Reactive Single-Page Applications with Dynamic Dataflow

Simon Fowler[1], Loïc Denuzière[2], and Adam Granicz[2]

[1] University of Edinburgh,
simon.fowler@ed.ac.uk
[2] IntelliFactory,
{loic.denuziere, granicz.adam}@intellifactory.com,
http://www.intellifactory.com

Abstract. Modern web applications are heavily dynamic. Several approaches, including functional reactive programming and data binding, allow a presentation layer to automatically reflect changes in a data layer. However, many of these techniques are prone to unpredictable memory performance, do not make guarantees about node identity, or cannot easily express *dynamism* in the dataflow graph.
We identify a point in the design space for the creation of statically-typed, reactive, dynamic, single-page web applications for the WebSharper framework in the functional-first language F#. We provide an embedding abstraction to link a dynamic dataflow graph to a DOM presentation layer in order to implement dynamic single-page applications, and show how the technique can be used to support declarative animation.

Keywords: Functional Programming; Reactive Web Applications; F#

1 Introduction

The web has grown from a collection of static, textual websites to a platform allowing complex, fully-fledged applications to run in a browser. A key advance has been the ability of page content to change, in particular as a result of changes to underlying data.

Changing the DOM via callback functions is adequate for small applications, but the inversion of control introduced by callback functions makes it difficult to maintain larger applications, and the code to update the presentation layer invariably becomes entangled with application logic. Techniques such as data binding allow mutable data to be inserted into the DOM, with the presentation layer automatically reflecting these changes. Functional reactive programming (FRP) [7] introduces *Signals* and *Behaviours*, where values can be treated as a function of time. Several successful implementations exist: React [1] provides an efficient data-binding system, and Elm [5] is a popular language designed for creating reactive web applications using FRP.

The design space, however, is vast. FRP, while having an extremely clean and expressive semantics, is prone to memory leaks when using higher-order signals.

As a result, Elm's type system forbids higher-order signals and the creation of new signals, using signal transformers from arrowised FRP [13] to achieve dynamism. Applications written with React are not statically-typed and make few guarantees about the preservation of the *identity*, including internal state, such as focus, of DOM nodes.

WebSharper[3] is a framework allowing web applications to be written entirely in the functional-first language F# [17]. This is achieved by compiling quoted F# expressions to JavaScript, with raw DOM elements and events encapsulated using a functional interface. Designing a framework, `UI.Next`, for *reactive single-page applications* in WebSharper required us to identify a point in the design space fulfilling the following key properties.

Dynamism It must be possible for the dataflow graph to consist of dynamic sub-graphs, where the structure of these sub-graphs may change over the course of the application's execution.

Predictable Memory Usage Purely monadic FRP systems must sometimes retain the entire history of a value in order to use higher-order signals. The framework must not mandate such memory leaks in order to preserve the semantics of the reactive system.

Composability It should be simple to compose elements in both the dataflow and presentation layers. Layers in the dataflow layer should compose using applicative and monadic abstractions, and it should be simple to integrate the dataflow and presentation layers.

Standard Type Systems The system should not require any non-standard type system features in order to fulfil the above properties.

Control over Node Identity The user should be able to explicitly specify whether DOM nodes are shared or regenerated upon changes in data.

1.1 Contributions

As a result of our design and implementation guided by the above principles, we report on the following scientific contributions.

- We describe a dynamic dataflow graph consisting of parameterised views of data sources, connected in a weak fashion by `Snap`s, a specialised extension of the `IVar` primitive [15]. This connects parameterised views of data sources in the dataflow graph, supporting asynchronous loading of variables, preventing glitches, and ensuring the graph is amenable to garbage collection (Section 3).
- We introduce a monoidal API for specifying DOM elements, provide abstractions to integrate this reactive DOM layer with the dynamic dataflow graph, and describe the implementation of this integration (Section 4).
- We demonstrate how a declarative animation API can be integrated with the DOM layer, making use of limited history-dependence, and can be backed by the dataflow graph (Section 5).

[3] `http://www.websharper.com`

UI.Next is freely available online at http://www.github.com/intellifactory/websharper.ui.next. Example applications can be found at http://intellifactory.github.io/websharper.ui.next.samples; the samples site itself is also written using UI.Next.

2 UI.Next by Example

UI.Next focuses on the creation of reactive, single-page applications. Before describing the implementation in detail, we provide an example of a calculator application, with standard and scientific modes.

We begin by defining data types for modes, a set of binary and unary operations, and a record to model the calculator. The calculator has one number in its memory in order to support binary operations, and a current operand and operation. We also define functions to execute the numerical operations.

```
type Mode = Standard | Scientific
type BinOp = Add | Sub | Mul | Divide | Exp | Mod
type UnOp = Sin | Cos | Tan | Squared
type Op = BinaryOp of BinOp | UnaryOp of UnOp
type Calculator = { Memory : float ; Operand : float ; Operation : Op }

let binOpFn = function
    | Add -> (+)    | Sub -> (-)
    | Mul -> ( * )  | Divide -> (/)
    | Exp -> ( ** ) | Mod -> (%)

let unOpFn = function
    | Squared -> fun x -> pown x 2
    | Sin -> sin    | Cos -> cos
    | Tan -> tan
```

There are two main reactive primitives in UI.Next: Vars, which can be thought of as observable mutable reference cells, and Views, which are read-only projections of Vars in the dataflow graph, and can be combined using applicative and monadic functional abstractions. In the following functions, rvCalc is a Var containing the current calculator state. Var.Update updates a variable based on its current value.

