Galactic: an Ur/Web Course Management System

by

Patrick M. Hurst II

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

© Patrick M. Hurst II, MMXIV. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2014

Certified by .....
Adam Chlipala
Assistant Professor
Thesis Supervisor

Accepted by .....
Albert R. Meyer
Chairman, Department Committee on Graduate Theses

Galactic: an Ur/Web Course Management System

by

Patrick M. Hurst II

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Ur/Web is a modern ML-like Web-oriented programming language with a type system that both ensures program correctness and provides powerful metaprogramming capabilities. This thesis describes techniques that allow construction of modular applications where the individual modules can be specified simply, using metaprogramming and module functors to generate the actual code from a few simple lines of mostly declarative code. In particular, I present a novel technique for indirect cross-module recursion and a flexible abstraction for form elements.

Thesis Supervisor: Adam Chlipala
Title: Assistant Professor

Acknowledgments

I would like to express my gratitude to Professor Chlipala for his supervision and useful advice throughout this thesis project.

I would also like to thank my parents for their support throughout my entire life.

Contents

1 Introduction **11**

1.1 Background 11

1.2 Project goals 12

2 Background on Ur/Web **13**

2.1 Type-level metaprogramming 13

2.2 Functors 15

2.3 HTML literals 16

2.4 Functional reactive programming 18

3 Goals **21**

3.1 Modularity 22

3.2 Modern interface 22

4 Building Galactic **25**

4.1 Variant-based recursion 25

4.2 GUI-building abstraction 31

4.3 The 'material' abstraction 42

4.4 The 'gradable' abstraction 43

4.5 Templating 49

4.6 Datetime library 50

4.7 Data attributes 51

5 Related work **55**

6 Further work **59**

6.1 Nested variant-based recursion 59

6.2 Computer-generated modules 60

7 Results **61**

List of Figures

2-1 Escaped and unescaped HTML 17

4-1 A record-based handler approach to indirect recursion 26
4-2 Working with handlers in variant-based routing 30
4-3 The functor responsible for 'tying the knot' in variant-based recursion 31
4-4 Using the `Close` functor 32
4-5 A sample form element 32
4-6 The form element from figure 4-5 in the browser 33
4-7 A composite form element 33
4-8 The source code listing for a basic material module 44
4-9 The editor created from figure 4-8 45
4-10 The source code listing for the reading question module 47

Chapter 1

Introduction

1.1 Background

The modern Web is developed in many different languages; some of the most popular choices are Ruby, Python, PHP, and JavaScript (both for the client and the server in the form of Node.js). However, none of these languages have strong, statically checked type systems, which leaves them open to bugs such as broken links and ill-formed HTML, as well as more serious security vulnerabilities such as SQL injection, cross-site scripting, and other bugs that result from failing to properly sanitize user input. Web frameworks like Haskell's Yesod and its templating language can help with these problems using the type system to enforce correctness. But they fail to address another problem: keeping client-side AJAX in sync with the server-side handlers that process and respond to the requests. The easiest way to do so is to have a single, unified language for writing both client-side and server-side code; this is part of the driving motivation behind Node.js, but it lies under the dynamic weak typing regime and therefore falls victim to the same security vulnerabilities in addition to the usual programming pitfalls of dynamic weak typed languages.

1.2 Project goals

The main goal of this project was to build a flexible Web application in Ur/Web [1], a strongly typed Web-oriented programming language, in order to develop a sense of how to build applications with modular components. These components should be easy to create; part of the strength of Ur/Web's type system lies in its inference, and a modular system that requires users to laboriously write out types is significantly less usable. I also wanted to test Ur/Web's integration with modern HTML frameworks such as Twitter Bootstrap to ensure that Ur/Web is capable of building applications with the 'Web 2.0' look many users have come to associate with modern Web applications. Part of that is using AJAX for form submissions instead of the non-JavaScript `<form>` approach. As a result, support for non-JavaScript clients was explicitly not a goal for the project; in a 2010 study [7], Yahoo! found that about 99% of all users had JavaScript enabled, so this is a safe assumption to make.

