# The Essence of Form Abstraction

# The Essence of Form Abstraction*

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

School of Informatics, University of Edinburgh

**Abstract.** Abstraction is the cornerstone of high-level programming; HTML forms are the principal medium of web interaction. However, most web programming environments do not support abstraction of form components, leading to a lack of compositionality. Using a semantics based on idioms, we show how to support compositional form construction and give a convenient syntax.

## 1 Introduction

Say you want to present users with an HTML form for entering a pair of dates (such as an arrival and departure date for booking a hotel). In your initial design, a date is represented just as a single text field. Later, you choose to replace each date by a pair of pulldown menus, one to select a month and one to select a day.

In typical web frameworks, such a change will require widespread modifications to the code. Under the first design, the HTML form will contain *two text fields*, and the code that handles the response will need to extract and parse the text entered in each field to yield a pair of values of an appropriate type, say, an abstract date type. Under the second design, however, the HTML will contain *four menus*, and the code that handles the response will need to extract the choices for each menu and combine them in pairs to yield each date.

How can we structure a program so that it is isolated from this choice? We want to capture the notion of *a part of a form*, specifically a part for collecting values of a given type or purpose; we call such an abstraction 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 be insulated from the choice of HTML presentation, and also from the calculation that yields the abstract value. And, of course, we should be able to compose formlets to build larger formlets.

Once described, this sort of abstraction seems obvious and necessary. But remarkably few web frameworks support it. Three existing web programming frameworks that do support some degree of abstraction over form components are WASH [28], iData [23] and WUI [11, 12], each having distinctive features and limitations. (We discuss these further in Section 6.)

Our contribution is to reduce form abstraction to its essence. We use *idioms* [19] (also known as *applicative functors*), a notion of effectful computation,

* Supported by EPSRC grant number EP/D046769/1

related to both monads [20] and arrows [14]. We define a semantics for formlets by composing standard idioms, show how to support compositional form construction, and give a convenient syntax. Furthermore, we illustrate how the semantics can be extended to support additional features (such as checking form input for validity), either by composing with additional standard idioms or by generalising to indexed and parameterised idioms.

We originally developed formlets as part of our work on Links [6], a programming language for the web. Like many other systems the original design of Links exposed programmers to the low-level details of HTML/CGI. We introduced formlets as a means to abstract away from such details.

In this paper we present a complete implementation of formlets in OCaml. We take advantage of the extensible Camlp4 preprocessor to provide syntactic sugar, without which formlets are usable but more difficult to read and write. Both the library and the syntax extension are available from

```
http://groups.inf.ed.ac.uk/links/formlets/
```

The Links implementation of formlets also provides the syntax presented here. The complete Links system includes many features, such as a full suite of HTML controls (textareas, pop-up menus, radio buttons, etc.), which are not described here. Steve Strugnell has ported a commercial web-based project-management application originally implemented in PHP to the Links version of formlets [26]. He gives an in-depth comparison between Links formlets and forms implemented in PHP. Chris Eidhof has released a Haskell implementation of formlets [8].

The remainder of this paper is organised as follows. Section 2 presents formlets, as they appear to the programmer, through examples. Section 3 gives a semantics for formlets as the composition of the three idiom instances that capture the effects needed for form abstraction. Section 4 defines formally the formlet syntax used throughout the paper and relates it to the formlet idiom. Section 5 shows how to extend the basic abstraction with additional features: static XHTML validation, user-input validation, and an optimised representation based on multi-holed contexts. Section 6 examines the relationship with existing form-abstraction features in high-level web frameworks.

## 2 Formlets by example

Now we illustrate formlets, as they might appear to the programmer, with an example (Fig. 1). We assume familiarity with HTML and OCaml. This section covers our OCaml implementation, and so has features that may vary in another implementation of formlets. We use a special syntax (defined formally in Section 4) for programming with formlets; this syntax is part of the implementation, and makes formlets easier to use, but not an essential part of the abstraction.

The formlet *date_formlet* has two text input fields, labelled "Month" and "Day." Upon submission, this formlet will yield a *date* value representing the date entered. The user-defined *make_date* function translates the day and month into a suitable representation.

```
let date_formlet : date formlet = formlet
  <div>
    Month: {input_int ⇒ month}
    Day: {input_int ⇒ day}
  </div>
yields  make_date month day

let travel_formlet : (string × date × date) formlet =
  formlet
    <#>
      Name: {input ⇒ name}
      <div>
        Arrive: {date_formlet ⇒ arrive}
        Depart: {date_formlet ⇒ depart}
      </div>
      {submit "Submit"}
    </#>
  yields  (name, arrive, depart)

let display_itinerary : (string × date × date) → xml =
  fun (name, arrive, depart) →
    <html>
      <head><title>Itinerary</title></head>
      <body>
        Itinerary for: {xml_text name}
        Arriving: {xml_of_date arrive}
        Departing: {xml_of_date depart}
      </body>
    </html>

handle travel_formlet display_itinerary
```

