An idiom’s guide to formlets

Citation for published version:

Document Version:
Early version, also known as pre-print

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
An idiom’s guide to formlets*

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop

The University of Edinburgh

Abstract. We present a novel approach to HTML form abstraction, which we call formlets. A formlet hides the underlying representation of a data type. For instance, a date formlet may allow a user to enter a date as a single text field, or separate fields for day, month, and year, or a combination of pulldown menus, or a custom JavaScript calendar widget; any consumer of the data from the form would see only the abstract data type Date. Remarkably, this form of abstraction is unsupported in almost all web frameworks, and not fully supported in any web framework of which we are aware.

Formlets are defined in terms of Conor McBride’s idioms (also called applicative functors), a program structuring technique halfway between Moggi’s monads and Hughes’s arrows.

We have implemented formlets in the Links programming language. The core functionality is implemented entirely in library code. In addition we offer syntactic sugar which makes formlets particularly easy to use. We show how to extend formlets to support validation by composing the underlying idiom with an additional idiom.

1 Introduction

Raw HTML forms, together with the code that handles them, give a low-level interface to a browser’s form facilities which poses several problems:

Accordance The code which handles the input submitted through a form is separate from the form’s definition, and they have no static association; changes to one are not propagated to the other, causing runtime errors.

Aggregation Form-field inputs are always submitted individually and always as strings: HTML provides no facility for submitting structured data.

Composition 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.

Formlets are abstract form components that: statically check that a form and its handler are compatible, translate raw form strings into structured values, and automatically generate distinct names for distinct fields. Thus formlets solve all three problems.

Conventional web programming frameworks such as PHP [14] and Ruby on Rails (RoR) [17] break abstraction by exposing to programmers each field in

* draft (October 2007)
a form. Research frameworks such as JWIG [4], scriptlets [7] and our previous design for Links [5] all fall short in the same way.

Three existing web programming frameworks that do support some degree of abstraction over form components are WASH [18], iData [15] and WUI [8, 9]. A key feature of formlets that is not supported by any other framework, to our knowledge, is form abstraction over function types. Such abstractions allow forms to be composed using straightforward function application, and are fundamental to the simplicity of the idiomatic semantics of formlets described in Sect. 4.

Table 1 summarises the features supported by a range of web programming frameworks. Several features require explanation. Parametric form abstractions are form abstractions that can be used to input values of any type (some systems restrict the input type). Functional form composition is the use of functional form abstractions for composing forms using straightforward function application.

The technical contributions of this paper are:

– A unified design for an abstraction over HTML forms which is compositional, type-safe, parametrically polymorphic, and easy to use;
– A definition of this feature in terms of idioms, a simple semantic framework;
– An extended design supporting form validation;
– A comparison of form-abstraction features in web-programming systems.

The rest of the paper is organised as follows. Section 2 gives a tour of formlets by example. Section 3 defines them precisely, while Section 4 factors this definition into standard idioms. Section 5 adds support for validation. Section 6 compares formlets with other web-programming systems. Section 7 concludes.

2 Formlets by Example

We illustrate formlets at a high level with an example. We assume familiarity with HTML and use Links syntax. Fig. 1 gives a Links refresher; more detail
Fig. 1. Syntax reference

is available in our earlier work [5]. One point bears noting: 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. (This is an instance of the common phenomenon whereby a type
system may not be precise enough to capture some patterns of use. Though well
studied [13], static validity checking for XML is not a clear win. It can lead to
large types, for instance, and handling polymorphism is challenging.)

The following Links expression creates a formlet, called date, with two text
input fields, labeled “Month” and “Day”:

```
sig date : Formlet(Date);
var date = formlet <#>Month: {inputInt → month},
    Day: {inputInt → day}</#>
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 expression formlet body yields result constructs a formlet. The value
yielded by the formlet is given by the expression result. The expression body
is an XML quasiquote augmented with a facility for binding formlets. A formlet
binding \{f → p\} binds the value yielded by f to the pattern p in result.
To evaluate a formlet expression, each antiquote term in the *body* expression is evaluated and its result replaces the antiquote term; the *body* expression determines the “rendering” of the formlet. The *result* expression is evaluated when the form is submitted; it determines the “collector” of the formlet.

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.

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.

