Reactive Web Applications with Dynamic Dataflow in F#

Anton Tayanovskyy Simon Fowler Loïc Denuzière Adam Granicz
IntelliFactory, `http://www.intellifactory.com`
{anton.tayanovskyy, simon.fowler, loic.denuziere, granicz.adam}@intellifactory.com

Abstract

Modern web applications depend heavily on data which may change over the course of the application's execution: this may be in response to input from a user, information received from a server, or DOM events, for example.

Much recent work has been carried out with the hope of improving upon the current callback-driven model: in particular, approaches such as functional reactive programming and data binding have proven to be promising models for the creation of reactive web-based user interfaces.

In this paper, we present a framework, `UI.Next`, for the creation of reactive web applications in the functional-first language F#, using the WebSharper web framework. We provide an elegant abstraction to integrate a dataflow layer built on the notion of a dynamic dataflow graph—a dataflow graph which may vary with time—with a DOM frontend, allowing updates to be automatically propagated when data changes. Additionally, we provide an interface for the specification of declarative animations, and show how the framework can ease the implementation of existing functional web abstractions such as Flowlets [3] and Piglets [11].

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Data-flow languages; Applicative (functional) languages

Keywords Dataflow; Web Programming; Functional Programming; Graphical User Interfaces; F#

1. Introduction and Background

Modern web applications depend hugely on data which changes during the course of the application's execution. A common approach to handling this in JavaScript is through the use of callbacks, meaning that whenever a piece of data changes, a callback is executed which performs the appropriate updates.

While this suffices for smaller applications, callbacks become unwieldy as an application grows: inversion of control reduces the ability to reason about applications, often resulting in concurrency issues such as race conditions. Applications also become increasingly difficult to structure, as view modification code becomes intertwined with application logic, decreasing modularity.

The *reactive programming* paradigm provides a promising solution to this problem: instead of relying on state which is mutated by callbacks, reactive approaches work by defining interdependent variables, and rely on a *dataflow graph* to propagate changes to dependent elements. Research into *functional* reactive programming [12] (FRP) builds upon this by using concepts from functional programming to model time-varying data.

FRP systems are based around the notion of a *Signal* or *Behaviour*—a value which varies with time—and an *Event*, which can be thought of as either a *discrete* occurrence such as a mouse click, or a *predicate* event which occurs exactly when a signal satisfies a set of predicates.

Functional Reactive Programming provides a very expressive and clear semantics, with powerful higher-order combinators to work with continuously-varying values. Although the theory is clear, implementing such systems poses several challenges such as space leaks, in particular with regard to higher-order stream operations: the canonical example of this is that by allowing a signal of signals, it therefore becomes necessary to record the *entire history* of the signal after its creation in order to allow future signals to depend on previous values. This naturally results in a memory leak, as memory usage must grow linearly with execution time.

To overcome this issue, different approaches have been taken by different systems. In particular, *Arrowised FRP* [16] disallows signals from being treated as first-class altogether, relying instead on a set of primitive signals and a set of stream combinators to manipulate these, while the *Elm* language [10] prohibits the creation of higher-order signals directly in the type system. Real-time FRP [32] and event-driven FRP [33] systems further restrict operations on streams to those which can be implemented in a manner that will yield acceptable real-time behaviour.

The use of higher-order signals is, however, natural when creating graphical applications. By ruling out higher-order signals, the underlying dataflow graph remains *static*, meaning that it cannot change during the course of the application's execution. In practice, this means that it is difficult to create applications with multiple sub-pages, such as in the single-page application (SPA) paradigm.

Our alternative solution consists of a dataflow layer which interacts with a DOM frontend, using F# and the WebSharper [1] functional web framework. We introduce reactive variables, or `Vars`, and `Views`, read-only projection of `Vars` within the dataflow graph. A key difference between this and the traditional FRP paradigm is that the system we propose only makes use of the *latest* available value in the system, but as a result supports *monadic* combinators to support *dynamic* composition of `Views`.

We designed our framework, `UI.Next`, taking into account the following key principles:

Modularity: The dataflow graph should be defined separately to the DOM representation, meaning that it should be possible to display the same data in different ways on the view layer in a similar way to the Model-View-Controller architecture. It should

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

also be possible to perform different transformations to the same node in the dataflow graph.

Leak-Freedom: A primary design decision was to prevent space leaks. Traditional pure monadic FRP systems permit the inclusion of space leaks by allowing higher-order event stream combinators, whereas other dataflow systems often keep strong links between nodes of the dataflow graph and as a result require manual unsubscription from data sources. Our solution does not keep strong links between nodes in the dataflow graph, and as a result prevents this class of leaks. space leaks are avoided by working purely with the latest value of reactive variables.

Preservation of DOM Node Identity: DOM nodes consist of more than is described in the DOM tree. State such as whether an element is currently in focus, or the current text in an input box, is preserved upon a DOM update, and only subtrees which have been explicitly marked as time-varying will change on an update.

Composability and Ease-of-Integration: Elements in the DOM representation compose easily due to a monoidal interface, and we introduce an elegant embedding abstraction to allow time-varying DOM fragments to be integrated with the remainder of the DOM tree representation.

1.1 Contributions

- An implementation of a dynamic dataflow system which is amenable to garbage collection by not retaining strong links between nodes in the dataflow graph through the use of an approach inspired by Concurrent ML [28] (Section 2).
- A reactive DOM frontend for the WebSharper web framework, allowing DOM nodes to depend on dataflow nodes and update automatically (Section 3).
- A declarative animation API which integrates with the DOM frontend, and can be driven by the dataflow system (Section 4).
- Implementations of functional abstractions such as Flowlets [3] and Piglets [11] using `UI.Next`, showing how the framework can ease the implementation of such abstractions, and how these abstractions can be used to build larger applications (Section 5).
- Example applications making use of the framework (Section 6).

Source code for the framework can be found at `http://www.bitbucket.org/IntelliFactory/websharper.ui.next`, and a website containing samples and their associated source code can be found at `http://intellifactory.github.io/websharper.ui.next`.

2. 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 primarily of two primitives: reactive variables, `Var`s, and reactive views, `View`s.

