Piglets to the rescue

Declarative User Interface Specification with Pluggable View Models

Loïc Denuzière Ernesto Rodriguez Adam Granicz
{loic.denuziere,ernesto.rodriguez,granicz.adam}@intellifactory.com
IntelliFactory, http://intellifactory.com

Abstract

This paper introduces Pluggable Interactive GUI-lets (or Piglets, for short) as a mean for constructing reusable, reactive graphical user interfaces that can be instantiated over different view models and presentation layers. Piglets therefore provide an attractive alternative for pragmatic user interface specification that can target multiple content delivery channels from the same declarative, type-safe user interface definition, keeping programmers happy and highly productive.

Categories and Subject Descriptors Human-centered computing [*Human computer interaction (HCI)*]: Interactive system and tools; Software and its engineering [*Software notations and tools*]: Functional languages

Keywords functional reactive programming, GUI, piglets, F#, WebSharper

1. Introduction

Keeping functionally different parts of an application separate is an important step to take in order to avoid excessive complexity while retaining flexibility. In particular, in the domain of graphical user interfaces, the separation between the user input on one hand, and the construction and validation of result values on the other hand, has been a closely studied area in recent years. To various degrees, paradigms such as MVC, MVVM, functional reactive programming have provided elements of response to this issue.

Piglets are a new approach based on functional reactive programming and inspired by Formlets [4] [2] that help specify the structure and validation of data in a succint and type-safe way, while providing the flexibility to build an interactive user interface to input and visualize this data, and keeping these two aspects modular and reusable. They are designed around these fundamental principles:

Modularity: The "data definition and validation" component (sometimes called controller) can be specified once and used with different views within the same application, or even in different applications, such as the web and mobile version of a given application.

Genericity: Piglets do not assume the use of any particular GUI framework. Piglets have been successfully tested with various frameworks such as WebSharper HTML [8], Sencha Touch [3] and Windows Presentation Foundation [13].

Type safety: Individual form fields, as well as resulting data structures, are strongly typed, which provides extra guarantees with regards to the correctness of the data. In addition, while the terseness of Piglets can certainly be equaled by dynamically typed libraries, strong typing prevents a number of errors, in particular mismatch errors between different modules where one has not been correctly updated to reflect changes in the other.

DRYness: While many frameworks perform data validation at the UI level, Piglets provide a simple data validation system that performs at the controller level. This means that different views connected to the same back-end do not need to repeat any data validation in the UI. Similarly, common components of different controls can trivially be re-used between these controls.

This paper presents the concept of Piglets together with an implementation in F#, using WebSharper HTML for UI construction. An additional Sencha Touch user interface, using the WebSharper bindings for Sencha Touch, for the running example in this paper is provided in Appendix C.

2. Structure of a Piglet

2.1 Streams

The Piglets library relies on the fundamental notion of a *stream*, represented by type `Stream<'a>`. A stream represents a reactive sequence of values that can be subscribed to by multiple clients, and written to by multiple producers. Only the latest value of a stream is kept in memory, and clients are notified in real-time whenever a producer pushed a new value into the stream. Piglet streams are similar to knockout Observables [14] and Rx hot observables [11].

Streams are further refined into *readers* and *writers*, represented by types `Reader<'a>` and `Writer<'a>`, respectively. A Reader is a stream to which clients can subscribe, but to which producers cannot push values. Conversely, a Writer is a stream to which producers can push values, but to which clients cannot subscribe. Thus, a simple `Stream<'a>` as described in the previous paragraph is a subtype of both `Reader<'a>` and `Writer<'a>`. The distinction between the two is useful when a stream needs to process the written values and, for example, conditionally trigger a value depending on the written data. In such a case, the stream can be a subtype of both `Writer<'a>` and `Reader<'b>`, with `'a` $\neq$ `'b`.

2.2 Piglets

A Piglet is a data structure composed of two parts:

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

- A *stream* which represents the successive values returned by the Piglet. It can both be read (by composed Piglets and the view) and written to (by the view).
- A *view builder* responsible of feeding the individual streams that compose the result data into a *view function*. The view function then returns an actual user interface capable of interacting with the streams.

Two combinators are provided to create Piglets:

```
val Return : 'a → Piglet<'a, 'b → 'b>
```

`Return x` creates a Piglet with its stream initialized with `x`, one whose view builder doesn't feed any extra argument to the view function.