```plaintext
var travelFormlet =
formlet
  <#><label>Name: {input → name}</label>
  <div>
    <label>Arrival date: {date → arrival}</label>
    <label>Departure date: {date → departure}</label>
  </div>
  {submit("Submit")}
</#>

yields (name, arrival, departure);
```

This formlet includes a submit button, created using `submit`, which simply returns the HTML for a submit button. Although `submit` does not produce a formlet, the generated element cannot cause a name clash because it is not given a `name` attribute.

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. To allow this, we introduce the `page q` construct. Here `q` is an XML quasiquote augmented with a facility to associate formlets with handlers. A formlet/handler association is written `{f ⇒ h}`. Intuitively, the value of `page q` is an HTML value where each formlet/handler association `{f ⇒ 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 form applies the handler `h` to the result of invoking `f`’s collector.

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

```plaintext
sig displayItinerary : ((String, Date, Date)) → Xml
fun displayItinerary((name, arrival, departure)) { 
  <html><body>
    Itinerary for: {stringToXml(name)}.
    Arriving: {dateToXml(arrival)}.
    Departing: {dateToXml(departure)}.
    Duration of trip: {dateToXml(duration(departure, arrival))}.
  </body></html>
}
```
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.

3 Formlets by Definition

Having shown how formlets look to the programmer, we now develop a definition. We begin by asking what operations we would like to perform on formlets. We then show how the formlet syntax can be desugared into the fundamental formlet operations. In Appendix A we give a concrete Links implementation of the operations.

3.1 Operations

From the programmer’s viewpoint, formlets consist of two functions: a renderer and a collector. The renderer returns the HTML rendering of a form, and the collector transforms raw submitted form data into the desired value. The renderer corresponds roughly to the body of a formlet expression and the collector to the result part of a formlet expression.

As functional programmers, we ask for a repertoire of transformations of these parts: we would like to be able to operate on the value yielded by the collector, transforming a Formlet(\(A\)) into a Formlet(\(B\)) by means of a function \(\text{Fun}(A) \rightarrow B\). Such an operation can be expressed applicatively by means of ordinary functions—but we use a specialised notion of function application, written with the operator \((\odot)\) :

\[
(\text{Fun}(\alpha) \rightarrow \beta, \text{Fun}(\alpha)) \rightarrow \text{Fun}(\beta).
\]

In addition, we want to be able to lift any value of type \(A\) into a formlet of type Formlet(\(A\)), which we do using pure : \(\text{Fun}(\alpha) \rightarrow \text{Fun}(\alpha)\). As we shall see in Sect. 4, these two operations are exactly what we need for defining an idiom.

Further, we want to be able to transform the HTML rendering, so we define an operation plug : (XmlContext, Formlet(\(\alpha\))) \rightarrow Formlet(\(\alpha\)) and define the type

\[
\text{typeName} \ \text{XmlContext} = \text{Fun}(\text{Xml}) \rightarrow \text{Xml}
\]

(The names XmlContext and plug will make more sense when we discuss certain restrictions on their intended use, in Sect. 3.3.)

Finally, we define 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. It is easy to add similar basic operations to the library. (We could define a single operation that generalises all
typename Formlet(α)

fundamental operations
sig pure : (α) → Formlet(α)
infixl ⊗
sig ⊗ : (Formlet((α) → β), Formlet(α)) → Formlet(β)
sig plug : (XmlContext, Formlet(α)) → Formlet(α)

library operations
sig input : Formlet(String)
sig inputInt : Formlet(Int)
sig submit : (String) → Xml

rendering
sig render : (Formlet(α), (α) → Xml) → Xml

Fig. 2. The basic formlet operations

of these, which handles only the name-generation and is parameterised on the HTML rendering; but we find the above more convenient.)

A formlet can be rendered as HTML using the render function, which takes a formlet and a handler, and produces an HTML form as output. The render function is not intended to be used directly by programmers; page expressions should be used instead.

The basic formlet operations are summarised in Fig. 2.

3.2 Desugaring

In order to define the translation from the formlet syntax to the fundametal operations, we give a precise definition of the Links syntax for XML quasiquotes:

(trees) \[ r ::= s \mid <t as>xs</t> \]
(nodes) \[ x ::= r \mid \{e\} \mid \{f \to p\} \]
(forests) \[ xs ::= \mid x xs \]
(XML quasiquotes) \[ q ::= <\#>xs</\#> \mid <t as>xs</t> \]

The desugaring translation \((\cdot)\circ\) compiles away the syntactic sugar.

\[ (\text{formlet } q \text{ yields } e)\circ = \text{pure } (\text{fun } (q^\dagger)(e^{\circ})) \otimes q^\ast \]
Its definition uses two auxiliary operations \((\cdot)^*\) and \((\cdot)^\dagger\).

\[
\begin{align*}
  s^\dagger &= \_{e} & s^\dagger &= \text{body}(\text{stringToXml}(s)) \\
  \{e\}^\dagger &= \_{e} & \{e\}^\dagger &= \text{body}(e^\circ) \\
  \{f \rightarrow p\}^\dagger &= p & \{f \rightarrow p\}^\dagger &= f^\circ \\
  <t \rightarrow xs</t>^\dagger &= xs^\dagger & <t \rightarrow xs</t>^\dagger &= \text{plug}(\text{fun}(x)\{<t \rightarrow x</t>\}, xs^\dagger) \\
  <#>xs</#>^\dagger &= xs^\dagger & <#>xs</#>^\dagger &= x^s \\