A `Var` is a time-varying variable, and can be thought of as very similar to a standard F# `ref` cell. The difference, however, is that a `Var` may be observed by `View`s: changes to a `Var` therefore update any dependent `View`s in order to trigger change propagation through the remainder of the dataflow graph.

2.1 `Vars`

`Var`s are parameterised over a particular type. The actions that may be performed on `Var`s are straightforward: they may be created, their value may be set, and they may be updated using the current value.

One additional operation, `SetFinal`, marks the value as finalised, meaning that no more writes to that variable are permitted. This is included in order to prevent a class of memory leaks: if it is known that a value does not change after a certain point, then `SetFinal` may be used to optimise accesses to the variable.

The operations that may be performed on `Var`s are detailed in Listing 1.

Listing 1. Basic operations on Vars
```
type Var =
  static member Create : 'T -> Var<'T>
  static member Get : Var<'T> -> 'T
  static member Set : Var<'T> -> 'T -> unit
  static member SetFinal : Var<'T> -> 'T -> unit
  static member Update : Var<'T> -> ('T -> 'T) ->
      unit
```

2.2 `Views`

A `View` provides a way of observing a `Var` as it changes. More specifically, a `View` can be thought of as a node in the dataflow graph which is dependent on a data source (`Var`), or one or more other dataflow nodes (`View`).

The power of `View`s comes as a result of the implementation of applicative and monadic combinators, allowing multiple views to be combined: these operations are shown in Listing 2.

Listing 2. Operations on Views
```
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>
```

A `View` can be created from a `Var` using the `FromVar` function. Additionally, it is possible to create a `View` of a constant value using the `Const` function.

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. This function is crucial in the implementation of the reactive DOM layer described in Section 3.

The remaining abstractions are ubiquitous in the functional domain: `Map` allows a function to be applied to the new value of an observed `Var` whenever it changes, yielding another view. In terms of the dataflow graph, this results in an additional node which depends on the original view. `MapAsync` is a helper function which facilitates asynchronous calls as supported within F# and WebSharper.

The applicative combinators `Map2` and `Apply`, as first exposited by McBride and Paterson [21], allow for *static composition* of `View`s. Using these combinators, it is possible to apply functions of arbitrary arity to nodes within the dataflow graph.

Finally, the monadic combinators Join and Bind allow *dynamic composition* of graphs – that is, a dataflow graph may consist of nodes which are themselves time-varying dataflow graphs, allowing the graph to change during the course of execution. Although this dynamism is natural in GUI programming, most implementations of FRP systems do not support this for efficiency reasons as outlined in Section 7.2.

2.3 Models and Collections

When working with collections which may change with time, it is often better to work with higher-level *models* than simple Vars and Views. A Model<'I, 'M> represents a mutable model, providing a projection between a mutable type 'M (such as an F# ResizeArray) and an immutable type 'I (such as an F# list). This proves very useful when rendering a collection, for example.

A specialisation of the Model type is the ListModel, which internally represents a time-varying collection as a ResizeArray but allows the model to be viewed as a list. Additionally, several operations to modify the collection are provided: these are shown in Listing 3.

Listing 3. Operations on a ListModel

```
type ListModel<'Key,'T> with
  member Add : 'T -> unit
  member Remove : 'T -> unit

type ListModel with
  static member Create<'Key,'T when 'Key : equality>
       : ('T -> 'Key) -> seq<'T> -> ListModel<'Key,'
      T>
  static member FromSeq<'T when 'T : equality> : seq
      <'T> -> ListModel<'T,'T>
  static member View : ListModel<'Key,'T> -> View<
      seq<'T>>
```

A ListModel is created using a function which derives a key to be used for equality testing, and a sequence of elements. Addition and removal of elements can be performed with the Add and Remove functions, but more importantly it is possible to obtain a View of the collection using the ListModel.View function.

2.4 Implementation

2.4.1 Vars

The implementation of a Var is shown in Listing 4. A value of type Var<'T>, where 'T is a polymorphic type variable, consists of mutable value field, a flag to specify whether or not the Var has been set as final, and a method by which any dependent views may be notified that the variable has been updated. This is implemented as a Snap, discussed in Section 2.4.2

Listing 4. Implementation of a Var

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

2.4.2 Snaps

The implementation of the dataflow layer depends largely on the notion of a Snap: an observable snapshot of a value.

The IVar Abstraction

At its core, a Snap is based on the notion of an *immutable variable*, or IVar [28]. An IVar is created as an empty cell, which can be written to only once: multiple writes to an IVar are not permitted. 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 1.

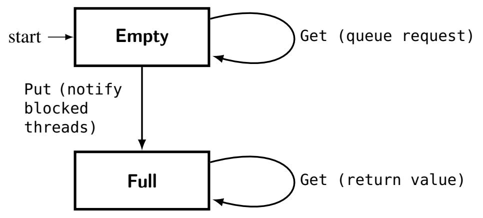

Figure 1. State Transition Diagram for an IVar

The IVar abstraction heavily inspires the method by which change propagation in the graph is handled. In this sense, there are no explicit links within the dataflow graph: that is, edges in the dataflow graph are not represented using concrete links – instead, dependent nodes can be thought of as attempting to retrieve a value from an IVar indicating obsoleteness. If the value is not obsolete, indicated in the IVar model as trying to retrieve a value from an empty cell, then the requests are queued[2]. As soon as the value in the dataflow node has been updated, meaning that it should be propagated through the graph, then all threads are notified with the latest value and continue execution.

Snap Implementation

While the IVar abstraction encapsulates the essence of a Snap, in reality the implementation is slightly more complex. A Snap can be thought of as a state machine consisting of four separate states:

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.

Forever: A snap in the Forever state indicates that it contains a value that will never change. This is an optimisation as it prevents nodes waiting for the Snap to become obsolete when this will never be the case.

Obsolete: A snap in the Obsolete state indicates that the snap contains obsolete information.

The state transition diagram for a Snap is shown in Figure 2.