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

`Yield x` creates a Piglet with its stream initialized with `x` and with its view builder feeding this stream to the view function.

The most important operation on Piglets is combination, denoted ⊗, which allows to build complex Piglets out of simpler ones in order to represent a resulting data structure. It is used similarly to the applicative composition operator. It has the same effect as applicative composition on the Piglets' streams, and composes their view builders into a new builder that passes arguments from both original builders to the view function.

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

For example, the following code fragment builds a complex Piglet for a record type based on simple Piglets for its members. The resulting view builder feeds a stream for each member to the view function.

```
type Person = { first: string; last: string }

let PersonPiglet (init: Person) =
  Return (fun first last →
    { first = first; last = last })
  ⊗ Yield init.first
  ⊗ Yield init.last
```

The type of this Piglet is:

```
val PersonPiglet : Person →
  Piglet<Person, (Stream<string>
                  → Stream<string>
                  → 'b
                 ) → 'b>
```

which can be read as "a Piglet which returns a `Person`, and feeds two string streams to the view function".

Once this Piglet has been built, a user interface for it can be written. In the above example, the UI widget type is `'b`, showing that the same Piglet can be used with any user interface technology.

In this paper, the embedded language for HTML provided by WebSharper [8] will be used to illustrate Piglet user interfaces, which should be easy to understand for the reader familiar with HTML. The following built-in F# constructs will also be used:

```
// Reversed function application
let (|>) x f = f x
```

A Piglet is connected to a view function using the combinator `Render`, as follows:

```
let initUser =
  { first = "Alonzo"; last = "Church" }

PersonPiglet initUser
|> Render (fun first last →
  // Here, first and last are both Stream<string>
  Div [
    Controls.Input first
    Controls.Input last
    Div [
        Text "Your name is "
        Span [] |> Controls.ShowString first id
        Text " "
        Span [] |> Controls.ShowString last id
    ]
  ])
```

The functions in the `Controls` module are provided by the Piglets library for easy integration with WebSharper. `Input` takes a string stream and shows an input box which reads and writes the stream. `ShowString` takes a stream, a mapping function `f` to string and an HTML element into which it reactively writes the value of the stream mapped through `f`.

The above code can be inserted into a standard WebSharper application to provide reactive text inputs and display, as shown in Figure 1.

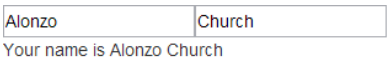

Figure 1. The rendered `PersonPiglet`

The purpose of constructing a value from simpler streams is generally to perform an action on this resulting value, such as submitting it to a server-side or saving it. To accomplish this, the constructed value can be forwarded to an arbitrary function using the `Run` combinator. This combinator doesn't modify the Piglet itself, but simply reads its stream.

```
PersonPiglet initUser
|> Run (fun name →
    Alert ("Hi, " + name.first))
|> Render ...
```

3. Data validation

Piglets provide a way to integrate input validation in the controller part, ie. the Piglet itself, rather than the view as is often done by various other approaches [16].

Streams actually carry a value of type `Result<'a>`, defined as:

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

All Piglets that have been described so far will only trigger `Success` values. These can be combined with validation combinators, which act as a filter on a Piglet's stream and transform its value into `Failure` when a given condition is not fulfilled. A validator creates a new stream that reads the underlying stream and produces:

- the underlying error message if the underlying stream already fails;

- a new error message if the underlying stream succeeds and the test fails;
- the same value if the underlying stream succeeds and the test succeeds.

A validator can be applied to any Piglet, either simple or complex. For example, we can check that neither the first name nor the last name of our user are empty, and that the resulting full name is present in a given dictionary:

```
let PersonPiglet (init: Person) =
  Return (fun first last →
    { first = first; last = last })
  ⊗ (Yield init.first
     |> Validation.Is Validation.NotEmpty
        "Please enter a first name.")
  ⊗ (Yield init.last
     |> Validation.Is Validation.NotEmpty
        "Please enter a last name.")
  |> Validation.Is (fun fullName →
      dictionary.Contains fullName) "Unknown user."
```

Whenever one of the validator fails, its failure propagates through the combinators and the resulting `Person` stream fails as well. If several individual streams fail, then their error messages accumulate. For example, if both input fields are empty, the result value is:

```
Failure [ "Please enter a first name.";
          "Please enter a last name." ]
```

In order to run an action even when the Piglet fails, a refinement of `Run` called `RunResult` can be used.

```
PersonPiglet initUser
|> RunResult (function
    | Failure ms → Alert (String.concat ", " ms)
    | Success x → Alert ("Hi, " + name.first))
```

4. Advanced Piglets operations

4.1 Submitting

The previously presented Piglets produce a new value, either `Failure` or `Success`, every time the user modifies one of the input fields. In a real-world situation, one generally wants to provide a way to confirm the data entered, and obtain a stream that only gets updated when this confirmation happens. This would be typically displayed as a "Submit" button.

