Vrije Universiteit Brussel

FACULTY OF SCIENCE AND BIO-ENGINEERING SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

Projection of Reactive Programming onto Dataflow Engines

Master thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Applied Sciences and Engineering: Applied Computer Science

Alexander Moerman

Promoter: Prof. Dr. Wolfgang De Meuter
Advisor: Mathijs Saey, Florian Myter and Thierry Renaux

Academic year 2016-2017

Abstract

Declaration of Originality

I hereby declare that this thesis was entirely my own work and that any additional sources of information have been duly cited. I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this thesis has not been submitted for a higher degree to any other University or Institution.

Acknowledgements

Contents

1 Introduction

2 Background
2.1 Introduction . . . 3
2.2 Reactive Programming . . . 4
2.2.1 Example . . . 4
2.2.2 Advantages of Reactive Programming . . . 5
2.3 The Dataflow Model . . . 6
2.3.1 Introduction . . . 6
2.3.2 Example . . . 7
2.3.3 Advantages . . . 7
2.4 Conclusion . . . 8

3 Language
3.1 Introduction . . . 9
3.2 FrDataFlow . . . 9
3.2.1 Sample program . . . 9
3.2.2 Sample programs . . . 11
3.2.3 Implementation . . . 12
3.3 Conclusion . . . 15

4 Engine

5 Evaluation

6 Future work and limitations

7 Related work

8 Conclusion

A Your Appendix

List of Figures

2.1 Graph of signals 5
2.2 Graph of instructions 7

List of Tables

1
Introduction

2 Background

2.1 Introduction

This research builds on two existing paradigms in software development: reactive programming and the dataflow model. While intimate knowledge about these concepts is not required, a basic understanding of both will be necessary to follow the ideas and implementation of this dissertation. Reactive Programming is situated in the category of higher level software development, serving as an abstraction tool for events and reactions to those events. The dataflow model on the other hand can be considered more 'low level', providing a strategy for the implicitly parallel execution of programs.

2.2 Reactive Programming

Reactive Programming is a software development paradigm focused on reactions, i.e. the handling of external events, user interactions, etc. In this paradigm, the application state is derived from the previous state and any events that may occur, for example user interactions or current environmental factors. This deviates from more traditional approaches, where values and state can be written at any point and for any reason. In a reactive program however, the flow of dependencies is recorded as a (possibly cyclic!) directed graph, making the derivation of the application state very explicit. At the core of reactive programming are the following main concepts:

- The first-class reification of events (making events a first-class citizen)
- The composition of these events through lifted functions
- The automatic tracking of dependencies and re-evaluation by the language runtime

A number of implementations exist for reactive programming, in this thesis we will focus on the interpretation taken in FrTime (Cooper & Krishnamurthi, 2006).

2.2.1 Example

The canonical metaphor for Reactive Programming is spreadsheets, which typically track changes across input cells and automatically recompute values in other cells if the formulas they contain reference the aforementioned input cells. In essence, cells react to modifications made in other cells if their formulas depend on them. These cells are what we call *observables* or *signals* in Reactive Programming. Imagine a simple program in an imperative programming setting:

```
a = b + c
```

When this statement is executed, it assigns the result of adding b and c to the variable a, mutating a in the current scope. Note that this only happens once. A snapshot is taken of the current value of b and c, to determine the new value of variable a. Of course, this assumes that the variable b and c are provided to the program.

In a reactive programming setting, a would subscribe to the values of b and c, essentially asking to be notified whenever the variables b or c change, at which point the value of variable a changes. See figure 2.1 for the reactive

