An idiom's guide to formlets

Ezra Cooper Sam Lindley Philip Wadler Jeremy Yallop
The University of Edinburgh

Abstract

Formlets in Links decouple user interface from data, a vital form of abstraction supported by very few web frameworks. Formlets are best defined in terms of idioms, not monads or arrows as one might suppose from the existing literature.

1. Introduction

Say you want to present the user with an HTML form for entering a pair of dates. In your initial design, a date is represented as a pair of pulldown menus, one to select a month and one to select a day. Later, you choose to replace each date by a single text field, for entering a date as text.

In the usual web frameworks, such a change will require widespread changes to the code. Under the first design, the HTML form will contain four menus, and code that handles the response will need to extract the menu entries for each month and day, and combine these to yield a pair of values of an abstract type Date. Under the second design, the HTML will contain two text input fields, and the code that handles the response will need to extract the texts and parse each to yield the pair of dates.

How can we structure a program so that it is isolated from this choice? What we want is the notion of a part of a form for representing dates—we call this a *formlet*. The designer of the formlet should choose the HTML presentation, and decide how to process the input into a Date value. Clients of the formlet should not need to know the HTML in the form, or how it is processed to yield an abstract value. And, of course, we should be able to compose formlets to build larger formlets.

Once described, this form of abstraction seems obvious and necessary. But remarkably few web frameworks support it. In particular, we failed to support it in the first version of our web language, Links.

The standard HTML form interface presents several problems in realising the above. For example:

- There is no static association between the form definition and the code that handles it; thus the interface is fragile. The form and the handling code need to be kept manually in sync.
- Field values are always submitted individually and always as strings: HTML provides no facility for processing data or giving it structure.
- Given two forms, there is generally no easy way to combine them into a new form without fear of name clashes amongst the fields—thus it is not easy to write a form that uses subcomponents abstractly. In particular, there is no easy way to use the same form twice in a larger form.

Formlets address all of these problems. They statically check that a form and its handler are compatible, they translate raw form inputs into structured values, and they automatically generate distinct names for distinct fields, enabling composition.

Conventional web programming frameworks such as PHP (PHP) and Ruby on Rails (Ruby on Rails) prevent abstraction by exposing to programmers the individual fields. Research frameworks such as PLT Scheme (Graunke et al. 2001), JWIG (Christensen et al. 2003), scriptlets (Elsman and Larsen 2004), Ocsigen (Balat 2006) and Lift (Lift) and our previous design for Links (Cooper et al. 2007) all fall short in the same way.

Three existing web programming frameworks that do support some degree of abstraction over form components are WASH (Thiemann 2005), iData (Plasmeijer and Achten 2006) and WUI (Hanus 2006, 2007).

Semantically, a formlet is an *idiom* (McBride and Paterson 2007). Idioms generalise monads (Moggi 1989; Wadler 1995; Benton et al. 2002). Idioms provide a model for *static computation*. Static computation is exactly what is needed for abstracting forms: monads are too restrictive as we describe in Section 3.

A particularly nice property of idioms is that they are closed under composition. We initially define formlets as the composition of a name generation idiom, an accumulation idiom and an environment idiom. Thus the formlet idiom is defined completely abstractly in terms of the basic idioms. Idioms promote modular programming. Once we have given the basic definition of formlets, we extend the formlet idiom to support validation by composing with a further accumulation idiom and an error idiom.

Entire implementations of non-validating and validating formlets are included in the paper. Links offers syntactic sugar for formlets, making them easier to use; even so, the basic idiom structure could be implemented in any functional language and is quite usable without the sugar. We have created basic implementations of formlets in Haskell, OCaml, Python and C♯.

This paper focuses on abstractions for building forms. There are many related issues that are not covered. In particular, we do not dwell on security issues, and we do not attempt to check the validity of XHTML statically. (Static validity checking for XML is well-studied (Møller and Schwartzbach 2005) and orthogonal to our treatment of formlets, so we focus elsewhere.)

The contributions of this paper are:

- a unified design for a compositional abstraction over HTML forms (Section 2);
- a definition of this feature in terms of idioms, a simple semantic framework (Section 3);

- a desugaring transformation that makes formlets easier to use (Section 4);
- an extended design supporting form validation (Section 5); and
- a comparison of form-abstraction features in web-programming systems (Section 6).

Section 7 concludes.

2. Formlets by example

Abstractly, formlets are composable templates that when instantiated yield two outputs: a *rendering* and a *collector*. The rendering is the HTML representation of the form, and the collector is a function that transforms raw submitted form data into a structured value. A single name source is used to ensure that the rendering and the collector are in accordance.

We illustrate formlets at a high level with an example (Figures 1, 3 and 2). We assume familiarity with HTML and use Links syntax. Links is a strict typed functional programming language designed for web-programming. Semantically it is close to the ML family of languages; syntactically it resembles JavaScript. For more details on Links see our earlier work (Cooper et al. 2007).

The following Links code creates a formlet, called $date$, with two text input fields, labelled "Month" and "Day":

```
sig date : Formlet(Date);
var date =
  formlet
    <div style="padding:8px">
     <span style="border:2px solid; padding:4px;">
      Month: {inputInt → month},
      Day: {inputInt → day}
     </span>
    </div>
  yields
    makeDate(month, day);
```

This defines $date$ as a value, of type $Formlet(Date)$, which can be embedded in a page as an HTML form. Upon submission of the form, this formlet will yield a $Date$ value representing the date entered; the user-defined $makeDate$ function translates the day and month into a suitable representation.

(The Links keyword sig simply declares a type signature for a variable binding or function definition. The expression var x = m; n binds the value of m to the variable x in n.)

A formlet expression has two components: a *body* and a *yields clause*. The body of the $date$ formlet is

```
<div style="padding:8px">
 <span style="border:2px solid; padding:4px;">
  Month: {inputInt → month},
  Day: {inputInt → day}
 </span>
</div>
```

and its yields clause is

```
makeDate(month, day);
```

Links allows XML to be embedded in Links code using *XML quasiquotes*. XML values in Links are forests represented as lists of XML nodes. An XML value has type Xml. An XML quasiquote is either a singleton forest consisting of one element <t as>...</t> or an arbitrary forest denoted by the special syntax <#>...</#>. A Links expression e of type Xml is embedded in a quasiquote using an *antiquote* written {e}.

The body of a formlet expression is a *formlet quasiquote*. Formlet quasiquotes augment XML quasiquotes with a further kind of antiquote: *formlet bindings*. A formlet binding {f → p} binds the value yielded by f to the pattern p in the yields clause. Thus the variables $month$ and day are bound to the values yielded by each instance of the $inputInt$ formlet. They are bound inside the yields clause. The value $inputInt$: $Formlet(Int)$ is a formlet that allows the user to enter an Int using an HTML text input element. Although the $inputInt$ formlet is used twice, the formlet library ensures that no field name clashes arise.

The body of a formlet expression corresponds roughly to the rendering and the yields clause to the collector of a formlet. More precisely, the quasiquote performs two roles: it defines how to combine the renderings of the sub-formlets to give a composite rendering for the whole formlet, and how to bind the values returned by the collectors of the sub-formlets. The yields clause defines how the collector should combine the bound values into a single return value.

Next we illustrate how user-defined formlets can be usefully combined to create larger formlets. We construct a travel formlet which asks for a name, an arrival date, and a departure date.

```
fun travel(title) {
  formlet
    <#>
     <h1>{stringToXml(title)}</h1>
     Name: {input → name}
     <div>
      Arrival date: {date → arrive}
      Departure date: {date → depart}
     </div>
     {submit("Submit")}
    </#>
  yields
    (name, arrive, depart)
}
```

The $input$ formlet allows the user to enter a $String$ using an HTML text input element. The library function $submit$ simply returns the HTML for a submit button without a name attribute. (The $submit$ function covers the common case where there is a single button on a form. We also provide a function $submitButton$: $(String) \rightarrow Formlet(Bool)$ which constructs a submit button formlet, allowing multiple buttons on the same form to be distinguished.)