This is the role of a *Submitter*. The type `Submitter<'a>` is a subtype of `Writer<unit>` and `Reader<'a>`. It possesses an underlying stream, and propagates the value of this stream to its subscribers whenever its `Writer` gets triggered. In order to create a Submitter, the Piglet is passed to the combinator `WithSubmit`. This combinator adds a `Submitter<'a>` to the view function arguments, and filters the Piglet's stream as described.

Figure 2 shows an example use of a Submitter. The module `Controls` provides the function `Submit` which displays a submit button that triggers the Submitter's `Writer<unit>` whenever clicked. When this happens, the Submitter's `Reader<Person>` is triggered with the current value of the Piglet.

Figure 2 also shows the use of `ShowResult`, which updates its display whenever the submitter `Reader<Person>` is triggered, i.e. whenever the user clicks "Submit".

4.2 Mapping

Mapping is a very common pattern in functional programming in which a function is applied to the value(s) in a given functor. Expectedly, Piglets provide different variants of this facility:

```
PersonPiglet initUser
|> WithSubmit
|> Render (fun first last submit →
  Div [
    Controls.Input first
    Controls.Input last
    Controls.Submit submit
    Div [
        Text "Your name is "
        Span [] |> Controls.ShowResult submit
            (function
             | Success x →
                 Text (x.first + " " + x.last)
             | Failure ms →
                 Text (String.concat ", " ms))
    ]
  ])
```

Figure 2. A view with a submit button and displaying errors

```
val Map : ('a → 'b)
       → Piglet<'a, 'v>
       → Piglet<'b, 'v>
val MapToResult : ('a → Result<'b>)
               → Piglet<'a, 'v>
               → Piglet<'b, 'v>
```

This makes it possible, for example, to provide a Piglet for values of a certain type which passes to the view a stream of a different (mapped) type. However, the use cases of `Map` itself are rare because mapping is generally performed at the applicative level, i.e. the function passed to `Return` in the `Person` example.

Another type of mapping, however, is very useful, especially in the context of web development. The `MapAsync` family of functions allow mapping over functions that can be executed asynchronously.

```
val MapAsync : ('a → Async<'b>)
            → Piglet<'a, 'v>
            → Piglet<'b, 'v>
val MapToAsyncResult : ('a → Async<Result<'b>>)
                    → Piglet<'a, 'v>
                    → Piglet<'b, 'v>
```

In WebSharper, for example, such a function can perform an Ajax call and return the response of this call; `MapAsync` then passes it as the value of the Piglet. In the case of a `Person`, if the goal is to save the input data to the server when `Submit` is triggered, then the Piglet can be mapped asynchronously:

```
PersonPiglet initUser
|> MapAsync (fun person →
    async {
      let! serverResult = SubmitToServer person
      return serverResult
   }
)
```

4.3 Fine-grained validation

When a validator is applied to a Piglet, the stream of this Piglet is not modified: the Piglet returned by the validator has its own stream, which listens to the underlying Piglet's stream. This is necessary so that input controls still have a value to show to the user even when this value is deemed invalid by the validator. It would be inconvenient to get the field wiped as soon as the user inputs an invalid value.

What this means, however, is that the stream passed to the view function by a simple Piglet does not contain any information about the validity of its value. This information would be useful to highlight the actual invalid input, rather than just showing the corresponding error message at the bottom of the form.