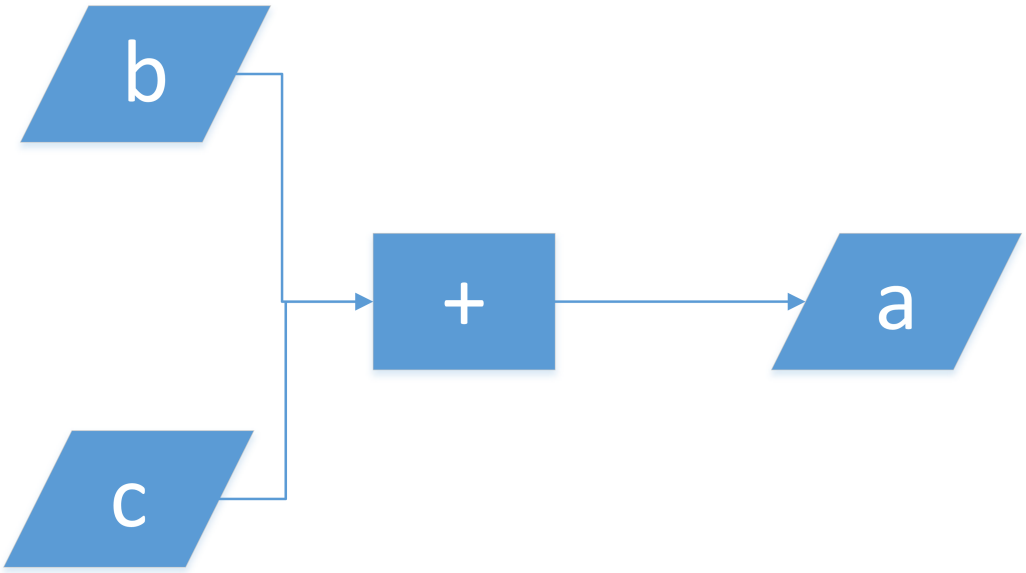

Figure 2.1: Graph of signals

graph. This process repeats every time the variables b or c are modified. Note that the value of a is undetermined until both b and c produce a value.

The implementation of this reactive mechanism can be provided by the language itself or by a framework or library.

2.2.2 Advantages of Reactive Programming

A signal can be described as "values over time", in contrast with a variable which only holds its latest value, revealing no information about the time that value was provided or what changed it. Signals can be used to model almost any concept in software development:

- mouse movements as a signal which emits the current position in real time
- click events as a signal which emits event objects
- the results of a database query as a signal which emits only one value
- an infinite sequence as a signal which never stops emitting

Even though the underlying mechanism will still be identical to more traditional approaches (attaching event listeners to DOM events in HTML, opening and connecting to a WebSocket connection, etc.), the fact that all

these concepts can be brought together under a single umbrella called *signals* allows for the modeling of higher order operators to map, combine and filter these flows of values in ways that were previously a lot harder.

2.3 The Dataflow Model

2.3.1 Introduction

The dataflow model is a paradigm focused on the parallel execution of programs. In this paradigm, instructions are seen as isolated units, which should be able to execute whenever the necessary parameters have been provided. Contrary to imperative programming, instructions are not invoked by a program counter, but rather whenever all of the parameters are present.

The execution of instructions in the dataflow model can be seen as a direct graph of nodes where each node represents an instruction and each edge is the output being sent to the next instructions that require the output as arguments. It is up to the dataflow engine to orchestrate the flow of arguments so that instructions are invoked correctly and in the correct order.

Whenever an instruction is invoked, the output is sent through to all connected instructions which depend on it. In Tagged Token Dataflow systems, instruction arguments are wrapped in tokens, which carry meta data about which execution context they belong to in order to isolate multiple calls to the same instruction from one another.

A large difference with Reactive Programming is that Dataflow Programming puts the instruction invocation at the center stage, while Reactive Programming puts forward signals as the core concept of its paradigm. In other words, while both systems have the notion of a dependency graph, the nodes in their graphs carry different concepts: instructions and signals respectively.

2.3.2 Example

Imagine a simple program in an imperative programming setting:

```
a = b + c
d = a + b
```

This assumes that the variable b and c are provided to the program. In a traditional execution, the variable a would be set to the sum of b and c and the variable d would be set to the sum of a and b. Note that the sequence in which these operations are executed is of vital importance: switching the two statements would result in different values for the variable d!