**Fig. 1.** Date example

```
let date_formlet : date formlet =
  pure (fun ((), month, (), day, ()) → make_date month day)
  ⊗ (tag "div" []
       (pure (fun () month () day () → ((), month, (), day, ()))
        ⊗ text "Month: " ⊗ input_int
        ⊗ text "Day: " ⊗ input_int ⊗ text "\n "))

let travel_formlet : (string × date × date) formlet =
  pure (fun ((), name, ((), arrive, (), depart), ()) →
        (name, arrive, depart))
  ⊗ (pure (fun () name ((), arrive, (), depart) () →
             ((), name, ((), arrive, (), depart), ()))
     ⊗ text "Name: " ⊗ input
     ⊗ (tag "div" []
          (pure (fun () arrive () depart → ((), arrive, (), depart))
           ⊗ text "Arrive: " ⊗ date_formlet
           ⊗ text "Depart: " ⊗ date_formlet))
     ⊗ xml (submit "Submit"))

let display_itinerary : (string × date × date) → xml =
  fun (name, arrive, depart) →
    xml_tag "html" []
      ((xml_tag "head" []
         (xml_tag "title" [] (xml_text "Itinerary"))) @
       (xml_tag "body" []
         ((xml_text "Itinerary for: ") @ (xml_text name) @
          (xml_text "Arriving: ") @ (xml_of_date arrive) @
          (xml_text "Departing: ") @ (xml_of_date depart))))

handle travel_formlet display_itinerary
```

**Fig. 2.** Date example (desugared)

A formlet expression consists of a *body* and a *yields clause*. The body of *date_formlet* is

```
<div>
  Month: {input_int ⇒ month}
  Day: {input_int ⇒ day}
</div>
```

and its yields clause is

*make_date month day*

The body of a formlet expression is a *formlet quasiquote*. This is like an XML literal expression but with embedded *formlet bindings*. A formlet binding $\{f \Rightarrow p\}$ binds the value yielded by $f$ to the pattern $p$ for the scope of the yields clause. Here $f$ is an expression that evaluates to a formlet and the type yielded by the formlet must be the same as the type accepted by the pattern. Thus the variables *month* and *day* will be bound to the values yielded by the two instances of the *input_int* formlet. The bound formlet $f$ will render some HTML which will take the place of the formlet binding when the outer formlet is rendered.

The value *input_int* : *int formlet* is a formlet that renders as an HTML text input element, and parses the submission as type *int*. It is built from the primitive formlet *input* which presents an input element and yields the entered string. Although *input_int* is used here twice, the system prevents any field name clashes.

It is important to realize that any given formlet defines behavior at two distinct points in the program's runtime: first when the form structure is built up, and much later (if at all) when the form is submitted by the user, when the outcome is processed. The first corresponds to the body and the second to the yields clause.

Next we illustrate how user-defined formlets can be usefully combined to create larger formlets. Continuing Fig. 2, *travel_formlet* asks for a name, an arrival date, and a departure date. The library function *submit* returns the HTML for a submit button; its string argument provides the label for the button. (This covers the common case where there is a single button on a form. A similar function *submit_button* : *string* → *bool formlet* constructs a submit button formlet, whose result indicates whether this button was the one that submitted the form.)

(The syntax `<#>` $\cdots$ `</#>` enters the XML parsing mode without introducing a root XML node; its result is an XML forest, with the same type as XML values introduced by a proper XML tag. We borrow this notation from WASH.)

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. The function *handle* attaches a handler to a formlet.

Continuing the above example, we render *travel_formlet* onto a full web page, and attach a handler (*display_itinerary*) that displays the chosen itinerary back to the user. (The abstract type *xml* is given in Fig. 3; we construct XML using

```
type xml = xml_item list
 and tag = string
 and attrs = (string × string) list
 and xml_item

val xml_tag : tag → attrs → xml → xml
val xml_text : string → xml
```

**Fig. 3.** The *xml* abstract type.

special syntax, which is defined in terms of the *xml_tag* and *xml_text* functions, as shown formally in Section 4.)

This is a simple example; a more interesting application might render another form on the *display_itinerary* page, one which allows the user to confirm the itinerary and purchase tickets; it might then take actions such as logging the purchase in a database, and so on.

This example demonstrates the key characteristics of the formlet abstraction: static binding (we cannot fetch the value of a form field that is not in scope), structured results (the month and day fields are packaged into an abstract *date* type, which is all the formlet consumer sees), and composition (we reuse the date formlet twice in *travel_formlet*, without fear of field-name clashes).