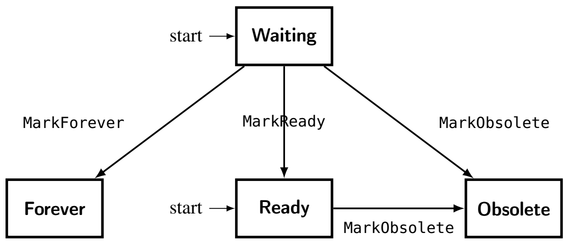

Figure 2. State Transition Diagram for a Snap

Snaps can be modified by four operations. These are:

[2] Native JavaScript is single-threaded, but we make use of the F# asynchronous workflow capabilities on the client by using a custom scheduler.

MarkForever: Updates the `Snap` with a value, transitioning to the `Forever` state to indicate that the value will never change.

MarkObsolete: Marks the `Snap` as obsolete, notifying all threads that are waiting for an updated value.

MarkReady: Marks the `Snap` as containing a new, up-to-date value, notifying all threads that are waiting for the initial value.

MarkDone: Marks the `Snap` as containing a value. If the `Snap` has been marked as constant, then transitions to the `Forever` state, otherwise transitions to the `Ready` state.

Additionally, `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.

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.

2.5 Change Propagation

As discussed in Section 2.4.1, a `Var` consists of a current value, and a `Snap` which is 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.

At its core, a `View` consists of a function `observe` to return a `Snap` of the current value.

```
type View<'T> =
    | V of (unit -> Snap<'T>)
```

The simplest `View` directly observes a single `Var`: this simply accesses the current `Snap` associated with that `Var`, updating whenever the `Snap` becomes obsolete.

Listing 5 shows the pattern of creating views which depend on other views. The `CreateLazy` function takes as its argument an observation function of type `(unit -> Snap<'A>)`, which is a function returning a `Snap` representing the latest value of the dependent dataflow nodes. This is created lazily for efficiency.

Listing 5. View Implementation
```
static member CreateLazy observe =
  let cur = ref None
  let obs () =
    match !cur with
    | Some sn when not (Snap.IsObsolete sn) -> sn
    | _ ->
      let sn = observe ()
      cur := Some sn
      sn
  V obs

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)
```

The implementations of `Snap.Map` and `Snap.Map2` are shown in Listing 6. We omit some optimisations for brevity.

Listing 6. Snap Combinator Implementation
```
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.Map2` 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` 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 `Snap`s `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 `Snap`s 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. Reactive DOM Layer

The reactive DOM layer allows data models described using the dataflow backend to be used to create reactive web applications which update automatically as a result of change propagation within the dataflow graph. In addition to providing a set of reactive input controls which depend on and modify `Var`s, the DOM layer provides combinators allowing dynamic DOM fragments to be directly composed with static fragments.

The simplest example of this is a text label which mirrors the contents of an input text box. This is shown in Listing 7.

Listing 7. A label mirroring the contents of an input 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 can place these components inside a `<div>` tag using the `Div0` function.

3.1 Monoidal Interface

A key design decision that was made in implementing the reactive DOM layer was the decision to use a *monoidal interface* for both DOM elements and DOM attributes. As the API is purely generative, meaning that it does not permit the deconstruction of nodes, we believe the use of a monoidal interface is an appropriate choice for DOM combinators as it does not differentiate between the absence of a node, a single node, or a list of nodes. Previous iterations of DOM node combinators within WebSharper did not use such an interface, and therefore often required explicit `yield` expressions within node lists.

All DOM elements in the reactive DOM layer are of type `Doc`, which represents either an empty DOM node, a single DOM node, or multiple DOM nodes. To form a monoid, `Doc` supports the operations shown in Listing 8. The same operations are supported by reactive attributes, of type `Attr`.

Listing 8. Monoidal operations on Doc
```
static member Empty : Doc
static member Append : Doc -> Doc -> Doc
static member Concat : seq<Doc> -> Doc
```

Here, `Empty` is the neutral identity element, which represents an empty DOM tree. `Append` is an associative binary operation, which combines two DOM subtrees: more precisely, the two DOM subtrees become sibling nodes, and the second subtree is rendered after the first.

In accordance with the monoid laws, appending an element to the empty `Doc`, and appending the empty `Doc` to the element does not change the element. Concatenation is implemented as a fold over a sequence of `Doc`s, using `Doc.Empty` as the initial element.

3.2 Reactive Elements and Attributes

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. As discussed in section 3.1, these sequences are concatenated.

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

Reactive attributes can be static, dynamic, or animated. Static attributes correspond to simple key-value pairs, as found in traditional DOM applications, whereas dynamic attributes are instead backed by a `View<string>`. We defer discussion of animation attributes to Section 4.

```
static member Create :
  name: string -> value: string -> Attr
static member Dynamic :
  name: string -> value: View<string> -> Attr
static member Animated :
  name: string ->
  Trans<'T> ->
  view: View<'T> ->
  value: ('T -> string) -> Attr
```

3.3 Embedding Reactive Views

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 *time-varying* 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, for example, we can create a value of type `View<Doc>` to be embedded using `EmbedView`.

By way of example, consider rendering an item in a to-do list, where the item should be rendered with a strikethrough if the task has been completed. 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.

```
type TodoItem =
    { Done : Var<bool>
      TodoText : string }
```

It would then be possible to render such an item as shown in Listing 9. Note that here, `Del0` is a notational shorthand for an HTML `<del>` element without any attributes, and `Doc.TextNode` creates a DOM text node.

We also make use of the F# construct `|>`, pronounced ‘pipe’, which signifies reverse function application.