  (x_1 \ldots x_k)^\dagger &= (x_1^\dagger, \ldots, x_k^\dagger) & (x_1 \ldots x_k)^\dagger &= \text{pure}(\text{fun}(x_1^\dagger) \ldots (x_k^\dagger) \{ (x_1^\dagger, \ldots, x_k^\dagger) \}) \odot x_1^s \ldots \odot x_k^s.
\end{align*}
\]

The translation \((\cdot)^\circ\) commutes with all other syntax constructors. Given an XML quasQuote \(q\), the operation \(q^*: \text{Formlet}(\alpha)\) returns a formlet expression; the operation \(q^\dagger: \alpha\) returns a pattern. The desugaring translation uses a helper function to lift an HTML value to a trivial formlet that renders to the supplied HTML value and whose collector returns the unit value.

\[
\begin{align*}
  \text{sig body : (} \text{Xml} \rightarrow \text{Formlet}(\_)) \\
  \text{fun body}(x) \{ \text{plug}(\text{fun}(y)\{<#>{x}{y}</#>\}, \text{pure}(\_)) \}
\end{align*}
\]

Note that the target of the \((\cdot)^\circ\) operation is just ordinary applicative Links syntax, without any formlet-expressions or -bindings. An essential property of this desugaring is that every \text{XmlContext} it produces is linear. We rely on this property in Sect. 3.3.

Without the aid of the sugar, the travel formlet is written as follows.

\[
\begin{align*}
  \text{pure}(\text{fun} (\text{name})(\text{arrival}, \text{departure})(\_) \{ (\text{name}, \text{arrival}, \text{departure}) \}) \odot \\
  \text{plug}(\text{fun} (x)\{<\text{label}>\text{Name: } \{x\}</\text{label}>\}, \text{input}) \odot \\
  \text{plug}(\text{fun} (x)\{<\text{div}>\{x\}</\text{div}>\}, \\
  \text{pure}(\text{fun} (x)(y) \{(x, y)\}) \odot \\
  \text{plug}(\text{fun} (x)\{<\text{label}>\text{Arrival date: } \{x\}</\text{label}>\}, \text{date}) \odot \\
  \text{plug}(\text{fun} (x)\{<\text{label}>\text{Departure date: } \{x\}</\text{label}>\}, \text{date}) \odot \\
  \text{body}(\text{submit}("\text{Submit}"))
\end{align*}
\]

Compare this with the sugared version on page 4. The sugar has allowed us to flatten out the formlet composition into a single formlet expression; note that the plugging operation falls out naturally from the XML quasQuote syntax.

Page expressions are desugared using the operation \((\cdot)^*: \\}

\[
\begin{align*}
  \{f \Rightarrow h\}^* &= \{\text{render}(f, h)\} & (\text{page } q)^* &= q^*
\end{align*}
\]

The \((\cdot)^*\) operation commutes with all other syntax constructors. The desugaring of page expressions becomes more interesting when we add validation (Sect. 5).

### 3.3 Limitations of static safety

One of the aims of formlets is to ensure compatibility between the field names in the HTML and the names read by the collector. More precisely, each name appearing in the HTML should be distinct, and the set of names appearing
in the HTML should be exactly the set of names the collector expects. These properties are almost guaranteed by construction—the formlet library functions ensure that whenever a name is used in the rendered HTML, the same name is read in the collector. However, there are two ways in which things could go wrong.

If we implement HTML contexts as functions and expose the plug function then there is nothing to stop the programmer passing a context to plug that does not use its argument linearly. This would allow names to be duplicated or deleted altogether in the HTML whilst leaving them unchanged in the collector. Our solution is to hide plug, and insist that programmers use the syntactic sugar instead. As noted, desugaring produces only linear contexts.

Still, a programmer may try to plug a formlet into an HTML context that contains input elements or other HTML form components, potentially leading to a name clash. In our current implementation we leave it up to the programmer to ensure that the elements form, input, button, textarea, select, option and optgroup never appear in the source, outside of the formlet library. The library can be used when these elements are needed.

4 The formlet idiom

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 [2] provide the standard semantic tools for reasoning about side-effects. It is not difficult to see that there is no monad corresponding to the formlet type. Intuitively, the problem is that if we try to define a bind operation for the formlet type, then it 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 formlet. In fact, the formlet idiom is the composition of three primitive idioms.

Recall [11] that an idiom is simply a type constructor together with operations pure and ⊗, 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. Typically, an idiom will also come with some operations for constructing impure (or effectful) values.

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

\[ \text{pure} : (\alpha) \rightarrow I(\alpha) \quad \odot : (I((\alpha) \rightarrow \beta), I(\alpha)) \rightarrow I(\beta) \]

that satisfy the following equations:

\[ \text{pure}(id) \odot u \equiv u \quad (id \text{ is the identity function}) \]
\[ \text{pure}(\circ) \odot u \odot v \odot w \equiv u \odot (v \odot w) \quad (\circ \text{ is function composition}) \]
\[ \text{pure}(f) \odot \text{pure}(x) \equiv \text{pure}(f(x)) \]
\[ u \odot \text{pure}(x) \equiv \text{pure}(\text{fun}(f)(f(x))) \odot u \]
These laws guarantee that pure computations can be reordered. In particular, an idiomatic effectful computation cannot depend on the result of a pure computation, and any expression built from pure and ⊗ can be rewritten in the canonical form: \( \text{pure}(f) \otimes u_1 \otimes \cdots \otimes u_k \), where \( f \) is the pure part of the computation and \( u_1 \ldots u_k \) are the effectful parts of the computation.

