# Reactive Abstractions for Functional Web Applications

Loïc Denuzière
IntelliFactory
loic.denuziere@intellifactory.com

Adam Granicz
IntelliFactory
granicz.adam@intellifactory.com

Simon Fowler
University of Edinburgh
simon.fowler@ed.ac.uk

## Abstract

Web frameworks and functional programming are a natural fit. Numerous web frameworks leverage the concise, declarative nature of functional programming languages to allow client and server code to be written in a more direct, idiomatic manner.

Of particular interest are abstractions for web programming. Formlets [5] are a compositional abstraction based on the notion of applicative functors [15] for the creation of statically-typed web forms. More recent abstractions [7] based on Formlets allowing dynamic composition and customisable rendering functions rely on reactive programming concepts. However, the underlying implementations for the reactive segments of these abstractions have been somewhat ad-hoc: we firstly consolidate the work on the reactive Formlets and Piglets abstractions using the UI.Next [9] reactive framework, simplifying and clarifying their implementations.

Secondly, we describe how reactive data models and lensed reactive variables may be used to allow reactive web abstractions such as Flowlets and Piglets to interact with external data sources.

## 1. Introduction

Web applications are ubiquitous. Traditionally, web applications are written in multiple different languages: HTML and JavaScript on the client side; languages such as Ruby or Python on the back-end; and SQL for database access. Additionally, much web development on both the client and server side is undertaken in languages with *dynamically-checked* type systems, enabling rapid development but losing type information between the client and server, and making type-directed development more difficult.

Inspired in part by the Links [4] functional web language, WebSharper[1] is a web framework allowing client, server, and database code to be written in the F# [24] functional programming language. F# is a strongly, statically typed language from the ML family, with built-in interoperability with the .NET framework. WebSharper leverages language-level reflection in the form of quoted expressions to compile F# code to JavaScript for use on the client side, which can interact with F# code running on the server.

A key contribution of the Links team was the *formlet* [5]: a compositional abstraction for constructing web forms based on the idea of an applicative functor [15]. As an example, consider a small form where we wish to collect the name and species of a pet, shown in Listing 1. We use the F# reverse function application notation `x |> f = f x`.

**Listing 1.** Pet Formlet

```
type Pet = { Name : string; Species : PetSpecies; }
let SpeciesOptions = [("Dog", Dog) ; ("Cat", Cat) ; ("Piglet",
      Piglet)]

let PetFormlet =
  Formlet.Yield (fun name species -> { Name = name; Species =
      species})
  ⊗ (Controls.Input "" |> Enhance.WithTextLabel "Name")
  ⊗ (Controls.RadioButtonGroup (Some 0) SpeciesOptions
      |> Enhance.WithTextLabel "Species")
```

`Pet` is an F# record containing fields for a pet's name and species. `PetFormlet` is a formlet of type `Formlet<Pet>`, where `Controls.Input` and `Controls.RadioButtonGroup` are formlets representing HTML input boxes and radio button groups respectively. The `Enhance.WithTextLabel` function adds a text label to the form control.

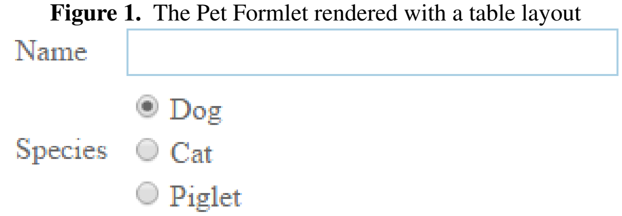
The `Yield` function 'lifts' a value into a formlet with an empty body, with type `Yield : 'a -> Formlet<'a>`. The ⊗ operator is the applicative 'apply' function, with type ⊗: `Formlet<'a -> 'b> -> Formlet<'a> -> Formlet<'b>`: given a function lifted into an applicative environment—in this case, that of a formlet—and a value lifted into the same environment, the ⊗ operator applies the argument to the function, returning the result lifted into the same environment. As a result, the pattern of creating a type-safe form is simple: lift a function using the `Yield` operation, and use the ⊗ operator to statically combine sub-formlets.

As a result of their definition as applicative functors, formlets are naturally compositional: smaller sub-formlets can be composed in order to make larger formlets. Formlets are then 'promoted' to forms by associating them with a handler, and embedding them into a webpage, as shown in Listing 2.

**Listing 2.** A formlet with a handler function embedded in a page

```
Div [
  PetFormlet.Run (fun s -> processResult s)
]
```

**Figure 1.** The Pet Formlet rendered with a table layout




Formlets are concise, compositional, and have well-defined semantics. On the other hand, through our use of formlets in practice, we identified two limitations: firstly applicative functors (or *idioms*) only support static composition—they are *oblivious* [14]—so later parts of a formlet may not depend on previous parts of a formlet. Secondly, the *layout* of the formlet is conflated with the underlying formlet data model: the visual structure of the form follows directly from the structure of the model, so changing the order of two form components would require a change to the underlying model.

In previous work, Bjornson et al. [2] extended formlets to handle dynamic composition by implementing the monadic bind operation `>>=`. The resulting abstraction, *flowlets*, allow dependency

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

within forms. Continuing the example of pets, consider the case where we wish to create a form for insuring a pet. We have three types of pets: dogs, cats, and piglets, each of which have different breeds. Additionally, should a dog be selected, we also wish to know whether or not the dog has attended any training sessions.

We begin by defining the data model, defining the species of the pet, the breeds for each, and a representation of the insurance information for each type of pet (Listing 3). Throughout this example, we elide the definitions for cats and piglets for brevity.

**Listing 3.** Data model for pet insurance flowlet

```
type PetSpecies = Dog | Cat | Piglet
type DogBreed = Husky | Boxer | Poodle
type CatBreed, PigletBreed = ...

type PetInsuranceInfo =
  | DogInfo of DogBreed * bool | CatInfo of CatBreed
  | PigletInfo of PigletBreed

type InsuredPet = { Name : string; Species : PetSpecies;
      InsuranceInfo : PetInsuranceInfo}
```

With the data model defined, we may now begin to define the flowlet. We begin by creating lists which we can use for selection boxes: these have types `List<string * 'T>`, where the first item in the pair is the text that is displayed in the list, and the second item in the pair is the data type to which the selection corresponds. We then define formlets for the insurance information of dogs, cats, and piglets (Listing 4).