```
let (|>) x f = f x
```

Listing 9. Embedding Reactive Views
```
View.FromVar todo.Done
|> View.Map (fun isDone ->
    if isDone
        then Del0 [ Doc.TextNode todo.TodoText ]
        else Doc.TextNode todo.TodoText)
|> Doc.EmbedView
```

We use this pattern extensively when developing applications using `UI.Next`.

3.4 Implementation

Doc

We store an in-memory representation of DOM trees, propagating these to the DOM when necessary. At the outermost layer, a `Doc` consists of information about its associated subtree, and a unit view which is used to propagate updates upwards through a tree. This is shown in Listing 10.

Listing 10. Implementation of the Doc type
```
type Doc =
    { DocNode : DocNode; Updates : View<unit> }
```

A `DocNode` is a node in the in-memory skeleton DOM representation. Defined as an algebraic data type, a `DocNode` may represent the concatenation of two `Docs` as a result of an `Append` operation, a `DOM` element, an embedding of a reactive DOM subtree, a DOM text node, or an empty node.

When creating nodes, the `Updates` view is combined with any dependent sub-views using the monadic and applicative combinators discussed in Section 2.2. For example, the `Updates` view of an `AppendDoc` is dependent on the `Updates` views of both sub-nodes, and as such is constructed using the `Doc.Map2` combinator, as shown in Listing 11. Note that the `Docs.Mk` is simply a constructor function for a `Doc` record, taking a `DocNode` and an `Updates` view as its arguments. The `||>` operator is similar to `|>`, but instead takes a tuple of arguments to pass to a function.

```
let (||>) (a, b) f = f a b
```

Listing 11. Construction of an `AppendDoc` `DocNode`

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

EmbedView

As discussed in Section 3.3, the `EmbedView` allows a time-varying DOM segment to be embedded within the DOM tree, with any updates in this segment being reflected within the DOM. The implementation of this is based on the idea of 'dirty-checking', as employed by many reactive DOM libraries such as Facebook React [1].

The `DocNode` representation of a time-varying DOM node is a `DocEmbedNode`, shown in Listing 12.

Listing 12. The `DocEmbedNode` type

```
type DocEmbedNode =
    { mutable Current : DocNode
      mutable Dirty : bool }
```

The record has two mutable fields: the `Current` field represents the current value of the embedded view, and the `Dirty` field is set to true if the `View<Doc>` has changed, indicating that the DOM subtree should be updated.

The implementation of the `EmbedView` function is shown in Listing 13.

Listing 13. Implementation of the `EmbedView` function

```
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)
```

The `EmbedView` function works by creating a new entry in the dataflow graph, depending on the time-varying 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

As previously discussed, any updates in the DOM representation are propagated *upwards* through the tree representation. In order to trigger a DOM update, we use the `Sink` imperative observer function discussed in Section 2.2, which triggers a function whenever a `View` (in this case the `Updates` view of the root node in the `Doc` tree) changes.

Synchronisation between the virtual DOM skeleton and the physical DOM representation is performed using an $\mathcal{O}(n)$ traversal of the virtual DOM tree, in a similar way to existing libraries. While at first this may seem prohibitive for a responsive web application, such an approach has been proven by libraries such as *React* to yield acceptable performance since such traversals are generally not computationally expensive, and there tend to be few physical DOM changes.

For an element node, the synchronisation algorithm recursively checks whether any child nodes have been marked as dirty. In the case of `EmbedNodes`, 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.

An important consideration when implementing the synchronisation algorithm was 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.

4. Declarative Animation

Animation is increasingly used in modern web applications, especially when visualising data, when processing user input, or advancing state within a control flow.

In the context of web applications, animations are typically implemented as an interpolation between attribute values over time. Such animations must be composable in order for different animations to run either sequentially or concurrently, must support the specification of interpolation strategies for a given type, and *easing* functions, which specify how quickly the animation progresses at different points during the animation.

Native CSS provides animation functionality which can interpolate values, apply easing functions, and apply animations both sequentially and concurrently through the use of keyframes. While this is sufficient and intuitive for simple applications, the approach founders when animations depend explicitly on dynamic data and cannot be determined statically.

The D3 library [4] provides more powerful animation functionality. In particular, the library enables animations to depend directly on data sets, for animations to be delayed, and for the specification of *transitions*—animations which are triggered when a node is added, changed, or removed from the DOM.

D3 is an extremely powerful library, and its use has led to some very impressive animated visualisations. The API, however, does not lend itself particularly well to a functional, statically typed language: in particular, animations are generally constructed using *selections*, using function chaining to add animations and transitions. This results in animations being declared in a more imperative style.

The declarative animation library in `UI.Next` allows animations (an interpolation of a value over time) and transitions to be specified separately. These are then integrated with the reactive DOM layer in one of two ways: more commonly, they can be attached directly to elements as attributes and therefore react directly to changes within the dataflow graph, but they may also be scheduled imperatively.

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. As shown in Listing 14, an `Anim<'T>` type is internally represented as a function `Compute`, mapping a normalised time (a value between 0 and 1 denoting progress through the animation) to a value, and the duration of the animation.

Listing 14. Implementation of the `Anim<'T>` type

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

An animation can be constructed using the `Anim.Simple` or `Anim.Delayed` functions: `Anim.Simple` can be seen as a delayed animation with a delay of 0. This 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.

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

To describe collections of animations, we once again make use of a monoidal interface: in this case, the semantics of monoid concatenation are that the animations play concurrently as opposed to sequentially. Collections of animations are represented by the `Anim` type and supports the monoidal `Empty`, `Append` and `Concat` operations, as well as a function `Pack` to lift an `Anim<unit>` type into a singleton animation collection.

```
static member Append : Anim -> Anim -> Anim
static member Concat : seq<Anim> -> Anim
static member Empty : Anim
static member Pack : Anim<unit> -> Anim
```

Concatenation of a list of animations involves creating a new animation with the length of the longest constituent animation, and a compute function which 'prolongs' shorter animations by not performing further interpolation after the end of the original animation.

Transitions are specified using the `Trans` type. Functions for creating and modify `Trans` types are shown in Listing 4.

```
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. Upon a DOM update, a set intersection is performed between the nodes that have enter and exit transitions and the nodes which have been added and removed respectively, and these are concatenated and played as an animation collection.

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

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

This function takes the name ofthe 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. Functional Web Abstractions

Functional programming and static type systems can ease web programming by facilitating the implementation of functional web abstractions. In particular, Formlets [8] provide a structured means, based on the notion of an applicative functor [21], of retrieving input from a user. Using Formlets, it is possible to define a statically-typed *model* of the input (for example as a record), and populate this model with input gained from form controls.