### 2.1 Syntactic sugar

Fig. 2 shows the desugared version of the date example. XML values are constructed using the *xml_tag* and *xml_text* functions and the standard list concatenation operator, `@`. Formlet values are slightly more complicated. The *xml_tag* and *xml_text* functions have formlet counterparts *tag* and *text*; composition of formlets makes use of the standard idiom operations *pure* and $\otimes$. The formlet primitives are covered in detail in Section 3.

The sugar makes it easier to freely mix static XML with formlets. Without the sugar, dummy bindings are needed to bind formlets consisting just of XML (see the calls to *pure* in Fig. 2), and formlets nested inside XML have to be rebound (see the second call to *pure* in the body of *travel_formlet* in Fig. 2). A desugaring algorithm is described in Section 4.

### 2.2 Life without formlets

Now consider implementing the above example using the standard HTML/CGI interface. We would face the following difficulties with the standard interface:

- There is no static association between a form definition and the code that handles it, so the interface is fragile. This means the form and the handling code need to be kept manually in sync.
- Field values are always received individually and always as strings: the interface 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

```
module type Idiom = sig
  type α t
  val pure : α → α t
  val (⊗) : (α → β) t → α t → β t
end
```

```
module type FORMLET = sig
  include Idiom
  val xml : xml → unit t
  val text : string → unit t
  val tag  : tag → attrs → α t → α t
  val input : string t
  val run : α t → xml × (env → α)
end
```

**Fig. 4.** The idiom and formlet interfaces

write a form that abstractly uses subcomponents. In particular, it's difficult to use a form twice within a larger form.

Conventional web programming frameworks such as PHP [22] and Ruby on Rails [25] facilitate abstraction only through templating or textual substitution, hence there is no automatic way to generate fresh field names, and any form "abstraction" (such as a template) still exposes the programmer to the concrete field names used in the form. Even advanced systems such as PLT Scheme [10], JWIG [5], scriptlets [9], Ocsigen [2], Lift [15] and the original design for Links [6] all fall short in the same way.

Formlets address all of the above problems: they provide a static association between a form and its handler (ensuring that fields referenced actually exist and are of the right type), they allow processing raw form data into structured values, and they allow composition, in part by generating fresh field names at runtime.

## 3 Semantics

We wish to give a semantics of formlets using a well-understood formalism. We shall show that formlets turn out to be *idioms* [19], a notion of computation closely related to monads [3, 20, 29]. We begin with a concrete implementation in OCaml, which we then factor using standard idioms to give a formal semantics.

### 3.1 A concrete implementation

Figs. 4 and 5 give a concrete implementation of formlets in OCaml.

The type $\alpha\, t$ is the type of formlets that return values of type $\alpha$ (the library exposes this type at the top-level as $\alpha$ *formlet*). Concretely $\alpha\, t$ is defined as a function that takes a *name source* (integer) and returns a triple of a *rendering* (XML), a *collector* (function of type $env \to \alpha$) and an updated name source. The formlet operations ensure that the names generated in the rendering are the names expected (in the environment) by the collector.

The *pure* operation is used to create constant formlets whose renderings are empty and whose collector always returns the same value irrespective of the

```
module Formlet : FORMLET = struct
  type α t = int → (xml × (env → α) × int)

  let pure x i = ([], const x, i)
  let (⊗) f p i = let (x1, g, i) = f i in
                  let (x2, q, i) = p i in
                    (x1 @ x2, (fun env → g env (q env)), i)

  let xml x i = (x, const (), i)
  let text t i = xml (xml_text t) i
  let tag t attrs fmlt i = let (x, f, i) = fmlt i in (xml_tag t attrs x, f, i)

  let next_name i = ("input_" ^ string_of_int i, i + 1)
  let input i = let (w, i) = next_name i in
                (xml_tag "input" [("name", w)] [], List.assoc w, i)

  let run c = let (x, f, _) = c 0 in (x, f)
end
```

**Fig. 5.** The formlet idiom

environment. The $\otimes$ operation applies an $A \to B$ formlet to an $A$ formlet. The name source is threaded through each formlet in turn. The resulting renderings are concatenated and the collectors composed. Together *pure* and $\otimes$ constitute the fundamental idiom operations. (To be an idiom, they must also satisfy some laws, shown in Section 3.2.)

As before, the *xml* and *text* operations create unit formlets from the given XML or text, and the *tag* operation wraps the given formlet's rendering in a new element with the specified tag name and attributes.

The primitive formlet *input* generates HTML input elements. A single name is generated from the name source, and this name is used both in the rendering and the collector. The full implementation includes a range of other primitive formlets for generating the other HTML form elements (e.g. textarea, option, etc.).

The *run* operation "runs" a formlet by supplying it with a name source (we use 0); this produces a rendering and a collector function.