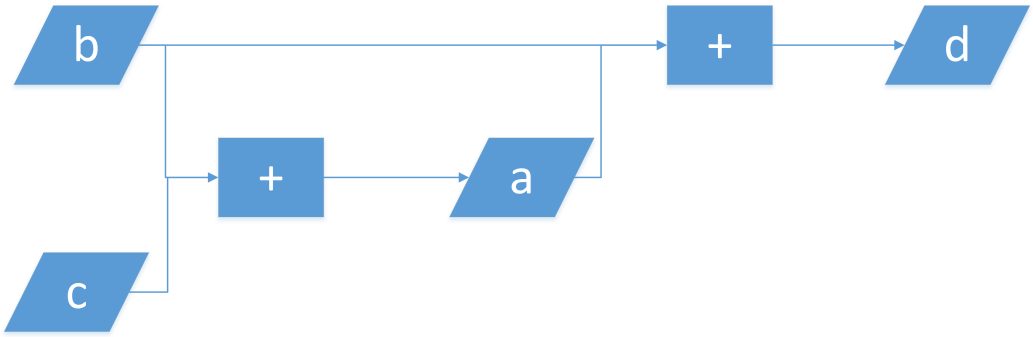

Figure 2.2: Graph of instructions

In a dataflow engine, these instructions would be registered as instructions in the dependency graph, as visualized by figure 2.2. The values of b and c would be added to the queue at application startup. B would be entered as a token twice; once for the instruction "+" which computes a and once for the instruction "+" which computes d. When the dataflow engine spins up and starts processing arguments, it sends the tokens for b and c to the first "+" instruction, which is triggered because all of its inputs are present and valid. This produces a value for variable a, which gets added to the token queue again as the first parameter for the second "+" instruction. This instruction now also has all of its inputs present, which allows it to compute the value for variable d at this point.

If at any point in the future, b or c (which should be seen as the output of other instructions not shown in the sample code) produce new values, these would be enqueued again for further processing.

2.3.3 Advantages

The key advantage of the data flow model is that only the data dependencies of the instructions decide when an instruction can be executed. Since data

flow instructions are not allowed to access or manipulate shared state, each instruction is completely isolated. This means that all dataflow instructions can be run in parallel, across different processes and even separate machines.

2.4 Conclusion

Two paradigms were presented: reactive programming and the dataflow model. Reactive programming is targeted more towards events and reactions and shines best in environments where these things are plenty, for example in user interfaces and other places where events can come from any direction. The dataflow model on its part focuses more on the parallel execution of instructions by streaming parameters to them in isolated scopes. We do however note similarities between the two, namely that they both work with an update graph that guides the data along the nodes.

3 Language

3.1 Introduction

For the purposes of this thesis, we have implemented a lightweight language *FrDataFlow* using two different AST interpreters in Racket, supporting most of the basic constructs found in Racket. This work is based on earlier work detailed in Abelson et al. (1999). The first interpreter supports reactive patterns on top of the already existing Racket language. The second interpreter also supports the exact same language, but executes its instructions using an underlying dataflow engine. We chose to implement this custom language *FrDataFlow* (rather than a framework or library) to maintain maximum control over the inner workings and to facilitate experimentation atop the dataflow engine. The main goal was to have a reactive superset of Racket for experimental purposes during this thesis.

3.2 FrDataFlow

3.2.1 Sample program

Take for example a roadside digital billboard that displays the current date and temperature. FrDataFlow provides an environment with most Racket

language constructs (defining variables, procedures, lambda, primitive operators, ...) and some built in signals, described in listing 3.1

```
;; A signal which emits the current seconds since 1 Jan 1970, every second
current-unix-timestamp
;; A signal which emits the current temperature every 1 to 10 seconds
current-temp-fahrenheit
```

Listing 3.1: Built in signals

```
(define (fahrenheit->celsius fahrenheit)
  (quotient (* (- fahrenheit 32) 5) 9))
(define current-temp-celsius
  (lift fahrenheit->celsius current-temp-fahrenheit))
(define current-date
  (lift seconds->date current-unix-timestamp))
(define billboard-label
  (lift
    (lambda (temperature date)
      (string-append "Temperature: " (number->string temperature) "CÂř, Date
          : " (date->string date)))
     current-temp-celsius
     current-date))
```

The initial version of the language was written in Racket, making use of the Racket evaluator to execute primitive procedures (e.g. addition, multiplication, etc.) and simulating a program memory by storing variables and scopes in simple Racket lists. This evaluator builds on earlier work by professor Theo D'Hondt, who provided a basic metacircular evaluator during the course 'Interpretation of Computer Programs'.

