Functional, Reactive Web Abstractions

Loïc Denuzière and Adam Granicz

IntelliFactory, Budapest, Hungary
{loic.denuziere,granicz.adam}@intellifactory.com

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 web programming abstractions that target robust user interface (UI) implementation and client- and/or server-side request routing. Formlets [1] are a compositional UI abstraction based on the notion of applicative functors [2] for the creation of statically-typed, composable web forms.
More recent abstractions such as Flowlets[3] and Piglets[4] provide, in addition, dynamic composition and customizable rendering functions using reactive programming concepts.
In this paper, we consolidate our previous work on UI abstractions (Formlets, Flowlets, and Piglets) and the `UI.Next` reactive library [5] into a new set of reactive web abstractions that offer more features, most notably two-way data binding and client-side routing, and are easier to reason about, now in terms of a simple reactive model, while also simplifying their implementation.
Next to a brief survey of these abstractions and their implementation details, we show how an extension of the earlier, simple `UI.Next` data models with lensed reactive variables may be used to allow interacting with more complex and structured data sources and binding them to the reactive Formlets introduced here.

Keywords: functional reactive programming, web programming, WebSharper, F#

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 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 [6] project, and its idea of 'tierless web development,' WebSharper[1] is a web framework enabling client, server, and database

[1] http://websharper.com

code to be written in F# [7], a strongly, statically-typed language from the ML family, with built-in interoperability with the .NET framework. WebSharper leverages various metaprogramming techniques, including language-level reflection using quotations to compile F# code to JavaScript for use on the client side, which can interact with F# code running on the server.

`UI.Next` [5] 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, an operates [8] via `Var`s: reactive variables, `View`s: their current values, and a series of `Doc` combinators and helpers that enable working with and embedding into reactive markup.

One important contribution of the Links team was the idea of a *Formlet* [1]: a compositional abstraction for constructing web forms based on applicative functors [2]. In previous work, [3] extended formlets to handle dynamic composition by implementing the monadic bind operation `>>=` and introduced a wizzard-like presentation and its associated machinery, *Flowlets*, for modeling sequences of dependent formlets.

Formlets inherently suffer from presentation artifacts being tied to formlet definition: formlet constructors carry their built-in markup and a given rendering strategy (using tables, or `DIV`s, or others), making it difficult to adapt formlet-based applications to different content delivery channels.

Piglets [4] address these issues by separating form definition from the presentation layer, and by providing a subsequent step of rendering form definitions to actual markup when forms are "run". This effectively decouples form definitions from their presentation, allowing multiple presentation formats to be specified depending on how and where forms are used or reused. This makes Piglets an ideal, retargetable form abstraction.

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. It is increasingly common for web applications to want to synchronize web forms in real time with various data sources. The advent of WebSockets, in particular, has largely participated in the popularity of such live applications.

Formlets and Piglets, as described in earlier work, were very inadequate for this new 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 heterogeneous, more complex external data sources.

1.1 Contributions

In this paper, we extend Formlets and Piglets with explicit two-way data binding to make them more readily applicable to actual, real-life applications and discuss further reactive abstractions that greatly improve the robustness of web development practices. In particular:

– We introduce data binding via lensed `UI.Next` reactive variables, and show how they can be used to associate a Formlet with a composite reactive model [8].
– We show how the dataflow layer and functional combinators provided by `UI.Next` [5] simplify the implementation of Formlets and Piglets, and remove some of the technical challenges of earlier implementations.
– We introduce a new reactive abstraction for client-side routing in single-page applications (SPAs) to model transitions between separate UI sections.

2 Data binding and data models in `UI.Next`

The most common use case for `UI.Next` data 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#).

In particular, the management of item identity in such collections is an important aspect. Many applications require the capability to insert, modify or delete a specific item, and only have the DOM nodes corresponding to this item be modified, rather than the DOM nodes corresponding to the whole collection. This is for obvious performance reasons, but also for more subtle behaviour reasons. For example, if one of the affected DOM nodes was focused, then it will lose this focus when deleted and replaced.

In order to manage such collections in `UI.Next`, we implemented the type `ListModel<'k, 't>` and the `SeqCached` family of functions [8].

2.1 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 1.1. The `I` prefix is a .NET convention for interface types.

Listing 1.1. 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 [9,10].

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 [11].

```
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 1.2.

Listing 1.2. 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 =
    Doc.BindSeqCachedViewBy people.Key (fun k v ->
        nameForm (people.Lens k)) people.View
```

3 Reactive Formlets

It is trivial to model web forms using plain `UI.Next` by creating reactive variables and combining their views into markup. However, abstractions such as Formlets and Piglets provide better concision, composability, and, in the case of Piglets, separation of concerns between data combination and display.