Piglets provide a solution to this issue. Every error message is tagged with a unique identifier that can be used to track which base stream caused it. To make use of this facility, the base `Reader` provides a `Through` method that takes the outer `Reader` as argument (for example, the `Submitter` for the global result) and returns the same value as the original, except when the outer `Reader` has error messages associated with the base `Reader`; in which case these error messages are returned instead.

```
type Reader<'a> =
  // ...
  member Through : Reader<'b> -> Reader<'a>
```

This allows the example to be enriched with error messages positioned at the source of each error.

```
PersonPiglet initPerson
|> WithSubmit
|> Render (fun first last submit ->
  Div [
    Controls.Input first
    Controls.ShowErrors (first.Through submit)
      (fun errors -> String.concat ", " errors)
    Controls.Input last
    Controls.ShowErrors (last.Through submit)
      (fun errors -> String.concat ", " errors)
    Controls.Submit submit
  ])
```

Logically, `Controls.ShowErrors` shows the error messages associated with the given `Reader`. In this case, these error messages only show up when the user clicks "Submit", since the `Reader`s passed to `ShowErrors` are passed through `submit`.

If the error messages need to be shown "in real-time", i.e. on every change of the base streams as opposed to only when submitted, then the submitter's input stream is the one needed. It is accessible using the `Input` property of the `Submitter` class.

5. Piglet collections

A common necessity in user interfaces is to provide the user with a way to input multiple values of the same type, with the possibility to add, remove, and reorder these values. Visually, this can take the shape of a grid, a list, or simply a sequence of similar sub-forms.

For example, the `PersonPiglet` can be extended to include a list of pets, defined as follows:

```
type Species = Cat | Dog | Piglet
type Pet = { species: Species; name: string }
type Person =
  { first: string; last: string; pets: Pet[] }
```

The Piglet for a single Pet can be defined as follows:

```
let PetPiglet (init: Pet) =
  Return (fun species name →
    { species = species; name = name })
  ⊗ Yield init.species
  ⊗ Yield init.name
```

To keep the code modular, it would be preferable to use `PetPiglet` as part of the definition of `PersonPiglet`. With the related Formlets approach[4], this can be achieved with the following `Many` combinator[2]:

```
val Many : Formlet<'a> → Formlet<'a[]>
```

With Piglets, the presence of the view builder complicates this combinator. It requires a mechanism to specify the rendering of each inner view, and a way to attach it to the outer view. The individual streams of each inner Piglet have to be combined consistently to generate the final sequence value.

The corresponding `Many` combinator for Piglets therefore has a more complex type signature.

```
val Many : 'a
        → ('a → Piglet<'a, 'v → 'w>)
        → Piglet<
            'a[],
            (Many.Stream<'a, 'v, 'w> → 'x) → 'x>
```

- The first argument, of type `'a`, is the value with which newly inserted inner Piglets will be initialized.
- The second argument, of type `'a → Piglet<'a, 'v → 'w>`, defines how inner Piglets are created from an initial value. In the previous example, it is the function `PetPiglet`.
- The combinator returns a Piglet whose stream has values of type `'a[]`, and adds a `Many.Stream` to the view function arguments. `Many.Stream` provides functionality to edit and display the collection. It can be read as a `Reader<'a[]>`, but it can only be written to by its `Add` method, which appends an element to the back of the collection and a corresponding inner view at the end of the container, or by inner items through its `Render` method.

```
type Many.Stream<'a, 'v, 'w> =
  // ...
  member Render : Container<'w, 'u>
               → (Operations → 'v)
               → 'u
```

This method is responsible for rendering inner elements into a container, and providing these elements with the streams they need to be able to delete and reorder items.

- The first argument to `Render` is the container into which the inner Piglets will be sequentially rendered. It satisfies the following interface:

  ```
  type Container<'in, 'out> =
      abstract member Add : 'in → unit
      abstract member Remove : int → unit
      abstract member MoveUp : int → unit
      abstract member Container : 'out
  ```

  where `'in` specifies the type of element returned by the view function of inner Piglets, and `'out` is the type of element returned by the view function of the outer Piglet. Every rendering engine (such as WebSharper HTML elements) needs to implement this interface in order to be used with `Many`.
- The second argument is the function that gets called when a new inner Piglet needs to be rendered. It receives an `Operations` object, containing `Writer`s for deleting and reordering this item, and any arguments fed by the inner view builder.

  Concretely, the `Operations` object contains:

  - Two reordering writers, `MoveUp` and `MoveDown` which, when triggered, move the current inner item in the view and in the resulting collection.
  - A `Delete` writer which, when triggered, removes the current inner item from the view and from the resulting collection.