(Functions in Links take multiple arguments. The declaration fun $f(ps)\{e\}$ defines a k-ary function with arguments bound by the patterns in $ps = p_1 \ldots p_k$ and body e. Anonymous functions are written in the same way, but omitting the name of the function. Curried functions can be defined using declarations of the form fun $f(ps_1)\ldots(ps_k)\{e\}$. A function of k arguments is given the type $(A_1, \ldots, A_k) \rightarrow B$ where $A_1, \ldots, A_k$ are the types of the arguments and B is the return type.)

Having created a formlet, how do we use it? For a formlet to become a form, we need to connect it with a *handler*, which will consume the form input and perform the rest of the user interaction and render it as part of a web page. In Links, web pages have type $Page$. The function $xmlPage$ outputs raw XML as a web page. The function $formletPage$ takes a formlet and a handler, renders the formlet with the handler attached, and outputs the resulting form element wrapped in html and body tags. (When we add validation in Section 5 we will replace $xmlPage$ and $formletPage$ with more general constructs.)

Continuing the above example, we render $travel$ onto a simple page, and attach a handler that displays the chosen itinerary back to the user.

```
sig date : Formlet(Date);
var date =
  formlet
    <div style="padding:8px">
     <span style="border:2px solid; padding:4px;">
      Month: {inputInt → month},
      Day: {inputInt → day}
     </span>
    </div>
  yields
    makeDate(month, day);

fun travel(title) {
  formlet
    <#>
     <h1>{stringToXml(title)}</h1>
     Name: {input → name}
     <div>
      Arrival date: {date → arrive}
      Departure date: {date → depart}
     </div>
     {submit("Submit")}
    </#>
  yields
    (name, arrive, depart)
}

sig displayItinerary : ((String, Date, Date)) → Page
fun displayItinerary((name, arrive, depart)) {
  xmlPage(
    <html>
     <body>
      Itinerary for: {stringToXml(name)}.
      Arriving: {dateToXml(arrive)}.
      Departing: {dateToXml(depart)}.
     </body>
    </html>)
}

formletPage(
  travel("Welcome to Bruntsfield Travel Services"),
  displayItinerary)
```

Figure 1. Date example

```
sig displayItinerary : ((String, Date, Date)) → Page
fun displayItinerary((name, arrive, depart)) {
  xmlPage(
    <html>
     <body>
      Itinerary for: {stringToXml(name)}.
      Arriving: {dateToXml(arrive)}.
      Departing: {dateToXml(depart)}.
     </body>
    </html>)
}

formletPage(
  travel("Welcome to Bruntsfield Travel Services"),
  displayItinerary)
```

A more interesting application might render another form on the *displayItinerary* page, one which allows the user to confirm the itinerary and purchase tickets; it might then take actions such as lodging the purchase in a database, and so on.

2.1 Syntactic sugar

Figure 2 shows the desugared version of the date example. XML values can be constructed using the tag^x and $text^x$ functions in conjunction with the standard list concatenation operation. Formlets are slightly more complicated. The tag^x and $text^x$ functions are lifted into the formlet as *tag* and *text* in order to allow raw XML to be mixed with formlets. Composition of formlets makes use of the standard idiom operations *pure* and $\otimes$. The XML and formlet primitives are covered in detail in Section 3.

The sugar makes it easier to freely mix raw XML with formlets. Without the sugar, dummy bindings (underscores) are needed to bind formlets consisting of raw XML (e.g. the calls to *pure* in Figure 2). Furthermore, formlets nested inside XML would have to be rebound (e.g. the second call to *pure* in the body of *travel* in Figure 2).

```
sig date : Formlet(Date);
var date =
  tag("div", [("style","padding:8px")],
   tag("span", [("style",
                 "border:2px solid; padding:4px")],
    pure(fun (_)(month)(_)(day) {
           makeDate(month, day)
         }) ⊗
    text("Month: ") ⊗ inputInt ⊗
    text("Day: ") ⊗ inputInt));

fun travel(title) {
  pure(fun (_)(_)(name)(arrive, depart)(_) {
         (name, arrive, depart)
       }) ⊗
  tag("h1", [], xml(stringToXml(title))) ⊗
  text("Name: ") ⊗ input ⊗
  tag("div", [],
      pure(fun (_)(x)(_)(y) {(x, y)}) ⊗
      text("Arrival date: ") ⊗ date ⊗
      text("Departure date: ") ⊗ date) ⊗
  xml(submit("Submit"))
}

sig displayItinerary : ((String, Date, Date)) → Page
fun displayItinerary((name, arrive, depart)) {
  xmlPage(
    tag^x("html", [],
     tag^x("body", [],
      text^x("Itinerary for: ") ++
      xml^x(stringToXml(name)) ++
      text^x("Arriving: ") ++
      xml^x(dateToXml(arrive)) ++
      text^x("Departing: ") ++
      xml^x(dateToXml(depart)) ++
      text^x("."))))
}

formletPage(
  travel("Welcome to Bruntsfield Travel Services"),
  displayItinerary)
```

Figure 2. Desugared date example

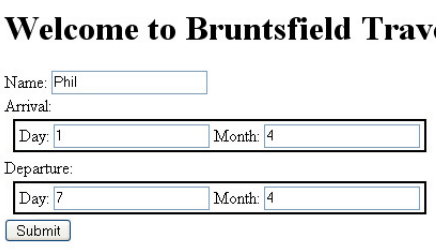

Figure 3. Date example screenshot

In fact, with a different representation of formlets, it is possible to write unsugared code where XML freely mixes with formlets, without the need for dummy bindings or rebinding. The idea is to compose the renderings of sub-formlets by plugging them into a multi-holed context rather than using tag and $text$. We have implemented a prototype of this technique in OCaml. Statically typing multi-holed contexts is non-trivial, so we defer the details to another paper.

It seems that some form of syntactic sugar is necessary if one wants formlets to appear next to their bindings; without the sugar the binding may appear a great distance from the formlet being bound.

2.2 Separating logic from layout

The benefits of separating logic from layout in web application development are well documented (Kerer and Kirda 2001). At first sight it may seem that formlets require logic and layout to be intertwined. We argue that in fact formlets *encourage* separation of logic from layout.

For modularity we allow the expressions appearing in formlet bindings and yields clauses to be arbitrary well-typed Links expressions. However, in practice, as exemplified by the date example, the formlet bindings typically refer only to names of other formlets and the yields clause is typically only used for aggregation of values returned by sub-formlets.

In other words, the parts of formlets that look like logic are in fact merely the interface between the logic and the layout. Though it can be a good idea to separate logic from layout, it is certainly important to expose the interface between the two, whether one is working on the logic or the layout.

3. Semantics

A natural question to ask is whether formlets fit into a well-understood semantic framework. Clearly formlets involve side-effects, in the form of name generation and user interaction. Monads (Moggi 1989; Wadler 1995; Benton et al. 2002) provide a standard semantic tool for reasoning about side-effects. Briefly, a monad is given by a type constructor T together with operations:

$$\begin{array}{rcl} \mathit{return} & : & (\alpha) \to \mathrm{T}(\alpha) \\ \star & : & (\mathrm{T}(\alpha), (\alpha) \to \mathrm{T}(\beta)) \to \mathrm{T}(\beta) \end{array}$$

that satisfy the following laws:

$$\begin{aligned} \mathit{return}(u) \star f &= f(u) \\ u \star \mathit{return} &= u \\ (u \star f) \star g &= u \star (\mathsf{fun}(x)\{f(x) \star g\}) \end{aligned}$$

It is not difficult to see that there is no monad corresponding to the formlet type. Intuitively, the problem is that a *bind* operation ($\star$) for the formlet type would have to read some of the input submitted by the user before the form had been rendered, which is clearly impossible. Idioms are a generalisation of monads that *are* suitable for modelling formlets. In fact, the formlet idiom is the composition of three primitive idioms.