Primitive values

```
1      ;; a number
"ABC"  ;; a string
'()    ;; the empty list
```

Primitive procedures

```
(cons x xs) ;; prepends x to a list of xs
(car xs)    ;; gets the head of a list
(cdr xs)    ;; gets the tail of a list
(null? xs)  ;; checks if a list is empty
```

Variables and procedures

```
(define x 3)              ;; defines a new variable x with value 3
x                         ;; gets the value of x
(define (x a b) (+ a b)) ;; defines a new procedure x
(x 1 2)                   ;; calls a procedure x with arguments 1 and 2
(lambda (a b) (+ a b))    ;; defines an anonymous procedure
```

For the reactive language, two new concepts were introduced to the metacircular evaluator:

1. Signals: wrappers around values which track their parents and children in the dependency graph.
2. Lift: a procedure that allows the creation of a new signal deriving from on one or more existing signals and a lambda that produces a single value given the values of the parent signals.

Signals and lift

```
(value signal)            ;; gets the current value of a variable called
    signal
(lift + signal1 signal2) ;; creates a new signal that emits the sum of
    signal1 and signal2
```

3.2.2 Sample programs

```
;; signal that emits the current date
(define current-date (lift seconds->date current-seconds))

;; prints the current value
(value current-date)

;; signal that adds current seconds to random integer
(define sec-rand (lift + current-seconds random-integer))

;; signal that emits whether the aforementioned signal is even
(define sec-rand-even? (lift even? sec-rand))
```

These are some sample usages of the lift function and existing source signals. There is no limit to the amount of parents a signal can have or to the depth of the dependency chain. Note that the 'value' function is necessary to extract the current value of a signal, without this the signal wrapper would be returned by the evaluator.

3.2.3 Implementation

When signals are created, they are registered with the parents as children. This implicitly creates a graph that tracks the dependencies between signals. A background loop constantly enumerates the signals, detecting stale children and computing new values for them by executing their lift function. This function is evaluated in the lexical scope of the Racket evaluator again, allowing for closures and other possible language constructs. The reactive language provides a few sample source signals, which update automatically and have no parents. The signal 'current-seconds' is such an example, it emits every second a new value indicating the number of seconds since midnight UTC, January 1, 1970. Another signal called 'random-integer' produces random numbers between 1 and 100 in randomized time intervals. All new signals have to be derived from at least one parent signal, and must output a value when one of the parents emits a new value.

Creating a new signal

```
;; ======================================
;; Signals: signals can derive from other signals
;;          taking their values as input and
;;          producing something of their own
;; ======================================
;; Object structure: [
;;   value           : any,
;;   has-value?      : boolean
;;   up-to-date?     : boolean,
;;   parents         : [signal],
;;   value-provider  : val1, val2, ... => value,
;;   children        : [signal]
;; ]
(define (make-signal parents value-provider)
  ;; make a vector representing the signal (see object structure above)
  (define $signal
        (list->vector (list null #f #f parents value-provider '())))

  ;; enumerate the parents and register the new child
  (for-each (lambda ($parent) (signal-add-child! $parent $signal)) parents)

  ;; return the newly created signal
  $signal)
```

Signals are stored in memory as simple vectors, keeping a reference to both parents and children to facilitate the implementation of the update loop.

