Meadowview: Language Guide

August 12, 2014

1 Introduction

The **Meadowview (MV)** is a language for describing individual or collective behavior of **Meadow devices**. It is a dynamically-typed, statically-scoped language with type inference, which facilitates *event-based, reactive programming model*.

2 Basic Types

Meadowview programs manipulate **values**, where each value has a **type**. Unlike C++ or Java, where variables have types, Meadowview is a dynamically-typed language.

2.1 Scalar vs Aggregate Types

A type is either a **scalar type** or an **aggregate type**.

2.2 Booleans

A boolean is a truth value, which can be either true or false.

```
true
false
```

2.3 Integers

```
2014
0
-10
```

2.4 Real numbers

```
3.14
0.0
-13.0
1E6
-1.3E-8
```

2.5 Time

```
05-13-2014/23:11:10
Aug-3-2014/00:00:00.000012
```

2.6 Characters

```
'a'
'b'
#13
```

2.7 Symbols

A **symbol** is a value which presents the "name" of the symbol.

```
symbol
com.mv.dataserver
```

2.8 Strings

A **string** is a finite sequence of characters witch fixed length and thus represent arbitrary Unicode texts.

```
"abc"
"Program Transformation"
```

2.9 Pairs

A **pair** is a aggregate data with two components.

```
(1, 3)
sym = (a, "bc")
sym.first    // a
sym.second   // "bc"
```

2.10 Lists

```
[1, 2, 3],
sym = [1, 2, (3, "str"), [a.b, b.c]]
sym.head     // 1
sym.tail     // [2, (3, "str"), [a.b, b.c]]
```

2.11 Vectors

```
#[1, 2, 3]
sym = #[10, 20, 30, "a"]
sym[2]      // 30
```

2.12 Named maps

```
{ name => "John", age => 25}
sym = { first => 1, second = "str" }
sym.first     // 30
```

2.13 Events

An **event** is an aggregate data which is transferred between devices. An event supports operations such as `event_publish` and `event_wait`. Events are treated as *first-class citizens* in Meadowview, meaning that they can be passed as a function argument, returned from a function call, or assigned to a variable.

Technically, an event is a named map with following predefined entries:

- `event_uuid`: symbol
- `device_uuid`: symbol
- `timestamp`: value of date type

Examples are:

```
// constructor
EV(com.mv.orderCompleteEvent, {price => 100.34})

// event generation
sym = EV(com.mv.orderCompleteEvent, {price => 10})
event_publish(sym)

// event reception
reactor (com.mv.orderCompleteEvent e) {
  e.ev_uuid
  e.value
}
```

2.14 Devices

A **device** is an entity which can offer services – properties, events, functions, and reactors. In particular, device can be the target of standard functions such as `prop_get`, `prop_set`, and `call` and `call_async`.

```
// no constructor
// only bound when process is instantiated
```

Technically, a device is a named map with following predefined entries:

- `device_uuid`: symbol

2.15 Functions

Functions are first-class citizens in Meadowview. A **function** consists of *function parameters* and the *function body*.

```
function foo(a, b) {
  return a + b;
}
```

2.16 Reactors

2.17 Event reactors

A **reactor** consists of an *event*, to which the reactor are sensitized to, and the *reactor body*. The body of the reactor is executed whenever the event is triggered.

```
reactor R(com.mv.shippingNotice notice) {
  a = notice.timestamp
  b = notice.data
}
```

2.18 Timed reactors

```
timed_reactor R(⟨time⟩]) {
  a = notice.timestamp
  b = notice.data
}
```

2.19 Processes

A **process** consists of *process parameters* and a collection of *reactors* and *functions*. When a process is executed, reactors and functions will be executed.[1]

```
process P(dev1, dev2) {
  property ntransfers = 0;

  reactor consumer(dev1:dataReady e) {
    data = dev1:get_data();
    dev2:consume_data(data);
    ntransfers++;
  }

  function getTransfers()  return ntransfers;
}

process ButtonAndLight(toggleButton, light) {
  reactor consumer(toggleButton:pressed e) {
```

[1] Actually, MV code contains *process definitions* rather than *processes*. In this document, the terms "process definition" and "process" will be used interchangeably but it should be clear from the context which is referred to.

```
    if (e.buttonOn)
      light:turnOn();
    else
      light:turnOff();
  }
}

process MapReduce(deviceList) {
  HOW TO REPRESENT MAP REDUCE OR GRAPH WORK
}
```

3 Functions

3.1 Overview

A **function** is a piece of paramterized code.