**Listing 4.** Sub-formlets for pet insurance information

```
let SpeciesOptions = [("Dog", Dog) ; ("Cat", Cat) ; ("Piglet",
      Piglet)]
let DogBreedOptions = [("Husky", Husky) ; ("Boxer", Boxer) ; ("
      Poodle", Poodle)]
let CatBreedOptions, PigletBreedOptions = ...

let DogInsuranceFlowlet =
  Formlet.Yield (fun breed isTrained -> DogInfo (breed, isTrained))
  <*> (Controls.Select 0 DogBreedOptions |> Enhance.WithTextLabel "
       Breed")
  <*> (Controls.Checkbox false |> Enhance.WithTextLabel "Has the
       dog attended training sessions?")

let CatInsuranceFlowlet, PigletInsuranceFlowlet = ...
```

Finally, we may create the larger flowlet (Listing 5). We define a function `PetInsuranceFlowlet` to define the insurance formlet to display when given the name of a pet, and define the main flowlet, `PetFlowlet`. We make use of F# computation expression syntax [18] to make the syntax more readable. The `let!` construct can be thought of as a monadic binding notation, allowing the result of the `species` flowlet to be used as an argument to `PetInsuranceFlowlet`.

**Listing 5.** Pet Insurance Flowlet

```
let PetInsuranceFlowlet = function
  | Dog -> DogInsuranceFlowlet
  | Cat -> CatInsuranceFlowlet
  | Piglet -> PigletInsuranceFlowlet

let PetFlowlet =
  Formlet.Do {
    let! name = Controls.Input "" |> Enhance.WithTextLabel "Name"
    let! species =
      Controls.RadioButtonGroup (Some 0) SpeciesOptions
      |> Enhance.WithTextLabel "Species"
    let! insuranceInfo = PetInsuranceFlowlet species
    return {Name = name ; Species = species ; InsuranceInfo =
          insuranceInfo}
  }
```

One problem remains: the *presentation* of both formlets and flowlets is intrinsically tied into the specification of the formlet or flowlet itself. Even an update as simple as switching the order of two fields requires the argument order of the underlying function to be changed, and users have little control over how the form is rendered.

Piglets [7] address these issues by separating the data layer from the presentation layer: a *Piglet* (shown in Listing 6) consists of a *stream*, representing the successive values returned by the Piglet, and a *view builder function*, which is a rendering function provided with the streams of the Piglet components. The key idea behind Piglets lies in the definition of the ⊗ operator, which not only performs standard applicative composition on streams, but also composes the view builders into a new builder, passing arguments from the previous builders into the new function.

**Listing 6.** Piglet Definition

```
type Piglet<'a, 'v> =
  { stream: Stream<'a>; viewBuilder: 'v }
val Yield : 'a -> Piglet<'a, (Stream<'a> -> 'b) -> 'b>
val ⊗ : Piglet<'a -> 'b, 'c -> 'd> -> Piglet<'a, 'd -> 'e> ->
     Piglet<'b, 'c -> 'e>
```

As a concrete example, let us revisit our pet formlet example, shown in Listing 1. We retain the same data model as before, but may now specify a separate rendering function: in this case, we render the function using the WebSharper HTML DSL.

**Listing 7.** Pet Piglet

```
let fido = { Name = "Fido" ; Species = Dog }

let PetPiglet (init: Pet) =
    Piglet.Return (fun name species -> { Pet.Name = name; Pet.Species =
           species})
    ⊗ Piglet.Yield init.Name
    ⊗ Piglet.Yield init.Species

let RenderPetPiglet name species =
    Div [
        Controls.Input name
        Controls.RadioLabelled species [
            (Dog, string Dog)
            (Cat, string Cat)
            (Piglet, string Piglet)
        ]
    ]

let PetForm =
    PetPiglet fido |> Piglet.Render RenderPetPiglet
```

Listing 7 shows a Piglet implementing the same functionality as the Formlet in Listing 1. Note that we define the Piglet with respect to an initial value, as there must be an initial value to render. The `RenderPetPiglet` function takes the streams for the name and species, passing them to Piglet `Controls` functions, which render the input box and radio buttons respectively.

The implementation of Flowlets and Piglets have one thing in common: the use of *reactive value streams*, in which values are pushed to a stream and retrieved in a publish-subscribe fashion. In the case of flowlets, the monadic bind operation requires subscription to the values of dynamic sub-formlets. In the case of Piglets, rendering functions are provided with streams for each form element to both display the current value in the model, and update the model with new values.

### 1.1 Contributions

- We show how the dataflow layer and functional combinators provided by `UI.Next` simplify the implementation of Formlets and Piglets.

- We introduce data binding via lensed UI.Next reactive variables, and show how they can be used to associate a Formlet with a reactive model.

## 2. Reactive Web Abstractions using UI.Next

### 2.1 The UI.Next Reactive Library

UI.Next [9] is a reactive library for WebSharper, based on the idea of a *dynamic dataflow graph*. The library consists of two layers: a dataflow layer, and a presentation layer.

The dataflow layer consists of two primitives: Vars, which are observable mutable reference cells, and Views, which are projections of Vars in the dataflow graph, and can be manipulated using standard functional combinators such as Map and Bind.

The dataflow is implemented using an extension of Concurrent ML's IVar [22] to propagate changes through the graph. This means that the edges of the graph are not explicit links; instead, dependent nodes can be thought of as attempting to retrieve a value from an IVar indicating the obsoleteness of the current value. This is an important feature, as it allows nodes that are swapped out by Bind to be garbage-collected if they are not otherwise referenced.

The reactive DOM layer is a presentation layer for the dataflow layer. It consists of a monoidally-composable type Doc, which represents a possibly-reactive, possibly-empty DOM subtree.