More information about Galactic, including links to the source repository, can be found at `http://plv.csail.mit.edu/galactic/`.

Chapter 2

Background on Ur/Web

My advisor, Adam Chlipala, developed the Ur/Web programming language in order to create a strong, statically typed functional programming language specifically oriented towards Web development. The syntax and type system are both similar to the ML family of languages, although there are several key semantic differences. Ur/Web encapsulates all potentially side-effectful operations in the `transaction` monad, similar to Haskell's `IO` monad; the name is derived from the fact that all actions in the `transaction` monad should be able to be rolled back.

2.1 Type-level metaprogramming

One of Ur/Web's main features is its metaprogramming; Ur/Web values are typically represented as records, and 'folder' values provide the ability to iterate over the fields of a record, building a new value along the way. In a formal type-theoretical framework, a folder is essentially an abstracted catamorphism over the record. The compiler can construct a folder for any concrete type automatically; for more complicated types, the standard library provides functions that can be used to build up folders. The type system allows the type of the built-up value to vary with the fields of the folded value. For example, Ur/Web has an `option` type that represents an optional value, analogous to `option` in SML or `Maybe` in Haskell; if all the fields in a record are wrapped in an `option`, then a function to attempt to 'unwrap' them all,

succeeding only if each wrapped value is in fact present, can be written as follows:

```
fun join [ts ::: {Type}] (fl : folder ts) (r : $(map option ts))
  : option $ts =
    @foldR [option] [fn ts => option $ts]
     (fn [nm ::_] [v ::_] [r ::_] [[nm] ~ r] v vs =>
         case (v, vs) of
             (Some v, Some vs) => Some ({nm = v} ++ vs)
           | _ => None)
     (Some {}) fl r
```

The first line of the code declares `join` as a function that takes three arguments. The first argument is a type-level argument (since it is enclosed in square brackets) and has kind `{Type}`, meaning a record of types. The triple colon bracket indicates an optional argument; the Ur/Web compiler will attempt to infer it if possible. The second argument is a value of type `folder ts`; note that the types of value-level arguments can refer to previous type-level arguments. `folder` arguments are special in that the compiler will attempt to infer them automatically. The third argument is of type `$(map option ts)`; `map option ts` applies the `option` type constructor to every type in the `ts` record, and `$` is an operator that builds a type out of a record of types. For example, if `con person = [Name = string, Age = int]` is a type-level record representing a person, we have `$person = {Name : string, Age : int}` and `val alyssa : $person = {Name = "Alyssa P. Hacker", Age = 21}`.

Here, the standard library function `foldR` is being applied; the `R` in the name indicates that it folds over records. Other functions in the same family include `foldUR`, which folds over records where the values are the same type, and `foldR2`, which folds over pairs of records with the same field names. The `@` prefix indicates that folders and disjointness arguments, which are normally inferred, are being explicitly passed; `@@` would indicate that optional type-level arguments are being explicitly passed as well.

A `foldR` operates over some record whose type is derived from some 'base' type (in this case `fs`); we pass `[option]` to indicate that all the fields in the record we are folding over have had `option` applied to them, and `[fn ts => option $ts]` to indicate that the result will be of type `option $fs`.

The third argument is the fold function; it takes in the name of the field, the type of the value, the type of the accumulator, a proof that the field is not present in the accumulator, the value, and the accumulator itself and simply combines the value and the accumulator. The `t` and `r` arguments are used implicitly in order to determine the types of the value-level arguments `v` and `vs`, but are not used explicitly. The disjointness is implicitly used by `++`.

Finally, with `(Some {}) fl r` we specify the initial value, the folder, and the value to be folded over.