When a number button is pressed, the number is added to the current operand multiplied by 10 (pushInt). Pressing a unary operation button applies it to the current operand. When a binary operation is pressed, the number is placed into the memory, the operation is stored, and the operand is set to zero (shiftToMem). The user can then type another number, and pressing the equals button will apply the operation to the number in memory and current operand (calculate).

```
let pushInt x rvCalc =
  Var.Update rvCalc (fun c -> { c with Operand = c.Operand * 10.0 + x})
let shiftToMem op rvCalc =
  Var.Update rvCalc (fun c ->
    { c with Memory = c.Operand; Operand = 0.0; Operation = op }

let calculate rvCalc =
  Var.Update rvCalc (fun c ->
    let ans =
      match c.Operation with
        | BinaryOp op -> binOpFn op c.Memory c.Operand
        | UnaryOp op -> unOpFn op c.Operand
  { c with Memory = 0.0 ; Operand = ans ; Operation = BinaryOp Add } )
```

The next step is to create a view for the model, allowing it to be embedded into a web page. In order to do this, we create and combine elements of type `Doc`, a monoidally-composable representation of a DOM tree, which may contain both static and reactive fragments.

The "screen" of the calculator should display the current operand. This is done by mapping a serialisation function onto the current operand, converting it to a string (resulting in a type of `View<string>`), and creating a `Doc.TextView` representing a DOM text node which will update every time the `View` updates. We make use of the F# 'pipe' operator (`a |> f = f a`).

```
let numberDisplay rvCalc =
  let rviCalc = View.FromVar rvCalc
  View.Map (fun c -> string c.Operand) rviCalc |> Doc.TextView
```

We next define the "keypad" of the calculator. We define several button creation functions using the `Doc.Button` function, which takes as its arguments a caption, list of attributes, and a callback function to update the calculator state. `Div0` constructs a `Doc` representing a `<div>` tag, without attributes.

```
let calcBtn i rvCalc = Doc.Button (string i) [] (fun _ -> pushInt i rvCalc)
let cbtn rvCalc = Doc.Button "C" [] (fun _ -> Var.Set rvCalc initCalc)
let eqbtn rvCalc = Doc.Button "=" [] (fun _ -> calculate rvCalc)
let uobtn o rvCalc = Doc.Button (showOp o) [] (fun _ -> setOp o rvCalc; calculate
     rvCalc)
let bobtn o rvCalc = Doc.Button (showOp o) [] (fun _ -> shiftToMem o rvCalc)
let keypad rvCalc =
  let btn num = calcBtn num rvCalc
  Div0 [
    Div0 [btn 1.0 ; btn 2.0 ; btn 3.0 ; bobtn (BinaryOp Add) rvCalc]
    Div0 [btn 4.0 ; btn 5.0 ; btn 6.0 ; bobtn (BinaryOp Sub) rvCalc]
    Div0 [btn 7.0 ; btn 8.0 ; btn 9.0 ; bobtn (BinaryOp Mul) rvCalc]
    Div0 [btn 0.0 ; cbtn rvCalc; eqbtn rvCalc; bobtn (BinaryOp Divide) rvCalc]
  ]
```

We may then declare the operations which are present in scientific mode, and two rendering functions, `standardCalc` and `scientificCalc`, composing each set of components.

```
let scientificOps rvCalc =
  Div0 [
    bobtn (BinaryOp Exp) rvCalc ; bobtn (BinaryOp Mod) rvCalc
    uobtn (UnaryOp Sin) rvCalc  ; uobtn (UnaryOp Cos) rvCalc
    uobtn (UnaryOp Tan) rvCalc  ; uobtn (UnaryOp Squared) rvCalc
  ]
let standardCalc rvCalc = Div0 [ numberDisplay rvCalc; keypad rvCalc ]
let scientificCalc rvCalc =
  Div0 [ numberDisplay rvCalc; scientificOps rvCalc; keypad rvCalc ]
```

Finally, we create two radio buttons to switch between standard and scientific modes, which set the `rvMode` variable accordingly, and create a `View` of `rvMode`. We then map the appropriate rendering function to create a `View<Doc>`, which can be embedded using the `Doc.EmbedView: View<Doc> -> Doc` function.

```
let calcView rvCalc rvMode =
  let modeButtons =
    [Div0 [Doc.Radio [] Standard rvMode ; Doc.TextNode "Standard"]
    Div0 [Doc.Radio [] Scientific rvMode ; Doc.TextNode "Scientific"]] |> Doc.Concat
  View.FromVar rvMode
  |> View.Map (fun mode ->
    let body =
      match mode with | Standard -> standardCalc | Scientific -> scientificCalc
    [body rvCalc; modeButtons] |> Doc.Concat
  ) |> Doc.EmbedView
```

3 Dataflow Layer

The dataflow layer exists to model data dependencies and consequently to perform change propagation. The layer is specified completely separately from the reactive DOM layer, and as such may be treated as a render-agnostic data model.