3.2 Syntactic form

The following is the syntactic form of a function.

```
function ⟨function_name⟩(⟨args⟩) {
  ⟨function_body⟩
}
```

where ⟨*function_body*⟩ consists of zero or more statements.

3.3 Example

```
function foo(x) {
  return x + 1;
}
```

4 Reactors

4.1 Overview

A **reactor** is a piece of code parameterized by an *event*, which is invoked whenver the corresponding event occurs.

- **Q**: should we allow multiple events? if yes, is it "OR" or "AND"
- **Q**: how can we support "JOIN" behavior?

4.2 Syntactic form

The following is the syntactic form of a reactor.

```
reactor ⟨reactor_name⟩(⟨event_class⟩ ⟨event_name⟩) {
  ⟨reactor_body⟩
}
```

In the form, ⟨*reactor_name*⟩, ⟨*event_class*⟩, and ⟨*event_name*⟩ are **symbols**.

4.3 Semantics

A reactor will be evaluated whenever an event of the event class, ⟨*event_class*⟩, is generated. The given event value will be bound to the ⟨*event_name*⟩.

4.4 Example

```
reactor myreactor(com.mv.shipper:ShippingNotice notice) {
  customerid = notice.customerid;
  trackid = notice.trackingId;
  func_call_async(com.mv.mailserver, sendmail, customerid, trackid);
  event_publish(shippingDone);
}
```

4.5 Use of event class

While meadowview is a dynamically-typed language, we still need include the class of event when specifying a reactor. This is to ensure only events of the given class will wake up the reactor. Without event class, each reactor have to wake up on event of any event class.

5 Processes

5.1 Syntactic form

The following is the syntactic form of a reactor.

```
process ⟨process_name⟩(⟨device_class⟩ ⟨device_name⟩) {
  ⟨process_body⟩
}
```

In the form, ⟨*process_name*⟩, ⟨*device_class*⟩, and ⟨*device_name*⟩ are **symbols**. The ⟨*process_body*⟩ can contain definitions of *properties*, *events*, *functions*, and *reactors*.

5.2 Example

```
process com.mv.monitorTemperature(com.device.Thermostat dev) {
  property batteryLife;    // pulled data
  event batteryLow;        // pushed data
  event needMaintenance;   // pushed data

  // ? need a way to import functions, properties, and events
  native function getTemperature();
  native function getBatteryLife();
  native event com.device.Thermostat:needMaintenance;

  function isBatteryMoreThanHalfFull() {
    return (batteryLife > 0.5) ? true : false;
  }

  // relay
  reactor ThermostatWatcher(com.device.Thermostat:needMaintenance e) {
```

```
    :generate(needMaintenance(e.getReason() /* string */));
  }

  function checkBatteryLoop() {
    sleep(30*3600*1000 /* 30 minutes */);
    batteryLife = getBatteryLife();
    if (batteryLife < 0.2) {
      :generate(batteryLow);
    }
    checkBatteryLoop();
  }

  // on process instantiation
  function start() {
    checkBatteryLifeLoop();
  }

  // on process instantiation and VM reboot
  function init() {
  }
}
```

5.3 Process names

Process name must be unique over the entire name space. This will be enforced during compilation of process.

5.4 Events

Events are values contained in a process. Events are special class of values, which can be **pushed** across processes.

5.5 Properties

Properties are values contained a process. Properties are special class of values which can be **pulled** across processes.