Now we can show the idioms that comprise formlets. The name-generation idiom threads a source of names through all of its computations; this permits the effect of fresh name generation.

\[
\text{typename } I_n(\alpha) = (Gen) \rightarrow (\alpha, Gen)
\]

\[
\text{fun pure}_n(v) \{ \text{fun}(gen) \{ (v, gen) \} \}
\]

\[
\text{op } f \otimes_n a \{ \text{fun } (gen) \{ \begin{array}{l}
\text{var } (v, gen) = f(gen); \\
\text{var } (w, gen) = a(gen);
\end{array}
\}
\]

\[
(v(w), gen)
\]

The accumulation idiom over the monoid of XML forests carries a value from the monoid alongside its computations. The idiom’s application operation also combines the two computations’ monoidal values using the monoid multiplication.

\[
\text{typename } I_x(\alpha) = (Xml, \alpha)
\]

\[
\text{fun pure}_x(v) \{ (<#/>, v) \}
\]

\[
\text{op } (x, f) \circ_x (y, a) \{ (<#>{x}{y}</#>, f(a)) \}
\]

The environment idiom passes some environment (e.g. an association list) through all its computations; the available effect is reading from the environment.

\[
\text{typename } I_e(\alpha) = (Env) \rightarrow \alpha
\]

\[
\text{fun pure}_e(v) \{ \text{fun } (env) \{ v \} \}
\]

\[
\text{op } f \circ_e a \{ \text{fun } (env) \{ f(env)(a(env)) \} \}
\]

(Readers familiar with monads can check that these are just standard monads, viewed as idioms. The transformation from a monad to an idiom takes \( \text{pure} = \text{return} \) and \( f \otimes x = f \star (\lambda f. x \star (\lambda x. f x)) \) where \( \star \) is the Kleisli star or “bind” operation.)

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 \( \text{pure}_1, \otimes_1 \) and \( \text{pure}_2, \otimes_2 \) (respectively), we obtain the idiom I pre-composed with J as \( I \circ J \) where

\[
\begin{array}{l}
\text{var } \text{pure}_{1J} = \text{pure}_1 \circ \text{pure}_2; \\
\text{op } f \circ_{1J} a \{ \text{pure}_1((\circ_2)) \otimes_1 f \otimes_1 a \}
\end{array}
\]

The formlet idiom We now give an implementation of formlets as the composition of the above idioms. The reader should see that this provides the basic architecture of formlets and that it corresponds to the complete implementation in Appendix A.

\[
I = I_n \circ I_x \circ I_e
\]
5 Form 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 Adding validation

In order to keep track of error messages we modify the Formlet type, changing the return type of the collector component from \( \alpha \) to \((\text{Xml}, \text{Maybe}(\alpha))\):

\[
\text{typename Maybe}(\alpha) = [| \text{Just}: \alpha | \text{Nothing} |]
\]

\[
\text{typename Formlet}(\alpha) = (\text{Gen}) \rightarrow (\text{Xml}, \text{Collector}(\text{Xml}, \text{Maybe}(\alpha))), \text{Gen})
\]

When passed an environment a Collector((\text{Xml}, \text{Maybe}(A))) attempts to extract a value \( v \) of type \( A \), returning Just(\( v \)) if the extraction succeeds, or Nothing if it fails. The second component of the return value, the error rendering, is the HTML that should be displayed in an error situation, whether that arose due to this or some other collector. When no validator is attached to the component, the component succeeds, using the original HTML as the error rendering. Given these adjustments we can add a validating operation, satisfies, on formlets:

\[
\text{typename Validator}(\alpha) = (\alpha, \text{Xml}) \rightarrow (\text{Xml}, \text{Maybe}(\alpha))
\]

\[
\text{sig satisfies : (Formlet}(\alpha), \text{Validator}(\alpha)) \rightarrow \text{Formlet}(\alpha)
\]

An auxiliary function, \( \text{err} \), constructs a validator from a predicate and a function that builds an error message; if the predicate fails then the message function will be passed the failing value.

\[
\text{sig err : ((\alpha) \rightarrow \text{Bool}, (\alpha) \rightarrow \text{String}) \rightarrow \text{Validator}(\alpha)}
\]

\[
\text{fun err}(\text{pred}, \text{error})(\text{val}, \text{xml}) =
\begin{align*}
\text{if (pred(\text{val})) (\text{xml, Just (\text{val})})} \\
\text{else (<##><span class="errorinput">{\text{xml} } </span> \text{error}(<##>\text{stringToXml(error(x))} </##>, \text{Nothing})}
\end{align*}
\]

(In the implementation given in Appendix A we flatten the nested pairs into triples.)
Besides these new operations, we must adjust the fundamental formlet operations to support validation. An implementation is given in Appendix B.

Using \( \text{satisfies} \) and \( \text{err} \) we can add error-checking to a formlet component. For example, we can now improve the definition of \( \text{inputInt} \) (Appendix A) so that a suitable message is displayed if parsing the string fails:

\[
\begin{align*}
\text{fun } \text{isInt}(s) & \{ s \sim /^-?[0-9]+$/ \} \\
\text{fun } \text{intError}(s) & \{ s ++ \text{ " is not an integer!" } \} \\
\text{sig } \text{inputInt} & : \text{Formlet}(\text{Int}) \\
\text{var } \text{inputInt} & = \text{formlet} <\text{\#}\{\text{input } \text{satisfies } \text{\'}}(\text{isInt } \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{\'} \text{
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>

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 pages represent composable web-page fragments containing validated forms. A page consists of a k-holed XML context, k formlets and k handlers. Following the presentation of formlets, we first give the operations on pages, then describe the desugaring rules, and finally the implementation in Links.