2.2 Functors

In addition to metaprogramming via type-level arguments, Ur/Web also has support for objects that function similarly to functions over modules, known as functors (which have no relation to functors in Haskell or C++). A functor takes in a module and outputs a module, similarly to how a function takes in a value and outputs a value. As a simple example, consider the following Ur/Web snippet from `refFun.urs` in the Ur/Web standard demo, which constructs a notion of 'reference' that supports the standard create/read/update/delete actions:

```
functor Make(M : sig
                 type data
                 val inj : sql_injectable data
             end) : sig
    type ref
    val new : M.data -> transaction ref
    val read : ref -> transaction M.data
    val write : ref -> M.data -> transaction unit
```

```
    val delete : ref -> transaction unit
end
```

The name of the functor is `Make`, and the functor's argument is named `M` (both of these are standard conventions). The argument must contain both a type named `data`, and a value named `inj`; here `inj` serves as a 'witness' that the type `data` can be inserted into a SQL table column. The functor 'returns' a module with one type-level component named `ref` and four functions. The module can be used like so:

```
structure IR = RefFun.Make(struct
                               type data = int
                           end)

fun mutate () =
    ir <- IR.new 3;
    IR.write ir 10
```

The `inj` is a typeclass argument and so can be inferred in this case, since there is only one value in scope of type `sql_injectable int`. The functor is invoked by calling it with an anonymous structure as its argument, and then assigned to the name `IR`; functions from the `IR` module can then be used using the standard qualified name syntax.

2.3 HTML literals

Ur/Web has native support for HTML literals:

```
val foo = "This is some string data! It will be <b>escaped</b>."
val bar : xbody = <xml>
  <p><b>This will actually be bold.</b> {[foo]}</p>
</xml>
val main : transaction page = return <xml>
```

```
  <body>
    Here's some HTML inlined: <div>{bar}</div>
  </body>
</xml>
```

Figure 2-1 shows how the above Ur/Web code will be actually rendered in the browser: while the bold tags included in the HTML literal `bar` show up directly in the output, those included in `foo` are escaped. This is because Ur/Web has two different types of interpolation into HTML:

- `{[foo]}` takes a string, or any value that Ur/Web can turn into a string, and entity-escapes it (e.g., replaces < with <) before inserting it into the surrounding HTML.

- `{bar}` inserts the value `bar` directly into the surrounding HTML; this can only be done with values that represent HTML. This prevents a whole class of attacks where user-inputted data is included without escaping in server responses, which can, e.g., allow a malicious user to execute JavaScript on the site of anybody who visits their profile. The HTML literals actually are syntax sugar for language primitives whose names correspond to the names of the tags.

Ur/Web also has some notion of what contexts a tag is permitted in; the `xbody` type corresponds to HTML that can be inserted inside a `body` tag, and `page` corresponds to an entire HTML document. For example, the `<head>` tag can only be used in a `page`, not an `xbody`.

Here's some XML inlined:

This will actually be bold. This is some string data! It will be escaped.

Figure 2-1: Escaped and unescaped HTML

2.4 Functional reactive programming

In functional reactive programming, or FRP, time-varying data is not explicitly computed; rather, it's specified as a function of other time-varying data. For example, in a spreadsheet program such as Excel, a formula such as `A1+B2` indicates that the cell's value should be automatically updated whenever either of the two dependent cells, `A1` or `B2`, has its value changed; the program then does the necessary bookkeeping behind the scenes to ensure that this update automatically takes place.

Ur/Web's functional reactive programming model is based around sources and signals. A source represents some kind of data source; sources are exclusively client-side, and the main way to use them is to 'connect' them to a form input element via pseudo-tags such as `ctextbox` that create an element and associate a source with it. (Sources also support `get` and `set` operators, making them useful for shared mutable global state.)