The dataflow layer consists of two primitives: reactive variables, `Var`s, and reactive views, `View`s. A `Var` is a data source, parameterised over a type: this is equivalent to a mutable reference cell with the notable exception that it may be *observed* by `View`s. A `View` represents a snapshot of a `Var`, and may be composed using applicative and monadic functional combinators.

In terms of the dataflow graph, a `Var` is a source node, and can have no incoming edges. A `View` is a node which must have at least one incoming edge. Edges in the graph are *not* direct pointers between nodes: nodes can be abstractly considered as communicating processes using a `Snap`, a novel, specialised variation of the Concurrent ML IVar primitive. As a result, the dataflow layer is amenable to garbage collection: if a `Var` or `View` becomes eligible for garbage collection, all dependent `View`s in the dataflow graph will be automatically garbage collected without the need for explicit unsubscription.

```
type View =
  static member Const : 'T -> View<'T>
  static member FromVar : Var<'T> -> View<'T>
  static member Sink : ('T -> unit) -> View<'T> -> unit
  static member Map : ('A -> 'B) -> View<'A> -> View<'B>
  static member MapAsync : ('A -> Async<'B>) -> View<'A> -> View<'B>
  static member Map2 : ('A -> 'B -> 'C) -> View<'A> -> View<'B> -> View<'C>
  static member Apply : View<'A -> 'B> -> View<'A> -> View<'B>
  static member Join : View<View<'T>> -> View<'T>
  static member Bind : ('A -> View<'B>) -> View<'A> -> View<'B>
```

`Var`s can be initialised, their values can be set, or they can be marked as finalised if their value no longer changes. `FromVar` creates a `View` which observes a `Var`, and `Const` creates a `View` which consists of a static, non-changing value. The `Sink` function acts as an *imperative observer* of the `View` – that is, the possibly side-effecting callback function of type `('T -> unit)` is executed whenever the value being observed changes. We use the `Sink` function to integrate the dataflow layer with the reactive DOM layer, which is further explained in Section 4.

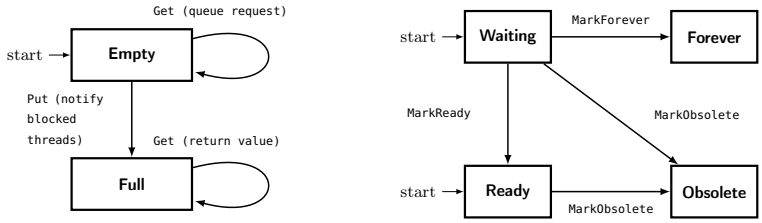
The remaining abstractions are standard combinators for applicative and monadic composition. Monadic composition allows dynamism in the dataflow graph, which is crucial for implementing dynamic single-page applications.

3.1 Implementation

In this section, we describe the implementation of the dataflow layer. A `Var` consists of a current value, a flag describing whether or not the `Var` is finalised and will not change, and a `Snap`.

```
type Var<'T> = { mutable Const : bool; mutable Current : 'T; mutable Snap : Snap<'T> }
```

A `Snap` can be thought of as an observable and stateful snapshot of the contents of a `Var`. At its core, a `Snap` is based on the notion of an *immutable variable*, or IVar [15]. An IVar is created as an empty cell, which can be written to only once. Attempting to read from a 'full' IVar will immediately yield the value contained in the cell, whereas attempting to read from an 'empty' IVar will result in the thread blocking until such a variable becomes available. This is shown in Figure 1a.



(a) `IVar` (b) `Snap`

Fig. 1: State Transition Diagrams for `IVar`s and `Snap`s

A simple way of implementing change propagation using `IVar`s instead of pointers is to associate `Var`s and `View`s with an IVar `obsolete` of unit type. Dependent nodes read an initial value from the data source, attempt to perform the `Get` operation on `obsolete`, and block since `obsolete` is empty[4]. Upon changing the value, `Put` is called on `obsolete`, and all dependent nodes are notified and can fetch the latest value. Finally, `obsolete` is replaced by a fresh IVar, and the process repeats.

This model is intuitive and conveys the essence of the approach. The realisation of this technique in `UI.Next`, a `Snap`, is slightly more complex in order to support applicative and monadic combinators, perform certain optimisations, prevent certain classes of leaks, and to better support asynchronously populating a `View` from an external data source using the `MapAsync` operation. A `Snap` can be thought of as a state machine consisting of four states:

[4] We make use of the F# asynchronous programming model on the client by using a custom scheduler built into the WebSharper runtime: creating threads is done by queueing functions for execution, which are executed in a round-robin style.

Ready: A `Snap` containing an up-to-date value, and a list of threads to notify when the value becomes obsolete.

