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

Takuo Watanabe
Department of Computer Science, Tokyo Institute of Technology

META 2017
Oct. 22, 2017, Vancouver
About This Work

• Proposing a new reflection mechanism for a pure FRP language designed for small-scale embedded systems
  - Reactive Reflection = Reflection via Time-Varying Values (Signals)
  - TVVs serve as interface between an application and its metalevel

• Objective
  - High-level abstraction mechanism for flexible and efficient execution of FRP languages

• Challenges
  - Limited runtime resource
  - Declarative representation of reflective behaviors
Emfrp

• Pure FRP language designed for small-scale embedded systems [Sawada & Watanabe, CROW2016]
  - "small-scale" = resource-constrained systems (microcontrollers)
    • 8-32 bit processor (\leq 100MHz), program storage & RAM (\leq 1MB)
    • w/o conventional operating systems or w/ simple RTOS
    • ex) ARM Cortex-M, AVR, PIC, etc.

• Why pure FRP for small-scale embedded systems?
  - To support efficient development by replacing patterns as follows, with declarative and uniform language constructs.
    • polling
    • callbacks (interrupts, timers, events, etc.)
    • state machines
    • threads
Emfrp: Language Design

• Pure functional language with:
  - Algebraic datatypes, Parametric polymorphism, Type inference
  - No higher-order functions, No recursion (datatypes and functions)

• Node(\(\tau\)) : Datatype for TVVs (signals)
  - The syntax and type system of Emfrp disallow that nodes are passed around as function arguments.
  - Functions applied to nodes are automatically \emph{lifted}.
  - N@last notation provides access to the value of N at the previous moment

• The above restrictions and properties guarantee that runtime memory size can be statically determined.
module RotResist
in  gyroZ : Int,  # gyroscope (z-axis)
     t    : Int    # current time (usec)
out motorL : Int,  # left motor
              motorR : Int    # right motor
use Std

const maxSpeed = 400
fun motorSpeed(s) = min(max(s, -maxSpeed), maxSpeed)
const kp = 11930465 / 1000
const kd = 8

node dt = t - t@last
node angle = gyroZ * dt * 14680064 / 17578125
node turn = motorSpeed(-angle / kp - gyroZ / kd)
node motorL = -turn
node motorR = turn
Execution Model

Iteration

1. read gyroscope and update gyroZ
2. read clock and update t
3. update dt
4. update angle
5. update turn
6. update motorL and write to $M_L$
7. update motorR and write to $M_R$
Motivation

- The static and fixed construction of DAGs in Emfrp realizes:
  - small runtime overhead, and
  - predictable memory space.
- However, it is not easy to introduce adaptive executions without modifying runtime libraries or generated C code.
- To cope with this problem, we designed a simple reflection mechanism for Emfrp [Watanabe & Sawada, LASSY 2017]
- This talk reports the current status of the project, a redesigned reflective FRP language Xfrp and its usage.
Related Work

• Applications to robot control [Pembeci et al, 2002][Hudak et al, 2003]
  - Remote-controlling of robots using Haskell-based Arrowized FRP
• RT-FRP [Wan et al, 2001], Event-Driven FRP (E-FRP) [Wan et al, 2002],
  E-FRP with Priorities [Kaiabachev et al, 2007]
  - FRP with (global) clocks and events
• Elm [Czaplicki 2012]
• Céu [Sant’Anna et al, 2015]
• Juniper [Helbling et al, FRAM 2016]
• Push-Based Reactive Layer [Kamina et al, COP 2017]
• Potato [De Troyer et al, REBLS 2017]
  - High-Level DSL for CPS / Middleware for Elixir
Reflection Mechanism

- **Metamodule**
  - Xfrp module that governs the execution of an application (base-level) module

- **in_world & out_world**
  - Input and output nodes that reify the intermediate states of the application module
Vanilla Metamodule

```haskell
module VanillaMeta

in  inWorld : World
out outWorld : World
use Reflect

node outWorld =
  let (xs, ys) = inWorld in
  if isEmpty[Node](xs)
    # Finishes a single base-level iteration
    then (ys, xs)
    # Updates a base-level node
  else let (x, xs') = dequeue[Node](xs) in
    let (n, p, c, e) = x in
    case eval(e, xs', ys) of
      # Updates the current value of the node
      Just(v) -> (xs', enqueue[Node](ys, (n, c, v, e)));
      # Does not update if evaluation fails
      Nothing -> (xs', enqueue[Node](ys, (n, p, c, e)));

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

Note: Seq, String, Value and Expr are primitive types. The 1st argument of dequeue and enqueue should be an identifier and should not appear after the call of these functions.
```

