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:

1 Introduction
Functional Reactive Programming (FRP)\cite{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\cite{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\footnote{FRP based on purely functional languages}) 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\cite{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\cite{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.

2 Overview of Xfrp
Xfrp is a reflective extension of Emfrp\cite{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\footnote{This example is adapted from an existing example for Pololu Zumo 32U4 Robot. https://github.com/pololu/zumo-32u4-arduino-library}. It runs on Pololu Zumo 32U4 Robot\footnote{https://www.pololu.com/category/170/zumo-32u4-robot} (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.
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.

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.
The value of `@last` is the value of `n` in the last iteration. At the first iteration, where no nodes have their previous values, `@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 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

# 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 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