Previous work has built upon Formlets in various ways. By extending Formlets with a monadic interface in addition to an applicative one, *sequences* of Formlets called Flowlets [3] can be created, where each stage in the flow can depend on previously-submitted input. Additionally, Formlets by default do not allow any flexibility in how forms are rendered: this is addressed by a Pluggable GUI-let, or Piglet [11].

`UI.Next` greatly simplifies the implementation of Piglets and Flowlet-style combinators. In this section, we discuss the implementation of these abstractions, and how their implementation has been eased using the framework. Additionally, we discuss a method by which pages and application state may be synchronised with the current URL, to allow easier sharing of locations within single-page applications.

5.1 Flowlets

The Flowlet-style combinators we have implemented are shown in Listing 15. It is important to note that this is not a direct implementation of Flowlets: in particular, we do not build forms using static, applicative composition, instead allowing each stage of the flow to handle the retrieval and processing of user input. The primary objective of these combinators, however, is to allow applications with a linear control flow to be constructed in a simple, intuitive fashion.

Listing 15. Flowlet Combinators

```
type Flow =
  // Definition
  static member Define : (('A -> unit) -> Doc) ->
       Flow<'A>
  static member Static : Doc -> Flow<unit>
  // Mapping
  static member Map : ('A -> 'B) -> Flow<'A> -> Flow
       <'B>
  // Monadic Combinators
  static member Bind : Flow<'A> -> ('A -> Flow<'B>)
       -> Flow<'B>
  static member Return : 'A -> Flow<'A>
  // Rendering function
  static member Embed : Flow<'A> -> Doc
  // Helper function
  static member Do : FlowBuilder
```

Pages in the flow are defined using the `Define` function. To define a page, we require function takes a callback of type `('A -> unit)`, used to pass the resulting value to subsequent stages of the flow, and renders the page as a `Doc`. It is also possible to define a static page which does not progress the flow by using the `Static` function.

Internally, a `Flow<'T>` is represented as a singleton record containing one member, a function `Render` which takes as its arguments a `Var` to be used for rendering the flow and a continuation function `('T -> unit)`, resulting in a unit value.

Listing 16. Implementation of the `Flow` type

```
type Flow<'T> =
  { Render : Var<Doc> -> ('T -> unit) -> unit }
```

To combine multiple stages of a flow, the `Bind` function is used. This is shown in Listing 17.

Listing 17. Implementation of the `Flow.Bind` function

```
static member Bind m k  =
    { Render = fun var cont -> m.Render var (fun r
         -> (k r).Render var cont) }
```

The `Bind` function is implemented by creating a new `Flow` record from a flow `m` of type `Flow<'A>` and a continuation function `('A -> Flow<'B>)`. The newly-created flow renders `m` to the `Var` `var`, with the value `r` returned by that particular stage of the flow applied to the continuation function `k` to construct the next stage in the flow.

The eliminator function for `Flow` types, `Embed`, is defined in Listing 18.

Listing 18. Implementation of `Flow.Embed`

```
static member Embed fl =
    let v = Var.Create Doc.Empty
    fl.Render v ignore
    Doc.EmbedView (View.FromVar v)
```

We begin by creating a `Var` `v` used to contain the current page rendering, initially consisting of the empty document. The flow is 'executed' by invoking the `Render` function with the variable, which is updated by the bind operation, and finally by embedding the resulting `View`.

Through the use of *computation expressions* [24], it is possible to specify flows in a manner analogous to do-notation in Haskell. Consider the following example flow:

1. A user is asked for a name and address, which is used to create a `Person` record.
2. The user is then asked to specify whether they wish to specify a phone number or e-mail address.
3. Based on the previous answer, the user is asked for either a phone number or an e-mail address.
4. The user is shown the data that they entered.

Such a flow would be described by the computation expression shown in Listing 19.

Listing 19. A flow described as a computation expression

```
let ExampleFlow () =
    Flow.Do {
        let! person = personFlowlet
        let! ct = contactTypeFlowlet
        let! contactDetails = contactFlowlet ct
        return! Flow.Static (finalPage person
             contactDetails)
    }
    |> Flow.Embed
```

This format provides a simple and expressive way of describing applications with a linear control flow.

5.2 Piglets

Formlets [8] allow user input to be retrieved in a type-safe fashion through the use of applicative functors to aid static composition. In spite of their advantages including type-safety, composability, and formal definition, formlets suffer from a lack of *modularity*: that is, the rendering of a formlet is tightly coupled to its data model. In order to change the ordering of components within the formlet, for example, it is necessary to modify the underlying data model.

Piglets [11] alleviate these concerns by separating the model and the view. Piglets consist of two separate components: a *stream* composed of values of components within the Piglets, and a *view builder function* which is provided with the values in the stream, and returns a rendering of the form. This is shown in Listing 20.

Listing 20. Structure of a Piglet

```
type Piglet<'a, 'v> =
    { stream: Stream<'a>; viewBuilder: 'v }
```

In order to create a Piglet, the `Yield` function is used. The argument to this can be a function, in which case static composition can be achieved through the use of the applicative-style composition operator ⊗. Both operations are shown in Listing 21.

Listing 21. Piglet Construction and Composition Functions

```
val Yield :
  'a -> Piglet<'a, (Stream<'a> -> 'b) -> 'b>

val ⊗:
  Piglet<'a -> 'b, 'v1 -> 'v2> ->
  Piglet<'a, 'v2 -> 'v3> ->
  Piglet<'b, 'v1 -> 'v3> ->
```

We are currently working towards an implementation of Piglets using the `UI.Next` framework. In particular, `UI.Next` replaces the `Stream` with primitives from the dataflow layer, as shown in Listing 22.

Listing 22. Piglets using UI.Next primitives

```
type Piglet<'a, 'v> =
    { read : View<Result<'a>> ; render : 'v }

val Yield : 'a -> Piglet<'a, (Var<'a> -> 'v) -> 'v>
```

In particular, a Piglet implemented using `UI.Next` consists of a `View` containing the current state of the form, and a rendering function. Creating a Piglet using `Yield` creates a `Var` which is used within the rendering function, and the implementation of ⊗ uses `View.Map2` to create a dependent `View`.