The abstract type is given in Fig. 3. Pages have a monoid structure. The unit value represents an empty page fragment with no forms or handlers; \( g_1 \otimes g_2 \) concatenates two page fragments together by concatenating the contexts, formlets and handlers of the pages \( g_1 \) and \( g_2 \). The plug operations is analogous to the equivalent formlet operation. The form function realises a pair of a formlet and a handler as a page.

In order to support validation the desugaring and type of pages needs to be adjusted. Top-level pages are automatically rendered to HTML using the internal `renderPage` function.

\[
\begin{align*}
(xs)^\nabla &= xs^\nabla \\
\langle t as xs\rangle^\nabla &= \text{plug}(\text{fun}(x)\{\langle t as \{x\} as t\rangle, x\})^\nabla \\
\{e\}^\nabla &= \text{body}(e^* \\
\{f \Rightarrow h\}^\nabla &= \text{form}(f^*, h^*) \\
\{f | g\}^\nabla &= g^* \\
\{\langle g | q\rangle\}^\nabla &= q^\nabla
\end{align*}
\]

Formlet instantiations are desugared to calls to `form`. Note the difference between embedded XML expressions (which are desugared using `body`—analogous to the equivalent formlet operation) and embedded pages (which are desugared by simply recursing). The `joinMany` function is the k-ary version of the multiplication operation `join`.

fun joinMany(gs) { foldl(join, unit, gs) }

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

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

sig unit : Page
sig ⊗ : (Page, Page) → Page

sig plug : (XmlContext, Page) → Page
sig form : (Formlet(α), Handler(α)) → Page

sig renderPage : (Page) → Xml

Fig. 3. The abstract type of pages

The implementation of validating pages is given in Appendix B.2. We briefly outline some of the interesting aspects. Recall that a page consists of a multi-holed XML context, and a list of formlet/handler pairs. 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 MultiXmlContext = ([Xml]) → Xml
typename Page = (Int, MultiXmlContext, [∃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 [12], which introduces existentials for encoding heterogeneous environments.

Another interesting aspect of the typing arises from the validation loop. In order to tie the recursive knot we make use of a recursive type. An abstraction is introduced for each HTML form, such that it is parameterised over the list of all HTML forms on a page. This allows every form to be updated when one of them changes (due to a validation error).

6 Related work

**JWIG** JWIG [4] is an extension of Java for building web services. It builds on ideas developed in MAWL [1] and <bigwig> [3]. 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 run-time. 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.

**Scriptlets** Scriptlets are build on top of SMLserver [6], a webserver for serving web applications written in Standard ML. Elsman and Larsen [7] 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.

**WUI** The WUI (Web User Interface) library [8, 9] implements form abstractions for the functional logic programming language Curry. The WUI library composes form abstractions using simple combinators for building up tuples, but WUIs do not directly support functional form composition.

Of the existing web form frameworks, WUI is the one that is closest to formlets in spirit. Indeed there is an embedding of formlets into WUIs. Formlets are simpler than WUIs in that unlike WUIs they do not take an input argument. This is not an important restriction as input arguments can be straightforwardly simulated using functional abstraction. In fact using functional abstraction for inputs is more flexible, as it allows both the rendered HTML and the collector to depend on the input value. The negative occurrence of the input argument in the datatype for WUIs means that it is not possible to directly characterise WUIs as idioms. It is, however, possible to characterise a generalisation of WUIs as arrows [10], a generalisation of idioms (exercise).

**WASH** The WASH/CGI Haskell library [18] 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.

A page generated by WASH has at most one form on it. The library provides a notion of “form” distinct from the HTML notion of form. This prevents varying the HTML attributes, such as the form encoding, between forms; furthermore, all form data must be submitted whenever any submit button is pressed. This can affect privacy and network usage. Consider the result if the user has tentatively selected a file to upload in one form and then decides to submit a different form: the entire file must be transferred although the user may not intend this.

**iData** The iData library [15] 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. At a lower-level form abstractions can be constructed in a non-type-directed manner.

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 \[16\] builds on top of iData and addresses this issue by enabling or disabling iData according to the state of the program.

7 Conclusion

We have presented formlets as 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.

Our current implementation of formlets always runs form handlers on the server. Links supports code running on the client as well as the server, so a natural extension would be to allow form handlers to run on the client. A more challenging area of future work is to extend formlets to respond to events other than submitting the form. It is straightforward to support certain client-side functionality such as validation, but seems harder to give a general model. Particular difficulties arise if we allow client-side code to dynamically modify forms.

References

Here we give the implementation of the basic formlet functions. The concrete type of formlets is as follows:

