological sorting on the DAG gives a sequence of the nodes. For Figure 2, we have: `gyroZ`, `t`, `dt`, `angle`, `turn` , `motorL`, `motorR`.

The Xfrp runtime system updates the values of the nodes by repeatedly evaluating the elements of the sequence. We call a single evaluation cycle an *iteration*. The order of updates (scheduling) in an iteration must obey the partial order determined by the above mentioned DAG.

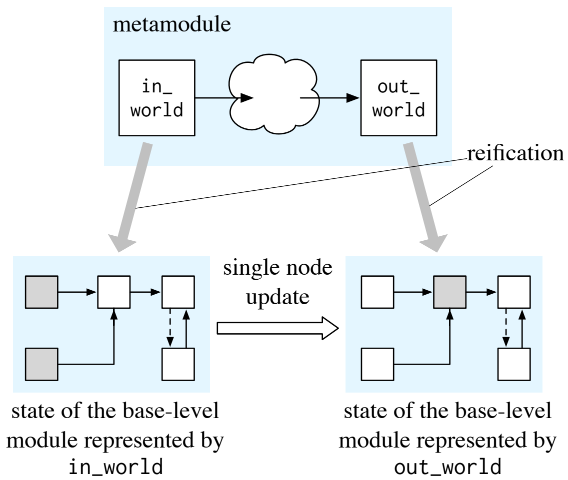

Figure 3. Metamodule

The value of *n*`@last` is the value of *n* in the last iteration. At the first iteration, where no nodes have their previous values, *n*`@last` refers to the initial value *c* specified with **`init`**`[`*c*`]` in the definition of *n*. In this example, since `t` is an input node, its initial value is specified at the header section of the module (line 3).

The Xfrp compiler translates a module definition into a platform-independent C program that repeatedly updates the values of nodes. The generated code is usually linked with some platform-dependent code (runtime system) to be deployed on an actual device.

3 Reflection Mechanism

To provide a high-level representation of the Xfrp runtime system, we introduce the notion of *metamodule* that governs an application level (base-level) module. Figure 3 depicts the concept. A metamodule contains at least one input node (`inWorld`) and one output node (`outWorld`), each of which represents an intermediate state of its corresponding base-level module.

Listing 2 shows the vanilla metamodule that expresses the basic execution model of Xfrp. Specifically, this module plays the role of the runtime function that repeatedly updates the node values.

Two nodes `inWorld` and `outWorld` represent an intermediate state of an iteration in the base-level module. The type of them (`World`) is defined as a pair type

```
type World = (Seq[Node], Seq[Node])
```

where `Seq[Node]` is the sequence type whose element type is `Node`. In the current version of Xfrp, functions using parametric types like this require explicit type parameters.

The elements of `World` respectively represent the nodes to be updated and the nodes already updated. The order of the nodes in the sequences should obey the order of the nodes in

```
1  module VanillaMeta
2  in  inWorld  : World
3  out outWorld : World
4  use Reflect
5
6  node outWorld =
7    let (xs, ys) = inWorld in
8    if isEmpty[Node](xs)
9    # Finishes a single base-level iteration
10   then (ys, empty[Node]())
11   # Updates a base-level node
12   else let (x, xs') = dequeue[Node](xs) in
13        let (n, p, c, e) = x in
14        case eval(e, xs', ys) of
15        # Updates the current value of the node
16        Just(v) ->
17            (xs', enqueue[Node](ys, (n, c, v, e)));
18        # Does not update if evaluation fails
19        Nothing ->
20            (xs', enqueue[Node](ys, (n, p, c, e)));
```

Listing 2. Vanilla Metamodule of Xfrp

the dependency graph explained in Section 2.2. A single base-level iteration starts with `(xs, empty[Node]())` and ends with `(empty[Node](), ys)` where `xs` and `ys` respectively correspond to the sets of nodes before and after the iteration.

The type of reified nodes is defined as

```
type Node = (String, Value, Value, Expr)
```

where `String`, `Value` and `Expr` are types of strings, reified data values (see next paragraph) and expressions. Thus, a node is represented as a quadruple `(n, p, c, e)` where `n`, `p`, `c` and `e` are the name, the last (previous) value, the current value, and the expression (RHS of the definition) of the node respectively. Values of the type `Value` represent base-level values of any data types.

Upon a successful update of a node, the previous *current* value of the node becomes the new *last* value and the evaluated value becomes the new *current* value (Line 17 in Listing 2). If the evaluation of the node fails, the current state of the node is just used as the result (line 20 in Listing 2)

4 Example: Robot Facing Uphill

This section describes an example using reflective features of Xfrp. The example, also runs on Zumo 32U4 Robot, uses the accelerometer to detect whether the robot is on a slanted surface. If it is on a slanted surface, then it turns itself to face uphill. It also uses the motor-rotation encoders to avoid rolling down the surface. Listing 3 show the controller module of the robot.