Putting all of this together, the new `PersonPiglet` is implemented in Appendix A and rendered in Appendix B as follows:

- First, a Piglet for individual Pets is created (`PetPiglet`, Appendix A l. 22), including appropriate validation.
- Then, the outer Piglet for the Person is defined (`PersonPiglet`, Appendix A l. 30), also including appropriate validation. It uses the `Many` combinator to include the list of pets (Appendix A l. 41).
- After the Piglets are defined, they are rendered (Appendix B l. 1). The pets are represented by the view argument `pets`, of type `Many.Stream<Pet, PetRender, Element>` where `PetRender` is the view function for the `PetPiglet`:

```
type PetRender =
  Stream<Pet> → Stream<string> → Element
```

  Note that the return type of `PetRender` is specialized to the `Element` type from WebSharper because it is used as such in the body of the view function; but the Piglet itself is still generic in terms of the view type.
- The `Render` method from the `Many.Stream` is called in the view function (Appendix B l. 8) to display the Piglets for every Pet. It receives an `HTMLContainer`, which is an implementation of `Container<Element, Element>` provided by `Controls`. It also receives a view function for individual Pets. This view function uses the Writers it receives from its argument `ops` to display buttons which allow moving the current pet up or down, or deleting it.
- The `Add` method from the `Many.Stream` is used in the view function (Appendix B l. 23) to display a button which, when clicked, adds a new pet to the collection, initialized as `defaultPet`.
- Finally, since WebSharper provides combinators to generate HTML elements, it is easy to display the final value (Appendix B l. 25). The pets are displayed by mapping over the pets array and returning HTML elements (Appendix B l. 32).

Figure 3. The rendered Piglet from Appendix B after the user added two pets

A commonly needed variant is to create new entries in the collection, not from a fixed initial value, but interactively. For example, one can have a complex form to create entries, and an editable grid that displays the collection of created entries. The Piglet used for the grid entries then needs to be different from the one used for the creation form: they might have different individual input fields, and most importantly, the creation form will require a Submit button while grid entries might not.

For these reasons, such a case requires a more general version of `Many`, which we call `ManyPiglet`, using a generalized `Many.Stream` called `Many.GenStream`.

```
val ManyPiglet : Piglet<'a, 'y -> 'z>
        → ('a → Piglet<'a, 'v → 'w>)
        → Piglet<
            'a[],
            (Many.GenStream<'a, 'v, 'w, 'y, 'z>
                → 'x) → 'x>
```

This combinator is similar to `Many`, but instead of providing an initial value as first argument, we provide the Piglet for the creation form. This form can then be displayed by the view function using the `AddRender` method of the `GenStream`.

6. Generalizing the View Function

Often enough it is desired to provide the same functionality across different user interfaces. The common scenario is extending an application to work on both a mobile device and a desktop computer. The genericity of the Piglets rendering mechanism enables such possibility without requiring any modification to the Piglet itself. It is only necessary to provide the Piglet with a different rendering function that generates the desired view output. To illustrate this feature, the example from Appendix B was also extended to work with the Sencha Touch bindings for WebSharper, as well as Windows Presentation Foundation (WPF) for desktop interfaces. The code for Sencha Touch can be found at Appendix C. The Sencha Touch output is portrayed in Figure 4, and the WPF output is visible in Figure 5.

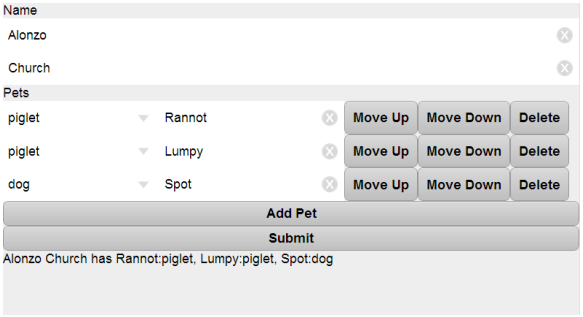

Figure 4. The example Piglet rendered using Sencha Touch[3]

7. Related work

The Piglet approach is inspired by Formlets [2, 4, 17]. Formlets provide a type-safe and declarative mechanism to generate user interfaces for a given data structure, and are speficied in many ways similarly to Piglets. However, since Formlets generate their view under the hood, it is difficult to customize appearance. Reactive behavior is also limited due to the fact that the input fields are generated as part of the automatic process. Piglets are a promising alternative to Formlets where finer control over the appearance of the resulting user interfaces is desired. For instance, the running example of this paper was implemented using both WebSharper HTML markup and Sencha Touch without any modification to the underlying Piglet.