In this section and the next one, we describe these abstractions, and detail using `UI.Next` as a reactive basis for their implementation. In previous work [5], 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.

Let us first describe Formlets. As an example, consider a small form where we wish to collect the name and species of a pet, shown in Listing 1.3. Here, we use the common F# reverse function application notation `x |> f = f x`.

Listing 1.3. 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.

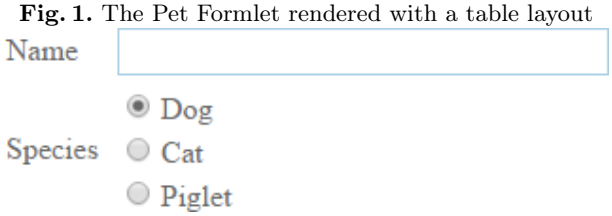
[2] `http://www.fsblogger.com`

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 1.4.

Listing 1.4. A formlet with a handler function embedded in a page

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

Fig. 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* [12]—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.

3.1 Implementing Formlets using `IntelliFactory.Reactive`

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) [13], 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.

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

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 `HotStream`s.

Another advantage of `UI.Next` over `IntelliFactory.Reactive` is that it makes it easier to reason about streams and track their composition. In `UI.Next`, it is guaranteed that the current value of a `View` `v` is the result of a computation defined when declaring `v`. For example, if `v` is defined as `let v = View.Map f u`, then it is guaranteed that at any time the current value of `v` is indeed the result of calling the function `f` on a value of the `View` `u`. Contrast with `IntelliFactory.Reactive`, where it is possible to explicitly trigger a `HotStream`. This means that with a similar definition `let v = Stream.Map f u`, it is not guaranteed that the current value of `v` is the result of calling the function `f` on a value of the `HotStream` `u`. Indeed, another piece of code might have called `v.Trigger(x)` in the meantime with some arbitrary value `x`.

Finally, another inconvenience of `WebSharper.Formlets` is that its display is managed with `WebSharper.Html.Client`, a 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.

3.2 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`. `View`s can be composed much more simply than `HotStream`s. 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 the `UI.Next`-based implementation is shown in Listing 1.5.

Listing 1.5. 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 `Var`s and output `View`s 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.

The type `Layout` represents the layout to be rendered, as shown in Listing 1.6. 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 1.6. 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 `Doc`s 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 1.7 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 `div`s, or using the CSS3 flexbox functionality [14].

Listing 1.7. 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 1.8 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 1.8. 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)] })
```

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 1.9. 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)})
```

3.3 Formlets for collections

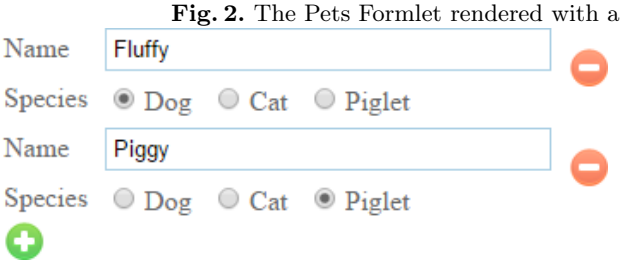
The necessity to prevent two renderings of the same `Formlet` from using the same `Var` is most evident when considering the `Many` combinator. This function takes a simple `Formlet<'a>` and returns a `Formlet<seq<'a>>`, where `seq<'a>` is F#'s abstract type for sequences, including lists and arrays. 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 1.10. 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).

Fig. 2. The Pets Formlet rendered with a table layout



3.4 Dependent Formlets

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.

This is a situation where a part of the form depends on the user input in a previous part. The list of breeds to choose from depends on what species the user has selected, and so does whether a subform for training sessions is displayed or not. This requires something more powerful than the applicative formlets we have dealt with so far: this requires *monadic* formlets.

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 1.11). Throughout this example, we elide the definitions for cats and piglets for brevity.

Listing 1.11. Data model for pet insurance formlet

```
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 formlet. 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 1.12).

Listing 1.12. 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 DogInsuranceFormlet =
  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 CatInsuranceFormlet, PigletInsuranceFormlet = ...
```

Finally, we may create the larger formlet (Listing 1.13). We define a function `PetInsuranceFormlet` to define the insurance formlet to display when given the name of a pet, and define the main formlet, `PetFormlet`. We make use of F# computation expression syntax [15] to make the syntax more readable. The `let!` construct can be thought of as a monadic binding notation, allowing the result of the `species` formlet to be used as an argument to `PetInsuranceFormlet`.