**??** is a simple UI.Next program consisting of a text box and a label, and the label updated with the contents of the text box, but capitalised.

**Listing 8.** A simple UI.Next interactive program

```
let rvText = Var.Create ""
let textView = View.FromVar rvText
let inputField = Doc.Input [] rvText
let capitalisedText = View.Map (fun txt -> txt.ToUpper ()) textView
let label = textView capitalisedText
div [
   inputField
   label
]
```

The rvText variable is of type Var<string>: an observable mutable reference cell containing values of type string. The inputField is Doc representing an HTML input box, bidirectionally bound to the rvText variable: should rvText change, the value in the text box will change, and any changes made by the user will be reflected by rvText. The textView variable is of type View<string>, and is a projection of rvText, whereas capitalisedText is a View the toUpper function mapped over the contents of textView.

The label variable is a Doc representing a DOM text node, and finally div creates an Doc representing an HTML <div> tag containing inputField and label.

The key to linking the dataflow and reactive DOM layers is the EmbedView function, which has the type:

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

Consequently, EmbedView allows possibly-reactive DOM-segments to be embedded in the remainder of the DOM tree: users may therefore define a data model of type View<'T> (where 'T is a polymorphic type variable), map a rendering function 'T -> View<Doc>, and embed the result into the remainder of the tree.

### 2.2 Implementing Formlets using UI.Next

In this section and the next one, we detail work on using UI.Next as a reactive basis for the implementation of web abstractions. In previous work [9], we have demonstrated how reactive web applications can be implemented using UI.Next, including larger sites such as a blogging platform[2]. Here, we aim to show that UI.Next is a sufficient reactive foundation to replace the ad-hoc implementation of streams in previous implementations of Formlets and Piglets.

#### The existing Formlets implementation

The existing WebSharper implementations of Formlets, called WebSharper.Formlets, is based on a library called IntelliFactory.Reactive[3]. This library's design is strongly inspired by Reactive Extensions (Rx) [16], which is a much more imperative approach to reactive programming. IntelliFactory.Reactive provides a type HotStream<'a> which is conceptually similar to Rx's hot observables. This type provides two imperative methods:

- Subscribe : IObserver<'a> -> IDisposable subscribes to future values of the stream. IObserver is an interface whose members are callbacks that will be called by the HotStream on new value, error, and termination, respectively. IDisposable is an interface with a member Dispose which, when called, unsubscribes the observer from the stream.
- Trigger : 'a -> unit pushes a new value to the stream.

This library therefore requires explicit subscription to and unsubscription from an observed stream. This makes the implementation of dynamic Formlet combinators such as Many and Bind tedious and prone to memory leaks. It also makes it particularly ill-suited to be inserted in an otherwise UI.Next-based application: whether a Formlet is displayed or not can depend on a View, whose changes therefore need to be manually propagated to the HotStreams.

Another inconvenient of WebSharper.Formlets is that its display is managed with WebSharper.Html.Client, which is a fairly straightforward wrapper around the standard DOM API. This means that dynamic Formlets need to imperatively remove and insert DOM nodes based on the current value of a Stream. This also contributes to making the code complex and difficult to reason about.

#### The new UI.Next-based implementation

The new UI.Next-based implementation of Formlets, called UI.Next.Formlets, alleviates the issues caused by using IntelliFactory.Reactive. Views can be composed much more simply than HotStreams. There is no need to worry about the lifetime of a subscription, because it is automatically managed by the dataflow graph. The Formlet type in this implementation is shown in Listing 9.

**Listing 9.** The type Formlet<'a>

```
type Formlet<'a> =
  | Formlet of unit -> FormletData<'a>

and FormletData<'a> =
  { view : View<Result<'a>>
    layout : list<Layout> }

and Result<'a> =
  | Success of 'a
  | Failure of list<ErrorMessage>
```

Semantically, rendering the same Formlet in two different places, either separately or composed into the same larger Formlet, creates two completely independent instances. That is, they are not "entangled": their internal Vars and output Views are not the same. This is ensured by Formlet<'a> being a (wrapped) function from unit rather than directly a record containing the View.

The type Result<'a> represents the value returned by the Formlet, which is either successful or a list of error messages.

[2] http://www.fsblogger.com

[3] http://github.com/intellifactory/reactive

The type `Layout` represents the layout to be rendered, as shown in Listing 10. A `FormletData` contains a list of layouts that represents items in reverse order; this way, the most common use of the applicative functor (adding a Formlet composed of a single field at the end of a larger Formlet) is efficient. When rendering a Formlet whose layout is a list of several items, those are implicitly considered a `Vertical` layout.

**Listing 10.** The `Layout` of a Formlet

```
type Layout =
  { shape : LayoutShape
    label : option<Doc> }

and LayoutShape =
  | Item of Doc
  | Varying of View<list<Layout>>
  | Horizontal of list<Layout>
  | Vertical of list<Layout>
  | Wrap of LayoutShape * (Doc -> Doc)
```

The layout contains the actual `Docs` that represent fields and labels, but the structure is represented abstractly. This is for two reasons:

- Combinators may alter the structure of a Formlet, like the function `ToHorizontal` in Listing 11 which transforms an arbitrary Formlet into a horizontal-layout Formlet. An example of horizontal layout is visible in Figure 2.
- The caller can choose exactly how to render the layout by providing their own rendering function of type `Layout -> Doc`. For example, they can choose to render using tables (like in Figure 1 and Figure 2), or simple `divs`, or using the CSS3 flexbox functionality [1].

**Listing 11.** A layout-altering Formlet combinator

```
let ToHorizontal (Formlet fl) =
  Formlet (fun () ->
    let fldata = fl ()
    let rec toHorizontal = function
      | [] -> []
      | [l] ->
        match l.Shape with
        | Vertical ls -> [{l with shape = Horizontal ls}]
        | Varying v ->
          [{l with shape = Varying (View.Map toHorizontal v)}]
        | _ -> [l]
      | ls -> [{shape = Horizontal ls; label = None}]
    {fldata with layout = toHorizontal fl.Layout})
```

With these defined, it is quite straightforward to define the standard functorial, applicative and monadic combinators for Formlets.