Functional reactive programming (FRP) has been researched as an approach to developing interactive user interfaces [1, 5, 6]. The main advantage of this approach is that heavily reduces the amount of code required to update a user interface and hides these details in the notation. One notable recent example is the Elm Programming

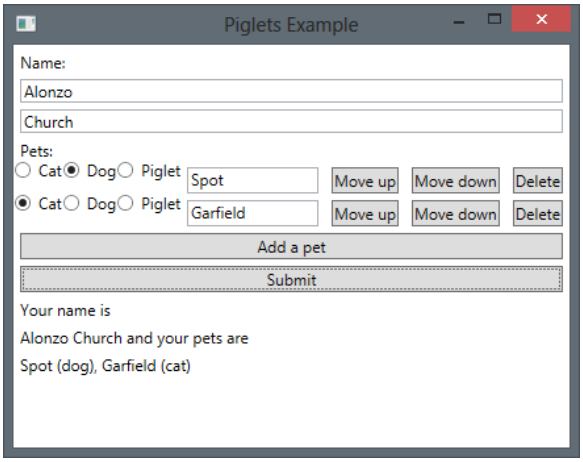

Figure 5. The example Piglet rendered using Windows Presentation Foundation

Language[6]. It is a web-oriented functional language inspired by Haskell and uses FRP as its approach to handle user input. It has similar combinators to those in Piglets and enable building interactive user interfaces easily.

Piglets extend functional reactive programming with a generic mechanism to change view models for collecting user input. Different user interfaces can be created to work against the same input data with minimal amount of extra code. Additionally, Piglets associate the data validation logic with the input data structures in a single place, which enforces the derived views to respect the validation rules. Since the error notification mechanism for Piglets is generic, user interfaces can be built without requiring detailed knowledge of the validation process. This ensures a more consistent behavior across multiple user interfaces and eliminates the possibility of propagating invalid data due to a faulty interface.

An alternative approach to reactive programming is shared state, as used by workflow management systems such as iTasks[12]. An advantage of this approach is to provide tools such as merging algorithms that allow multiple tasks to work on a shared state. This allows GUIs to apply modifications across a distributed system. User interfaces are derived from a task and specified in a declarative style. The system automatically takes care of rendering it, providing the advantage of simplicity at the cost of less flexibility in the rendering process.

In the context of dynamic languages, similar approaches have been developed to provide generic mechanisms that update data with different display mechanisms [9, 14, 15]. The approaches are very flexible and easily allow implementing reactive behavior inside the interface. Piglets take advantage of the full power of the ML type system allowing the compiler to assist the developer in massively reducing semantic errors in the application. Furthermore, our typed approach also provides valuable insights for developing additional user interfaces to existing data. The type system also provides strong guarantees that all views asociated with a Piglet will be modified accordingly when the underlying data structure is modified. Static typing comes with a cost, though. Currently it is impossible to recursively embed a Piglet inside itself with the `Many` combinator since doing so results in an infinite type even though the code would behave correctly at run-time.

Finally, Piglets have some similarities to the MVC pattern [10]. In particular, this approach also separates the mechanism to display and manipulate data. In this approach, the controller usually provides an interface that the view uses to perform operations over the data. The Piglet approach is fundamentally more data-oriented than MVC. This is due to the fact that they are inspired from FRP where the core idea is to interpret data as a stream over time. This provides the developer with the flexibility to work in a generic way over any data structure instead of having to heavily depend on the interface of the controller.

8. Conclusions

Functional reactive programming has been an active field of research in recent years. It provides a simple way to combine values gathered interactively from a user interface into more complex data structures. The user interface uses a minimal amount of extra code to take care of updating the data, and a separate controller then gathers this data and processes it as needed.

In this paper, we have developed an approach, which we call Pluggable Interactive GUI-lets, or Piglets, derived from functional reactive programming, which extends the above capabilities and allows to describe the controller in a concise, declarative way. The Piglet controller uses the type system to ensure that data is collected correctly. It decides which data streams the user interface needs to be able to interact with. It also includes the validation logic which is common to all potential different views associated with it. It is independent from the GUI frameworks used to produce these views, and the same Piglet controller can be used with different view instantiations. This approach is especially useful for writing applications that need to target multiple content delivery channels, and may have desktop, web, and mobile user interfaces at the same time.