Sources themselves cannot be used for FRP; since a source typically corresponds exactly to the state of some input widget such as a text area or a radio checkbox, there is no way to 'apply' a function to a source and get another source. Instead, they have to be converted to a `signal` via the function `signal : t ::: Type -> source t -> signal t`. `signal` forms a monad; that is, these two functions exist:

```
val signal_bind : a ::: Type -> b ::: Type -> signal a
                  -> (a -> signal b) -> signal b
val signal_return : a ::: Type -> a -> signal a
```

and these functions satisfy identities known as the 'monad laws'. [6] gives an in-depth overview of how monads can be used in functional programming; for our purposes, the important characteristic is that the monadic nature of `signal` lets us write code like the following, which takes two values of type `signal int` and produces a `signal int` corresponding to their sum:

```
fun signalSum a b =
    a' <- a;
```

```
b' <- b;
return (a' + b')
```

This code will be desugared by the compiler into an expression using `signal_bind` and `signal_return`.

Chapter 3

Goals

At an abstract level, my goal for this project was to investigate ways of implementing modularity in information-management systems. Many systems boil down to being able to create, read, update, and delete information stored in some form of persistent storage. For example, a system that lets a privileged class of users upload documents that any user can read must let privileged users create new documents as well as update or delete existing ones, as well as let unprivileged users read (and browse a list of) documents. A system with many different kinds of information should present a uniform interface, both to the programmer and to the user, for each kind.

Conversely, these components should not be completely isolated from each other. For example, it may be useful to have one component that summarizes data from other components; e.g., a calendar that lists upcoming events such as due dates or quizzes. In this case, the calendar's items should also contain backlinks: clicking the name of an assignment should take the user to the page for that assignment.

Course management systems help teachers administrate a class by providing a central location for them to upload homework assignments, reading material, etc., as well as giving students a convenient way to check their grades, upload their submissions for assignments, and otherwise interact with the course. They therefore provide a good test application for exploring modularity in Ur/Web. In this project I focused on aspects that relate to management of course-related data, such as quizzes, reading material, and homework. I considered components such as a forum or a question-and-

answer service, though potentially useful and interesting to investigate, to be outside the scope of this thesis.

3.1 Modularity

A common pattern in a course management system is the notion of something that teachers can create and edit and students can only view. One of the most obvious examples of this is reading material; teachers can upload new files as reading material, and students can then download them. Ideally, the system would not make significant assumptions about the structure of such objects; while allowing new object types to be created at run-time by course administrators may be difficult, it should not be difficult to create new compile-time modules.

Furthermore, it should be possible to specify new types of objects while writing minimal amounts of logic. For example, suppose a teacher wants to give students a discussion prompt that they should respond to before each class. This response should take the form of a paragraph or two of text. This is a simple use case, and so the administrator for the course Web site should not have to do anything complicated in order to add this functionality to their Galactic instance. In particular, I wanted to adhere to one of the principles of the original Ur/Web paper [1]: "The users of a metaprogram should need to write neither proofs nor types more complex than those found in mainstream programming languages."

3.2 Modern interface

The website should also be easy to use from the student's point of view and fit in with modern principles of Web design. Going back to the reading question example, there should be client-side JavaScript that validates the student responses and disallows the students from submitting completely empty responses (although server-side validation should also be done). Similarly, modern form design usually involves sending an AJAX request when the 'submit' button is clicked and displaying the success or

failure of the submission in an HTML pop-up or JavaScript alert box, rather by redirecting the user to a new page.

I also wanted to make the site conform to a modern Web design aesthetic; I therefore chose Twitter's Bootstrap 3.0 as a base for the Galactic CSS. Bootstrap is a fairly popular front-end design framework with both CSS and JavaScript components that make things such as navigation bars and nicely laid-out forms easy to design. It would also provide a good test for seeing how Ur/Web's HTML generation interacts with preexisting CSS toolkits, which can be rather opinionated on HTML structure and class names.

Chapter 4

Building Galactic

My efforts in building Galactic were primarily focused around setting up a system for teachers to upload content, optionally allowing students to respond and be graded. I therefore created two broad types of content, each of which is implemented as a functor:

- A 'material' is anything that a teacher can create one or more instances of, such as reading material.
- A 'gradable' is a material that students need to respond to, such as reading questions, homework, or group projects.