An idiom (McBride and Paterson 2007) is simply a type constructor together with operations pure and $\otimes$, pronounced "apply", obeying certain laws. These operations permit injecting values into the idiom as well as general applicative computations—but the idiom gives a special meaning to the notion of application. On top of the general applicative structure given by pure and apply, an idiom will also typically come with operations for constructing impure (or effectful) values in that particular idiom.

(Arrows (Hughes 2000) can also be used; we could simulate the formlet idiom—or any other idiom for that matter—using arrows. We choose not to as we do not require the extra power provided by arrows.)

Formally, an idiom is a type constructor I together with operations:

$$\begin{array}{rcl} \mathit{pure} & : & (\alpha) \to \mathrm{I}(\alpha) \\ \otimes & : & (\mathrm{I}((\alpha) \to \beta), \mathrm{I}(\alpha)) \to \mathrm{I}(\beta) \end{array}$$

that satisfy the following laws:

$$\begin{aligned} \mathit{pure}(\mathit{id}) \otimes u &= u \\ \mathit{pure}(\circ) \otimes u \otimes v \otimes w &= u \otimes (v \otimes w) \\ \mathit{pure}(f) \otimes \mathit{pure}(x) &= \mathit{pure}(f(x)) \\ u \otimes \mathit{pure}(x) &= \mathit{pure}(\mathsf{fun}(f)\{f(x)\}) \otimes u \end{aligned}$$

where id is the identity function and $\circ$ is function composition.

We note that every monad is an idiom.

$$\begin{aligned} \mathrm{I} &= \mathrm{T} \\ \mathit{pure} &= \mathit{return} \\ f \otimes a &= f \star \mathsf{fun}(f)\{a \star \mathsf{fun}(a)\{\mathit{return}(f(a))\}\} \end{aligned}$$

Indeed every monad generates two idioms, as we can swap the order of $f \star \mathsf{fun}(f)$ and $a \star \mathsf{fun}(a)$ on the right-hand-side of the last equation.

Like standard function application, idiom application is left-associative. The idiom laws guarantee that pure computations can be reordered. In particular, an effectful computation cannot depend on the result of a pure computation, and any expression built from pure and $\otimes$ can be rewritten in the canonical form:

$$\mathit{pure}(f) \otimes u_1 \otimes \cdots \otimes u_k$$

where f is the pure part of the computation and $u_1 \ldots u_k$ are the effectful parts of the computation. This canonical form captures the essence of idioms as a tool for modelling computation. The intuition is that an idiomatic computation consists of a series of side-effecting computations, each of which returns a value. As with monads, the order in which computations are performed is significant, but unlike monads subsequent computations cannot depend on the values returned by prior computations. The final return value is obtained by aggregating the values returned by each of the side-effecting computations using a pure function.

3.1 XML

Before giving some concrete examples of idioms, we make a small digression to outline how XML is implemented in Links (Figure 4). (A Links declaration of the form `typename T = A` binds a type alias T to the type A. Type aliases can also have type parameters. The type `[A]` denotes the type of lists whose elements have type A.) In addition to the basic operations on lists, we need to be able to create text and element nodes. The operation $\mathit{text}^{\mathrm{x}} : (\mathit{String}) \to \mathit{Xml}$ converts a string to a text node, and the operation $\mathit{tag}^{\mathrm{x}} : (\mathit{Tag}, \mathit{Attributes}, \mathit{Xml}) \to \mathit{Xml}$ takes a three arguments: t, as and x, and returns an element node with tag t, attributes as and body x.

The Links type Xml captures the general notion of XML documents; in this paper we use the type only for HTML, and thus we will refer to HTML values although they have type Xml. Links makes no attempt to statically validate HTML, but it does make certain well-formedness guarantees: for instance, there is no way to create HTML with mismatched start and end tags.

3.2 The basic idioms

Throughout the remainder of the paper we make liberal use of superscripts to distinguish related operations on different data structures (usually idioms). Of course, Links source code does not support superscripts; they should be translated as capital letter suffixes.

```
typename XmlItem;
typename Tag = String;
typename Attributes = [(String, String)];
typename Xml = [XmlItem];

sig text^x : (String) -> Xml
sig tag^x : (Tag, Attributes, Xml) -> Xml
```

Figure 4. XML

```
typename I^n(α) = (Gen) -> (α, Gen);

sig pure^n : (α) -> I^n(α)
sig ⊗^n : (I^n(α -> β), I^n(α)) -> I^n(β)
sig nextName^n : I^n(String)
sig run^n : I^n(α) -> α

fun pure^n(v) {
  fun(gen) { (v, gen) }
}
op f ⊗^n a {
  fun (gen) {
    var (v, gen) = f(gen);
    var (w, gen) = a(gen);
    (v(w), gen)
  }
}
typename Gen = Int;
fun nextName^n(gen) {
  ("input_" ++ intToString(gen), gen + 1)
}
fun run^n(c) {
  var (v, _) = c(0);
  v
}
```

Figure 5. The name-generation idiom

In a language that supported some form of overloading, such as type classes, we might dispense with many of the superscripts.

Each new idiom we introduce will be presented in its own figure and should be read as an abstract type supporting a fixed set of operations. (Links does not yet support abstract types, but will do in a future version.)

Recall that idioms always compose. The formlet idiom is the composition of three basic idioms. Now we can show the idioms that comprise formlets. The *name-generation idiom* (Figure 5) threads a source of names through all of its computations; this permits the effect of fresh name generation. (The declaration op $p_1 \odot p_2$ {e} defines a binary infix operator $\odot$ with arguments bound by the patterns p_1 and p_2. It has type $(A_1, A_2) \to B$ where A_1 and A_2 are the types of its arguments and B is the return type.) The *accumulation idiom over the monoid of XML forests* (Figure 6) carries an XML value alongside its computations. The idiom's application operation concatenates the two computations' XML values. The additional operations ($text^a$, xml^a and tag^a) provide other ways of manipulating the XML. Note in particular that $text^a$ and tag^a lift the corresponding operations on XML into the idiom. The *environment idiom* (Figure 7) passes some environment (e.g. an association list) through all its computations; the available effect is reading from the environment.

In fact, each of these primitive idioms is simply a standard monad interpreted as an idiom. The reason for interpreting them as idioms rather than monads is that idioms always compose, whereas monads do not, and in particular these monads do not compose.

```
typename I^a(α) = (Xml, α);

sig pure^a : (α) -> I^a(α)
sig ⊗^a : (I^a(α -> β), I^a(α)) -> I^a(β)
sig text^a : (String) -> I^a( () )
sig xml^a : (Xml) -> I^a( () )
sig tag^a : (Tag, Attributes, I^a(α)) -> I^a(α)
sig plug^a : ((Xml) -> Xml, I^a(α)) -> I^a(α)
sig run^a : I^a(α) -> (Xml, α)

fun pure^a(v) {
  ([], v)
}
op (x, f) ⊗^a (y, a) {
  (x ++ y, f(a))
}
fun plug^a(k, (x, v)) {
  (k(x), v)
}
fun xml^a(e) {
  plug^a(fun (_) {e}, pure^a( () ))
}
fun text^a(s) {
  xml^a(text^x(s))
}
fun tag^a(t, as, v) {
  plug^a(fun (x) {(tag^x(t, as, x))}, v)
}
var run^a = id;
```

Figure 6. The accumulation idiom over the monoid of XML forests

```
typename Env = [(String, String)];
typename I^e(α) = (Env) -> α;

sig pure^e : (α) -> I^e(α)
sig ⊗^e : (I^e(α -> β), I^e(α)) -> I^e(β)
sig lookup^e : (String) -> I^e(String)
sig run^e : I^e -> ((Env) -> α)

fun pure^e(v) {
  fun (env) {v}
}
op f ⊗^e a {
  fun (env) {f(env)(a(env))}
}
fun lookup^e(n) {
  fun (env) {
    switch (env) {
      case [] -> error("Not found " ++ n)
      case (m,v)::env -> if (n == m) v
                         else lookup^e(n)(env)
    }
  }
}
var run^e = id;
```

Figure 7. The environment idiom

Why do the monads fail to compose? The failure to compose the monads that induce these idioms arises at the first step when we try to compose the accumulation monad with the environment monad. The difficulty is that in order to define $u \star f$ we need the return value of u which we can only extract once we have the environment, but the environment is not in scope when we need the accumulated monoid, so we cannot extract the accumulation

resulting from applying f to the return value of u. We end up trying to write something like:

```
(x, u) * f = (x ++ y,
              fun (env) {
                var (y, v) = f(u(env));
                v(env)
              })
```

which is clearly not well-scoped. This is exactly the issue mentioned at the beginning of the section—there cannot be a formlet monad because it would necessitate being able to read some of the input before rendering the whole of the form. It seems that attempting to precompose any non-trivial monad with the environment monad will lead to similar problems.