The original `Stream` implementation relied on manual subscription and pushing of values, whereas this is all handled by dataflow combinators in the `UI.Next` implementation. Replacing much of this imperative-style logic with functional combinators results in more concise, understandable, and readable code. Use of `UI.Next` primitives also avoids the need to specify explicit disposal functions, as was the case with Streams.

5.3 Sites with Multiple Pages

This work focuses on facilitating the creation of *single-page applications*: applications which run in a single page in the browser. Such applications often consist of multiple sub-pages, using JavaScript to transition between them.

Using our approach, implementing such functionality is simple and idiomatic. We begin by declaring a data type which describes the different pages in the site, and rendering functions (producing a `Doc`) for each:

```
type IFLPage = | Home | CallForPapers | Registration
      | Submission |

let renderHome v =
    Div0 [
        H10 [txt "Home"]
        ...
    ]

let renderCFP v = ...
```

We then create a Var<IFLPage>, representing the current page. Using this, we may then create a View, and map the appropriate rendering function, resulting in a view of the rendering of the current page. This may then be embedded into a page using EmbedView; navigation between pages is possible by changing the previously-created Var.

```
let v = Var.Create Home
View.FromVar v
|> View.Map (fun pg ->
    match pg with
    | Home -> renderHome v
    | CallForPapers -> renderCFP v
    | Registration -> renderRegistration v
    | Submission-> renderSubmission v)
|> Doc.EmbedView
```

6. Examples

In this section, we present two examples using the framework: the first of which showcases reactive animation and the treatment of object identity, and the second of which describes a form rendered using the Piglets implementation backed by UI.Next.

6.1 Object Constancy

Object Constancy is a technique for allowing an object representing a particular datum to be tracked through an animation. In particular, consider the case where the underlying data does not change, but the user controls filtering criteria: changes in such criteria may add new data to the visualisation, remove currently-displayed data, and assuming sorting criteria remains constant, change the ordering of the data.

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. Such an example is discussed by Bostock [5], using the D3 [4] library.

The example described by Bostock [5] displays the percentage of the population in a particular age bracket for a number of different states, where 10 states are displayed. The percentages for each state are displayed in descending order. To recreate this example using our declarative animation framework, we begin by defining a data model using F# records.

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

Here, AgeBracket and State are representations of age brackets and states respectively, and DataSet is a representation of the entire data set as read in from an external data source. The StateView record specifies details about how a state should be displayed based on other visible items: MaxValue specifies the maximum percentage, Position specifies the rank of the item in the visible set, State specifies the name of the state, Total specifies the total number of items within the set and Value specifies the percentage value of the item.

```
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. In particular, upon entry, the element will transition from the origin position to the desired position; and will transition back to the origin position on exit.

We now specify a rendering function taking a View<StateView> and returning a Doc to be embedded within the tree. We elide some of the function in the interest of brevity, but you can find the complete source of the example online[3].

```
let Render (state: View<StateView>) =
  let anim name kind (proj: StateView -> double) =
      Attr.Animated name kind (View.Map proj state)
           string
  // Projection functions
  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
    ] []
  ]
```

The helper function anim takes the name of the attribute to animate, the transition to use, and a projection from the state view to the value to use within the transition. We then specify three projection functions: one for the width of the bar, based on the value as a proportion of the maximum value in the set; one for the Y-position of the bar, and and one for the height of the bar. These may then be specified as attributes of the object to animate.

Finally, we create a selection box to allow the user to modify the age bracket, which in turn modifies the current list of StateViews. To implement object constancy, a key which uniquely identifies the data is required [14]. In the case of StateView, this is State: we use this when embedding the current set of visible elements using the ConvertSeqBy function, which is a memoising conversion function useful for preserving node identity. It is then possible to embed this into the DOM using EmbedView.

```
S.Svg ["width" ==> string Width; "height" ==> string
     Height] [
    shownData
```

[3] https://github.com/intellifactory/websharper.ui.next/blob/master/src/ObjectConstancy.fs

```
        |> View.ConvertSeqBy (fun s -> s.State) Render
        |> View.Map Doc.Concat
        |> Doc.EmbedView
    ]
```

6.2 Reactive Piglets

In this simple example, we define a form which consists of three fields: a first name, a last name, and a type of pet. We begin by defining a data model.

```
type Pet = | Cat | Dog | Piglet
type Person = { firstName: string; lastName: string;
     pet : Pet }
let Pets = [ Cat ; Dog ; Piglet ]
let showPet = function
  | Cat -> "Cat" | Dog -> "Dog" | Piglet -> "Piglet"
```

The next step is to use the Return and Yield operation to construct a Piglet. We also use the Validation.Is function to add validation to the form: should either the first or last name be empty, the failure will be propagated and displayed upon submission. The WithSubmit function adds a Submitter type, which can be used to snapshot the state of the form stream when a submission button is pressed. This can then be used as part of an AJAX call to a server, or to display errors.

```
let Person init =
    Piglet.Return (fun f l p -> { firstName = f;
         lastName = l ; pet = p})
    <*> (Piglet.Yield init.firstName
        |> Validation.Is Validation.NotEmpty "Please
           enter your first name.")
    <*> (Piglet.Yield init.lastName
        |> Validation.Is Validation.NotEmpty "Please
           enter your last name.")
    <*> (Piglet.Yield init.pet)
    |> Piglet.WithSubmit
```

Finally, we define a view for the Piglet. The Render function is provided with Vars for the fields in the Piglet, which are in turn used with the built-in form components in UI.Next.