In order for these content types to be editable, I developed a form-building abstraction where each form element can model any number of fields, including zero; form elements can be combined, resulting in their fields being concatenated and their HTML output being joined.

4.1 Variant-based recursion

Ur/Web does not support mutually recursive functions across module boundaries; if two functions are mutually recursive, they need to be part of the same `val rec` declaration. This can cause difficulty when building GUIs using the GUI-building

```
con recHandler (requires :: {Type}) (param :: Type) =
    $(map (fn t => t -> transaction page) requires) -> param
      -> transaction page

con recHandlers (requires :: {Type}) (provides :: {Type}) =
    $(map (handler requires) provides)

val fooRecHandlers : recHandlers [Bar = int] [Foo = int] = {
  Foo = fn r x => return <xml><body>
    The current value of x is {[x]}.
    <a link={r.Bar (x + 1)}>This link goes to Bar</a>.
    </body></xml>
}

val barRecHandlers : recHandlers [Foo = int] [Bar = int] = {
  Bar = fn r x => return <xml><body>
    The current value of x is {[x]}.
    <a link={r.Foo (x + 1)}>This link goes to Foo</a>.
  </body></xml>
}
```

Figure 4-1: A record-based handler approach to indirect recursion

abstraction that I developed. One concrete example is in the 'submit' page that the `Gradable` module generates; we want the `submit` page to have a link back to the page for the assignment itself. It is possible to generate URLs via string manipulation and invoke `bless : string -> url`, which checks the URL against a compile-time-specified list of allowed patterns, but this would defeat part of the purpose of using Ur/Web in the first place. Instead, I asked my advisor for a solution, and he gave me the first draft of an approach that I've decided to call 'variant-based recursion'.

4.1.1 Handler records

A naive approach to the lack of inter-module recursion is to not expose functions of type `t -> transaction page` directly; instead, the functions should take a function of type `handler -> t -> transaction page`, where `handler` is some record of functions that provides 'routing'.

Figure 4-1 is a simple example of this approach with two mutually recursive han-

dlers. While for the sake of simplicity this example is located entirely in one file, the `fooRoute` and `barRoute` declarations do not depend on each other and so they can be located in separate files. `recHandler` and `recHandlers` are utility type constructors; the first argument to `recHandler` indicates the kind of routes that the handler requires, and the second argument is the parameter. For example, `fooRecHandlers` depends on a handler named `Bar` that takes an `int` as a parameter and provides one named `Foo` that takes an `int` as a parameter.

Next, we need some way to actually 'tie the knot' on these records, so to speak; if we have some set of handlers that provides everything it requires, we need some way to construct the `r` values that the individual handlers require. This is accomplished via the `recClose` function:

```
fun recClose [provides ::: {Type}] (fl : folder provides)
          (handlers : recHandlers provides provides)
    : $(map (fn t => t -> transaction page) provides) =
    @mp [recHandler provides] [fn t => t -> transaction page]
     (fn [t] f param => f (@recClose fl handlers) param) fl handlers
```

This function is dense, and requires some explanation. `mp` here has the type (ignoring the type-level parameters):

```
mp :: (t ::: Type -> recHandler provides t -> (t -> transaction page))
      -> (recHandlers provides provides)
      -> $(map (fn t => t -> transaction page) provides)
```

In other words, its first argument is a function that given a handler requiring `provides` and a `t`, produces a page; its second argument is a 'closed' set of handlers providing and requiring `provides`, and it produces a record with one field for each field in `provides` that takes that field's type and produces a `transaction page`.

Producing the function is the tricky part: the type expands into

```
t ::: Type
```

```
  -> ($(map (fn t => t -> transaction page) provides) -> t
      -> transaction page)
  -> t -> transaction page
```

The only way to construct a function of this type is to provide a value of type `$(map (fn t => t -> transaction page) provides)`. But that's just the return value of our `close` invocation, so we can pass it back into itself!