Waiting: A `Snap` without a current value. Contains a list of threads to notify when the value becomes available, and a list of threads to notify should the `Snap` become obsolete prior to receiving a value. This is required for the implementation of the `MapAsync` combinator, and represents a `Snap` wherein a request has been made for a value, but it has not yet been received.

Forever: A `Snap` containing a value that will never change. This prevents nodes waiting for the `Snap` to become obsolete when this will never be the case.

Obsolete: A `Snap` containing obsolete information, signifying that the `View` should obtain a new snapshot.

The state transition diagram for a `Snap` is shown in Figure 1b. `Snap`s can be modified by three operations. `MarkForever` updates the `Snap` with a value which will never change, transitioning to the `Forever` state. `MarkReady` marks the `Snap` as containing a new value, notifying all waiting threads. Finally, `MarkObsolete` marks the `Snap` as obsolete. An additional operation `MarkDone` checks if the `Snap` is in the `Forever` state, and if not, transitions to the `Ready` state. `Snap`s support a variety of applicative and monadic combinators in order to implement the operations provided by `View`s: to implement `Map2` for example, a `Snap` must be created which is marked as obsolete as soon as either of the two dependent `Snap`s becomes obsolete.

`Var`s support an operation, `SetFinal`, which marks the value as finalised, preventing further writes to the variable. This prevents a class of leaks wherein a `Var` which remains static is continually observed.

`Snap`s are used to drive change propagation. When the value of a `Var` is updated, the current `Snap` is marked as obsolete and replaced by a new `Snap` in the `Ready` state.

```
type View<'T> = V of (unit -> Snap<'T>)
static member FromVar var = V (fun () -> var.Snap)
static member Set var val =
  if var.Const then () // Invalid
  else Snap.MarkObsolete var.Snap;
    var.Current <- val; var.Snap <- Snap.CreateWithValue val
```

At its core, a `View` consists of a function `observe` to return a `Snap`. The simplest `View` directly observes a single `Var`: this simply accesses the current `Snap` associated with that `Var`, updating whenever the `Snap` becomes obsolete.

At a high level, implementing `View` combinators for applicative and monadic composition involves creating a `View` with an observation function which uses the underlying `Snap` combinators. `View`s are created lazily, and results are cached for efficiency. When a `Snap` becomes obsolete, the observation functions are called to yield new `Snap`s.

```
static member Map fn (V observe) =
  View.CreateLazy (fun () -> observe () |> Snap.Map fn)
static member Map2 fn (V o1) (V o2) =
  View.CreateLazy (fun () -> let s1 = o1 (); let s2 = o2 () Snap.Map2 fn s1 s2)
```

```
static member CreateLazy observe =
 let cur = ref None
 let obs () =
   match !cur with
   | Some s when not (Snap.IsObsolete s) -> s
   | _ -> let sn = observe (); cur := Some sn; sn
 V obs
```

In order to react to lifecycle events and trigger change propagation through the dataflow graph, the `When` eliminator function is used.

```
val When : Snap<'T> -> ready: ('T -> unit) -> obsolete: (unit -> unit) -> unit
```

The `When` function takes a `Snap` and two callbacks: `ready`, which is invoked when a value becomes available, and `obsolete`, which is invoked when the `Snap` becomes obsolete. This is implemented by matching on the state of the `Snap`, and adding the callback to the appropriate queue.

```
let Map fn sn =
  let res = Create ()
  When sn (fn >> MarkDone res sn) (fun () -> MarkObsolete res) ; res

let Map2 fn sn1 sn2 =
  let res = Create (); let v1 = ref None; let v2 = ref None
  let obs () = v1 := None; v2 := None; MarkObsolete res
  let cont () =
    match !v1, !v2 with
      | Some x, Some y -> MarkReady res (fn x y)    | _ -> ()
  When sn1 (fun x -> v1 := Some x; cont ()) obs
  When sn2 (fun y -> v2 := Some y; cont ()) obs ; res
```

The `Snap.Map` function takes a dependent `Snap sn` and a function `fn` to apply to the value of `sn` when it becomes available. Firstly, an empty `Snap`, `res`, is created. This is passed to the `When` eliminator along with two callbacks: the first, called when `sn` is ready, marks `res` as ready, containing the result of `fn` applied to the value of `sn`. The second, called when `sn` is obsolete, marks `res` as obsolete.

The `Snap.Map2` function applies a function to multiple arguments, which can in turn be used to implement applicative combinators. In order to do this, a `Snap res` and two mutable reference cells, `v1` and `v2`, are used. When either of the dependent `Snaps` `sn1` or `sn2` update, the corresponding reference cell is updated and the continuation function `cont` is called. If both of the reference cells contain values, then the continuation function marks `res` ready, containing the result of `fn` applied to `sn1` and `sn2`. If either of the dependent `Snaps` become obsolete, then `res` is marked as obsolete. This avoids glitches, which are intermediate states present during the course of change propagation, and avoids such intermediate states being observed by the reactive DOM layer.