Any two idioms can be composed, producing an idiom. The composition of two idiom triples is defined pointwise. Given idioms I and J with associated operations $pure^{\mathrm{I}}$, $\otimes^{\mathrm{I}}$ and $pure^{\mathrm{J}}$, $\otimes^{\mathrm{J}}$ (respectively), we obtain the idiom I composed with J as I$\circ$J where

```
var pure^{I∘J} = pure^I ∘ pure^J;
op f ⊗^{I∘J} a {
  pure^I(curry((⊗^J))) ⊗^I f ⊗^I a
}
```

(The standard library function *curry* converts a binary function to a curried function.) We can *lift* any idiomatic computation of type $\mathrm{J}(A)$ to an idiomatic computation of type $\mathrm{I}(\mathrm{J}(A))$, *refine* any idiomatic computation of type $\mathrm{I}(A)$ to an idiomatic computation of type $\mathrm{I}(\mathrm{J}(A))$, and *map* any function of type $\mathrm{J}(A) \to \mathrm{J}(B)$ to a function of type $\mathrm{I}(\mathrm{J}(A)) \to \mathrm{I}(\mathrm{J}(B))$.

```
sig lift^{I,J} : (J(α)) → I(J(α))
var lift^{I,J} = pure^I;

sig refine^{I,J} : (I(α)) → I(J(α))
fun refine^{I,J}(v) { pure^I(pure^J) ⊗^I v }

sig map^{I,J}
  : ((J(α)) → J(β)) → (I(J(α))) → I(J(β))
fun map^{I,J}(f)(c) { pure^I(f) ⊗^I c }
```

Using a combination of lifting, refinement and mapping, it is straightforward to transform all of the side-effecting operations from the basic idioms into the formlet idiom.

3.3 The formlet idiom

We now give an implementation of formlets as the composition of the above idioms (Figure 8).

$$Formlet = \mathrm{I}^{\mathrm{n}} \circ \mathrm{I}^{\mathrm{a}} \circ \mathrm{I}^{\mathrm{e}}$$

The actual implementation included with Links differs slightly in that we flatten the nested pairs into triples and inline many of the occurrences of *pure* and $\otimes$ for performance. The XML manipulation operations and *run* operation are lifted into the formlet idiom. It would be inappropriate to lift the *nextName* and *lookup* operations into the formlet idiom as lower-level access to the generated names is needed in order to ensure accordance. Instead of *nextName* and *lookup* we provide a library of formlet operations corresponding to HTML input elements, each of which generates one or more names. These include *input*, *textarea* and *button* (which give rise to the eponymous HTML elements), as well as *choice* (corresponding to HTML `option/select` elements), *submit* (which produces HTML for a submit button) and others. Some additional library operations that are used in the examples in this paper are defined in Figure 9.

3.4 Pickling continuations

Links maintains session state by pickling continuations (Cooper et al. 2007). In particular, the continuation to be invoked by a form is stored as a pickled continuation in a hidden field in the form.

```
typename Formlet(α) = I^n(I^a(I^e(α)));

sig pure : (α) → I^f(α)
sig ⊗ : (I^f(α → β), I^f(α)) → I^f(β)
sig xml : (Xml) → Formlet( () )
sig text : (String) → Formlet( () )
sig tag : (Tag, Attributes, Formlet(α)) → Formlet(α)
sig run : Formlet(α) → I^a(I^e(α))
sig input : Formlet(String)

var pure = pure^n(pure^a(pure^e))
op f ⊗ a {
  pure^n(fun (f)(a) {
           pure^a(curry((⊗^e))) ⊗^a f ⊗^a a
         }) ⊗^n f ⊗^n a
}
fun xml(x) {
  pure^n(pure^a(pure^e) ⊗^a xml^a(x))
}
fun text(s) {
  pure^n(pure^a(pure^e) ⊗^a text^a(s))
}
fun tag(t, as, f) {
  pure^n(fun (v) {tag^a(t, as, v)}) ⊗^n f
}
var run = run^n;
var input =
  pure^n(fun (name) {
           tag^a("input",
                 [("name", name)],
                 pure^a(lookup^e(name)))
         }) ⊗^n nextName^n
```

Figure 8. The formlet idiom

```
sig inputInt : Formlet(Int)
sig submit : (String) → Xml

var inputInt =
  formlet
    <#>{input → s}</#>
  yields
    stringToInt(s);
fun submit(text^x) {
  <button type="submit">{stringToXml(text^x)}</button>
}
```

Figure 9. Formlet library operations

In order to implement this functionality for formlets we add some extra operations to the environment idiom and the accumulation idiom (Figure 10). A handler of type α is a function from α to $Page$. A continuation is a handler whose input comes from the CGI environment—that is, a handler of type $Handler(Env)$. The *makeCont* function composes a handler with a collector to give a continuation. The *pickleCont* function is a built-in Links function for pickling a continuation as a string. The expression $makeForm(e, cont)$ returns a form whose action is to invoke *cont* and whose body is e. When a form is submitted, the special name `_k` is recognised by the Links run-time as containing the pickled continuation. The continuation is unpickled, and invoked with the current CGI environment.

```
typename Handler(α) = (α) → Page;
typename Cont = Handler(Env);

sig makeCont : (Handler(α)) → (I^e(α)) → I^e(Page)
sig pickleCont : (Cont) → String
sig makeForm : (Xml, Cont) → Xml

fun makeCont(h)(c) {
  pure^e(h) ⊗^e c
}
fun makeForm(contents, cont) {
  var (x, _) =
    tag^a("form",
        [("enctype",
          "application/x-www-form-urlencoded"),
         ("action", "#"),
         ("method", "POST")],
        tag^a("input",
            [("type", "hidden"),
             ("name", "_k"),
             ("value", pickleCont(cont))],
            contents)));
  x
}
```

Figure 10. Pickling continuations

```
typename Page = Xml;
sig form^p : (Formlet(α), Handler(α)) → Page
sig render^p : (Page) → Xml
sig xmlPage : (Xml) → Page
sig formletPage : (Formlet(α), Handler(α)) → Page

fun form^p(f, h) {
  var (x, h) =
    run^a(pure^a(makeCont(h)) ⊗^a run(f));
  makeForm(x, run^e(h));
}
var render^p = id;
var xmlPage = id;
fun formletPage(f, h)
  tag^x("html", [],
   tag^x("body", [],
    form^p(f, h)))
```

Figure 11. Pages

3.5 Pages

Recall that the top-level value returned by a Links program must have type $Page$. For the basic formlet idiom without validation, we can simply define the $Page$ type to be Xml (Figure 11). (When we add validation in Section 5, pages become more involved.)

The only interesting function here is $form^{\mathrm{p}}$. The expression $form^{\mathrm{p}}(f)$ first runs f, then makes a continuation by composing the handler with the formlet's collector and finally invokes the function $form^{\mathrm{a}}$ to render the form as HTML.

