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 are a compositional abstraction based on the notion of applicative functors for the creation of statically-typed web forms. More recent abstractions 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 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 backend; and SQL for database access. Additionally, much web development on both the client and server side is undertaken in dynamically-checked languages with development on both the client and server side is undertaken in a more direct, idiomatic manner.

Secondly, the formlet data model: the visual structure of the form follows directly of the formlet is conflated with the underlying components would require a change to the underlying model. Formlets are then ‘promoted’ to forms by associating them with a handler, and embedding them into a webpage, as shown in Figure 1.

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—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. extended formlets to handle dynamic composition by implementing the monadic bind operation >>=. The resulting abstraction, flowlets, allow dependency

Listing 1. Pet Formlet

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

let PetFormlet =

http://www.websharper.com
```

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

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

Figure 1. The Pet Formlet rendered with a table layout

Name

Species

Dog
Cat
Piglet

Pets are written in multiple different languages: HTML and JavaScript on the client side; languages such as Ruby or Python on the backend; and SQL for database access. Additionally, much web development on both the client and server side is undertaken in dynamically-checked languages with development on both the client and server side is undertaken in a more direct, idiomatic manner.

Secondly, the formlet data model: the visual structure of the form follows directly of the formlet is conflated with the underlying components would require a change to the underlying model. Formlets are then ‘promoted’ to forms by associating them with a handler, and embedding them into a webpage, as shown in Figure 1.

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—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. extended formlets to handle dynamic composition by implementing the monadic bind operation >>=. The resulting abstraction, flowlets, allow dependency
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
```

```fsharp
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
```

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

let DogInsuranceFlowlet = Formlet.Yield init.Species
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
```

```fsharp
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
```

```fsharp
type Piglet<'a, 'v> = |
    | Yield : 'a -> (Stream<'a> -> viewBuilder: 'v ) -> 'v
    | ⊗ : Piglet<'a, 'v> -> 'b -> (Stream<'b> -> viewBuilder: 'v) -> 'v

let DogBreedOptions = [('Husky', Husky); ('Boxer', Boxer); ('Poodle', Poodle)]

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
```

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


let RenderPetPiglet name species = Div {
    Controls.RadioButtonGroup (Some 0) SpeciesOptions
    Controls.Checkbox false |> Enhance.WithTextLabel "Has the dog attended training sessions?"
}

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

```fsharp
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

let capitalisedText = View.Map (fun txt -> txt.ToUpper ()) textView
let label = textView capitalisedText

let inputField = Doc.Input [] rvText

let label = textView capitalisedText

let div [] inputField
```

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 the text box.

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.

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 : IObservable<'a> -> IDisposable subscribes to future values of the stream. IObservable 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>

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

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

Semitantically, 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 data 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.

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

```fsharp
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 -> [ ]
        | Horizontal v  -> [[ with shape = Horizontal v ]]
        | Varying v     -> [[ with shape = Varying (View.Map toHorizontal v) ]]
        | Item (Doc x) -> [ x ]
        | Wrap _ -> [ ]
        | Varying _ -> [ ]
    (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 `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.

### 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))
```

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.

### 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 `HotStreamResult<'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

```fsharp
type Piglet<'a, 'v> =
    { read : View.Result<'a> ; render : () -> 'v }

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

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

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

```fsharp
let Yield x =
    (read = View.Map Success (View.FromVar x);
     render = fun _ -> x)

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 -> 'b, 'v -> 'w> 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