3.2 Identity-Preserving Conversion Functions

We provide several transformation functions on reactive collections, which allow stateful conversion by using shallow memoisation: that is, where inputs are equal,

previous outputs are re-used. Only one previous value for each entry in the sequence is stored, meaning that the memory usage of these functions is linear in the size of the longest sequence in the `View`. This allows *identity* to be preserved: this is particularly useful for sharing `Doc`s upon updates, preventing needless DOM node regeneration and loss of internal DOM node state. This allows the transformations to have an amount of history-dependence: this is important when incorporating the notion of identity into animations, for example, as described in Section 5.2. Conversion functions are parameterised over either two or three type parameters; `'A` and `'B` are the input and output types respectively, while `'K` is the type of an equality key. The `when 'A : equality` constraint specifies that the `'A` type must support equality testing.

```
static member Convert<'A,'B when 'A : equality>:
  ('A -> 'B) -> View<seq<'A>> -> View<seq<'B>>
static member ConvertBy<'A,'B,'K when 'K : equality>:
  ('A -> 'K) -> ('A -> 'B) -> View<seq<'A>> -> View<seq<'B>>
static member ConvertSeq<'A,'B when 'A : equality>:
  (View<'A> -> 'B) -> View<seq<'A>> -> View<seq<'B>>
static member ConvertSeqBy<'A,'B,'K when 'K : equality>:
  ('A -> 'K) -> (View<'A> -> 'B) -> View<seq<'A>> -> View<seq<'B>>
```

The `Convert` function can be thought of as converting a sequence of values, and re-using output values from the previous step should the inputs be determined to be equal. The `ConvertSeq` function is an extension of this notion, wherein the conversion function accepts a reactive view: changes to each individual item of the collection (as detected by either a machine- or user-specified notion of equality) are propagated on the item-level using this `View`.

4 Reactive DOM Layer

The Reactive DOM layer exists as a presentation layer for the dynamic dataflow graph, allowing changes in the dataflow graph to be automatically propagated to the DOM. In this section, we detail the design and implementation of the reactive DOM layer, showing how an in-memory representation of the DOM can be linked with the dataflow graph. We show how this can be used to batch updates, prevent visual glitches, and preserve the *identity* (internal state such as focus) of nodes. The simplest example of the integration of the dataflow and DOM layers is a text label which mirrors the contents of an input text box.

```
let rvText = Var.Create "" ; let inputField = Doc.Input [] rvText
let label = Doc.TextView rvText.View ; Div0 [ inputField; label ]
```

We begin by declaring a variable `rvText` of type `Var<string>`, which is a reactive variable to hold the contents of the input box. Secondly, we create an input box which is associated with `rvText`, meaning that whenever the contents of the input field changes, `rvText` will be updated accordingly. Next, we create a label using `Doc.TextView`, which we associate with a view of `rvText`. Finally, we construct a `<div>` tag using a monoidal DOM API.

Another example is that of a to-do list, where the item should be rendered with a strikethrough if the task has been completed. Arguably the most important function within the Reactive DOM layer is the `Doc.EmbedView` function:

```
static member EmbedView : View<Doc> -> Doc
```

Semantically, this allows us to embed a *reactive* DOM fragment into a larger DOM tree. This is the key to creating reactive DOM applications using the dataflow layer: by using `View.Map` to map a rendering function onto a variable, we can create a value of type `View<Doc>` to be embedded using `EmbedView`.

We begin by defining a simple type, with a reactive variable of type `Var<bool>` which is set to true if the task has been completed. An item can be rendered by mapping a rendering function onto a `View` of this variable; note that in the code listing below, `Del0` is a notational shorthand for an HTML `<del>` element without any attributes, and `Doc.TextNode` creates a DOM text node.

```
type TodoItem = { Done : Var<bool> ; TodoText : string }
View.FromVar todo.Done
|> View.Map (fun isDone ->
  if isDone then Del0 [ Doc.TextNode todo.TodoText ] else Doc.TextNode todo.TodoText)
|> Doc.EmbedView
```

4.1 Design

Reactive elements are created using the `Doc.Element` function, which takes as its arguments a tag name, a sequence of attributes, and a sequence of child elements.

```
static member Element : name: string -> seq<Attr> -> seq<Doc> -> Doc
```

Reactive attributes have type `Attr` and can be static, dynamic, or animated. Static attributes correspond to simple key-value pairs, as found in traditional static sites, whereas dynamic attributes are instead backed by a `View<string>`. We defer discussion of animation attributes to Section 5.

A key design decision is to use a *monoidal interface* for both DOM elements and attributes. All DOM elements in the reactive DOM layer are of type `Doc`. To form a monoid, `Doc`s support `Empty`, and `Append` and `Concat` functions. Reactive attributes of type `Attr` support the same interface.

4.2 Implementation

The Reactive DOM layer consists of a skeleton representation of the DOM tree in memory. Each node in this skeleton representation contains a `View` of unit type, and updates are propagated upwards through the tree. When the DOM skeleton is marked as changed, a message is sent to an update process, which applies the changes to the DOM.