```
let radioButtons (v: Var<Pet>) = ...
let Person init =
 ViewModel.Person init
 |> Piglet.Render (fun first last pet submit ->
     Doc.Concat [
      Div0 [Doc.TextNode "First Name: " ; Doc.Input
          [] first]
      Div0 [Doc.TextNode "Last Name: " ; Doc.Input
          [] last]
      radioButtons pet
      Div0 [Doc.Button "Submit" [] submit.Trigger]
      Div0 [Doc.TextView (submit.Output |>
       View.Map (function
       | Success u ->
          "Person: " + u.firstName + " " +
          u.lastName + ", Pet: " + (showPet u.pet)
       | Failure errs ->
          List.fold (fun out (str: ErrorMessage) ->
          out + " " + str.Message) "" errs)) ] ])
```

7. Related Work

7.1 Dataflow Systems

Synchronous dataflow languages originated as a means of specifying, designing, and implementing real-time systems such as those used within hardware. Languages such as ESTEREL [2], LUSTRE [13], and Lucid Synchrone [27] can compile programs to transition systems, in such a way that they may be formally verified using techniques such model checking. Such approaches are generally limited to the hardware domain, as the languages do not support features required in more general programs such as recursion or dynamic memory allocation.

REACTIVEML [20] is an extension of OCaml embedding the synchronous dataflow paradigm, providing primitives such as signal and await to express dataflow within the language. Cooper and Krishnamurthi [9] extend the Scheme programming language with dynamic dataflow to create a system, FrTime, which works by modifying the Scheme evaluator.

Scala.React [19] embeds the dataflow paradigm into Scala, introducing an imperative dataflow language, time-varying values, and event streams. The implementation is driven by a scheduler which proceeds in discrete steps, known as *propagation turns*, and the graph is constructed using weak pointers. A similar technique is used within OCaml React [6].

7.2 Functional Reactive Programming

Functional Reactive Programming [12, 15, 31] has served a large inspiration for the dataflow-based model for reactive user interfaces that we have described. FRP systems are based around primitive abstractions modelling time-varying data, referred to as Behaviours or Signals, and Events which occur either as a response to events such as user input, or when a signal satisfies a set of predicates.

The semantics of FRP are extremely attractive and clear: time-varying values are simple to transform and reason about, and event streams provide a method by which interaction can modify these. Despite their mathematical simplicity, the implementation of FRP semantics is notoriously difficult. In particular, early implementations of FRP such as Fran [12] remained very true to FRP semantics, at the cost of introducing memory leaks: in order to fully implement the FRP semantics (which allowed signals to depend on any past, present or future value), it was necessary to store every signal value, regardless of whether or not it would be used. This in turn led to memory usage growing linearly with time.

Subsequent approaches favour less expressive forms of FRP to provide better runtime guarantees. Real-time FRP [32] only allows signals to be used in ways which can be implemented efficiently, and can be reasoned about when developing real-time systems. Event-driven FRP [33] takes this further by only propagating changes as a result of a discrete *event*.

Arrowised FRP [16, 23] disallows signals to be treated as first-class values instead providing only *transformers* or *combinators* on primitive signals. Manipulating signals in this way is eased through the use of the Arrow abstraction [17]. Such an approach, although less expressive than purely-monadic FRP approaches such as Fran, are far more efficient and practical. The issue with arrowised FRP as it pertains to GUI programming is that it cannot adequately express *dynamic* dataflow graphs, as it becomes impossible to specify monadic combinators on time-varying values. Since our implementation of signals (Views), works purely on the latest available value, it *is* possible to specify monadic combinators, meaning that dynamic composition is possible.

The *Elm* programming language [10] is a language providing first-class FRP primitives with the goal of easing the creation of responsive GUIs. Elm implements static signal composition operators such as lift which work on the latest value in the signal, equivalent to View.Map in our dataflow layer, and $\texttt{lift}_n$, which is equivalent to View.Map2 and View.Apply. In addition, Elm provides a construct foldp to perform transformations based on previous signal values. The asynchronous capabilities of Elm are mirrored in UI.Next through the use of the MapAsync function, which is supplemented by the RPC functionality supported by

WebSharper. In order to prevent space leaks, the creation of higher-order signals is prohibited by the type system. Such an approach is a good solution to the problem, but is not feasible when working within an existing ML type system. Instead, we forego the ability to perform time-dependent transformations as a primitive operation within the reactive layer, instead postulating that such functionality may be attained either using simple single-layer callbacks, or an approach based on concurrent processes such as Concurrent ML [28].

More theoretical recent work [18**?**] focuses on languages implementing FRP semantics, including higher-order signals, while guaranteeing leak-freedom. In particular, the approach described by **?**] divides expressions into those which may be evaluated immediately, and those which depend on future values and whose evaluation must be delayed. In order to prevent space leaks, obsolete behaviour values are aggressively deleted. The approach relies on a specialised type system and an explicit notion of time being exposed to the programmer, which limits its applicability to our problem domain.

7.3 Reactive DOM Libraries

Facebook React [1] is a library which, in a similar way to our approach, allows developers to construct DOM nodes programmatically. This process is facilitated through `JSX`, an HTML-like markup language with facilities for property-based data binding. The key concept behind React is the use of an automated 'diff' algorithm, driven by a global notion of time instead of a dataflow system: as a result, DOM updates are batched for efficiency. We decided to use a dataflow-backed system instead of purely a diff algorithm to avoid losing control over DOM node identity. Our approach uses some aspects of React, such as dirty-checking, but this is localised to DOM fragments which have been specifically embedded.

Flapjax [22] is a dataflow-backed programming language providing full FRP functionality which can also be used as a JavaScript library. As the library does not prohibit higher-order signals, it is possible to introduce space leaks as previously discussed.

7.4 Functional Web Programming

Functional programming has been found to be very applicable to the web programming domain. In particular, functional programming and the static type systems associated with many functional programming languages allow for the development of many powerful web abstractions to ease the structuring and development of web applications.

WebSharper takes inspiration from Links [7], a language which allows client, server, and database code to be written in a single source language, thus mitigating the impedance mismatch problem. F# functions are compiled to JavaScript aided by quotations [30], and AJAX calls are easily represented using F# asynchronous workflows.

Formlets [8] are an abstraction for retrieving typed user input from HTML forms, which have been extended by Denuzière et al. [11] to enable the specification of customised rendering functions. Flowlets [3] augment Formlets with a monadic interface to enable the construction of multiple dependent formlets.