- The functorial `Map` maps the result within the view and doesn't change the layout. A quasi-functorial `MapResult`, which maps over `Result<'a>` instead of `'a`, is also provided.
- `Return`, also named `Yield`, creates a Formlet with a constant view and an empty layout.
- The applicative `Apply`, also named ⊗, performs the applicative operation on the result within the views and concatenates the layouts.
- The monadic `Bind` performs a monadic bind on the views and combines the layouts using `Varying`.

Aside from these usual combinators, the primary way to create a Formlet is using `Controls`. A `Control` is displayed as an input field of some kind, and its `View` is derived from the `Var` bound to this input field. Listing 12 shows the implementation of the `Input` control, which is displayed as a simple text box. Similar implementations are provided for text areas, checkboxes, radio buttons, dropdowns, and other form components. It is important that the `Var` is created inside the unit function in order to preserve composability.

**Listing 12.** The `Input` Formlet control

```
let Input initValue =
  Formlet (fun () ->
    let var = Var.Create initValue
    { view = View.FromVar var |> View.Map Success
      layout = [Layout.Item (Doc.Input [] var)] })
```

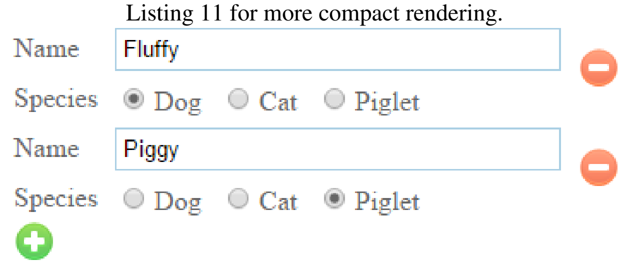
The necessity to prevent two renderings of the same `Formlet` from using the same `Var` is most evident when considering the `Many` combinator. This function takes a simple `Formlet<'a>` and returns a `Formlet<seq<'a>>`, where `seq<'a>` is F#'s abstract type for sequences, including lists and arrays. When rendering a `Many` combinator, the same `Formlet` is used for different items of the sequence, and therefore needs each of these renderings to be associated with different `Vars`.

**Listing 13.** An example use of the `Formlet.Many` combinator

```
let PetsFormlet =
  PetFormlet
  |> Formlet.Many
```

When rendering a `Formlet.Many`, buttons are automatically added to insert a new element (green button in Figure 2) and delete an element (red buttons in Figure 2).

**Figure 2.** The Pets Formlet rendered with a table layout
The radio button group has been passed to `ToHorizontal` from Listing 11 for more compact rendering.




Input validation works in a very similar way as in `WebSharper.Formlets`. It simply maps the `View` of a Formlet, transforming a `Success` into `Failure` if a predicate is false.

**Listing 14.** Formlet input validation

```
module Validation =
  let Is pred error (Formlet fl) =
    Formlet (fun () ->
      let fldata = fl()
      {fldata with view =
        fldata.view |> View.Map (function
          | Success x as r -> if pred x then r else Failure [error]
          | Failure _ as r -> r)})
```

## 2.3 Implementing Piglets using `UI.Next`

### The existing Piglets implementation

Like `WebSharper.Formlets`, the existing `WebSharper.Piglets` is based on `IntelliFactory.Reactive`'s `HotStreams`. More precisely, the type `Stream<'a>` is a thin wrapper around a `HotStream<Result<'a>>`.

Instead of `Streams`, `UI.Next.Piglets` use reactive values from `UI.Next`. They are replaced in the `Yield` function by a `Var`: this means that the rendering function will receive a `Var` instead of a `Stream`. The `Var` can be bound to form controls to update the Piglet state, and can be used to create a `View` in order to display the current value of a variable. This way, the whole `Controls` module from `WebSharper.Piglets` can be dropped from `UI.Next.Piglets`, and standard `UI.Next` functions such as `Doc.Input` and `Doc.EmbedView` can be used to input and display reactive values, respectively.

**Listing 15.** Piglet defined using `UI.Next` primitives

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

val Yield : 'a -> Piglet<'a, (Var<'a> -> 'v) -> 'v
val ⊗ : Piglet<'a -> 'b, 'v -> 'w> -> Piglet<'a, 'w -> 'x> ->
        Piglet<'b, 'v -> 'x>
```

However, similarly to `UI.Next.Formlets`, the `Stream<'a>` in the Piglet itself is replaced with a `View<'a>`: this enables views of the data to be combined by the ⊗ operator. As a concrete example, Listing 16 shows the implementation of the `Yield` and ⊗ operations.

Another aspect where the use of `UI.Next` brings more type safety is in managing failure. The possibility of failure is represented by the same type `Result<'a>` as in Formlets. In Piglets like in Formlets, the following property holds: input is always successful, and failure is only introduced by validation filters. However, since `WebSharper.Piglets` represents both inputs and internal reactive values as the same type `Stream<'a>`, this property is not enforced statically, and the implementation of input elements needs to explicitly trigger the `Stream` with a `Success` value.

In `UI.Next`-based Piglets, on the other hand, this property can be enforced. The type `Result<'a>` is now explicitly present in the `View` of a Piglet, but absent from the `Var` passed to the render function. This way, the `Yield` function can implement an always-successful Piglet and validation filters can then map this `Success` to a `Failure` as needed.

**Listing 16.** Operations on UI.Next Piglets

```
let Yield x =
  let v = Var.Create x
  { read = View.Map Success (View.FromVar v);
    render = fun f -> f v }

let ⊗ (pf: Piglet<_, _>) (px: Piglet<_, _>) =
  let v = View.Map2 Result.Apply pf.read px.read
  Piglet.Create v (pf.render >> px.render)
```

Recall that the Piglet `Yield` operation takes an initial value as its argument. The function creates a `Var` to be passed as an argument to the rendering function, and a `View` to represent the current value.

The ⊗ function takes as its arguments a function-valued Piglet of type `Piglet<'a -> 'b, 'v -> 'w>`, and applies the argument `Piglet<'a, 'w -> 'x>` within the Piglet context. In order to implement this, a new `View` is created from the `Views` of the argument Piglets. `View.Map2` and `Result.Apply` are applicative functor operations for `Views` and `Results`, respectively. `Result.Apply` concatenates lists of error messages if both `Results` are `Failures`. The rendering functions are simply composed: `(>>)` is the standard F# operator for flipped function composition.

Returning to our example, the `PetPiglet` function can stay the same: while we change the implementation of the Piglets library, the interface remains compatible. We do, however, change the render function to use the `UI.Next` reactive DOM layer:

**Listing 17.** Rendering Function for `UI.Next` Pet Piglet