Finally, we need some way to combine handlers. This is relatively simple: we write a 'weakener' that makes a record of handlers require more things, and then we can combine them via simple record concatenation:

```
fun recWeakens [requires ::: {Type}] [requires' ::: {Type}]
               [provides ::: {Type}] [requires ~ requires']
    (fl : folder requires) (fl2 : folder provides)
    (handlers : recHandlers requires provides)
    : recHandlers (requires ++ requires') provides =
    @mp [recHandler requires] [recHandler (requires ++ requires')]
      (fn [u] h r => h (r --- map (fn t => t -> transaction page)
                                  requires'))
      fl2 handlers
```

The majority of the complexity of `recWeakens` is in its type signature: given a handler that has a set of requirements, we can build a handler that requires more things by taking the record of handlers and removing the extra things before passing it to the original handler; we then do this for each individual handler.

Finally, we can weaken and combine our handler records, close them, and hook the resulting record up to `main`:

```
val closed : {Foo : int -> transaction page,
              Bar : int -> transaction page} =
    recClose (recWeakens fooRecHandlers ++ recWeakens barRecHandlers)

val main = closed.Foo 0
```

. . . but if we try to compile the above code, we get a compiler error! Although our code successfully type-checks, the compiler is unable to generate valid code from it; the specific error message indicates that it's unable to actually generate links. There are two reasons for this inability to generate code: one is that we should have the handler dependencies generate values of type `url` instead of `transaction page`. In general values of type `url`, which represent a URL, are easier to work with than values of type `transaction page`, which usually represent a handler that will be run server-side. The second reason is that Ur/Web's URL generation mechanism cannot correctly handle dispatching between the fields of a large record like this in a case where the record's structure isn't 'known' to the `recClose` function ahead of time. The solution is to use variant types.

4.1.2 Handler variants

Any record of types `t :: {Type}` has a corresponding variant type `variant t`, which has one constructor for each field in the record which takes an argument corresponding to the type of that field. Variant values are constructed using `make` and consumed using `match`, like so:

```
con t :: {Type} = [ Foo = int, Bar = string ]
val var : variant t = make [#Foo] 8
val matched : string = match var { Foo = show, Bar = fn x => x }
(* matched will be "8" *)
```

The first argument to `match` is the variant to consume, and the second is a record with fields corresponding to the original type; each field's type maps the original record's type to some fixed type. In this case, each value in the match record maps the argument to a `string`.

In particular, instead of `$(map (fn t => t -> url) requires)` we can use a `variant requires -> url`. Seeing that the two types are identical conceptually is easy; each one gives you a way, given a field and a value in that field's type, to get a value of type `url`. Writing the handler code to use variants is fairly straightforward

```
type router (requires :: {Type}) = variant requires -> url

type handler (requires :: {Type}) (param :: Type) =
     router requires -> param -> transaction page

type handlers (requires :: {Type}) (provides :: {Type}) =
     $(map (handler requires) provides)

type closable (linkage :: {Type}) = handlers linkage linkage

fun weakenHandler [requires ::: {Type}] [requires' ::: {Type}]
                  [requires ~ requires'] (fl : folder requires)
                  [param ::: Type] (old : handler requires param)
    : handler (requires ++ requires') param  =
 fn r => old (fn x => r (@V.weaken ! fl x))

fun weaken [requires ::: {Type}] [unused ::: {Type}]
           [provides ::: {Type}] [requires ~ unused]
           (fl : folder provides)
           (fl2 : folder requires)
           (fs : handlers requires provides)  =
    @mp [handler requires] [handler (requires ++ unused)]
     (fn [u] f => @weakenHandler ! fl2 f) fl fs
```

Figure 4-2: Working with handlers in variant-based routing

for the most part; figure 4-2 is a straightforward translation of the record-based equivalents above.

We break out the weakener for an individual handler into a separate function for simplicity, and also define a new type synonym `closable` to reflect the type of handlers that provide everything they require and vice versa.