The most interesting operation on pages is $form^{\mathrm{p}}$. This function takes a formlet and a handler and generates a rendering of the formlet with the handler attached. The function $render^{\mathrm{p}}$ is used internally by the Links run-time, which needs a way of converting pages back to XML for display by a web browser. The $xmlPage$ function allows raw XML to be lifted to a page. The $formletPage$ function invokes $form^{\mathrm{p}}$ and wraps the resulting XML in `html` and `body` tags.

The reader may be worried that these operations seem rather limited. For instance, they do not allow multiple formlets on a page, and they do not allow a header to appear on a page containing a formlet. These features are supported by the version of formlets discussed in Section 5 (where we abandon $xmlPage$ and $formletPage$ in favour of more general functions). We have omitted them here in an attempt to make the initial presentation simpler.

Terms

q^{x}	XML
formlet q^{f} yields e	formlet

XML quasiquotes

$$
\begin{array}{rcll}
n & ::= & s \mid \texttt{\{}e\texttt{\}} & \\
 & \mid & \texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>} & \text{node} \\
q^{\mathrm{x}} & ::= & \texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>} & \\
 & \mid & \texttt{<\#>}n_1 \ldots n_k\texttt{</\#>} & \text{quasiquote}
\end{array}
$$

Formlet quasiquotes

$$
\begin{array}{rcll}
n & ::= & s \mid \texttt{\{}e\texttt{\}} \mid \texttt{\{}f \to p\texttt{\}} & \\
 & \mid & \texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>} & \text{node} \\
q^{\mathrm{f}} & ::= & \texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>} & \\
 & \mid & \texttt{<\#>}n_1 \ldots n_k\texttt{</\#>} & \text{quasiquote}
\end{array}
$$

Meta variables

e	expression	s	string
p	pattern	t	tag
f	formlet	as	attribute list

Figure 12. Quasiquote syntax

$$
\begin{array}{rcl}
(q^{\mathrm{x}})^{\circ} & = & [\![q]\!]^{\mathrm{x}} \\
(\text{formlet } q \text{ yields } e)^{\circ} & = & pure\,(\text{fun } (q^{\dagger})\texttt{\{}e^{\circ}\texttt{\}}) \otimes [\![q]\!]^{\mathrm{f}}
\end{array}
$$

$$
\begin{array}{rcl}
[\![s]\!]^{\mathrm{x}} & = & text^{\mathrm{x}}(s) \\
[\![\texttt{\{}e\texttt{\}}]\!]^{\mathrm{x}} & = & e^{\circ} \\
[\![\texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>}]\!]^{\mathrm{x}} & = & tag^{\mathrm{x}}(t,\ as,\ [\![\texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}]\!]^{\mathrm{x}}) \\
[\![\texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}]\!]^{\mathrm{x}} & = & [\![n_1]\!]^{\mathrm{x}} \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} [\![n_k]\!]^{\mathrm{x}}
\end{array}
$$

$$
\begin{array}{rcl}
[\![s]\!]^{\mathrm{f}} & = & text(s) \\
[\![\texttt{\{}e\texttt{\}}]\!]^{\mathrm{f}} & = & xml(e^{\circ}) \\
[\![\texttt{\{}f \to p\texttt{\}}]\!]^{\mathrm{f}} & = & f^{\circ} \\
[\![\texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>}]\!]^{\mathrm{f}} & = & tag(t,\ as,\ [\![\texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}]\!]^{\mathrm{f}}) \\
[\![\texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}]\!]^{\mathrm{f}} & = & pure\,(\text{fun } (n_1^{\dagger}) \ldots (n_k^{\dagger})\ \texttt{\{}\ (n_1^{\dagger}, \ldots, n_k^{\dagger})\ \texttt{\}}) \otimes [\![n_1]\!]^{\mathrm{f}} \cdots \otimes [\![n_k]\!]^{\mathrm{f}}
\end{array}
$$

$$
\begin{array}{rcl}
s^{\dagger} & = & \_ \\
\texttt{\{}e\texttt{\}}^{\dagger} & = & \_ \\
\texttt{\{}f \to p\texttt{\}}^{\dagger} & = & p \\
\texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>}^{\dagger} & = & n_1^{\dagger} \ldots n_k^{\dagger} \\
\texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}^{\dagger} & = & n_1^{\dagger} \ldots n_k^{\dagger}
\end{array}
$$

Figure 13. Desugaring XML and formlets

4. Desugaring

Having shown how formlets look to the programmer in Links, and described the idiom structure of formlets, we now show how to compile the syntactic sugar (Figure 12) for XML, formlets and pages.

The desugaring transformation $(\cdot)^\circ$ is given in Figure 13. It is a homomorphism except on XML quasiquotes and formlet expressions. The auxiliary operations $[\![\cdot]\!]^{\mathrm{x}}$, $(\cdot)^\dagger$ and $[\![\cdot]\!]^{\mathrm{f}}$ are defined on quasiquotes and on nodes. Let r range over quasiquotes and nodes. The operation $r^\dagger$ returns a pattern aggregating the sub-patterns of r. The operation $[\![r]\!]^{\mathrm{f}}$ returns a formlet aggregating the sub-formlets of r.

As well as pure and apply, the operations $text$, xml and tag are used for lifting raw XML into formlets.

5. Validation

Up to this point we have been proceeding on the impractical assumption that text entered into forms is always valid. For example, we have not made any provision for the case where the user enters non-digit characters into a field which is interpreted as an integer. We wish to provide the following behaviour: when an invalid form is submitted, the page on which it appeared should be redisplayed, together with error messages describing the problems with the form.

5.1 The validating formlet idiom

In order to keep track of error messages we modify the *Formlet* type, changing the return type of the collector component from α to $(Xml, Maybe(\alpha))$.

In Section 3 we saw that formlets arise as the composition of three idioms.

$$Formlet = \mathrm{I}^{\mathrm{n}} \circ \mathrm{I}^{\mathrm{a}} \circ \mathrm{I}^{\mathrm{e}}$$

In order to define a validating formlet idiom we need to introduce an idiom for modelling errors. The error idiom I_o (Figure 14) is derived from the standard error monad.

```
typename Maybe(α) = [|Just:α|Nothing|];
typename Io(α) = Maybe(α);
```

(Links supports variant types. The type $[|C_1 : A_1 \ldots C_k : A_k|]$ is a variant with constructors $C_1, \ldots, C_k$ of types $A_1, \ldots, A_k$.)

The validating formlet idiom (Figure 16) is obtained by the composition of the non-validating formlet idiom with idioms for XML accumulation and errors.

$$Formlet = \mathrm{I}^{\mathrm{n}} \circ \mathrm{I}^{\mathrm{a}} \circ \mathrm{I}^{\mathrm{e}} \circ \mathrm{I}^{\mathrm{a}} \circ \mathrm{I}^{\mathrm{o}}$$

```
typename Formlet(α) = In(Ia(Ie(Ia(Io(α)))));
```

When passed an environment, a collector of type $(Env) \to (Xml, Maybe(A))$ attempts to extract a value v of type A, returning $Just(v)$ if the extraction succeeds, or $Nothing$ if it fails. The first component of the return value, the *error rendering*, is the HTML that should be displayed in an error situation, whether that situation arose due to this or some other collector. If no validator is attached to the formlet then the extraction succeeds, using the original HTML as the error rendering. Given these adjustments we can add a validating operation, *satisfies*, on formlets (Figures 15 and 16).

```
typename Validator(α) =
  ((α) → Bool, (α) → (Xml) → Xml);
sig satisfies
  : (Formlet(α), Validator(α)) → Formlet(α)
```

A *validator* is a pair of a predicate and an error reporting function. The error reporting function takes a value and then transforms some existing XML to add an error message. An auxiliary function, *err*, constructs a validator from a predicate and a function that builds an error message as a string; if the predicate fails then the message function will be passed the failing value.

```
sig err
  : ((α) → Bool) → ((α) → String) → Validator(α)
```

The *satisfies* operation is defined by lifting the function *check*, which validates computations in the accumulation idiom precomposed with the error idiom, into the validating formlet idiom.

```
sig check
  : (Ia(Io(α))) → (Validator(α)) → Ia(Io(α))
```

The function *check* is divided into three parts. The first part checks whether: there is a value and the predicate is satsified, there is a value but the predicate is not satisfied, or there is no value. The second part changes the maybe value to *Nothing* if it fails to satisfy the predicate. The third part inserts an error message if there is a value that does not satisfy the predicate. Besides these new operations, we must adjust the basic formlet operations to support validation (Figure 16).