In line 8 of this example, a Boolean output node `needsTurn` is declared to be related to `meta(isBusy)`. The notation expresses that the value of `needsTurn` can also be referred as

```
module FaceUphill
in  accX      : Int, # accelerometer (x-axis)
    accY      : Int, # accelerometer (y-axis)
    encL      : Int, # left motor rotation encoder
    encR      : Int  # right motor rotation encoder
out motorL    : Int, # left motor
    motorR    : Int, # right motor
    needsTurn : Bool = meta(isBusy)
                # Connected to isBusy of the metamodule
use Std
meta AdaptiveSpeedMeta

# This function is used to constrain the speed of
# the motors to be between -maxSpeed and maxSpeed
const maxSpeed = 150
fun motorSpeed(s) = min(max(s, -maxSpeed), maxSpeed)

# True iff the robot is on a slanted surface.
# (incline of more than 5 degrees)
node init[False] needsTurn =
    accX * accX + accY * accY > 1427 * 1427

# Calculates the turning speed from the y-axis value
# of the accelerometer. It will be 0 if the incline
# is not significant.
node turn = if needsTurn then accY / 16 else 0

# Calculates the forwarding speed from the encoder
# values.
node forward = -(encL + encR)

node motorL = motorSpeed(forward - turn)
node motorR = motorSpeed(forward + turn)
```

Listing 3. Facing Uphill Robot

the value of isBusy of the metamodule AdaptiveSpeedMeta (Listing 4). This *inter-level node connection* is the central mechanism of reflection in Xfrp.

The metamodule AdaptiveSpeedMeta has an extra input node isBusy and an extra output node iterSleepMs as well as inWorld and outWorld. As described above, isBusy refers to the value of the node needsTurn in the base-level. The node iterSleepMs represents the sleep time between iterations. The larger the value of the node is, the slower the execution of the system becomes and the smaller the power consumption will be.

In this example, while the robot is on a level plane, it will slow itself down and lower the power consumption by sleeping 10ms between iterations. Once the robot finds itself on a slanted plane, needsTurn in the base-level module becomes true. This implies that isBusy is true in the metamodule because of the inter-level node connection. Thus the system runs as fast as possible by changing the value of iterSleepMs to zero and keeps the state for 1000 iterations.

```
module AdaptiveSpeedMeta
in  inWorld     : World,
    isBusy      : Bool # busyness of the base-level
out outWorld    : World,
    iterSleepMs : Int # sleep time between iterations
use Reflect

# Counts the iterations. Resets to 0 when detecting
# the falling edge of isBusy.
node init[0] count =
  if !isBusy && isBusy@last  # falling edge
  then 0
  else count@last + 1

# Keeps full-speed iterations while isBusy or 1000
# iterations after isBusy becomes False. After that,
# 10ms sleep is inserted at each iteration.
node iterSleepMs =
  if isBusy || count < 1000 then 0 else 10

# Same as VanillaMeta
node outWorld = ...
```

Listing 4. Adaptive Speed Metamodule

The runtime system used with this metamodule should insert specified sleep time between iterations. Such behavior can be implemented for example by inserting a sentence usleep(iterSleepMs * 1000); to the place before the invocation of the iteration process.

5 Concluding Remark

This work-in-progress paper presents a simple reflection mechanism for Xfrp, a reflective functional reactive programming language designed for resource-constrained embedded systems. The main purpose of introducing reflection is to provide a certain degree of flexibility and adaptability to the statically designed runtime system of the language.

The proposed reflection mechanism opens up the internals of a runtime system via nodes (time-varying values) connected to nodes in the metamodules. The mechanism based on the *inter-level node connection* can be classified as a behavioral reflection in a sense that the base-level module can modify its own behavior by affecting the execution of the metamodule via connected nodes.

The future research direction should focus on the investigation of the use of the proposed reflection mechanism as well as performance evaluation.

Acknowledgments

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

References

[1] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. ACM, 411–422. https://doi.org/10.1145/2499370.2462161

[2] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*. ACM, 263–273. https://doi.org/10.1145/258949.258973

[3] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*. Lecture Notes in Computer Science, Vol. 2638. Springer-Verlag, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

[4] Izzet Pembeci, Henrik Nilsson, and Gregory Hager. 2002. 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)*. ACM, 168–179. https://doi.org/10.1145/571157.571174

[5] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems. In *Modularity 2016 Constrained and Reactive Objects Workshop (CROW 2016)*. ACM, 36–44. https://doi.org/10.1145/2892664.2892670

[6] Takuo Watanabe and Kensuke Sawada. 2017. Towards Reflection in an FRP Language for Small-Scale Embedded Systems. In *Companion to the 1st International Conference on the Art, Science and Engineering of Programming (Programming 2017)*. ACM, Article 10, 10:1–10:6 pages. https://doi.org/10.1145/3079368.3079387