### 3.2 Idioms

Idioms were introduced by McBride [18] to capture a common pattern in functional programming.[1] An *idiom* is a type constructor I together with operations:

$$\mathit{pure} : \alpha \to \mathrm{I}\,\alpha \qquad\qquad \otimes : \mathrm{I}\,(\alpha \to \beta) \to \mathrm{I}\,\alpha \to \mathrm{I}\,\beta$$

[1] Subsequently McBride and Paterson [19] changed the name to *applicative functor* to emphasise the view of idioms as an "abstract characterisation of an applicative style of effectful programming". We stick with McBride's original "idiom" for brevity.

that satisfy the following laws:

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

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

The *pure* operation lifts a value into an idiom. Like standard function application, idiom application $\otimes$ is left-associative. The idiom laws guarantee that pure computations can be reordered. However, 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 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. The order in which computations are performed is significant, but a computation cannot depend on 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. As Lindley and others [17] put it: *idioms are oblivious*.

Formlets fit this pattern: the sub-formlets cannot depend on one another, and the final value yielded by a formlet is a pure function of the values yielded by the sub-formlets.

### 3.3 Factoring formlets

Now we introduce the three idioms into which the formlet idiom factors (Fig. 6). Besides the standard idiom operations in the interface, each idiom comes with operations corresponding to primitive effects and a *run* operation for executing the effects and extracting the final result. A computation in the *Namer* idiom has type $\mathit{int} \to \alpha \times \mathit{int}$; it is a function from a counter to a value and a possibly-updated counter. The *next_name* operation uses this counter to construct a fresh name, updating the counter. A computation in the *Environment* idiom has type $\mathit{env} \to \alpha$; it receives an environment and yields a value. The *lookup* operation retrieves values from the environment by name. A computation in the *XmlWriter* idiom (also known as a monoid-accumulator) has type $\mathit{xml} \times \alpha$ and so yields both XML and a value; the XML is generated by the primitive *xml*, *text* and *tag* operations and concatenated using $\otimes$. Each of these idioms corresponds directly to a standard monad [19].

The formlet idiom is just the composition of these three idioms (see Fig. 8). The *Compose* module composes any two idioms (Fig. 7).

```
module Namer : sig
  include Idiom
  val next_name : string t
  val run : α t → α
end = struct
  type α t = int → α × int
  let pure v i = (v, i)
  let (⊗) f p i = let (f', i) = f i in
                  let (p', i) = p i in
                      (f' p', i)
  let next_name i =
   ("input_"^string_of_int i, i+1)
  let run v = fst (v 0)
end
```

```
module Environment : sig
  include Idiom
  type env = (string × string) list
  val lookup : string → string t
  val run : α t → env → α
end = struct
  type α t = env → α
  and env = (string × string) list
  let pure v e = v
  let (⊗) f p e = f e (p e)
  let lookup = List.assoc
  let run v = v
end
```

```
module XmlWriter : sig
  include Idiom
  val text : string → unit t
  val xml : xml → unit t
  val tag : tag → attrs → α t → α t
  val run : α t → xml × α
end = struct
  type α t = xml × α
  let pure v = ([], v)
  let (⊗) (x, f) (y, p) = (x @ y, f p)
  let text x = (xml_text x, ())
  let xml x = (x, ())
  let tag t a (x,v) = (xml_tag t a x, v)
  let run v = v
end
```

**Fig. 6.** Standard idioms

```
module Compose (F : Idiom) (G : Idiom) : sig
  include Idiom with type α t = (α G.t) F.t
  val refine : α F.t → (α G.t) F.t
end = struct
  type α t = (α G.t) F.t
  let pure x = F.pure (G.pure x)
  let (⊗) f x = F.pure (⊗_G) ⊗_F f ⊗_F x
  let refine v = (F.pure G.pure) ⊗_F v
end
```

**Fig. 7.** Idiom composition

```
module Formlet : FORMLET = struct
  module AE = Compose (XmlWriter) (Environment)
  include Compose (Namer) (AE)
  module N = Namer module A = XmlWriter module E = Environment
  let xml x = N.pure (AE.refine (A.xml x))
  let text s = N.pure (AE.refine (A.text s))
  let tag t ats f = N.pure (A.tag t ats) ⊗_N f
  let input = N.pure (fun n → A.tag "input" [("name", n)]
                                (A.pure (E.lookup n))) ⊗_N N.next_name
  let run v = let xml, collector = A.run (N.run v) in (xml, E.run collector)
end
```

**Fig. 8.** The formlet idiom (factored)

To work with a composed idiom, we need to be able to lift the primitive operations from the component idioms into the composed idiom. Given idioms $F$ and $G$, we can lift any idiomatic computation of type $\alpha\, G.t$ to an idiomatic computation of type $(\alpha\, G.t)\ F.t$ using $F.pure$, or lift one of type $\alpha\, F.t$ to one of type $(\alpha\, G.t)\ F.t$ using $Compose(F)(G).refine$.