Piglets also provide more advanced facilities, such as the `Many` combinator that abstracts away the manipulation of collections of data, including the display of multiple inputs for items in such a collection, in a grid or any other type of nested layout. They also give user interfaces the flexibility in the way they display collected values and validation messages, whether grouped or associated with their corresponding individual fields.

Our next steps in the development of Piglets are the formalization of the semantics of the Piglet controller, and an evaluation of the feasibility and convenience of implementing classic functional reactive programming combinators such as combining, folding and accumulating. Another ambitious goal is to take further advantage of the modularity of Piglets to run in a client-server configuration, where the Piglet controller sits on the server and synchronizes potentially different view clients, using techniques such as operational transformations [18] or differential synchronization [7].

A. The common Person Piglet

```
type Species = | Cat | Dog | Piglet
type Pet = { species: Species; name: string }
type Person =
  { first: string; last: string; pets: Pet[] }

let showSpecies = function
    | Cat → "cat"
    | Dog → "dog"
    | Piglet → "piglet"

let dictionary =
    set [
        ("Alonzo", "Church")
        ("Alan", "Turing")
        ("Edsger", "Dijkstra")
        ("Charles", "Babbage")
    ]

let defaultPet =
```

```
  { species = Piglet; name = "Spot" }

let PetPiglet (init: Pet) =
  Return (fun species name →
     species = species; name = name )
  ⊗ Yield init.species
  ⊗ (Yield init.name
     |> Validation.Is Validation.NotEmpty
        "Please enter the pet's name.")

let PersonPiglet (init: Person) =
  Return (fun first last pets →
    { first = first;
      last = last;
      pets = pets })
  ⊗ (Yield init.first
     |> Validation.Is Validation.NotEmpty
        "Please enter a first name.")
  ⊗ (Yield init.last
     |> Validation.Is Validation.NotEmpty
        "Please enter a last name.")
  ⊗ Many defaultPet PetPiglet
  |> Validation.Is (fun fullName →
        dictionary.Contains
          (fullName.first, fullName.last))
      "Unknown user."
  |> WithSubmit

let initUser =
  {first = "Alonzo"; last = "Church"; pets = [||]}
```

B. A Piglet rendered using WebSharper HTML

```
PersonPiglet initUser
|> Render (fun first last pets submit →
  Div [
    H3 [Text "Name:"]
    Controls.Input first
    Controls.Input last
    H3 [Text "Pets:"]
    pets.Render (HtmlContainer(Div[]))
      (fun ops species name →
        Div [
          Controls.Radio species
            [ (Cat, "Cat");
              (Dog, "Dog");
              (Piglet, "Piglet") ]
          Controls.Input name
          Controls.Button ops.MoveUp
            -< [Text "Move up"]
          Controls.Button ops.MoveDown
            -< [Text "Move down"]
          Controls.Button ops.Delete
            -< [Text "Delete"]
        ])
    Controls.Button pets.Add -< [Text "Add a pet"]
    Controls.Submit submit
    Div [
        Text "Your name is "
        Span [] |> Controls.ShowResult submit
          (function
           | Success x →
              [ Text (x.first + " " + x.last +
                      " and your pets are ")
                Text (x.pets
                  |> Array.map (fun pet →
                      pet.name + " (" +
                      showSpecies pet.species + ")")
                  |> String.concat ", ") ]
           | Failure ms →
             [Text (String.concat ", " ms)])
    ]
  ])
```