DOM Skeleton Representation The internal structure of a `Doc` is a pair of a `DocNode`, which indicates what the `Doc` represents, and a `View updates` of type `View<unit>`, which is used to notify the update process that part of the tree has changed.

```
type DocNode =
  | AppendDoc of DocNode * DocNode   | ElemDoc of DocElemNode
  | EmbedDoc of DocEmbedNode | EmptyDoc | TextDoc of DocTextNode
type DocTextNode = { Text : TextNode; mutable Dirty : bool; mutable Value : string }
type DocElemNode = { Attr : Attrs.Dyn; Children : DocNode; El : Element; ElKey : int }
type DocEmbedNode = { mutable Current : DocNode; mutable Dirty : bool }
```

Moreover, `DocNode` is a discriminated union consisting of five possible types of node. To support the monoidal interface, `AppendDoc` denotes two sibling nodes, and `EmptyDoc` denotes the absence of an element.

An `ElemNode` represents a DOM element, consisting of the attributes associated with the elements, the skeleton representation of the children of the element, the DOM element itself, and a key which is used for equality testing.

A `TextNode` represents a DOM text node, consisting of the current value, the current in-memory DOM node, and a `Dirty` flag used for DOM synchronisation. Finally, an `ElemNode` is used to represent a reactive `View` embedded into the tree. This consists of a *mutable* `DocNode` to represent the changes, and `Dirty` flag to specify that either the entire subtree, or an element within the subtree has changed.

Integration with Dataflow Layer The main entry point to a reactive application is the `Doc.Run` function, which attaches a reactive DOM fragment of type `Doc` with a standard DOM element. The `Doc.Run` function is implemented by spawning an update process providing actor-like concurrency. Whenever a message is received by this update process, the update process firstly performs any animations that may be necessary (described further in Section 5.1), and synchronises the in-memory DOM representation with the physical DOM.

The key to the integration between the dataflow and reactive DOM layers is the `Updates View` associated with each `Doc`. The key idea for the integration of these two layers is that a notification for an update is propagated *upwards* through the tree. Once the notification propagates to the top of the `Doc` tree, the update process is notified in order to trigger any animations and synchronise the virtual and physical DOM representations.

Combining the `Views` associated with each `Doc` is done through the use of the standard `View` combinators. As an example, consider the `Doc.Append` function, which appends two `Docs` as siblings. The `AppendDoc` node requires an update either of the two contained `Docs` require an update: this can be achieved using the `Map2` combinator. `Docs.Mk` is simply a constructor for `Doc`. The `||>` operator is similar to `|>`, but takes a tupled argument, applying both arguments to the function.

```
static member Append a b =
  (a.Updates, b.Updates) ||> View.Map2 (fun () () -> ())
  |> Docs.Mk (AppendDoc (a.DocNode, b.DocNode))
```

EmbedView Implementation `EmbedView` allows a reactive DOM segment to be embedded within the DOM tree, with any updates in this segment being reflected within the DOM.

```
static member EmbedView view =
  let node = Docs.CreateEmbedNode ()
  view |> View.Bind (fun doc -> Docs.UpdateEmbedNode node doc.DocNode; doc.Updates)
  |> View.Map ignore |> Docs.Mk (EmbedDoc node)
```

`EmbedView` works by creating a new entry in the dataflow graph, depending on the reactive DOM segment. Conceptually, this can be thought of as a `View<View<Doc>>`, which would not be permissible in many FRP systems. Here, the monadic `Bind` operation provided by the dynamic dataflow layer is crucial in allowing us to observe not only changes *within* the `Doc` subtree (using `doc.Updates`), but changes *to* the `Doc` itself: when either change occurs, the `DocEmbedNode` is marked as dirty, and the update is propagated upwards through the tree.

Synchronisation The synchronisation algorithm recursively checks whether any child nodes have been marked as dirty.

In the case of `EmbedNode`s, it is not only necessary to check whether the `EmbedNode` itself is dirty but also whether the current subtree value represented by the `EmbedNode` is dirty: this ensures that both global (entire subtree changes) and local (changes within the subtree) changes have been taken into account. If so, then the updates are propagated atomically to the DOM.

An important consideration of the synchronisation algorithm is the preservation of node *identity* – that is, the internal state associated with an element such as the current input in a text box, and whether the element is in focus. For this reason, when updating the children of a node, simply removing and reinserting all children of an element marked dirty is not a viable solution: instead we associate a *key* with each item, which is used for equality checking, and perform a set difference operation to calculate the nodes to be removed.

As the synchronisation process is only triggered when updates are required, the synchronisation process applies updates in a *batched* fashion, meaning that there is no visible 'cascade' of updates.

5 Declarative Animation

Animations in web applications are typically implemented as an interpolation between attribute values over time. CSS has some native animation functionality, but the approach founders when animations depend explicitly on dynamic data and cannot be determined statically. The D3 library [4] provides more powerful animation functionality, with a particular focus on data visualisation, but targets a more imperative style of programming.