In defining the composed formlet idiom, a combination of $N.pure$ and $AE.refine$ is used to lift the results of the $A.xml$ and $A.text$ operations. The *tag* operation is lifted differently as its third argument is a formlet: here we apply the $A.tag\ t\ ats$ operation to it. The *run* operation simply runs each of the primitive *run* operations in turn. The *input* operation is the most interesting. It generates a fresh name and uses it both to name an input element and, in the collector, for lookup in the environment.

### 3.4 A note on monads

Monads [3, 20, 29] are a more standard semantic tool for reasoning about side-effects. However, it is not difficult to see that there is no monad corresponding to the formlet type. Intuitively, the problem is that a *bind* operation for formlets would have to read some of the input submitted by the user before the formlet had been rendered, which is clearly impossible. (Recall that the type of *bind* would be $\alpha\ formlet \to (\alpha \to \beta\ formlet) \to \beta\ formlet$ and to implement this would require extracting the $\alpha$ value from the first argument to pass it to the second argument; but the rendering of the $\beta\ formlet$ should not depend on the $\alpha$-type data submitted to the first formlet.)

Every monad is an idiom, though of course, being oblivious, the idiom interface is less powerful (see Lindley and others [17] on the relative expressive power of idioms, arrows and monads). Although the idioms in Fig. 6 are in fact also monads, their composition (the formlet idiom) is not a monad: although idioms are closed under composition, monads are not. Using monad transformers in place of functor composition recovers some compositionality, but there is no combination of monad transformers that layers these effects in the right order.

## 4 Syntax

The syntax presented in Section 2 can be defined as syntactic sugar, which desugars into uses of the basic formlet operations. Here we formally define the syntax and its translation. We add two new kinds of expression: XML quasiquotes, (or XML literals with embedded evaluable expressions), and formlet expressions, denoting formlet values. Fig. 9 gives the grammar for these expressions.

The desugaring transformations are shown in Fig. 10. The operation $[\![\cdot]\!]$ desugars the formlet expressions in a program; it is a homomorphism on all syntactic forms except XML quasiquotes and formlet expressions. The operation $(\cdot)^*$ desugars XML quasiquotes and nodes. The operation $z^\dagger$ denotes a pattern aggregating the sub-patterns of $z$ where $z$ ranges over formlet quasiquotes and nodes. In an abuse of notation, we also let $z^\dagger$ denote the expression that reconstructs

Expressions

$$
\begin{array}{lll}
e ::= \cdots \mid r & & \text{(XML)} \\
\quad \mid \mathsf{formlet}\ q\ \mathsf{yields}\ e & & \text{(formlet)}
\end{array}
$$

XML quasiquotes

$$
\begin{array}{lll}
m ::= s \mid \texttt{\{}e\texttt{\}} \mid \texttt{<}t\ ats\texttt{>}m_1\ \ldots\ m_k\texttt{</}t\texttt{>} & & \text{node} \\
r \ ::= \texttt{<}t\ ats\texttt{>}m_1\ \ldots\ m_k\texttt{</}t\texttt{>} \mid \texttt{<\#>}m_1\ \ldots\ m_k\texttt{</\#>} & & \text{quasiquote}
\end{array}
$$

Formlet quasiquotes

$$
\begin{array}{lll}
n ::= s \mid \texttt{\{}e\texttt{\}} \mid \texttt{\{}f \Rightarrow p\texttt{\}} \mid \texttt{<}t\ ats\texttt{>}n_1\ \ldots\ n_k\texttt{</}t\texttt{>} & & \text{node} \\
q \ ::= \texttt{<}t\ ats\texttt{>}n_1\ \ldots\ n_k\texttt{</}t\texttt{>} \mid \texttt{<\#>}n_1\ \ldots\ n_k\texttt{</\#>} & & \text{quasiquote}
\end{array}
$$

Meta variables

| | | | | | |
|---|---|---|---|---|---|
| $e$ | expression | $f$ | formlet-type expression | $t$ | tag |
| $p$ | pattern | $s$ | string | $ats$ | attribute list |

**Fig. 9.** Quasiquote syntax.

the value matched by the pattern. (Of course, we need to be somewhat careful in the OCaml implementation to properly reconstruct the value from the matched pattern.) Finally, $z^\circ$ is a formlet that tuples the outcomes of sub-formlets of $z$.