The collector of the *pure* function now returns an empty error rendering and a value wrapped in *Just*. The $\otimes$ operator concatenates the error renderers; collection now succeeds only if it succeeds for both operands. The *input* formlet includes HTML for both the regular renderer and the error renderer. If collection fails then the submitted value is available to the error renderer, so we can repopulate the field by supplying a value for the *value* attribute.

Using *satisfies* and *err* we can add error-checking to a formlet. For example, we can now improve the definition of *inputInt* so that a suitable message is displayed if parsing the string fails (Figure 17).

(Links supports matching strings against regular expressions. The expression s `~` r evaluates to true if the string s matches the regular expression r.)

Validation can be added to any formlet value regardless of whether there is validation code attached to the value already. For example, starting with the *inputInt* formlet we can construct a formlet that accepts only even numbers:

```
fun evenError(i) { intToString(i) ++ " is not even!" }
var inputEven =
  inputInt `satisfies` (even `err` evenError);
```

(The standard library function $even : (Int) \to Bool$ returns true if its argument is an integer.) If the user enters a non-integer into an *inputEven* field then the error message generated by *intError* will be displayed. If integer parsing succeeds but the parity check fails then the message generated by *evenError* will be displayed. In general, when additional validation is applied to a formlet f which already includes validation code, the validators are run from innermost outwards; only the first failing validator is used to label f with an error message. However, errors may also be displayed from other formlets, which are not descendents of f. For instance, a date formlet constructed from two *input* formlets for the day and the month may display errors arising from errors on both the day and the month.

We might similarly improve the formlets from Section 2 by adding validation that tests that the dates are within range, or that the departure date is no earlier than the arrival date. We have only shown an error message combinator (*err*) that takes a message and displays it in a standard place. The datatype permits user-defined combinators that indicate the error in other ways.

5.2 Pages with validation support

In order to support multiple forms on a page we introduce some additional syntactic sugar for pages (Figures 19 and 20). We introduce the `page` q construct. Here q is a *page quasiquote*, an XML quasiquote augmented with a facility to associate formlets with handlers. A *formlet/handler association* is written $\{f \Rightarrow h\}$. The value of `page` q is a *page fragment*. Intuitively, a page fragment is an HTML value where each formlet/handler association $\{f \Rightarrow h\}$ has been replaced by an HTML form. The body of that form is the rendering of the formlet f, and the action of the

```
typename Maybe(α) = [|Just:α|Nothing|];
typename I°(α) = Maybe(α);

sig pure° : (α) → I°(α)
sig ⊗° : (I°(α → β), I°(α)) → I°(β)
sig fail° : I°(α)
sig run° : I°(α) → Maybe(α)

fun pure°(v) {
  Just(v)
}
op f ⊗° a {
  switch (f, a) {
    case (Just(f), Just(a)) → Just(f(a))
    case _ → Nothing
  }
}
var fail° = Nothing;
var run° = id;
```

Figure 14. The error idiom

form applies the handler h to the result of invoking f's collector. In addition to formlet/handler associations, page quasiquotes also support *page antiquotes* `{|g|}`, which allow page fragments to be composed from smaller page fragments. Formlet/handler associations generalise the *formletPage* function and XML antiquotes inside page expressions generalise the *xmlPage* function.

```
fun formletPage(f, h) {
  page
    <html>
     <body>
     {f ⇒ h}
     </body>
    </html>
}
```

```
fun xmlPage(x) {
  page
    <#>{x}</#>
}
```

The example in Figure 18 creates a page containing two forms. As explained earlier, we want a formlet that fails validation to be presented a second time to the user along with error messages. Further, we want this formlet to be presented in its original context. If the user enters non-numeric text into the *inputEven* form then the implementation should re-display the entire page, with an error message beside the offending field. If the user subsequently submits invalid input in the *date* form then the entire page should be re-displayed with error messages accompanying both forms: the state of the *inputEven* form, including error messages, should be preserved.

In order to implement such behaviour we refine our notion of pages. Validating *page fragments* represent composable web-page fragments containing validated forms. A validating page fragment consists of a k-holed XML context, k formlets and k handlers. To support concatenation of contexts we must also store the number of holes in the context. The formlets in the list may have different types; we might elude the typing problem by hiding the types behind an existential.

```
typename Context = (Int, ([Xml]) → Xml);
typename Page =
  (Context, [∃α.(Formlet(α), Handler(α))]);
```

Links does not support existential types, so instead we store the code that will be used to eliminate the formlet/handler pairs. We hide the ∃-bound type using a closure, in essence the inverse of Minamide et al's *typed closure conversion* (Minamide et al. 1996), which introduces existentials for encoding heterogeneous environments.

```
typename Validator(α) =
  ((α) → Bool, (α) → (Xml) → Xml);

sig err
  : ((α) → Bool, (α) → String) → Validator(α)
sig check
  : (Iª(I°(α)), Validator(α)) → Iª(I°(α))

fun err(p,error) {
  (p,
   fun (v)(x) {
     <#>
      <span class="errorinput">{x}</span>
      <span class="error">{textˣ(error(v))}</span>
     </#>
   })
}
typename Result(α) = [|Pass:α|Fail:α|Dead|];
fun check(a, (p, error)) {
  var result =
    pureª(fun (o) {
      switch (run°(pure°(fun (v) {
                          if p(v) Pass(v)
                          else Fail(v)
                        }) ⊗° o)) {
        case Just(v) → v
        case Nothing → Dead
      }
    }) ⊗ª a;

  var w =
    pureª(fun (r) {
      switch (r) {
        case Pass(v) → pure°(v)
        case _ → fail°
      }
    }) ⊗ª result;

  switch (runª(result)) {
    case (_, Fail(v)) → plugª(error(v), w)
    case _ → w
  }
}
```

Figure 15. Validation operations

Another interesting aspect of the typing arises from the validation loop. Every form on the page has a handler, but every handler must be able to regenerate all the forms on a page. In order to tie this recursive knot we make use of a recursive type (Figure 21).

```
typename RecForms = [μα.([α]) → Xml];
```

The HTML forms are represented as a list of *recursive forms* of type *RecForms*. A recursive form is an HTML form that is parameterised over the list of all other recursive forms on a page. This allows every form to be updated when one of them changes (due to a validation error).