```fsharp
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

```fsharp
type Dependent<'b,'u,'w> =
    member View = View.Result<'b>
    member RenderPrimary = () -> Doc
    member RenderDependent = () -> 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 transitorily 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 mutableVars 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 \texttt{View<\texttt{msg}>>}. The API is described in Listing 19. Note how the \texttt{update} function acts imperatively on the \texttt{msg} value.

\begin{verbatim}
Listing 19. The API of the \texttt{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<\texttt{msg}>

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

\begin{verbatim}
Listing 20. The implementation of the \texttt{Model} type
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 \texttt{ResizeArray<\texttt{a}>}, and expose it to the dataflow graph as an immutable sequence \texttt{(type seq<\texttt{a}>)}. Such a model is implemented in \texttt{ListModel} as the type \texttt{ListModel<\texttt{k},\texttt{t} when \texttt{k} : equality>}. This type also exposes a number of \texttt{Convert} family of functions.

\begin{verbatim}
Listing 21. The API of the \texttt{ListModel} type

\begin{verbatim}
\end{verbatim}

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

\begin{verbatim}
Listing 22. The \texttt{View.\texttt{Convert}} family of functions

\begin{verbatim}
\end{verbatim}

These functions can be split in two groups:

\begin{itemize}
    \item \texttt{Convert} and \texttt{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. \texttt{Convert} is essentially \texttt{ConvertBy \texttt{id}}.
    \item \texttt{ConvertSeq} and \texttt{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. \texttt{ConvertSeq} is essentially \texttt{ConvertSeqBy \texttt{id}}.
\end{itemize}

Each of these functions also has a counterpart in the \texttt{Doc} module, defined based on the following template:

\begin{verbatim}
Listing 23. Additional \texttt{Views} provided by \texttt{ListModel}

\begin{verbatim}
\end{verbatim}

3.2 The \texttt{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 \texttt{View}, but must be able to write to it. For this purpose, we introduce the abstract type \texttt{IRef<\texttt{a}>}, described in Listing 24.

\begin{verbatim}
Listing 24. The type \texttt{IRef} for abstract settable reactive values

\begin{verbatim}
\end{verbatim}

This type represents a reactive value that can be read from or written to. The simplest form of \texttt{IRef<\texttt{a}>} is \texttt{Var<\texttt{a}>}, which stores its value directly as a reference cell. But more advanced \texttt{IRef}s can also be constructed from existing \texttt{Var} 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 [4].

\begin{verbatim}
\end{verbatim}

With these defined, it is now trivial to implement lensed \texttt{IRef}s, that is, \texttt{IRef}s that, instead of storing a value directly like \texttt{Var}, store it in another \texttt{IRef} by changing its value through a lens.
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](#).

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 [3] 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

---

[Listing 25]: A simple use of lensed `IRefs`

```
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)))
      member b.UpdateMaybe(f) =
        a.UpdateMaybe(fun t ->
          Option.map (fun v -> update v t) (f (get t)))
      member b.View = View.Map get a.View)

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>

let ManyWithModel (m: ListModel<'k, 'a>)
  (f: IRef<'a> -> Formlet<'b>) : Formlet<seq<'b>> =
  (new Formlet<seq<'b>> with
    member m.View |> View.Map (fun xs ->
      for x in xs do
        let k = m.Key x
        if not (m.ContainsKey k) then
          mf.Add(k, (f (m.Lens k)).Data ())
        for (k, v) in mf.Value do
          if not (m.ContainsKey k) then
            mf.RemoveByKey k (view = (v |> render = (* ... *)))
      let mf = ListModel.Create fst (m.Value |> Seq.map (fun x ->
        (f m.Lens x))
    let cb =
      (new Formlet<seq<'b>> with
        member b.View = View.Map (fun x ->
          Doc.Input [] (p.Lens Person.LastName))
      (new Formlet<seq<'b>> with
        member b.UpdateMaybe(f) =
          (new Formlet<seq<'b>> with
            member b.Update(f) =
              (new Formlet<seq<'b>> with
                member b.Get() = get (a.Get())
              b.Update(f))
            member b.View = View.Map get a.View)
  )

```

[Listing 26]: The `InputRef` Formlet control

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

Integrating lensed `ListModel`s 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.

---

[Listing 24]: A simple use of lensed `IRefs`

```
type Person =
  { firstName: string; lastName: string; id: Key; }

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.

```

[Listing 27]: The internal `ListModel` in `Formlet.ManyWithModel`

```
let ManyWithModel (m: ListModel<'k, 'a>)
  (f: IRef<'a> -> Formlet<'b>) : Formlet<seq<'b>> =
  (new Formlet<seq<'b>> with
    member m.View |> View.Map (fun xs ->
      for x in xs do
        let k = m.Key x
        if not (m.ContainsKey k) then
          mf.Add(k, (f (m.Lens k)).Data ())
        for (k, v) in mf.Value do
          if not (m.ContainsKey k) then
            mf.RemoveByKey k (view = (v |> render = (* ... *)))
      let mf = ListModel.Create fst (m.Value |> Seq.map (fun x ->
        (f m.Lens x))
    let cb =
      (new Formlet<seq<'b>> with
        member b.View = View.Map (fun x ->
          Doc.Input [] (p.Lens Person.LastName))
      (new Formlet<seq<'b>> with
        member b.UpdateMaybe(f) =
          (new Formlet<seq<'b>> with
            member b.Update(f) =
              (new Formlet<seq<'b>> with
                member b.Get() = get (a.Get())
              b.Update(f))
            member b.View = View.Map get a.View)
  )

```

---

[Listing 25]: A simple use of lensed `IRefs`

```

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)))
      member b.UpdateMaybe(f) =
        a.UpdateMaybe(fun t ->
          Option.map (fun v -> update v t) (f (get t)))
      member b.View = View.Map get a.View)

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.

```

[Listing 26]: The `InputRef` Formlet control

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

Integrating lensed `ListModel`s 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.

---

[Listing 24]: A simple use of lensed `IRefs`

```

```
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) [12,15] 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


