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 de Toegepaste Informatica

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

List of Figures

List of Tables

1
Introduction

2 Background

2.1 Introduction

This research builds on two existing paradigms in software development: Reactive Programming and Dataflow Programming. 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. Dataflow programming on the other hand can be considered more 'low level', providing a optimized engine for fast function execution.

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 between values must be recorded once, which can be seen as a (possibly cyclic!) directed graph. When a new value is recorded, it is appended to the end of the graph, where the new node "subscribes" to values from the source nodes. Whenever these source nodes produce new data, they will notify this node, at which point it can recompute what its own value should be. In this way, values ripple through the direct graph of nodes, updating nodes wherever they pass through. Note that values can only ever be appended at the end of the graph, i.e. add more derived values. Once such a "reactive node" - often called an observable or signal in literature - has been defined, it is impossible to modify the source nodes which will feed it data downwards in the graph.

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. 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, effectively mutating a. 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. 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

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. As it turns out, 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 Dataflow Programming

2.3.1 Introduction

Dataflow Programming is a paradigm focused on the optimal, parallel execution of functions. In this paradigm, functions are seen as isolated units of code which should be able to execute whenever the necessary parameters have been provided. Contrary to imperative programming, functions are therefore not called directly, but rather whenever all of the parameters are present. The output of that function is then pushed into the parameter queue again, ready to be sent to the next function which takes it as its input. Function parameters are typically wrapped in tokens, which carry meta data information about which execution context they belong to, to isolate multiple calls to the same function from one another. In this way, the execution of functions in Dataflow Programming can also be seen as a direct graph of nodes where each node represents a function and each edge represents the output of a function being sent to another function. It is up to the dataflow engine to orchestrate the queue of tokens so that functions are executed correctly and in the correct order.

A large difference with Reactive Programming is that Dataflow Programming puts the function call 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: function calls 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!

In a dataflow engine, the values of b and c would be added to the token queue at application startup. B would be entered as a token twice; once for the function call "+" which computes a and once for the function call "+" which computes d. When the dataflow engine spins up and starts processing tokens, it sends the tokens for b and c to the first "+" function, 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 "+" function. This function 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 functions not shown in the sample code) produce new values, these would be enqueued again for further processing.

Do note that, unlike in Reactive Programming, the execution of a function consumes the parameters, which means a function will not execute again until new tokens are present for all of its parameters.

2.3.3 Advantages

One of the key advantages of Dataflow Programming is its isolation of function executions, completely removing the need for shared state. Since Dataflow functions are only allowed to access data from its provided parameters, it cannot rely on external state outside the scope of the dataflow engine. This allows the execution of these functions to be distributed across different processors and even separate machines, ensuring optimal parallelization.

2.4 Conclusion

Two paradigms for software development were presented: Reactive Programming and Dataflow programming. 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. Dataflow Programming on its part focuses more on the optimal, parallel execution of functions 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 research, a lightweight reactive language was implemented to facilitate the process of porting it to a dataflow engine afterwards. Having control of the inner peculiarities of the timings and delivery mechanism of data would prove to be crucial in adapting to the dataflow world. The main goal was to provide a simple reactive interpreter that would keep the signals up to date in the proper manner. Later on, this mechanism would then be built using a dataflow engine. The interpreter does not compile down to assembly, but evaluates immediately in the underlying Racket language.

3.2 Reactive Metacircular Evaluator

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
```

3.2.1 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.

3.2.2 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

References