`UI.Next` animations can be attached directly to elements and therefore react directly to changes within the dataflow graph. An animation is defined using the `Anim<'T>` type, where the `'T` type parameter defines the type of value to be interpolated during the animation. An `Anim<'T>` type is internally represented as a function `Compute`, mapping a normalised time to a value, and the duration of the animation.

```
type Anim<'T> = { Compute : Time -> 'T; Duration : Time }
```

An animation can be constructed using the `Anim.Simple` function, which takes as its arguments an interpolation strategy, an easing function, the duration of the animation, the delay of the animation in milliseconds, and the start and end values. Collections of animations can be described using a monoidal interface.

```
static member Anim.Simple :
  Interpolation<'T> -> Easing -> duration: Time -> delay: Time -> startValue: 'T ->
       endValue: 'T -> Anim<'T>
```

Transitions are specified using the `Trans` type.

```
static member Create : ('T -> 'T -> Anim<'T>) -> Trans<'T>
static member Trivial : unit -> Trans<'T>
static member Change : ('T -> 'T -> Anim<'T>) -> Trans<'T> -> Trans<'T>
static member Enter : ('T -> Anim<'T>) -> Trans<'T> -> Trans<'T>
static member Exit : ('T -> Anim<'T>) -> Trans<'T> -> Trans<'T>
```

A transition can either be created with the `Trivial` function, meaning that no animation occurs on changes, or with an animation. Enter and exit transitions, which occur when a node is added or removed from the DOM tree respectively, can be specified using the `Enter` and `Exit` functions.

An animation is embedded within the reactive DOM layer as an attribute through the `Attr.Animated` function:

```
static member Animated : string -> Trans<'T> -> View<'T> -> ('T -> string) -> Attr
```

This function takes the name of the attribute to animate, a transition, a view of a value upon which the animation depends (for example, an item's rank in an ordered list), and a projection function from that value to a string, in such a way that it may be embedded into the DOM.

5.1 Implementation

Animations are triggered as a result of transitions. In order to support transitions, a set of nodes from the previous update is kept at each invocation of the update process. The update process can perform the appropriate set difference operations on these two sets in order to ascertain the sets of animations which must be played as a result of nodes being added or removed.

The JavaScript `requestAnimationFrame` notifies the browser of the intent to perform an animation, and schedules a callback to be performed upon the next browser redraw cycle. The argument provided to this callback is the current timestamp: by calculating the difference between this timestamp and the timestamp at the beginning of the animation, the current point in the animation can be passed to the `Compute` function to calculate the new attribute value.

Animated attributes have an `Updates View`, which is triggered whenever an animation updates the current value of the attribute. This is linked with the remainder of the DOM synchronisation function in the `ElemNode` to which the `Attr` is attached, as the `Updates View` of the element is triggered whenever the element or any of its attributes are updated.

5.2 Example: Object Constancy

Object Constancy is a technique for allowing an object representing a particular datum to be tracked through an animation: consider the case where the underlying data does not change, but can be filtered or sorted. In such a case, the objects representing the data remaining in the visualisation should not be removed and re-added, but instead should transition to their new positions: this relies crucially on the preservation of node identity. Bostock [3] discusses an example displaying the top ten US states for a particular age bracket, sorted by population percentage. We begin by defining a data model.

```
type AgeBracket = AgeBracket of string; type State = State of string
type StateView = {
  MaxValue : double; Position : int; State : string; Total : int; Value : double }
type DataSet =
   { Brackets : AgeBracket []; Population : AgeBracket -> State -> int;
     States : State [] }
```

Here, `AgeBracket` and `State` are representations of age brackets and states respectively, and `DataSet` represents data read from an external source. The `StateView` record specifies details about how a state should be displayed based on other visible items.

```
let SimpleAnimation x y =
  Anim.Simple Interpolation.Double Easing.CubicInOut 300.0 x y
let SimpleTransition = Trans.Create SimpleAnimation
let InOutTransition = SimpleTransition
  |> Trans.Enter (fun y -> SimpleAnimation Height y)
  |> Trans.Exit (fun y -> SimpleAnimation y Height)
```

Using this, it is possible to define an animation lasting for 300ms between 2 given values. With the animation, we can then create two transitions: an unconditional transition `SimpleTransition`, and a transition `InOutTransition` which is triggered when a DOM entry is added (`Enter`) and removed (`Exit`). The `Enter` and `Exit` transitions interpolate the `y` co-ordinate of a bar between the bottom of the SVG graphic (`Height`) and a given position. The element will transition from the origin position to the desired position on, and to the origin on exit.

We now specify a rendering function taking a `View<StateView>` and returning a `Doc` to be embedded within the tree.

```
let Render (state: View<StateView>) =
  let anim name kind (proj: StateView -> double) =
    Attr.Animated name kind (View.Map proj state) string
  let x st = Width * st.Value / st.MaxValue
  let y st = Height * double st.Position / double st.Total
  let h st = Height / double st.Total - 2.
  S.G [Attr.Style "fill" "steelblue"] [
    S.Rect [
      "x" ==> "0"; anim "y" InOutTransition y; anim "width" SimpleTransition x
      anim "height" SimpleTransition h ] []
   ]
```