```
typename Gen = Int
.typename Env = [(String, String)]
typename Collector(α) = Env \rightarrow α
.typename Formlet(α) = Gen \rightarrow (Xml, Collector(α), Gen)
```

In order to ensure that the input fields rendered by formlets have distinct names, a name generator of type $Gen$ is threaded through formlets. The name generator is implemented as an single integer. When generating a name, this is converted to a string; then it is incremented before being passed onward.

When the user submits a form, the Links runtime turns the submitted data into an environment (of type $Env$), which maps input-field names to input values. Currently an environment is implemented as an association list from strings to strings. A $Collector(α)$ produces a value of type $α$ from such an environment. Finally, a $Formlet(α)$ is a function taking a name generator and producing a triple of the HTML rendering, the collector, and the updated name generator.

Now the fundamental formlet functions are straightforward to define:

```
sig pure : (α) \rightarrow Formlet(α)
fun pure(α) { fun (gen) (<#/>), fun (_{a}, gen) }

infixl ⊛
sig ⊛ : (Formlet((α) \rightarrow β), Formlet(α)) \rightarrow Formlet(β)
op f ⊛ a { 
  fun (gen) { 
    var (x, c, gen) = f(gen); 
    var (y, d, gen) = a(gen); 
    (<#{x}{y}</#>), fun (env) {c(env)(d(env))}, gen 
  }
}
```
The basic formlet operations must generate fresh names using the name generator, which is defined by the following function:

```haskell
typename Gen = Int

sig nextName : (Gen) -> (String, Gen)
fun nextName(gen) { ("input_"+ intToString(gen), gen+1) }
```

The `nextName` function takes a name generator and returns a fresh string along with an updated name generator. The `input` formlet illustrates a use of `nextName`:

```haskell
fun input(gen) {
  var (name, gen) = nextName(gen);
  (<input name="{name}" />,
   fun (env) {lookup(name, env)},
   gen)
}
```

The generated name is used both in the rendered HTML, as a field name, and in the collector, to look up the input data. The implementations of the other library operations are similar.

The `inputInt` formlet is a straightforward wrapper over the `input` formlet.

```haskell
var inputInt = formlet<#>{input → s}</#>
yields stringToInt(s);
```

The `render` function is defined in terms of a helper, `mkForm`.

```haskell
fun mkForm(cont, contents) {
  <form enctype="application/x-www-form-urlencoded"
    action="#" method="POST">
    <input type="hidden" name="_k" value="{pickleCont(cont)}" />
    {contents}
  </form>
}
```
sig render : (Formlet(α), (α) → Xml) → Xml
fun render(f, handler) {  
  var (x, c, _) = f(0);  
  mkForm(fun (env) {handler(c(env))}, x)  
}

In render, the handler is composed with the collector, which together are pickled into a string and stored in the special hidden field _k which the Links runtime recognised as an entry point. The pickleCont function is a built-in Links function for marshalling a closure as a string.

sig pickleCont : ((Env) → Xml) → String
B Implementation of validation

B.1 Formlets

Here we give the implementation of the validating version of formlets.

```plaintext
typename Maybe (α) = [ | Just : α | Nothing | ]
typename Formlet (α) = (Gen) → (Xml, Collector ((Xml, Maybe(α))), Gen)
```

The collector of the pure function now returns an empty error rendering and a value wrapped in Just.