Listing 1.13. Pet Insurance Dependent Formlet

```
let PetInsuranceFormlet = function
  | Dog -> DogInsuranceFormlet
  | Cat -> CatInsuranceFormlet
  | Piglet -> PigletInsuranceFormlet

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

3.5 Data binding in Formlets

An extra advantage of implementing Formlets in `UI.Next` is that it is now possible to integrate data binding features.

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 1.14. 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 1.14. 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 1.15.

Listing 1.15. 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.

One problem with Formlets remains: the *presentation* of formlets is intrinsically tied into the specification of the formlet 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 while users have some control over how the form is rendered (via `Layouts`), it is limited.

4 Reactive Piglets

Piglets [4] address these issues by separating the data layer from the presentation layer: a *Piglet* (shown in Listing 1.16) 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 $\otimes$ 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 1.16. 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 1.3. 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 1.17. Pet Piglet

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

let PetPiglet (init: Pet) =
    Piglet.Return (fun name species ->
        { Name = name; 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 1.17 shows a Piglet implementing the same functionality as the Formlet in Listing 1.3. 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.

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

4.1 Implementing Piglets using `UI.Next`

The `UI.Next`-based implementation of Piglets is called `WebSharper.Forms`, and its equivalent of `Piglet<'a,'v>` is called `Form<'a,'v>`.

Like `UI.Next.Formlets`, instead of `Stream`s, `WebSharper.Forms` uses 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 `WebSharper.Forms`, and standard `UI.Next` functions such as `Doc.Input` and `Doc.EmbedView` can be used to input and display reactive values, respectively.

Listing 1.18. Piglet defined using `UI.Next` primitives

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

val Yield : 'a -> Form<'a, (Var<'a> -> 'v) -> 'v>
val ⊗ : Form<'a -> 'b, 'v -> 'w> -> Form<'a, 'w -> 'x> ->
        Form<'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 1.19 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 1.19. 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: Form<_, _>) (px: Form<_, _>) =
  let v = View.Map2 Result.Apply pf.read px.read
  { read = v; render = 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.

In order to be able to do data binding with external sources, there is also a variant `YieldVar` which directly takes a `Var` as argument, similarly to `UI.Next.Formlets`'s `InputVar` described in subsection 3.5.

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 `View`s of the argument Piglets. `View.Map2` and `Result.Apply` are applicative functor operations for `View`s and `Result`s, respectively. `Result.Apply` concatenates lists of error messages if both `Result`s are `Failure`s. 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 1.20. 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 `WebSharper.Forms`. This combinator allows the display of a "dependent" Piglet to react to the value of a "primary" Piglet. Its type is given in Listing 1.21.

Listing 1.21. 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 `WebSharper.Forms`, 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.

5 Reactive Routing in Single-Page Applications

Most WebSharper client-server applications are implemented using *sitelets*, a composable server-side abstraction, embedded into the type system, that defines how HTTP requests to application endpoints (a pair of a URL address and an HTTP verb) are routed to actual response content. Sitelets are parameterized over a type that contains the value space for endpoint arguments, typically encoded as a discriminated union. Internally, a `Sitelet<'T>` value is defined as a router and controller pair, each functions of `Request -> 'T option` and `'T -> Content`, respectively, where `Request` and `Content` are types defined in WebSharper for representing incoming requests and content to be served.

The following shows a simple sitelet, of type `Sitelet<EndPoint>`, marked with a special `Website` attribute for discoverability in web containers, serving two `GET` endpoints:

Listing 1.22. Simple Sitelet with Two Endpoints

```
type EndPoint =
    | [<EndPoint "GET /">] Home
    | [<EndPoint "GET /about">] About