C. A Piglet rendered using Sencha Touch

```
let view (c: Ext.Container) =
 PersonPiglet initUser
 |> Render (fun first last pets submit →

  let pets =
    SenchaContainer(ExtCfg.Container().Create())
    |> pets.Render (fun opts species name →
         let petOptions =
           Array.map (fun s →
               {text=showSpecies s;value=s})
             [|Piglet;Cat;Dog|]

         let elems : Ext.Component [] =
           [|
             ExtCfg.field.Select(Options=petOptions)
               .Create()
             |> Select.WithSelect(id,id,species);

             Ext.field.Text() |> Text.WithText name;

             ExtCfg.Button(Text="Move Up").Create()
             |> Button.WithTap opts.MoveUp;

             ExtCfg.Button(Text="Move Down")
               .Create()
             |> Button.WithTap opts.MoveDown;

             ExtCfg.Button(Text="Delete").Create()
             |> Button.WithTap opts.Delete
           |]

         ExtCfg.Container(Items=elems,Layout="hbox")
           .Create()
       )

  let showPerson p =
    let pets =
      p.pets
      |> Seq.map (fun x →
           String.concat ":"
             [
               x.name;
               (showSpecies x.species)
             ])
      |> String.concat ", "
    p.first + " " + p.last + " has " + pets

    let infoLabel =
      Ext.Label()
      |> Label.WithLabelGen (showPerson,submit)

    let errorContainer =
      let err = ExtCfg.Container().Create()
      submit.Subscribe(
        fun msg →
          err.RemoveAll (true,true);
```

```
            match msg with
              | Failure ms →
                ms |> Seq.map (fun m →
                        ExtCfg.Label(Html="m.Message")
                          .Create())
                   |> Seq.toArray
                   |> err.Add |> ignore
              | _ → ()
        ) |> ignore
        err

    let items : Ext.Component [] =
      [|
        ExtCfg.Label(Html="Name").Create();
        Ext.field.Text() |> Text.WithText first;
        Ext.field.Text() |> Text.WithText last;

        ExtCfg.Label(Html="Pets").Create();
        pets;

        ExtCfg.Button(Text="Add Pet").Create()
        |> Button.WithTap pets.Add;

        ExtCfg.Button(Text="Submit").Create()
        |> Button.WithTap submit

        infoLabel;
        errorContainer
      |]

  ExtCfg.Container(Items=items).Create()
  |> c.Add
 )
```

References

[1] H. Apfelmus. Reactive-banana. Library Home Page `http://www.haskell.org/haskellwiki/Reactive-banana`, Retrieved on August 1, 2013.

[2] J. Bjornson, A. Tayanovsky, and A. Granicz. Composing Reactive GUIs in F# using WebSharper. In *Implementation and Application of Functional Languages*. Springer, 2010.

[3] J. E. Clark. *Sencha Touch Mobile JavaScript Framework*. Packt Publishing Ltd, 2012.

[4] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. An Idioms Guide to Formlets. Technical report, University of Edinburg, 2008.

[5] A. Courtney. Genuinely Functional User Interfaces. In *ACM Sigplan Workshop on Haskell*, pages 41–69, 2001.

[6] E. Czaplicki and S. Chong. Asynchronous Functional Reactive Programming for GUIs. In *Programming Language Design and Implementation*, Seattle, WA, 2013. ACM.

[7] N. Fraser. Differential Synchronization. In *DocEng'09, Proceedings of the 2009 ACM Symposium on Document Engineering*, pages 13–20, 2 Penn Plaza, Suite 701, New York, New York 10121-0701, 2009. URL `http://neil.fraser.name/writing/sync/eng047-fraser.pdf`.

[8] A. Granicz, A. Tayanovskyy, and J. Bjornson. WebSharper. Home Page `http://websharper.com`, Retrieved on August 1, 2013.

[9] A. Gutierrez. Web application client side architecture with angularjs. 2013.

[10] G. Krasner and S. Pope. A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988. URL `http://citeseer.ist.psu.edu/krasner88description.html`.

[11] E. Meijer. Reactive extensions (Rx): curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 11. ACM, 2010.

[12] S. Michels, R. Plasmeijer, and P. Achten. iTask as a New Paradigm for Building GUI Applications. In *Implementation and Application of Functional Languages*. Springer, 2010.

[13] A. Nathan. *Windows Presentation Foundation Unleashed*. Sams Publishing, 2006.

[14] J. Papa. Knockout's Built-in Bindings for HTML and JavaScript. MSDN Magazine `http://msdn.microsoft.com/en-us/magazine/hh852598.aspx`, Retrieved on August 1, 2013.

[15] A. Ronacher. Pluggable views. Library Home Page `http://flask.pocoo.org/docs/views/`, Retrieved on August 1, 2013.

[16] S. Senf. Ketchup – Tasty Form Validation. Library Home Page `https://github.com/mustardamus/ketchup-plugin`, Retrieved on August 1, 2013.

[17] M. Snoyman. *Developing Web Applications with Haskell and Yesod*, chapter 8. O'Reilly Media, 2012.

[18] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, pages 59–68. ACM, 1998.