As a simple example of desugaring, consider the definition of the *input_int* formlet used earlier:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{input\_int} : \mathit{int\ formlet} = \\
\quad \mathsf{formlet}\ \texttt{<\#>}\{\mathit{input} \Rightarrow i\}\texttt{</\#>}\ \mathsf{yields}\ \mathit{int\_of\_string}\ i
\end{array}
$$

Under the translation given in Fig. 10, the body becomes

$$\mathit{pure}\ (\mathsf{fun}\ i \rightarrow \mathit{int\_of\_string}\ i) \otimes (\mathit{pure}\ (\mathsf{fun}\ i \rightarrow i) \otimes \mathit{input})$$

We can use the idiom laws (and $\eta$-reduction) to simplify the output a little, giving the following semantically-equivalent code:

$$\mathit{pure}\ \mathit{int\_of\_string} \otimes \mathit{input}$$

As a richer example, recall *date_formlet* from Fig. 1 and its desugaring in Fig. 2. We could easily optimise the desugared code by removing the extra units from the body of the inner *pure* and from the arguments to the function in the outer *pure*. One thing we cannot do is avoid the rebinding of *month* and *day*. Section 5.3 outlines an alternate desugaring that obviates this rebinding.

**Completeness** Everything expressible with the formlet operations can be expressed directly in the syntax. For example, the $\otimes$ operator of the formlet idiom may be written as a function *ap* using syntactic sugar:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{ap} : (\alpha \rightarrow \beta)\ \mathit{formlet} \rightarrow \alpha\ \mathit{formlet} \rightarrow \beta\ \mathit{formlet} = \\
\quad \mathsf{fun}\ f\ p \rightarrow \mathsf{formlet}\ \texttt{<\#>}\{f \Rightarrow g\}\{p \Rightarrow q\}\texttt{</\#>}\ \mathsf{yields}\ g\ q
\end{array}
$$

Under the desugaring transformation, the body becomes

$$
\begin{aligned}
[\![r]\!] &= r^* \\
[\![\mathsf{formlet}\ q\ \mathsf{yields}\ e]\!] &= \mathit{pure}\,(\mathsf{fun}\ q^\dagger \to [\![e]\!])\ \otimes\ q^\circ \\
\\
s^* &= \mathit{xml\_text}\ s \\
\{e\}^* &= [\![e]\!] \\
(\texttt{<}t\ \mathit{ats}\texttt{>}m_1\ \ldots\ m_k\texttt{</}t\texttt{>})^* &= \mathit{xml\_tag}\ t\ \mathit{ats}\ (\texttt{<\#>}m_1\ \ldots\ m_k\texttt{</\#>})^* \\
(\texttt{<\#>}m_1\ \ldots\ m_k\texttt{</\#>})^* &= m_1^*\ \texttt{@}\ \cdots\ \texttt{@}\ m_k^* \\
\\
s^\circ &= \mathit{text}\ s \\
\{e\}^\circ &= \mathit{xml}\ [\![e]\!] \\
\{f \Rightarrow p\}^\circ &= [\![f]\!] \\
(\texttt{<}t\ \mathit{ats}\texttt{>}n_1\ \ldots\ n_k\texttt{</}t\texttt{>})^\circ &= \mathit{tag}\ t\ \mathit{ats}\ (\texttt{<\#>}n_1\ \ldots\ n_k\texttt{</\#>})^\circ \\
(\texttt{<\#>}n_1\ \ldots\ n_k\texttt{</\#>})^\circ &= \mathit{pure}\ (\mathsf{fun}\ n_1^\dagger\ \ldots\ n_k^\dagger \to (n_1^\dagger, \ \ldots\ , n_k^\dagger)) \otimes n_1^\circ\ \cdots\ \otimes n_k^\circ \\
\\
s^\dagger &= \texttt{()} \\
\{e\}^\dagger &= \texttt{()} \\
\{f \Rightarrow p\}^\dagger &= p \\
(\texttt{<}t\ \mathit{ats}\texttt{>}n_1\ \ldots\ n_k\texttt{</}t\texttt{>})^\dagger &= (n_1^\dagger, \ldots, n_k^\dagger) \\
(\texttt{<\#>}n_1\ \ldots\ n_k\texttt{</\#>})^\dagger &= (n_1^\dagger, \ldots, n_k^\dagger)
\end{aligned}
$$

**Fig. 10.** Desugaring XML and formlets.

$$(\mathit{pure}\ (\mathsf{fun}\ (g,\ q) \to g\ q)) \otimes (\mathit{pure}\ (\mathsf{fun}\ g\ q \to (g,\ q)) \otimes f \otimes p)$$

which, under the idiom laws, is equivalent to $f \otimes p$. And *pure*, too, can be defined in the sugar as $\mathsf{fun}\ x \to \mathsf{formlet}\ \texttt{<\#></\#>}\ \mathsf{yields}\ x$. This shows that the syntax is complete for the formlet operations.

## 5 Extensions

The formlet abstraction is robust, as we can show by extending it in several independent ways.

### 5.1 XHTML validation

The problem of statically enforcing validity of HTML and indeed XML is well-studied [4, 13, 21, 27]. Such schemes are essentially orthogonal to the work presented here: we can incorporate a type system for XML with little disturbance to the core formlet abstraction.

Of course, building static validity into the type system requires that we have a whole family of types for HTML rather than just one. For instance, we might have separate types for `block` and `inline` entities (as in Elsman and Larsen's system [9]), or even a different type for every tag (as in XDuce [13]).