In order to avoid an existential type, we need to examine how the formlets and handlers are going to be consumed. The key component is the modified *makeCont* function (Figure 21).

```
sig makeCont
  : (Handler(α), Context, RecForms, Int) →
    (Iᵉ(Iª(I°(α)))) → Iᵉ(Page)
```

The non-validating version of *makeCont* (Figure 10) takes only two arguments: a handler h and a collector c. The validating version is augmented with three extra arguments for keeping track of error state: a context k, a recursive form list zs, and an index i into zs identifying the current form. The collector is run

```
typename Formlet(α) = I^n(I^a(I^e(I^a(I^o(α)))));

sig pure : (α) → I^f(α)
sig ⊗ : (I^f(α → β), I^f(α)) → I^f(β)
sig text : (String) → Formlet( () )
sig xml : (Xml) → Formlet( () )
sig tag : (Tag, Attributes, Formlet(α)) → Formlet(α)
sig run : Formlet(α) → I^a(I^e(I^a(I^o(α))))
sig input : Formlet(String)
sig satisfies
  : (Formlet(α), Validator(α)) → Formlet(α)

var pure = pure^n(pure^a(pure^e(pure^a(pure^o))))
op f ⊗ a {
  pure^n(fun (f)(a) {
    pure^a(fun (f)(a) {
      pure^e(fun (f)(a) {
        pure^a(curry((⊗^o))) ⊗^a f ⊗^a a
      }) ⊗^e f ⊗^e a
    }) ⊗^a f ⊗^a a
  }) ⊗^n f ⊗^n a
}
fun text(x) {
  var v = text^a(s);
  pure^n(pure^a(pure^e(pure^a(pure^o) ⊗^a v)) ⊗^a v)
}
fun xml(x) {
  var v = xml^a(x);
  pure^n(pure^a(pure^e(pure^a(pure^o) ⊗^a v)) ⊗^a v)
}
fun tag(t, as, f) {
  fun wrap(v) {tag^a(t, as, v)}
  pure^n(fun (v) {
    pure^a(pure^e(wrap)) ⊗^a wrap(v)
  }) ⊗^n f
}
var run = run^n;
var input =
  pure^n(fun (name) {
          fun wrap(v) {
            tag^a("input", [("name", name)], v)
          }
          pure^a(pure^e(wrap)) ⊗^a
          wrap(pure^a(pure^e(pure^a(pure^o)) ⊗^a
                      lookup^e(name)))
        }) ⊗^n nextName^n
fun satisfies(f, validate) {
  pure^n(fun (u) {
    pure^a(fun (v) {
      pure^e(fun (w) {
        check(w, validate)
      }) ⊗^e v
    }) ⊗^a u
  }) ⊗^n f
}
```

Figure 16. The validating formlet idiom

on the environment, returning some HTML x and an optional return value v. If validation succeeds then the value is simply passed to h. If validation fails then the HTML for the i-th form is updated and the page is re-rendered. The HTML for each form is generated by applying each z in zs to the entire list zs. This is where the recursive knot is tied. (The standard library function $map : ((\alpha) \to \beta, [\alpha]) \to [\beta]$ maps a function over a list. The standard library function $substAt : ([\alpha], Int, \alpha)$ when applied to

```
fun isInt(s) { s ~ /^-?[0-9]+$/ }
fun intError(s) { s ++ " is not an integer!" }

sig inputInt : Formlet(Int)
var inputInt =
  formlet
    <#>{input `satisfies` (isInt `err` intError) → s}</#>
  yields
    stringToInt(s);
```

Figure 17. Validating formlet library operations

```
fun displayEven (e) {
 <html>
  <body>
   An even number: {intToString(e)}
  </body>
 </html>
}

fun displayDate(d) {
 <html>
  <body>
   A date: {dateToString(d)}
  </body>
 </html>
}

page
 <html>
  <body>
   Enter an even number: {inputEven ⇒ displayEven}
   Alternatively, enter a date: {date ⇒ displayDate}
  </body>
 </html>
```

Figure 18. Two forms on a page

xs, i and x returns xs with the i-th element replaced with the value x.)

We can now give an implementation of page fragments that does not depend on an existential type (Figure 22).

```
typename CheckedFormBuilder =
  (Context, RecForms, Int) → Xml;
typename Page = (Context, [CheckedFormBuilder]);
```

A checked form builder is a function that takes a multi-holed context k, a recursive form list zs and an index i into this list, and returns the HTML rendering for the form $zs(i)$. Note that the context k and list zs are both necessary in order to be able to report errors.

Page fragments form a monoid structure (Figure 22). The unit is $empty^p : Page$ and the multiplication is $\oplus : (Page, Page) \to Page$. In addition we need to lift the usual XML manipulation operations into the $Page$ type. Most importantly we need a function for attaching a handler to a page and a way of rendering pages as XML. The definitions of the monoid and XML operations are quite straightforward. The function $form^p$ invokes $makeCheckedFormBuilder$ to construct a checked form builder from a formlet and a handler. Notice that the body of $makeCheckedFormBuilder$ is essentially the same as body of the non-validating $form^p$ function of Figure 11. The key difference is that the call to $makeCont$ takes the extra arguments for identifying the context of the form on the page. The function $render^p$ uses the checked form builders to generate a recursive form list. This list is then plugged into the context, tying the recursive knot in the same way as in the $makeCont$ function. (The standard library

Terms

	page q^{p}	page fragment

Page quasiquotes

n	$::=$	s	
		$\mid$ {e} $\mid$ {$f \Rightarrow e$} $\mid$ {| g |}	
		$\mid$ <t as>$n_1 \ldots n_k$</t>	node
q^{p}	$::=$	<t as>$n_1 \ldots n_k$</t>	
		$\mid$ <#>$n_1 \ldots n_k$</#>	quasiquote

Meta variables

h	handler	f	formlet
		g	page fragment

Figure 19. Page fragment syntax

$$(\mathsf{page}\ q)^{\circ} = [\![q]\!]^{\mathrm{p}}$$

$$\begin{aligned}
[\![s]\!]^{\mathrm{p}} &= \mathit{text}^{\mathrm{p}}(s) \\
[\![\{e\}]\!]^{\mathrm{p}} &= \mathit{xml}^{\mathrm{p}}(e^{\circ}) \\
[\![\{f \Rightarrow h\}]\!]^{\mathrm{p}} &= \mathit{form}^{\mathrm{p}}(f^{\circ}, e^{\circ}) \\
[\![\{\mid g \mid\}]\!]^{\mathrm{p}} &= g^{\circ} \\
[\![\texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>}]\!]^{\mathrm{p}} &= \mathit{tag}^{\mathrm{p}}(t,\ as,\ [\![\texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}]\!]^{\mathrm{p}}) \\
[\![\texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}]\!]^{\mathrm{p}} &= [\![n_1]\!]^{\mathrm{p}} \oplus^{\mathrm{p}} \ldots \oplus^{\mathrm{p}} [\![n_k]\!]^{\mathrm{p}}
\end{aligned}$$

Figure 20. Desugaring page fragments

```
typename RecForms = [μα.([α]) → Xml];
sig makeCont
  : (Handler(α), Context, RecForms, Int) →
    (I^e(I^a(I^o(α)))) → I^e(Page)
fun makeCont(h, k, zs, i)(c) {
  var hf = makeCont(h, k, zs, i)(c);
  pure^e(
    fun (a) {
      var (x, v) = run^a(a);
      fold^o(h,
             fun () {
               fun z(zs) {
                 makeForm(x, run^e(hf))
               }
               var zs = substAt(zs, i, z);
               k(map(fun (z) {z(zs)}, zs))
             },
             v)
    }) ⊗^e c
}
```

Figure 21. Validating continuations

function $mapi$: $((\alpha, \mathit{Int}) \to \beta, [\alpha]) \to [\beta]$ maps a function f : $(\alpha, \mathit{Int}) \to \beta$ over a list. The second argument to f is the index of the element being mapped.)