Evaluating the lift function

```
;; ============================================
;; Lifting: One or more signals can be lifted with
;;          a procedure to create a new signal
;; Syntax: (lift (lambda (value1 value2 ...) ( ... ))
;;          $signal1 $signal2 ...)
;; ============================================
(define (lift? exp) (tagged-list? exp 'lift))
(define (lift-operator exp) (cadr exp))
(define (lift-signals exp) (cddr exp))

(define (eval-lift operator-exp signal-exps env)
  ;; evaluate  the operator in the environment
  (define operator (eval operator-exp env))

  ;; evaluate the parent signals
  (define wrapped-parents (map (lambda (signal-exp) (eval signal-exp env))
      signal-exps))

  ;; unwrap the parents (signals are wrapped in tagged lists to
      differentiate them)
  (define parents (map signal-wrapper-unwrap wrapped-parents))

  ;; wrap the operation so that its execution happens in the scope of the
      environment
  (define value-provider (lambda parent-values (apply-in-scope operator
      parent-values)))

  ;; return the wrapped newly created signal
  (make-signal-wrapper (make-signal parents value-provider)))
```

Lifting signals creates a new signal, a scoped operator to produce values based on the parents and returns a wrapped signal to the environment.

The update loop

```
;; ========================================
;; The update loop: Continuously updates the signals
;; ========================================

;; signal-update: Procedure that updates the value of a signal
;;                provided all parents have values
(define (signal-update! $signal)
  (define parents (signal-parents $signal))
  (define children (signal-children $signal))
  (define parents-have-values (map signal-has-value? parents))
  (define all-parents-have-values? (foldl and-2 #t parents-have-values))
  (if all-parents-have-values?
    (let ((value-provider (signal-value-provider $signal))
          (parent-values (map signal-value parents)))
         (signal-value! $signal (apply value-provider parent-values))
         (signal-up-to-date! $signal #t)
         (for-each (lambda ($child) (signal-up-to-date! $child #f))
                   children))
  #f))

;; source-signals: signals without a parent
(define source-signals (list $current-seconds $random-integer))

;; get-topologically-sorted-signals: sorts the signals in topological order
;;                                   signals with fewest dependencies first
(define (get-topologically-sorted-signals)
  (define (topological-sort accumulator next-children)
    (if (null? next-children)
        accumulator
        (topological-sort (append accumulator next-children)
                          (foldl append '() (map signal-children
                                                 next-children)))))
  (topological-sort '() source-signals))

;; update-signals-loop: loops infinitely and updates signals if necessary
(define (update-signals-loop)
  (define signals (get-topologically-sorted-signals))
  (for-each signal-update! (filter (compose not signal-up-to-date?)
                                   signals))
  (sleep 0.05)
  (update-signals-loop))
```

The signal update loop repeats every 5ms and checks if any signals need to be updated. For those that do, it evaluates the lift operator with the values of the parents, provided all of them have values. It then flags the signal as being up to date, but flags the children as stale. These will be picked up in the next iteration of the loop. Note that if the update loop takes too long, it is possible that incoming values of the source-signals are never processed. This problem can be tackled by switching to a callback based update loop, where updates to a signal immediately trigger the updating of all children. Unfortunately this approach also suffers a few drawbacks: the application would freeze if signals provide values too quickly.

3.3 Conclusion

In this chapter the reactive language was presented with samples and snippets of its implementation. It is built as a metacircular evaluator, evaluating Racket expressions in a simulated runtime that forwards statements to the real underlying Racket implementation.

This language can be used to model reactive data flows and provides a built in update loop to manage the data dependencies between signals. It does this by looping over a node graph that tracks the dependencies between signals. New signals can be created by deriving from other signals and a lift function which transforms the values of the parent signals into a single value.

4
Engine

5
Evaluation

6

Future work and limitations

7
Related work

8
Conclusion

A

Your Appendix

Bibliography

Abelson, H., Sussman, G. J., & Sussman, J. (1999). *Structure and Interpretation of Computer Programs* (Second ed.). London, UK, UK: McGraw-Hill Book Company.

Cooper, G. H., & Krishnamurthi, S. (2006, March). Embedding Dynamic Dataflow in a Call-by-Value Language. In *Programming Languages and Systems* (pp. 294–308). Springer, Berlin, Heidelberg. Retrieved 2017-05-27, from `https://link.springer.com/chapter/10.1007/11693024_20` doi: 10.1007/11693024_20