module Site =
    [<Website>]
    let Main =
        Application.MultiPage (fun ctx endpoint ->
            match endpoint with
            | EndPoint.Home -> HomePage ctx
            | EndPoint.About -> AboutPage ctx
        )
```

Sitelets bring type safety to referencing endpoints, thus eliminating invalid URLs from within an application, and provide a range of additional shorthands for common use cases such as communicating form values, JSON, and `POST`/`GET` parameters.

Sitelets defined over the same endpoint type can be combined to build larger sitelets, while an attempt to combine sitelets serving different endpoint types will yield a compile-time error, and various sitelet combinators are available to create and combine sitelets. For instance, `Sitelet.Content` creates a singleton sitelet for a given URL (assuming `GET` for its HTTP protocol), `Sitelet.Sum` combines/-sums sitelets to form larger ones, `Sitelet.Protect` wraps its sitelet argument with authentication, and `Sitelet.Infer` creates a sitelet from an endpoint to content mapping (a variant of which is `Application.MultiPage` as shown in Listing 1.22), to name a few.

5.1 Implementing Client-Side Routing with `UI.Next`

Serving sitelets follows a typical setup: incoming requests are piped through the router function, and successful matches are transferred to the controller to yield servable content. While this works in server-side scenarios, it is often desired that some of the same capabilities are present for client-side code as well. In particular, many single-page applications routinely deal with segmenting application logic into user interfaces that are visible on demand, for instance by "switching" or "swiping" to the relevant functionality from the "home" screen of the application.

In these types of scenarios, one would like to represent the target of a section of UI in such a way that it can be invoked or switched to programmatically, freeing the programmer from having to worry about the ceremony and other concerns that accompany such switch. It is also usually expected that such actions can be reverted, usually using a Back button or a similar facility.

`UI.Next` enables a simple yet effective approach: modeling separate UI sections as a client-side endpoint type and using Uniform Resource Identifier (URI) fragment identifiers ("hashed URLs") to mark each section. Due to the complexity of passing heterogeneous types that encode arguments to the UI sections as strings in the URI, we chose to require a mapping from endpoints to lists of strings

that are then concatenated automatically to form the URI used (and provided helpers in `RouteMap` to create and plug-in such custom routing.)

The following gives an example of a simple SPA that combines two mutually exclusive UI segments, assuming that the host HTML document has a placeholder node with `ID="main"`:

Listing 1.23. SPA with Two UI Sections

```
[<JavaScript>]
module Client =
    type EndPoint = Home | Echo of string

    let routeMap =
        RouteMap.Create
        <| function
            | Home -> []
            | Echo s -> ["echo"; s]
        <| function
            | [] -> Home
            | ["echo"; s] -> Echo s
            | _ -> failwith "404"

    let Main =
        let router = RouteMap.Install routeMap
        let renderMain v =
            View.FromVar v
            |> View.Map (fun endpoint ->
                let go = Var.Set v
                match endpoint with
                | Home -> HomeSection go
                | Echo s -> EchoSection s go
            )
            |> Doc.EmbedView

        Doc.RunById "main" (renderMain router)
```

The key moment in the above code is using a `Var` to reflect the UI segment to be shown. This can then be set from any part of the SPA directly, causing the application to display the appropriate UI section in the placeholder node; effectively, modeling a visual switch to that section.

With this reactive abstraction over client-side routing, UI sections are directly addressable in the type system. Consider a simple `EchoSection`, with a link to the "home" section, which can now be expressed in a robust manner:

Listing 1.24. Type-safe Links to Client-Side UI Sections