Fortunately, it is easy to push the extra type parameters through our formlet construction. The key component that needs to change is the *XmlWriter* idiom.

As well as the value type, this now needs to be parameterised over the XML type. The construction we need is what we call an *indexed idiom*. It is roughly analogous to an effect-indexed monad [30]. In OCaml, we define an indexed idiom as follows:

```
module type XIdiom = sig
  type (ψ, α) t
  val pure : α → (ψ, α) t
  val (⊗) : (ψ, α → β) t → (ψ, α) t → (ψ, β) t
end
```

(For the indexed XML writer idiom the parameter $\psi$ is the XML type.) Like idioms, indexed idioms satisfy the four laws given in Section 3. They can be pre- and post-composed with other idioms to form new indexed idioms. Pre-composing the name generation idiom with the indexed XML writer idiom pre-composed with the environment idiom gives us an indexed formlet idiom.

As a proof of concept, we have implemented a prototype of formlets with XML typing in OCaml using Elsman and Larsen's encoding of a fragment of XHTML 1.0 [9]. It uses phantom types to capture XHTML validity constraints.

### 5.2 Input validation

A common need in form processing is validating user input: on submission, we should ensure that the data is well-formed, and if not, re-display the form to the user (with error messages) until well-formed data is submitted.

Formlets extend to this need if we incorporate additional idioms for error-checking and accumulating error messages and add combinators *satisfies* and *err*, which add to a formlet, respectively, an assertion that the outcome must satisfy a given predicate and an error message to be used when it does not. Any time the continuation associated with a formlet is invoked, the outcome is sure to satisfy the validation predicate(s).

The need to re-display a page upon errors also requires additional mechanics. Instead of simply attaching a continuation to a formlet and rendering it to HTML, the formlet continuation now needs to have a complete page context available to it, in case it needs to redisplay the page. To facilitate this, we add a new syntactic form, which associates formlets with their continuations *in the context of* a larger page.

Extending with input validation adds some complexity to the implementation, so we omit details here. We have implemented it in the Links version of formlets and provide details in a technical report [7].

### 5.3 Multi-holed contexts

The presentation of formlets we have given in this paper relies on lifting the *tag* constructor from the *XmlWriter* idiom into the *Formlet* idiom. As illustrated by the desugaring of the date example in Section 4 this makes it difficult to separate

the raw XML from the semantic content of formlets and requires nested formlet values to be rebound.

Besides obfuscating the code, this rebinding is inefficient. By adapting the formlet datatype to accumulate a list of XML values rather than a single XML value, and replacing *tag* with a general operation for plugging the accumulated list into a multi-holed context *plug*, we obtain a more efficient formlet implementation that does provide a separation between the raw XML and the semantic content. Further, this leads to a much more direct desugaring transformation. For example, the desugared version of the date example becomes:

```
let date_formlet : (_, date) NFormlet.t =
  plug (tag "div" [] (text "Month: " @ hole @ text "Day: " @ hole))
    (pure (fun month day → make_date month day) ⊗ input_int ⊗ input_int)
```

Statically typing *plug* in OCaml requires some ingenuity. Using phantom types, we encode the number of holes in a context, or the number of elements in a list, as the difference between two type-level Peano numbers [16]. As with XHTML typing the key component that needs to change is the *XmlWriter* idiom. This now needs to be parameterised over the number of XML values in the list it accumulates. The construction we need is the what we call a *parameterised idiom*, the idiom analogue of a parameterised monad [1]. In OCaml, we define a parameterised idiom as follows:

```
module type PIdiom = sig
  type (μ, ν, α) t
  val pure : α → (μ, ν, α) t
  val (⊗) : (μ, ν, α → β) t → (σ, μ, α) t → (σ, ν, β) t
end
```

(For the parameterised XML writer idiom the parameters $\mu$ and $\nu$ encode the length of the list of XML values as $\nu - \mu$.) Like idioms, and indexed idioms, parameterised idioms satisfy the four laws given in Section 3. They can be pre- and post-composed with other idioms to form new parameterised idioms. Pre-composing the name generation idiom with the parameterised XML writer idiom pre-composed with the environment idiom gives a parameterised formlet idiom.

We have implemented a prototype of formlets with a multi-holed plugging operation in OCaml. Statically-typed multi-holed contexts can be combined with statically typed XHTML [16]. Lifting the result to idioms gives either an *indexed parameterised idiom*—that is, an idiom with an extra type parameter for the XML type and two extra type parameters for the number of XML values in the accumulated list—or, by attaching the XML type to both of the other type parameters, a parameterised idiom.

### 5.4 Other extensions

These are by no means the only useful extensions to the basic formlet abstraction. For example, we might wish to translate validation code to JavaScript to run on the client [12], or enforce separation between those portions of the program that deal with presentation and those that treat application-specific computation, a

common requirement in large web projects. Either of these may be combined with the formlet abstraction without injury to the core design presented here.