The *i*nteractive *Data*, or `iData` abstraction [25] is an edit-driven approach to type-safe forms: edits to input fields trigger computations, with previous state being restored in the case of invalid data being entered. This is taken further by the `iTasks` workflow management system [26] which makes use of multiple high-level combinators such as recursion, sequence, and choice.

8. Conclusion

In this paper, we have presented a framework in F#, `UI.Next`, facilitating the creation of reactive DOM applications backed by a dynamic dataflow graph. Guided by previous work on functional reactive programming and dataflow systems, our framework consists of a dataflow layer consisting of `Vars`, representing time-varying variables, and `Views`, read-only representations of `Vars` in a dataflow graph. Our dataflow representation is modular as it is decoupled from the DOM layer, and amenable to garbage collection by not allowing higher-order event streams or keeping strong links between dataflow nodes. While inspired by functional reactive programming, we make several simplifications which facilitate the implementation of higher-order monadic operations on `Views` to allow dynamic dataflow graph composition, in turn supporting common GUI programming patterns.

The DOM layer uses a monoidal interface to aid composability, and through the use of the `EmbedView` function, allows time-varying DOM elements to be directly embedded into a larger tree. Updates to the in-browser DOM are performed only when necessary and build on the well-founded notion of dirty-checking, minimising needless node generation both as an efficiency measure and to preserve node identity. Such an approach within a strongly, statically-typed language has proven extremely useful in aiding the implementation of several functional web abstractions, such as Piglets and Flowlets.

Additionally, we have presented an interface for declarative animation based on the dataflow graph, which integrates directly into the reactive DOM layer as reactive attributes, and can be backed directly by reactive attributes. This enables the creation of rich, data-backed animations using a statically-typed, declarative interface.

8.1 Future Work

The current implementation of `UI.Next` is freely available for experimentation. Future work will be centred around effectively integrating event streams within the dataflow layer to aid handling of user interactions. We envisage that usage of the concurrent programming paradigm as in Concurrent ML [28] or Hopac [4] will prove to be a promising future direction for this purpose.

We are currently investigating how to further integrate the reactive layer with plain HTML through the use of an F# type provider [29]. A more ambitious goal involves implementing the dataflow layer in a distributed setting with updates to a `Var` on a server being propagated automatically to clients. We are additionally currently working on a frontend implementation for the Windows Presentation Foundation, to allow the dynamic dataflow backend to be used within traditional desktop applications.

Other planned work includes further efficiency benchmarking and optimisation: while we currently implement some optimisations to minimise physical DOM accesses, further optimisation is possible.

Strongly, statically-typed languages have been shown to aid the development of web applications by better allowing applications to be structured, and decreasing debugging time by detecting errors earlier in the development process. We hope that continued research into functional web programming will allow web developers to fully take advantage of these advances.

References

[1] React | A JavaScript Library for Building User Interfaces. `http://facebook.github.io/react/`, 2014.

[2] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992. ISSN 01676423. .

[3] Joel Bjornson, Anton Tayanovskyy, and Adam Granicz. Composing Reactive GUIs in F# using WebSharper. In *Implementation and Application of Functional Languages*, pages 203–216. Springer, 2011.

[4] `https://github.com/VesaKarvonen/Hopac`

[4] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-Driven Documents. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):2301–2309, 2011.

[5] Mike Bostock. Object Constancy. `http://bost.ocks.org/mike/constancy/`, 2012.

[6] Daniel Bünzli. React / Erratique. `http://erratique.ch/software/react`, 2010.

[7] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In FrankS de Boer, MarcelloM Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer Berlin Heidelberg, 2007. .

[8] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The Essence of Form Abstraction. In *Programming Languages and Systems*, pages 205–220. Springer, 2008.

[9] Gregory H. Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In Peter Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer Berlin Heidelberg, 2006. .

[10] Evan Czaplicki and Stephen Chong. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA, 2013. ACM. .

[11] Loïc Denuzière, Ernesto Rodriguez, and Adam Granicz. Piglets to the Rescue. In Rinus Plasmeijer, editor, *Proceedings of the 25th International Symposium on Implementation and Application of Functional Languages (IFL '13)*, 2013.

[12] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. ISSN 0018-9219. .

[14] Jeffrey Heer and Michael Bostock. Declarative Language Design for Interactive Visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 16(6):1149–1156, 2010.

[15] Paul Hudak. Functional Reactive Programming. In Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, chapter 1, page 1. Springer Berlin Heidelberg, Berlin, Heidelberg, March 1999. ISBN 978-3-540-65699-9. .

[16] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003.

[17] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, May 2000. ISSN 01676423. .

[18] Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. Higher-order Functional Reactive Programming in Bounded Space. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 45–58, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. .

[19] Ingo Maier, Tiark Rompf, and Martin Odersky. Deprecating the Observer Pattern. Technical Report EPFL-REPORT-148043, 2010.

[20] Louis Mandel and Marc Pouzet. ReactiveML: A Reactive Extension to ML. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 82–93, New York, NY, USA, 2005. ACM. ISBN 1-59593-090-6. .

[21] Conor McBride and Ross Paterson. Applicative Programming with Effects. *Journal of Functional Programming*, 18(01):1–13, May 2007. .

[22] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM. .

[23] Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. .

[24] Tomas Petricek and Don Syme. The f# Computation Expression Zoo. In *Practical Aspects of Declarative Languages*, pages 33–48. Springer, 2014.

[25] Rinus Plasmeijer and Peter Achten. iData for the World Wide Web âĂŞ Programming Interconnected Web Forms. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 242–258. Springer Berlin Heidelberg, 2006. .

[26] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 141–152, New York, NY, USA, 2007. ACM. .

[27] Marc Pouzet. Lucid Synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI*, 2006.

[28] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007.

[29] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, and Others. Strongly-typed language support for internet-scale information sources. Technical report, Technical Report MSR-TR-2012-101, Microsoft Research, 2012.

[30] Don Syme, Adam Granicz, and Antonio Cisternino. Language-Oriented Programming: Advanced Techniques. In *Expert F# 3.0*, pages 477–501. Springer, 2012.

[31] Zhanyong Wan and Paul Hudak. Functional Reactive Programming from First Principles. *SIGPLAN Not.*, 35(5):242–252, May 2000. ISSN 0362-1340. .

[32] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, volume 36 of *ICFP '01*, pages 146–156, New York, NY, USA, October 2001. ACM. ISBN 1-58113-415-0. .

[33] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-Driven FRP. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 155–172. Springer Berlin Heidelberg, 2002. .