In figure 4-3 we 'tie the knot' on the indirectly recursive handlers. Perhaps the most obvious change is that `Close` is now a functor; this helps 'break' the recursive chain, so to speak. We define a `page` function that, given a `variant M.provides` (i.e., a handler and the parameter for that handler), will actually dispatch to the appropriate handler. `page` is defined in terms of the `routed'` function that does all the knot-tying work. The reason that it's a function now when it was a value before is that mutually recursive values must all be functions. We then export `routed`, which

```
functor Close(M : sig
                  con provides :: {Type}
                  val fl : folder provides
                  val fs : handlers provides provides
              end) : sig
    val routed : $(map (fn t => t -> transaction page) M.provides)
end
  = struct
    fun page (v : variant M.provides) : transaction page =
        match v (routed' ())

    and routed' () = (@mp [fn t => handler M.provides t]
                      [fn t => t -> transaction page]
                      (fn [t ::_] v => v (fn x => url (page x)))
                      M.fl M.fs)

    val routed = routed' ()
end
```

Figure 4-3: The functor responsible for 'tying the knot' in variant-based recursion

actually does all the routing.

Figure 4.1.2 demonstrates how to use the `Close` functor. The code compiles and work exactly as expected, even if the definitions of the two handlers are moved into separate modules: clicking a link in either one increments the value of `x` and sends you to the other. It generates URLs that look like `/Recursion/Closed/page/Foo/2`, which are slightly unwieldy and make the URL harder to read, but Ur/Web's URL-rewriting directives can be used to clean the generated URLs up somewhat.

4.2 GUI-building abstraction

4.2.1 GUI elements

Galactic GUIs are typically built out of primitives called 'form elements' which are defined in `gui.ur`. Originally, the form element type constructor as defined in `gui.urs` was opaque, with the intent being that the type is an implementation detail. However, I ultimately realized that constructing a GUI element library that is flexible

```
val fooHandlers : handlers [Bar = int] [Foo = int] = {
  Foo = fn r x => return <xml><body>
    The current value of x is {[x]}.
    <a href={r (make [#Bar] (x + 1))}>This link goes to Foo</a>.
    </body></xml>
}

val barHandlers : handlers [Foo = int] [Bar = int] = {
  Bar = fn r x => return <xml><body>
    The current value of x is {[x]}.
    <a href={r (make [#Foo] (x + 1))}>This link goes to Foo</a>.
  </body></xml>
}


structure Closed = Close(struct
  val fs = weaken fooHandlers ++ weaken barHandlers
end)

val main = Closed.routed.Foo 0
```

Figure 4-4: Using the `Close` functor

```
val questionEl : formElement [] unit [Question = string]
                 (source string) body
                 = textboxEl [#Question] "Question"
                             "What is the meaning of life?"
                             Util.nonEmpty
```

Figure 4-5: A sample form element

enough to handle all the use cases would be essentially impossible, and any 'wrapper' that would wrap user-specified handlers up in an opaque `formElement` type would require input that would essentially be isomorphic to the type itself.

Figure 4-5 shows the construction of a simple form element using the `textboxEl` function. The `formElement` type constructor takes five parameters; here we indicate that the element uses no routes (i.e., its routing record is the empty record `[]`), it uses no metadata, it provides a field named `Question` with type `string`, its internal state is a `source string`, and it can be used in any context inside a `<body>` element. The arguments to `textboxEl` are the name of the field that the textbox models, the