```
let RenderPetPiglet name species =
  div [
    Doc.Input name
    Doc.Radio [] string [Dog, Cat, Piglet] species
  ]
```

Another feature of Piglets is the pseudo-monadic bind combinator, named `Choose` in `WebSharper.Piglets` and renamed to `Dependent` in `UI.Next.Piglets`. This combinator allows the display of a "dependent" Piglet to react to the value of a "primary" Piglet. Its type is given in Listing 18.

**Listing 18.** Dependent Piglet

```
type Dependent<'b,'u,'w> =
  member View : View<Result<'b>>
  member RenderPrimary : 'u -> Doc
  member RenderDependent : 'w -> Doc

val Dependent : primary: Piglet<'a, 'u -> 'v> ->
                dependent: ('a -> Piglet<'b, 'w -> 'x>) ->
                Piglet<'b, (Dependent<'b,'u,'w> -> 'y) -> 'y>
                when 'a : equality and 'v :> Doc and 'x :> Doc
```

The switch to `UI.Next` allows a much cleaner implementation of `Dependent`. Indeed, one feature of this operator is that it is memoized: if the primary Piglet's value has already been seen, then the dependent Piglet is not recomputed. In `WebSharper.Piglets`, this required a lot of care with regard to the lifetime of the subscriptions induced by the memoized dependent Piglet, and many explicit unsubscriptions and resubscriptions. With `UI.Next.Piglets`, this is once again managed by the dataflow graph, as `Views` which are not transitively observed by a `Sink` are implicitly disconnected from the graph until they become active again. The memoization can therefore be handled by the usual simple memoize function that just stores the result corresponding to a given argument.

## 3. Data binding in `UI.Next`

Web forms such as those created using Formlets and Piglets do not exist in a vacuum. They are generally intended to edit the data from a given data source. This data source can be the browser's local storage, or a database on the server side accessed via Ajax requests or Websockets. If it is acceptable for this data source to be updated only when the form is submitted, then it is sufficient to do so imperatively in the function passed to `Formlet.Run` or `Piglet.Run`.

However, it is increasingly common for web applications to synchronize the page in real time with the data source. The advent of WebSockets, in particular, has largely participated in the popularity of such live applications.

Formlets and Piglets, as described so far, were very inadequate for this paradigm. Their purpose was to provide the user with an interface to enter the components of a final return value, and the reactive components were used mainly to enhance this user interface. In order to accommodate live-updated applications, the reactive layer must be able to interact with external data sources.

To solve this problem, we first introduce *`Models`*, which generalize mutable `Vars` to store their data differently, and in particular *`ListModels`* which provide facilities to store collections of items. Then we present the abstract type `IRef` which encompasses `Vars` and `Models` and can also be created by applying a *lens* on an `IRef`. Finally, we show how Formlets can be enhanced to be backed by external models.

### 3.1 Models

`UI.Next` `Vars` are very simple: a reference cell storing a value of type `'a`, observable as a `View<'a>`. However, it is sometimes preferable to expose data to the dataflow graph differently from how it is stored. For example, one might want to store data as a mutable structure, while keeping the values passed to the graph immutable. This functionality is provided by `Models`.

A `Model<'i, 'm>` is conceptually a reference cell storing a value of type `'m` and a mapping function of type `'m -> 'i`, which exposes its contents to the dataflow graph as a `View<'i>`. The API is described in Listing 19. Note how the `Update` function acts imperatively on the `'m` value.

**Listing 19.** The API of the `Model` type

```
module Model =
  val Create : ('m -> 'i) -> 'm -> Model<'i, 'm>
  val Update : ('m -> unit) -> Model<'i, 'm> -> unit
  val View : Model<'i, 'm> -> View<'i>
```

Under the hood, this simple type of `Model` is implemented as a `Var` with a `Map` on its view, as shown in Listing 20.

**Listing 20.** The implementation of the `Model` type

```
type Model<'i, 'm> = M of Var<'m> * View<'i>

module Model =
  let Create proj init =
    let var = Var.Create init
    M (var, View.Map (View.FromVar var) proj)

  let Update f (M (var, _)) =
    Var.Update var (fun x -> f x; x)

  let View (M (_, view)) = view
```

The most common use case for models is to store a collection of items as a mutable, resizable array, implemented in F# as the type `ResizeArray<'a>`, and expose it to the dataflow graph as an immutable sequence (type `seq<'a>` in F#). Such a model is implemented in `UI.Next` as the type `ListModel<'k, 't>`.

The type `ListModel<'k, 't>` provides an API to insert, delete, and update individual items in the collection. Part of this API is shown in Listing 21. In order to be able to implement this functionality, items of type `'t` are identified by a key of type `'k`. This means that, for example, the method `Add` will replace an existing item with the same key, if any. The function to extract the key of an item is passed to the smart constructor `ListModel.Create`.

**Listing 21.** The API of the `ListModel` type

```
type ListModel<'k, 't when 'k : equality> =
  member View : View<seq<'t>>
  member Add : 't -> unit
  member RemoveByKey : 'k -> unit
  member UpdateBy : ('t -> option<'t>) -> 'k -> unit
  member Key : ('t -> 'k)

module ListModel =
  val Create : ('t -> 'k) -> seq<'t> -> ListModel<'k, 't>
```

A type `Key` is also provided to simplify the creation of keys when the stored datatype does not have an intrinsic unique identifier. The function `Key.Fresh()` creates a new unique key on each invocation.

Most of the time, the `View` from a `ListModel` is integrated into the dataflow graph using a function of the `View.Convert*` family shown in Listing 22. These functions map a `View<seq<'a>>` to a `View<seq<'b>>` using a function `'a -> 'b`, and use caching to avoid needing to call the function on all items of the sequence if only some of them have changed.

**Listing 22.** The `View.Convert` family of functions