```
let EchoPage s go =
    Doc.Concat [
        h1 [text "Echo"]
        p [text ("Got: " + s)]
        Doc.Link "Go to Home" [] (fun _ -> go EndPoint.Home)
    ]
```

6 Related Work

6.1 Functional Web Programming

Links [6] 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 [1] 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 [16] 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 [17] is an interactive workflow system based the idea of task-oriented programming (TOP). Task-oriented programming is a high-level paradigm centered around the concept of *tasks*—"abstract descriptions of interactive persistent units of work that have a typed value" [18]. 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.

6.2 Reactive Programming

The Reactive Extensions (Rx) [13,20] 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) [21] 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 [22], 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. [23] 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. [24] modify the original FRP interface of [21] to ensure that functions exposed by the library do not have to retain obsolete values.

Elm [25] 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.

7 Future Work

The reactive abstractions presented in this paper provide significant benefits for functional web developers. In parallel work [8], we presented an approach for automatic, type-safe synchronization of data models shared across multiple clients and a single server. This enables seamless push notifications about model changes to all participating clients, next to ordinary client-to-server model updates, and fully reactive UIs that accompany them. We also introduced *reactive templates* using F# *type providers* [26], where HTML template documents with special placeholders are automatically converted to `UI.Next` values, allowing typed, reactive F# data models to be seamlessly embedded within externalized HTML templates.

The lensed composite data models introduced here have been implemented for reactive Formlets. We plan to make them available to reactive Piglets as well.

Flowlets and Piglets are useful extensions to the original Formlet abstraction, but do not yet have a formal semantics. We are currently working on a semantics for `UI.Next`, with the goal of providing a unified semantics for all reactive web abstraction discussed here.

8 Conclusion

Web abstractions such as Formlets provide concise, compositional ways to structure web applications, and obtain data 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 implementations, and provide new tangible benefits.

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 more complex, composite, external data sources and be bound to our reactive Formlets.

And last, we have also shown a reactive abstraction for client-side routing that can be used to model in the type system the switching between various, possibly parameterized UI sections within single-page applications.

It is our hope that these abstractions will pave the road for a more robust, functional, reactive web development ahead.

References

1. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: The Essence of Form Abstraction. In: Programming Languages and Systems. Springer (2008) 205–220
2. McBride, C., Paterson, R.: Applicative Programming with Effects. Journal of Functional Programming **18**(01) (May 2007) 1–13
3. Bjornson, J., Tayanovskyy, A., Granicz, A.: Composing Reactive GUIs in F# using WebSharper. In: Implementation and Application of Functional Languages. IFL '10. Springer (2011) 203–216
4. Denuzière, L., Rodriguez, E., Granicz, A.: Piglets to the Rescue. In Plasmeijer, R., ed.: Proceedings of the 25th International Symposium on Implementation and Application of Functional Languages (IFL '13). (2013)
5. Fowler, S., Denuzière, L., Granicz, A.: Reactive Single-Page Applications with Dynamic Dataflow. In Pontelli, E., Son, T.C., eds.: Practical Aspects of Declarative Languages. Volume 9131 of Lecture Notes in Computer Science. Springer International Publishing (2015) 58–73
6. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web Programming Without Tiers. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: Formal Methods for Components and Objects. Volume 4709 of FMCO '06., Springer Berlin Heidelberg (2007) 266–296
7. Syme, D., Granicz, A., Cisternino, A.: Expert F# 3.0. APress (2012)

8. Denuzière, L., Granicz, A.: Enabling modular persistence for reactive data models in F# client-server web applications. Constrained and Reactive Objects Workshop (CROW) (2016)
9. O'Connor, R.: Functor is to lens as applicative is to biplate: Introducing multiplate. CoRR **abs/1103.2841** (2011)
10. van Laarhoven, T.: Overloading functional references. (2007)
11. Cherry, A.: Aether — total & partial lenses in F#. (2014)
12. Lindley, S., Wadler, P., Yallop, J.: Idioms are oblivious, arrows are meticulous, monads are promiscuous. Electronic Notes in Theoretical Computer Science **229**(5) (March 2011) 97–117
13. Meijer, E.: Reactive extensions (rx): Curing your asynchronous programming blues. In: ACM SIGPLAN Commercial Users of Functional Programming. CUFP '10, New York, NY, USA, ACM (2010) 11:1–11:1
14. Atkins Jr, T., Etemad, E.J., Atanassov, R.: CSS flexible box layout module. (2013)
15. Petricek, T., Syme, D.: The F# Computation Expression Zoo. In: Practical Aspects of Declarative Languages. Springer (2014) 33–48
16. Snoyman, M.: Developing Web Applications with Haskell and Yesod. O'Reilly Media, Inc., Sebastopol, CA. (2012)
17. Plasmeijer, R., Achten, P., Koopman, P.: 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, New York, NY, USA, ACM (2007) 141–152
18. Lijnse, B.: TOP to the Rescue—Task-Oriented Programming for Incident Response Applications. PhD thesis
19. Plasmeijer, R., Achten, P. In: iData for the World Wide Web – Programming Interconnected Web Forms. Volume 3945 of FLOPS '06. Springer Berlin Heidelberg (2006) 242–258
20. Liberty, J., Betts, P.: Programming Reactive Extensions and LINQ. 1st edn. Apress, Berkeley, CA, USA (2011)
21. Elliott, C., Hudak, P. In: Functional Reactive Animation. Volume 32(8) of ICFP '97. ACM, New York, NY, USA (1997) 263–273
22. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, Robots, and Functional Reactive Programming. In: Advanced Functional Programming. Springer (2003) 159–187
23. Krishnaswami, N.R.: Higher-order Functional Reactive Programming Without Spacetime Leaks. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. ICFP '13, New York, NY, USA, ACM (2013) 221–232
24. Ploeg, A., Claessen, K.: Practical principled FRP: Forget the past, change the future, FRPNow! In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ICFP 2015, New York, NY, USA, ACM (2015) 302–314
25. Czaplicki, E., Chong, S.: Asynchronous Functional Reactive Programming for GUIs. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '13, New York, NY, USA, ACM (2013) 411–422
26. Syme, D., Battocchi, K., Takeda, K., Malayeri, D., Fisher, J., Hu, J., Liu, T., McNamara, B., Quirk, D., Taveggia, M., Chae, W., Matsveyeu, U., Petricek, T.: Strongly-typed language support for internet-scale information sources. Technical report, Technical Report MSR-TR-2012-101, Microsoft Research (2012)