We specify three projection functions for the width, Y position, and height of the bar, and animated attributes for each. Finally, we create a selection box to allow the user to modify the age bracket. To implement object constancy, we use a key which uniquely identifies the data [9]. For `StateView`, this is `State`, used when embedding the current set of visible elements using `ConvertSeqBy`. The `shownData` argument is a `View` of the data to be displayed, of type `View<seq<StateView>>`.

```
S.Svg ["width" ==> string Width; "height" ==> string Height] [
  shownData |> View.ConvertSeqBy (fun s -> s.State) Render
  |> View.Map Doc.Concat |> Doc.EmbedView ]
```

6 Related Work

Functional Reactive Programming [7] provides *Behaviours* or *Signals*, representing values as a function of time. Early implementations of FRP [7] supported higher-order signals by storing every signal value, creating a memory leak. *Arrowised* FRP [13] allows only *combinators* on primitive signals, manipulated using the Arrow abstraction [10], but avoids memory leaks as a result. The lack of first-class signals makes many GUI programming patterns difficult to implement.

Elm [5] is an FRP-based web programming language. Higher-order signals are forbidden by Elm's type system, allowing history-dependent transformations and avoiding memory leaks. In order to achieve dynamism, Elm implements arrowised FRP. Elm's history-dependence allows the elegant implementation of applications such as games, but without first-class signals and monadic composition, does not support our dynamic SPA pattern. `UI.Next` does not implement FRP signals, but retains first-class dataflow nodes and monadic composition as a result.

Krishnaswami [11] describes a language implementing FRP semantics while guaranteeing leak freedom by dividing expressions into those which may be evaluated immediately, and those which depend on future values; obsolete behaviour values are aggressively deleted. The approach relies on a specialised type system.

React [1] is a reactive DOM library which uses an automated 'diff' algorithm driven by browser redraw cycles instead of the approach we have described. We decided on a dataflow-backed system instead of a diff algorithm to retain complete control over DOM node identity. Flapjax [12] provides similar functionality to `UI.Next`, but has an entirely different approach to the dataflow graph and integrates with the DOM layer differently: signals are instead inserted manually.

The `iTask` framework [14] allows applications to be developed using *workflows*. Interconnected forms are combined using a rich set of combinators. Task-oriented programming is high-level, but is not our target in the design space; abstractions such as Flowlets [2] can handle scenarios such as dependent sequential forms.

SMLtoJS [8] also compiles an ML language (SML) to JavaScript and provides an interface to the DOM API.

7 Conclusion and Future Work

In this paper, we have presented a framework in F#, `UI.Next`, facilitating the creation of reactive applications backed by a dynamic dataflow graph. `Snap`s, an

extension `IVars`, are used as weak links within the dataflow graph to make the graph more amenable to garbage collection and prevent glitches. The DOM layer allows reactive DOM fragments to be embedded using the `EmbedView` function, and uses a monoidal interface. Finally, we have presented an interface for declarative animation which integrates directly into the reactive DOM layer as reactive attributes. We are currently investigating the use of an F# type provider [16] for reactive templating, and are working on formalising the semantics of `UI.Next`, to give a semantics to reactive abstractions such as Flowlets [2] and Piglets [6].

Acknowledgements Fowler is supported by EPSRC grant EP/L01503X/1 (University of Edinburgh School of Informatics CDT in Pervasive Parallelism). Thanks to Anton Tayanovskyy, Adam Harries, and the anonymous reviewers for their useful comments.

References

[1] React | A JavaScript Library for Building User Interfaces. `http://facebook.github.io/react/`, 2014.
[2] J. Bjornson, A. Tayanovskyy, and A. Granicz. Composing Reactive GUIs in F# using WebSharper. IFL '10. 2011.
[3] M. Bostock. Object Constancy. `http://bost.ocks.org/mike/constancy/`, 2012.
[4] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-Driven Documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011.
[5] E. Czaplicki and S. Chong. Asynchronous Functional Reactive Programming for GUIs. PLDI '13, New York, NY, USA, 2013.
[6] L. Denuzière, E. Rodriguez, and A. Granicz. Piglets to the Rescue. 2013.
[7] C. Elliott and P. Hudak. *Functional Reactive Animation*, volume 32(8) of *ICFP '97*, pages 263–273. ACM, New York, NY, USA, 1997.
[8] M. Elsman. SMLtoJs: Hosting a Standard ML Compiler in a Web Browser. 2011.
[9] J. Heer and M. Bostock. Declarative Language Design for Interactive Visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 16(6):1149–1156, 2010.
[10] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37 (1-3):67–111, May 2000.
[11] N. R. Krishnaswami. Higher-order Functional Reactive Programming Without Spacetime Leaks. ICFP '13, New York, NY, USA, 2013.
[12] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. OOPSLA '09, New York, NY, USA, 2009.
[13] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. Haskell '02, New York, NY, USA, 2002.
[14] R. Plasmeijer, P. Achten, and P. Koopman. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. ICFP '07, New York, NY, USA, 2007.
[15] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007.
[16] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. Strongly-typed language support for internet-scale information sources. Technical report, Technical Report MSR-TR-2012-101, Microsoft Research, 2012.
[17] D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. APress, 2012.