```
module View =
val Convert : ('a -> 'b) -> View<seq<'a>> -> View<seq<'b>>
              when 'a : equality
val ConvertBy : ('a -> 'k) -> ('a -> 'b) ->
                View<seq<'a>> -> View<seq<'b>>
                when 'k : equality
val ConvertSeq : (View<'a> -> 'b) -> View<seq<'a>> -> View<seq<'b>>
                 when 'a : equality
val ConvertSeq : ('a -> 'k) -> ('k -> View<'a> -> 'b) ->
                 View<seq<'a>> -> View<seq<'b>>
                 when 'k : equality
```

These functions can be split in two groups:

- `Convert` and `ConvertBy` are intended for use when the value associated with a given key does not change with time. They call the mapping function for every element whose key was not in the previous sequence. This means that if the new sequence has an element whose key was already in the old sequence, then this new value is ignored. `Convert` is essentially `ConvertBy id`.
- `ConvertSeq` and `ConvertSeqBy` are intended for use when the value associated with a given key might change with time. They also call the mapping function for every element whose key was not in the previous state, but instead of passing it the corresponding value, it passes a view on the value. This means if the new sequence has an element whose key was already in the old sequence, then this new value is propagated to this view. `ConvertSeq` is essentially `ConvertSeqBy id`.

Each of these functions also has a counterpart in the `Doc` module, defined based on the following template:

```
module Doc =
  let Convert (f: 'a -> Doc) (v: View<seq<'a>>) : Doc =
    Doc.EmbedView (View.Map Doc.Concat (View.Convert f v))
```

The `ListModel` type also exposes a number of `Views` for characteristics of the sequence such as its length or the value of the item with a given key. Such `Views` could be constructed using `View.Map` on the `View<seq<'a>>`, but the provided implementation is optimized by mapping directly on the internal `Var<ResizeArray<'a>>`.

**Listing 23.** Additional `Views` provided by `ListModel`

```
type ListModel<'k, 't> =
  member LengthAsView : View<int>
  member TryFindByKeyAsView : 'k -> View<option<'t>>
  member ContainsKeyAsView : 'k -> View<bool>
```

### 3.2 The `IRef` abstraction and lensing

Models allow the storage of a reactive value in a different shape from its representation in the dataflow graph. However, user inputs such as `Doc.Input` do not only need to be able to read from an item's field using a `View`, but must be able to write to it. For this purpose, we introduce the abstract type `IRef<'a>`, described in Listing 24. The `I` prefix is a .NET convention for interface types.

**Listing 24.** The type `IRef` for abstract settable reactive values

```
type IRef<'a> =
    abstract Get : unit -> 'T
    abstract Set : 'T -> unit
    abstract Update : ('T -> 'T) -> unit
    abstract UpdateMaybe : ('T -> 'T option) -> unit
    abstract View : View<'T>
```

This type represents a reactive value that can be read from or written to. The simplest form of `IRef<'a>` is `Var<'a>`, which stores its value directly as a reference cell. But more advanced `IRefs` can also be constructed from existing `IRef` using lenses [17, 25].

Since F# does not support higher-kinded types, our implementation of lenses is the most basic version of the concept: a pair of a getter function and an updater function. This approach has already been used in F# by the Aether library [3].

```
type Lens<'a, 'b> = ('a -> 'b) * ('b -> 'a -> 'a)
```

With these defined, it is now trivial to implement lensed `IRefs`, that is, `IRefs` that, instead of storing a value directly like `Var`, store it in another `IRef` by changing its value through a lens.

```
type IRef<'a> with
  member a.Lens ((get, update) : Lens<'a, 'b>) : IRef<'b> =
    { new IRef<'b> with
      member b.Get() = get (a.Get())
      member b.Set(v) = a.Update (update v)
      member b.Update(f) =
        a.Update(fun t -> update (f (get t)) t)
      member b.UpdateMaybe(f) =
        a.UpdateMaybe(fun t ->
            Option.map (fun v -> up v t) (f (get t)))
      member b.View = View.Map get a.View }
```

Controls such as `Doc.Input` are then modified to take as argument `IRef<'a>` instead of `Var<'a>`. One can then implement a user interface in which several input fields reflect the value of different fields of the same record in a `Var`, as shown in Listing 25.

**Listing 25.** A simple use of lensed `IRefs`

```
type Person =
  { firstName: string; lastName: string; id: Key }
  static member FirstName : Lens<Person, string> =
    (fun p -> p.firstName), (fun n p -> { p with firstName = n })
  static member LastName : Lens<Person, string> =
    (fun p -> p.lastName), (fun n p -> { p with lastName = n })
let nameForm (p: IRef<Person>) =
  form [
    label [
      text "First name: "
      Doc.Input [] (p.Lens Person.FirstName)
    ]
    label [
      text "Last name: "
      Doc.Input [] (p.Lens Person.LastName)
    ]
  ]
let v = Var.Create {
  firstName = "John"; lastName = "Doe"; id = Key.Fresh() }
nameForm v
```

ListModels also provide lensed `IRefs` to modify a single element, referenced by its key.