6. Related work

JWIG JWIG (Christensen et al. 2003) is an extension of Java for building web services. It builds on ideas developed in MAWL (Atkins et al. 1999) and <bigwig> (Brabrand et al. 2002). JWIG allows HTML (including forms) to be composed using *templates*. Templates are first-class multi-holed HTML contexts with named holes. Both templates and simple values can be plugged into templates. Regular expressions are used to validate form input data at runtime. The field validation is performed both on the client and on the server. A flow analysis is used to statically check validity of generated HTML documents. The flow analysis requires that field names be constants, so it is not possible to abstract over form components.

```
typename CheckedFormBuilder =
  (Context, RecForms, Int) → Xml;
typename Page = (Context, [CheckedFormBuilder]);

sig empty^p : Page
sig text^p : (Xml) → Page
sig xml^p : (Xml) → Page
sig tag^p : (Tag, Attributes, Page) → Page
sig form^p : (Formlet(α), Handler(α)) → Page
sig render^p : (Page) → Xml

var empty^p = ((0, fun ([]) {[]}), []);
sig ⊕^p : (Page, Page) → Page
op ((i1, k1), ms1) ⊕^p ((i2, k2), ms2)) {
  ((i1 + i2,
    fun (xs) {
      k1(take(i1, xs)) ++ k2(drop(i1, xs))
    }),
   ms1 ++ ms2)
}
fun text^p(s) {
  xml^p(text^x(s))
}
fun xml^p(x) {
  ((0, fun ([]) {x}), fun (gen) {([], gen)})
}
fun tag^p(t, as, ((i, k), fs)) {
  ((i, fun (xs) {tag^x(t, as, k(xs))}), fs)
}
sig makeCheckedFormBuilder
  : (Formlet(α), Handler(α)) → CheckedFormBuilder
fun makeCheckedFormBuilder(f, h)(k, zs, i) {
  var (x, h) =
    run^a(pure^a(makeCont(h, k, zs, i)) ⊗^a run(f));
  makeForm(x, run^e(h))
}
fun form^p(f, h) {
  ((1,
    fun ([x]) {x}),
    [makeCheckedFormBuilder(f, h)])
}
fun render^p(((n, k), ms)) {
  var zs = mapi(fun (m, i)(zs) {m(k, zs, i)}, ms);
  k(map (fun (z) {z(zs)}, zs))
}
```

Figure 22. The validating page monoid

Scriptlets Scriptlets are built on top of SMLserver (Elsman et al. 2007), a webserver for serving web applications written in Standard ML. Elsman and Larsen (Elsman and Larsen 2004) implemented static typing for HTML on top of SMLserver. Their system uses phantom types to enforce validity of HTML, and SML functors called *scriptlets* for building statically checked forms. SML functors are not first class, which limits the scope for dynamically composing forms using scriptlets.

WASH The WASH/CGI Haskell library (Thiemann 2005) treats HTML forms in a well-typed manner, but does not support the same degree of abstraction as formlets.

The paradigm of WASH is monadic. The data produced by a form component is carried forward as values are carried forward

by a monad, and the HTML part of the component is accumulated as a monadic effect. Further, since handlers are attached to submit buttons (rather than to the entire form), a submit button is forced to appear below the fields that it depends on.

WASH supports using a user-defined type for an individual form field, and it supports aggregating data from multiple fields in a standard way, but it does not support aggregating multiple fields into an arbitrary user-defined type. Hence, the programmer cannot abstract over the HTML presentation of a component: the nature of its form fields is revealed in its type. For example, given a one-field component, a programmer cannot readily modify it to consist of two fields, without changing all the uses of the component.

iData The iData library (Plasmeijer and Achten 2006) takes a model-view-controller approach to editing program values using HTML forms. An *iData* is the fundamental abstraction for editing values in a web form. The iData library makes use of type-directed overloading to automatically derive editors for certain types.

As well as abstracting over forms, the iData library builds in a control flow mechanism which effectively forces the programmer to treat an entire program as a single web page consisting of a collection of interdependent iData. Whenever one of the elements is edited by the user, the form is submitted and then re-displayed to the user with any dependencies resolved. The iTasks library (Plasmeijer et al. 2007) builds on top of iData and addresses this issue by enabling or disabling iData according to the state of the program.

WUI The WUI (*Web User Interface*) library (Hanus 2006, 2007) implements form abstractions for the functional logic programming language Curry. Of the existing web form frameworks, the WUI library is the one that is closest to formlets in spirit. Indeed, WUIs are essentially values of type $(\alpha) \to \mathit{Formlet}(\alpha)$. The validation mechanism for WUIs is strikingly similar to the one we describe here for formlets. WUIs support validation on the client (Hanus 2007) as well as the server by compiling some of the validation code to JavaScript. The WUI library is not stateless; continuations are stored in a table in a persistent server-side process.

We briefly mention some less closely related work.

Hughes's CGI library (Hughes 2000) supports a form of stateless communication using arrows. It does not attempt to tackle the issue of building abstractions over forms.

Rather than presenting the programmer with HTML forms, the Google Web Toolkit (GWT) generates JavaScript from desktop Java applications, written against desktop windowing frameworks, and thus uses a very different paradigm for user interaction.

Ocsigen (Balat 2006) is a web programming extension for OCaml. It supports statically typed HTML, but does not support form abstractions.

Lift (Lift) is a web framework for the Scala language. Lift provides some support for form construction: for instance, it allows associating a function with a form field, which is applied to the field value upon submission. Such functions are associated only with a single concrete field and never with a constructed, abstract field as formlets allow. Also, there is no syntactic support for binding multiple field values within the same scope, thus the only way to collect values from multiple fields is through side-effects.

7. Conclusion

We have presented formlets, a form abstraction based on idioms. We have implemented formlets in Links and shown that they can be cleanly extended to support new features such as validation.

In our current implementation, form handlers always run on the server. Links, however, also supports code running on the client, so a natural extension would be to allow client-side form handling. A more challenging task is to extend formlets to respond to events other than form submission. The natural responses to client-side events are likely to favour manipulations of the DOM, whose API has a very imperative feel; how to integrate this with our largely functional approach is an open-ended question.

References

David L. Atkins, Thomas Ball, Glenn Bruns, and Kenneth C. Cox. Mawl: A domain-specific language for form-based services. *IEEE Trans. Software Eng.*, 25(3):334–346, 1999.

Vincent Balat. Ocsigen: typing web interaction with objective caml. In *ML*, pages 84–94, 2006.

Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *Applied Semantics: Advanced Lectures*, volume 2395 of *LNCS*, pages 42–122, 2002.

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The <bigwig> project. *ACM Trans. Internet Techn.*, 2(2):79–114, 2002.

Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level web service construction. *ACM Trans. Program. Lang. Syst.*, 25(6):814–875, 2003.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *FMCO 2006*, volume 4709 of *LNCS*, pages 266–296, 2007.

Martin Elsman and Ken Friis Larsen. Typing XHTML web applications in ML. In *PADL*, pages 224–238, 2004.

Martin Elsman, Niels Hallenberg, and Carsten Varming. *SMLserver—A Functional Approach to Web Publishing (Second Edition)*, April 2007. (174 pages). Available via `http://www.smlserver.org`.

Paul T. Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the web with high-level programming languages. In *ESOP*, pages 122–136, 2001.

GWT. Google Web Toolkit website, March 2008. `http://code.google.com/webtoolkit/`.

M. Hanus. Putting declarative programming into the web: Translating Curry to JavaScript. In *PPDP'07*, pages 155–166, 2007.

Michael Hanus. Type-oriented construction of web user interfaces. In *PPDP*, pages 27–38, 2006.

John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37 (1-3):67–111, 2000.

Clemens Kerer and Engin Kirda. Layout, content and logic separation in web engineering. In *Web Engineering, Software Engineering and Web Application Development*, volume 2016 of *LNCS*, pages 135–147, 2001.

Lift. Lift website, March 2008. `http://liftweb.net/`.

Conor McBride and Ross Paterson. Applicative programming with effects. *JFP*, 17(5), 2007.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *POPL '96*, pages 271–283, 1996.

Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23, 1989.

Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *ICDT '05*, January 2005.

PHP. PHP Hypertext Preprocessor, March 2008. `http://www.php.net/`.

Rinus Plasmeijer and Peter Achten. iData for the world wide web: Programming interconnected web forms. In *FLOPS*, pages 242–258, 2006.

Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. *SIGPLAN Not.*, 42(9):141–152, 2007.

Ruby on Rails. Ruby on Rails website, March 2008. `http://www.rubyonrails.org/`.

Peter Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. Inter. Tech.*, 5(1):1–46, 2005. ISSN 1533-5399. doi: http://doi.acm.org/10.1145/1052934.1052935.

Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. 1995.