Note: Seq, String, Value and Expr are primitive types. The 1st argument of dequeue and enqueue should be an identifier and should not appear after the call of these functions.
typedef int (*Expr_t)(Seq_Node_t, Seq_Node_t, Value_t);

typedef unsigned int String_t;
#define STRING_dt 0xc4363a5a
#define STRING_gyroZ 0x8ef8e1d0

// type Node = ( String, Value, Value, Expr )
typedef struct { String_t v1, Value_t v2, Value_t v3, Expr_t v4 }
  tuple_string_value_value_expr_t;
typedef tuple_string_value_value_expr_t Node_t;

void expr_angle(Seq_Node_t nsCurr, Seq_Node_t nsPast,
               Option_Value_t *ov) {
  Node_t *node_gyroZ = find_node(nsCurr, STRING_gyroZ);
  Node_t *node_dt = find_node(nsCurr, STRING_dt);
  if (node_gyroZ == NULL || node_dt == NULL) { ov->k = 0; return; }
  ov->k = 1;
  ((ov->u).v0.a0)->t = T_INT;
  (((ov->u).v0.a0)->u).v_int =
    (node_gyroZ->v2->u).v_int * (node_dt->v2->u).v_int *
    14680064 / 17578125;
Using Reflection

• AdaptiveMeta [Watanabe & Sawada, LASSY2017]
  - A metamodule that can avoid unnecessary node updates
  - We can specify some nodes as "lazy" in the base-level via a node which is connected to the metamodule.
  - The nodes specified as lazy will be updated only when they are needed.

• AdaptiveSpeedMeta (this paper)
  - A metamodule that provides a control mechanism of "execution speed"
  - We can control the iteration rate of an application via a node.
  - Applications:
    • Power saving
    • Expressing real-time execution
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)
use Std
meta AdaptiveSpeedMeta
...

node init[False] needsTurn =
  accX * accX + accY * accY > 1427 * 1427
node turn = if needsTurn then accY / 16 else 0
node forward = -(encL + encR)
node motorL = motorSpeed(forward - turn)
node motorR = motorSpeed(forward + turn)
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 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 = ...

# busyness of the base-level

• Assumption: iterations are triggered by a periodical timer of the processor
• The output node iterSleepMs specifies the timer period.
• While the robot is on a level plane, the metamodule slows down the interval.
• When the robot finds itself on a slanted surface, the metamodule starts full-speed iteration.
module Balancer
in gyroY : Int, # gyroscope (y-axis)
    accX : Int, # accelerometer (x-axis)
    encL : Int, # left motor encoder
    encR : Int # right motor encoder
out motorL : Int, # left motor
    motorR : Int, # right motor
    iterSleepMs : Int = meta(iterSleepMs)
use Std
meta AdaptiveSpeedMeta
...

node iterSleepMs = update_time_ms
node init[0] angle = (angle@last + gyroY * update_time_ms) * 99 / 100

node speedL = encL - encL@last
node speedR = encR - encR@last
node init[0] distL = distL@last + speedL
node init[0] distR = distR@last + speedR

node risingAngleOff = gyroY * angle_rate_ratio + angle
node init[0] motor =
    motorSpeed(motor@last +
        (angle_resp * risingAngleOff +
         dist_resp * (distL + distR) +
         speed_resp * (speedL + speedR)) / 100 / gear_ratio)

node diffSpeed = (distL - distR) * dist_diff_resp / 100
node motorL = if accX > 0 then motor + diffSpeed else 0
node motorR = if accX > 0 then motor - diffSpeed else 0
Summary & Future Direction

• A simple reflection mechanism based on base-meta node connections can introduce several adaptive behaviors to a pure FRP language for small-scale embedded systems.

• Future Direction
  - Expressing imperative (sequential) behaviors
    • e.g. initialization/setting up phase of embedded systems
  - Composition Mechanisms
  - Asynchronous Execution Models [Watanabe, AGERE 2016]
  - Case Studies
  - Semantics
  - Verification / Testing / Debugging