```
type ListModel<'k, 't> =
  member Lens : 'k -> IRef<'t>

let people = ListModel.Create (fun p -> p.id) []
let peopleForm =
  people.View |> Doc.ConvertSeqBy people.Key (fun k v ->
    nameForm (people.Lens k))
```

### 3.3 Integration into Formlets

In order to use these data binding features in abstractions like Formlets, we require some additions.

For simple Formlets, new versions of `Controls` are necessary which, instead of taking an initial value as argument and creating an internal `Var` every time it is instantiated, takes as argument an `IRef` and uses it as its backing reactive variable, as shown in Listing 26. Note that, unlike the previous kind of control, multiple instantiations of such a data-bound control will be "entangled", since they are backed by the same `IRef`.

**Listing 26.** The `InputRef` Formlet control

```
let InputRef (ref: IRef<'a>) : Formlet<'a> =
  Formlet (fun () ->
    { view = ref.View |> View.Map Success
      layout = [Layout.Item (Doc.Input [] ref)] })
```

Integrating lensed `ListModels` requires a more extensively modified version of `Formlet.Many`. To understand it, we need to first look at how the original `Formlet.Many` is implemented.

Internally, `Formlet.Many` uses a model of type `ListModel<Key, Key * FormletData<'t>>`. The `Key` of this model is internal to the implementation and isn't visible to the user; it is used to minimize recomputation of both returned value and rendered layouts via `View.ConvertBy`.

In order to implement a variant of `Formlet.Many` backed by a provided `ListModel<'k, 't>`, which we'll call `ManyWithModel`, a first intuition would be to simply pass this `ListModel`'s `View` to `ConvertBy` in order to obtain a `ListModel<'k, 'k * FormletData<'t>>`, and then follow the same implementation as previously. Unfortunately, this brings rendering-related issues. The difference is that in `Formlet.Many`, the underlying `ListModel` is only updated when an item is added or removed, whereas here the `ListModel` is updated every time a lensed `IRef` is updated. Even though no key is added or removed, and therefore the function passed to `ConvertBy` is never called, the update still propagates through the dataflow graph down to the `Doc` rendering. What this translates to visually is a re-render of the full `Formlet` as soon as the user types in an input box. This is clearly not acceptable user experience.

The solution is to have an internal model of type `ListModel<'k, 'k * FormletData<'t>>` which only gets updated on insert or remove. To ensure that this is the case, `ConvertBy` is called on the base `ListModel`'s `View`, and items are inserted into or removed from the internal `ListModel` by calling `Add` or `RemoveByKey` within the mapping function, as shown in Listing 27.

**Listing 27.** The internal `ListModel` in `Formlet.ManyWithModel`