## 6 Related work

The WASH, iData and WUI frameworks all support aspects of the form abstraction we have presented. WUI, in fact, meets all of the goals listed in the introduction. Underlying all these systems is the essential mode of form abstraction we describe, although they vary richly in their feature sets and limitations.

**WASH** The WASH/CGI Haskell framework [28] supports a variety of web application needs, including forms with some abstraction. WASH supports user-defined types as the result of an individual form field, through defining a `Read` instance, which parses the type from a string. It also supports aggregating data from multiple fields using a suite of tupling constructors, but it does not allow arbitrary calculations from these multiple fields into other data types, such as our abstract *date* type. In particular, the tupling constructors still expose the structure of the form fields, preventing true abstraction. For example, given a one-field component, a programmer cannot modify it to consist of two fields without also changing all the uses of the component.

**iData** The iData framework [23] supports a high degree of form abstraction, calling its abstractions *iData*. Underlying iData is an abstraction much like formlets. Unlike formlets, where form abstraction is separated from control flow (the function *handle* attaches a handler to a formlet), iData have control flow baked in. An iData program defines a single web page consisting of a collection of interdependent iData. Whenever a form element is edited by the user, the form is submitted and then re-displayed to the user with any dependencies resolved. The iTasks library [24] builds on top of iData by enabling or disabling iData according to the state of the program.

**WUI** The WUI (*Web User Interface*) library [11, 12] implements form abstractions for the functional logic programming language Curry. Here the basic units are called WUIs. WUIs enforce an assumption that each WUI of type $\alpha$ should accept a value of type $\alpha$ as well as generate one; this input value models the default or current value for the component. Thus a *WUI* $\alpha$ is equivalent, in our setting, to a value of type $\alpha \to \alpha$ *formlet*.

## References


1. Robert Atkey. Parameterised notions of computation. In *MSFP*, 2006.
2. Vincent Balat. Ocsigen: typing web interaction with objective caml. In *ML Workshop '06*, pages 84–94, 2006.
3. Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *Applied Semantics: Advanced Lectures*, volume 2395 of *LNCS*, pages 42–122, 2002.
4. Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *PASTE*, pages 38–45, 2001.
5. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level web service construction. *TOPLAS*, 25(6):814–875, 2003.

6. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *FMCO '06*, volume 4709 of *LNCS*, pages 266–296, 2007.
7. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. An idiom's guide to formlets. Technical Report EDI-INF-RR-1263, University of Edinburgh, 2008.
8. Chris Eidhof. Formlets in Haskell, 2008. `http://blog.tupil.com/formlets-in-haskell/`.
9. Martin Elsman and Ken Friis Larsen. Typing XHTML web applications in ML. In *PADL '04*, pages 224–238, 2004.
10. Paul T. Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the web with high-level programming languages. In *ESOP '01*, pages 122–136, 2001.
11. Michael Hanus. Type-oriented construction of web user interfaces. In *PPDP '06*, pages 27–38, 2006.
12. Michael Hanus. Putting declarative programming into the web: Translating Curry to JavaScript. In *PPDP '07*, pages 155–166, 2007.
13. Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.
14. John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.
15. Lift website, March 2008. `http://liftweb.net/`.
16. Sam Lindley. Many holes in Hindley-Milner. In *ML Workshop '08*, 2008.
17. Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. In Venanzio Capretta and Conor McBride, editors, *MSFP '08*, Reykjavik, Iceland., 2008.
18. Conor McBride. Idioms, 2005. Presented at SPLS June 2005 `http://www.macs.hw.ac.uk/~trinder/spls05/McBride.html`.
19. Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), 2008.
20. Eugenio Moggi. Computational lambda-calculus and monads. In *LICS '89*, pages 14–23, 1989.
21. Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *ICDT '05*, 2005.
22. PHP Hypertext Preprocessor, March 2008. `http://www.php.net/`.
23. Rinus Plasmeijer and Peter Achten. iData for the world wide web: Programming interconnected web forms. In *FLOPS '06*, pages 242–258, 2006.
24. 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.
25. Ruby on Rails website, March 2008. `http://www.rubyonrails.org/`.
26. Steve Strugnell. Creating linksCollab: an assessment of Links as a web development language. BSc thesis, University of Edinburgh `http://groups.inf.ed.ac.uk/links/papers/undergrads/steve.pdf`, 2008.
27. Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *J. Funct. Program.*, 12(4&5):435–468, 2002.
28. Peter Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. Inter. Tech.*, 5(1):1–46, 2005.
29. Philip Wadler. Monads for functional programming. In *Advanced Functional Programming '95*, volume 925 of *LNCS*, pages 24–52, 1995.
30. Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Log.*, 4(1):1–32, 2003.