5.6 Compilation of processes

While reactors are interpreted by the device runtime but processes are compiled by the process compiler. After compilation of the process, it can be instantiated multiple times with device parameters. Note that ⟨*event_class*⟩ is just another symbol,

5.7 Instantiation of processes

When class of devices "conform to" (who performs conformance check?) ⟨*device_class*⟩'s of a process definition, the process can be instantiated using the devices as actuals to the process definition. Process instantiation amounts to *deploying the process to the devices until this process is decomissioned*.

5.8 Lieftime of a process

Process is alive until

- it is explicitly decomissioned.
- `kill-process` builtin function is executed by a participating device.

5.9 Execution of processes

Instantiation of a process does not automatically execution of the process. The process will begin execution when mandatory `start` function of the process is executed.

5.10 Non-native exports

A process can contain non-native exports. An export is either an **event**, a **property**, or a **function**.[2]

Q: how to integrate these notions? native exports vs non-native exports defined by process definitions.

5.11 The `start` function

Every process definition must contain the definition of `start` function.

5.12 The `initial` function

An optional `initial` function is executed once in the following occasions.

1. When the process is first instantiated and deployed to a device, the device runtime executes the initial process.
2. When the device boots, it executes all intial functions of the processes which have been deployed to the device.

5.13 The `final` function

An optional `final` is executed when the process is dismissed from the device.

5.14 Dismissal of processes

A process can be a one-time process or a repetitive process. A one-time process is dismissed from the device to which it is deployed whenever the first termination condition is satisfied.

6 Expressions

An expression is a syntactic entity, which *evaluates to a value*.

[2] just like AllJoyn

6.1 Time

6.2 Variable references

6.2.1 List entry reference

6.2.2 Vector entry reference

6.2.3 Named map entry reference

6.3 Arithmetic expressions

6.4 Property References

A property reference has the form

⟨*device_name*⟩ : ⟨*property_name*⟩

6.5 Function calls

⟨*device_name*⟩ : ⟨*function_name*⟩ (⟨*args*⟩)

7 Statements

A function or a reactor consists of statements.

7.1 Sequence

⟨*statement*⟩ ;
⟨*statement*⟩

The "`;`" is a sequencing operator which indicates that the statement that comes before this operator is executed to its completion, and then, the statement that comes after the operator is executed to its completion.

7.2 Parallel-fork

```
fork {
  ⟨statement⟩
  ⟨statement⟩
} [join | join_none | join_any ]
```

For now, the difference between sequence and parallel-fork is that, when a statement in a parallel-fork is suspended due to request to an external device, the other statement in the fork can be executed.

7.3 Assignments

⟨*lhs_expr*⟩ = ⟨*rhs_expr*⟩

where ⟨*lhs_expr*⟩ is either a variable or a device property.

7.4 Event trigger

```
=> EV(⟨event_name⟩, ⟨event_payload⟩)
```

7.5 Function calls

A function call has the following synctactic form.

```
⟨device_name⟩:⟨function_name⟩(⟨args⟩) {
  timeout ⟨time⟩ => {
    ⟨timeout_body⟩
  }
  failure => {
    ⟨failure_body⟩
  }
}
```

The ⟨*function_name*⟩ is a symbol which notes the function to be invoked. It can be either a local function or a remote function. An optional ⟨*fail_body*⟩ can be executed in case the result is not available in given timeout time. This is mostly for remote procedure calls.

The ⟨*device_name*⟩ can be `local` is the callee is defined inside the same device.

7.6 Branches

```
if (⟨cond⟩) {
  ⟨statements⟩
}
else {
  ⟨statements⟩
}
```

7.7 Loops

```
for (⟨init⟩; ⟨cond⟩; ⟨step⟩) {
  ⟨statements⟩
}
```

The ⟨*statements*⟩ can contain continue or break statement.

8 Standard Library

8.1 Devices

8.1.1 `device_self`

Returns the device where this function is executed.