```
let ManyWithModel (m: ListModel<'k, 'a>)
                  (f: IRef<'a> -> Formlet<'b>) : Formlet<seq<'b>> =
  Formlet (fun () ->
    let mf = ListModel.Create fst (m.Value |> Seq.map (fun x ->
      let k = m.Key x
      let (Formlet fl) = f (m.Lens k)
      (k, fl ())))
    let cb =
      m.View |> View.Map (fun xs ->
        for x in xs do
          let k = m.Key x
          if not (mf.ContainsKey k) then
            mf.Add(k, (f (m.Lens k)).Data ())
        for (k, _) in mf.Value do
          if not (m.ContainsKey k) then
            mf.RemoveByKey k)
    {view = (* ... *); render = (* ... *)})
```

In order to ensure that the view `cb` is inserted in the dataflow graph, it is then mapped into a `Doc.Empty` and concatenated into the layout.

## 4. Related Work

### 4.1 Functional Web Programming

Links [4] is a functional web programming language which aims to address the impedance mismatch problem: that of having to use multiple programming languages for multiple tiers of development. Users can write client, server and database code in the Links language, which compiles the client code to HTML and JavaScript, and the server code to SQL. In WebSharper, we use the concept of writing all layers in a single language, but instead of writing a new language, we use F# by leveraging features such as language-level reflection and type providers. In contrast to Links, the server component of WebSharper is persistent as opposed to CGI-based.

The Links implementation of formlets [5] uses a preprocessing step: forms are written using HTML-like markup, and desugared into applicative style in a subsequent step. This offers some control over the layout, but the order of fields remains fixed. Links only provides applicative formlets, with data only accessible through form submission.

Yesod [23] is a web framework for the Haskell programming language. Concentrating on the server aspects of Haskell web applications, Yesod makes use of Haskell's type system and metaprogramming through Template Haskell to facilitate the creation of correct and secure web applications.

Interestingly, Yesod contains both applicative and monadic formlets. The monadic semantics are, however, different to those of flowlets: Yesod formlets are statically generated upon page loads, with data obtained through form submission. WebSharper formlets and flowlets are designed to allow the data contained within a form to be used within client code on the webpage. Consequently, the main aim of monadic Yesod formlets is to allow more flexibility in the presentation of the form. Monadic Yesod formlets separate the model and view components of form elements, allowing the model components to be combined applicatively, and the view components to be used within a rendering function. The rendering function takes the form of a Template Haskell representation of an HTML page, with the view components of form elements used as parameters to form components such as input boxes.

This mechanism is in contrast both with flowlets, as it does not allow dynamic sub-forms, and with Piglets, as the rendering function is specialised to HTML.

The iTask framework [20] is an interactive workflow system based the idea of task-oriented programming (TOP). Task-oriented programming is a high-level paradigm centred around the concept of *tasks*—"abstract descriptions of interactive persistent units of work that have a typed value" [13]. Task-oriented programming is powerful: tasks may be combined using a large number of combinators supporting recursion, monadic binding, parallel composition, and others. Although the iTask framework developed from iData [19], a way of constructing web forms, the paradigm targets a different level of abstraction, concentrating on the creation of, and interplay between tasks as opposed to the creation of reactive web forms.

### 4.2 Reactive Programming

The Reactive Extensions (Rx) [12, 16] library is designed to allow the creation of event-driven programs. The technology is heavily based on the observer pattern, which is an instance of the publish / subscribe paradigm. Rx models event occurrences, for example key presses, as observable event streams, and has a somewhat more imperative design style as a result. The dataflow layer in `UI.Next` models time-varying values, as opposed to event occurrences.

Functional Reactive Programming (FRP) [8] is a paradigm relying on values, called *Signals* or *Behaviours* which are a function of time, and *Events*, which are discrete occurrences which change the value of Behaviours.

FRP has spawned a large body of research, in particular concentrating on efficient implementations: naïvely implemented, purely-monadic FRP is prone to space leaks. One technique, arrowised FRP [10], provides a set of primitive behaviours and forbids behaviours from being treated as first-class, instead allowing the primitive behaviours to be manipulated using the arrow abstraction. Krishnaswami [11] provides an implementation of FRP without spacetime leaks by aggressively deleting obsolete behaviour values, and separating values into those which may be evaluated immediately, and those which depend on future values. Ploeg and Claessen [21] modify the original FRP interface of Elliott and Hudak [8] to ensure that functions exposed by the library do not have to retain obsolete values.

Elm [6] is a functional reactive programming language for web applications, which has attracted a large user community. Elm implements arrowised FRP, using the type system to disallow leak-prone higher-order signals.

While `UI.Next` draws inspiration from FRP, it does not attempt to implement FRP semantics. Instead, `UI.Next` consists of observable mutable values which are propagated through the dataflow graph, providing a monadic interface with imperative observers. Consequently, presentation layers such as the reactive DOM layer can be easily integrated with the dataflow layer. Such an approach simplifies the implementation of reactive web abstractions such as Flowlets and Piglets.

## 5. Conclusion

Web abstractions such as Formlets provide concise, compositional ways to structure web applications, and obtain information from users in a structured, type-safe manner.

Extensions to the original Formlet abstraction, such as Flowlets and Piglets, require reactive programming in order to support dynamic composition and custom rendering functions. In this paper, we have shown how the dataflow primitives from `UI.Next` can replace the previously imperative implementation of the reactive portions of the implementation of Formlets, Flowlets, and Piglets.

We have additionally demonstrated how reactive web abstractions can, through the use of reactive models and lensed reactive variables to implement data binding, be used to interact with external data sources.

Flowlets and Piglets are useful extensions to the original Formlet abstraction, but do not currently have a formal semantics. We are currently working on a semantics for `UI.Next`, with the goal of providing a unified semantics for reactive web abstractions. Additionally, we are currently investigating the use of F# type providers to embed typed, reactive data within web pages.

## References


[1] T. Atkins Jr, E. J. Etemad, and R. Atanassov. CSS flexible box layout module. 2013. URL `http://www.w3.org/TR/css3-flexbox/`.

[2] J. Bjornson, A. Tayanovskyy, and A. Granicz. Composing Reactive GUIs in F# using WebSharper. In *Implementation and Application of Functional Languages*, IFL '10, pages 203–216. Springer, 2011.

[3] A. Cherry. Aether — total & partial lenses in F#. 2014. URL `http://kolektiv.github.io/fsharp/aether/2014/08/10/aether/`.

[4] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4709 of *FMCO '06*, pages 266–296. Springer Berlin Heidelberg, 2007. . URL `http://dx.doi.org/10.1007/978-3-540-74792-5_12`.

[5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. The Essence of Form Abstraction. In *Programming Languages and Systems*, pages 205–220. Springer, 2008.

[6] E. Czaplicki and S. 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. . URL `http://dx.doi.org/10.1145/2491956.2462161`.

[7] L. Denuzière, E. Rodriguez, and A. Granicz. Piglets to the Rescue. In R. Plasmeijer, editor, *Proceedings of the 25th International Symposium on Implementation and Application of Functional Languages (IFL '13)*, 2013.

[8] C. Elliott and P. Hudak. *Functional Reactive Animation*, volume 32(8) of *ICFP '97*, pages 263–273. ACM, New York, NY, USA, 1997. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.7696`.

[9] S. Fowler, L. Denuzière, and A. Granicz. Reactive Single-Page Applications with Dynamic Dataflow. In E. Pontelli and T. C. Son, editors, *Practical Aspects of Declarative Languages*, volume 9131 of *Lecture*

*Notes in Computer Science*, pages 58–73. Springer International Publishing, 2015. . URL http://dx.doi.org/10.1007/978-3-319-19686-2_5.

[10] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003.

[11] N. R. Krishnaswami. Higher-order Functional Reactive Programming Without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 221–232, New York, NY, USA, 2013. ACM. . URL http://doi.acm.org/10.1145/2500365.2500588.

[12] J. Liberty and P. Betts. *Programming Reactive Extensions and LINQ*. Apress, Berkeley, CA, USA, 1st edition, 2011.

[13] B. Lijnse. *TOP to the Rescue—Task-Oriented Programming for Incident Response Applications*. PhD thesis.

[14] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science*, 229(5):97–117, Mar. 2011. ISSN 15710661. . URL http://dx.doi.org/10.1016/j.entcs.2011.02.018.

[15] C. McBride and R. Paterson. Applicative Programming with Effects. *Journal of Functional Programming*, 18(01):1–13, May 2007. . URL http://dx.doi.org/10.1017/s0956796807006326.

[16] E. Meijer. Reactive extensions (rx): Curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 11:1–11:1, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0516-7. . URL http://doi.acm.org/10.1145/1900160.1900173.

[17] R. O'Connor. Functor is to lens as applicative is to biplate: Introducing multiplate. *CoRR*, abs/1103.2841, 2011. URL http://arxiv.org/abs/1103.2841.

[18] T. Petricek and D. Syme. The F# Computation Expression Zoo. In *Practical Aspects of Declarative Languages*, pages 33–48. Springer, 2014.

[19] R. Plasmeijer and P. Achten. *iData for the World Wide Web – Programming Interconnected Web Forms*, volume 3945 of *FLOPS '06*, pages 242–258. Springer Berlin Heidelberg, 2006. . URL http://dx.doi.org/10.1007/11737414_17.

[20] R. Plasmeijer, P. Achten, and P. 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. . URL http://doi.acm.org/10.1145/1291151.1291174.

[21] A. Ploeg and K. Claessen. Practical principled FRP: Forget the past, change the future, FRPNow! In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 302–314, New York, NY, USA, 2015. ACM. . URL http://doi.acm.org/10.1145/2784731.2784752.

[22] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007.

[23] M. Snoyman. *Developing Web Applications with Haskell and Yesod*. O'Reilly Media, Inc., Sebastopol, CA., 2012.

[24] D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. APress, 2012.

[25] T. van Laarhoven. Overloading functional references. 2007. URL http://twanvl.nl/blog/haskell/overloading-functional-references.