```plaintext
sig pure : (α) → Formlet (α)
fun pure (a) { fun (gen) { (<#/> , fun (_) { (<#/> , Just (a)) } , gen) } }
```

The ⊛ operator concatenates the error renderers; collection now succeeds only if it succeeds for both operands.

```plaintext
sig ⊛ : (Formlet ((α) → β), Formlet (α)) → Formlet (β)
op f ⊛ a { 
  fun (gen) { 
    var (x, c, gen) = f (gen); var (y, d, gen) = a (gen);
    (<#>{x}{y}</#>,
    fun (env) { var (x, v) = c (env); var (y, w) = d (env);
      (<#>{x}{y}</#>, v ⊛ o w),
    gen) 
  }
}
```

The plug operation applies the XMLContext to both the regular and the error renderers.

```plaintext
sig plug : (XmlContext, Formlet (α)) → Formlet (α)
fun plug (k, f) { 
  fun (gen) { 
    var (x, c, gen) = f (gen);
    fun d (env) { 
      var (v, y) = c (env);
      (v, k (y))
    }
    (k (x), d, gen)
  }
}
```

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.
sig input : Formlet(String)

fun input(gen) {
    var (name, gen) = nextName(gen);
    (<input name="{name}"/>,
     fun (env) {
         var v = assoc(name, env);
         (<input name="{name}" value="{v}"/>, Just (v)),
         gen
     }
    )
}
B.2 Pages

Here we give the implementation of the validating version of pages. 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 `validate` function.

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

sig validate :
  (Collector(α), Handler(α), MultiXmlContext, RecForms, Int)
  → (Env) → Xml

fun validate(c, h, k, zs, i)(env) {  
  switch (c(env)) {  
    case (_, Just(v)) → h(v)  
    case (x, Nothing) → {  
      fun z(zs) { mkForm(validate(c, h, k, zs, i), x) }  
      var zs = substAt(zs, i, z);  
      k(map(fun z {z(zs)}, zs))  
    }  
  }  
}
```

The `validate` function takes six arguments: `c` and `h` are the collector and handler for the `i`-th form in the multi-holed context `k`, `zs` is a list of functions for generating the HTML for the forms to be plugged into `k` and `env` is the environment. The collector is run 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.

We can now give an implementation of pages that does not depend on an existential type.

```
typename CheckedFormBuilder =  
  (MultiXmlContext, RecForms, Int) → Xml

typename Page =  
  (Int, MultiXmlContext, (Gen) → ([CheckedFormBuilder], Gen))

sig renderPage : (Page) → Xml  
fun renderPage((n, k, fs)) {  
  var (ms, _) = fs(0);  
  var zs = mapi(fun (m, i) {m(k, zs, i)}, ms);  
  k(map(fun z {z(zs)}, zs))  
}
```

```
sig mkCheckedFormBuilder : (Xml, Collector(α), Handler(α))
  → (MultiXmlContext, RecForms, Int) → Xml

fun mkCheckedFormBuilder(x, c, h)(k, zs, i) {  
  mkForm(validate(c, h, k, zs, i), x)  
}
```
sig unit : Page
var unit = (0, fun ([]) {"/"}, fun (gen) ([], gen));

sig join : (Page, Page) → Page
fun join((i₁, k₁, fs₁), (i₂, k₂, fs₂)) {
  (i₁ + i₂,
   fun (xs) {
     "/>{%k₃(take(i₁, xs))(k₄(drop(i₁, xs)))%}<"/>
   },
   fun (gen) {
     var (gen, ms₁) = fs₁(gen);
     var (gen, ms₂) = fs₂(gen);
     (ms₁ ++ ms₂, gen)
   })
}

sig body : (Xml) → Page
fun body(x) {
  (0, fun ([]) {x}, fun (gen) {[[], gen]})
}

sig plug : (XmlContext, Page) → Page
fun plug(context, (i, k, fs)) {
  (i, fun (xs) {context(k(xs)), fs})
}

sig form : (Formlet(α), Handler(α)) → Page
fun form(f, h) {
  (1,
   fun ([x]) {x},
   fun (gen) {
     var (x, c, gen) = f(gen);
     ([mkCheckedFormBuilder(x, c, h]], gen)
   })
}