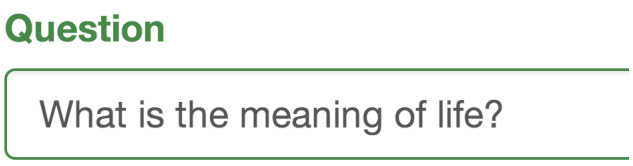

Figure 4-6: The form element from figure 4-5 in the browser

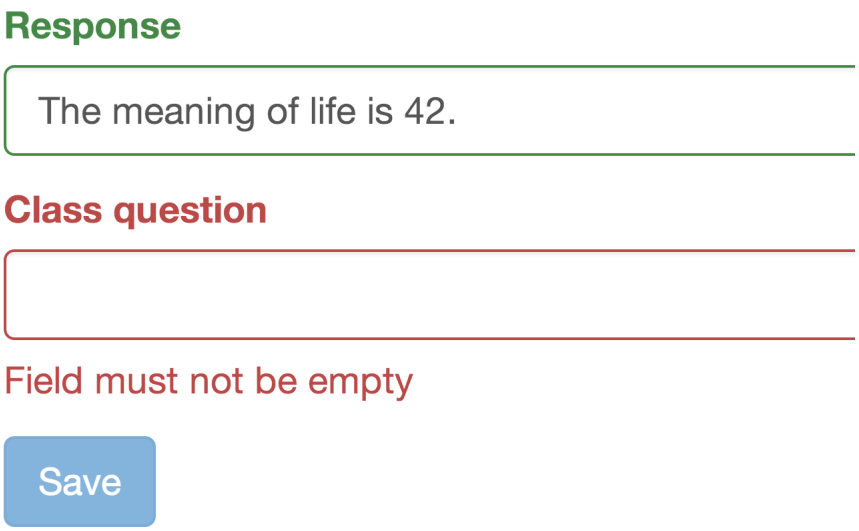

Figure 4-7: A composite form element

label for the form element, the default value, and the validator (which in this case errors if the field is non-empty).

Figure 4-6 shows how the element is rendered; in this case, both the label and the actual outline are rendered in green, indicating that the element passes validation. A more complicated form element modeling a response to a reading question and a question for discussion in the class is shown in figure 4-7. The class question field is empty, which is invalid, and so is outlined in red; the infrastructure for materials (covered in section 4.3) uses the fact that the data is invalid to dim the save button.

Form elements can be combined via the `concatForm` function, which takes two `formElements` that use compatible routers and metadata and model a value with disjoint record names, and produces a combined `formElement` whose editor is the concatenation of the editors of the two sub-elements. This is why the value a form

element represents must be a record; if arbitrary values were permitted then the concatenation of two elements would have to model a tuple of the elements' values, which would quickly lead to unwieldy types for composite forms with more than two or three elements. The record-based approach also allows the possibility of form elements that model zero fields, which can be used to represent some non-interactive HTML.

At a value level, a form element is a record with five fields:

- `Init` represents how to initialize a GUI. This is executed server-side, and will typically construct some number of `source` values that will then be bound to text boxes. The initializer takes as its arguments a variant-based recursion router, some metadata, and an `option $fs`, where `fs` is the type-level record of values the form element contains. The case where the value is `None` is used when creating a new item, since some form elements such as file uploads might not have sensible default values.

- `Signal` represents how to extract a value in the `signal` monad from the state, representing a value that is dynamically updated as the state changes; see section 4.2.3 for more information.

- `Edit` builds the state into an HTML fragment that can be included somewhere on the resulting page.

- `OnSave` is a handler that should be executed whenever the form element is 'saved'; for most elements, this does nothing, but for the file upload element, the uploaded file must be 'claimed', i.e., saved from automatic deletion.

- `Validate` indicates what validation should be performed on the server; while client-side validation is implicitly done by `Signal`, this abstraction should obviously be secure against malicious users!

The GUI module also exposes constructors for various simple form elements, such as strings (which can be input either via a textbox or a textarea), an integer (input

via a textbox), or a date (input via an external date/time picking widget that I wrote Ur/Web bindings for). These constructors take care of the majority of simple use cases, such as entering an assignment's title, due date, description, and maximum grade. More complicated elements can be custom written, such as in `gradable.ur` which creates a custom element to represent a student's grades.

Note that the form element framework does not specify how to display the edited data. Earlier versions included a field `View` with type `$fs -> xbody`, but this field was ultimately removed. Unlike editing, where generally the precise layout of the form is not terribly important as long as it's usable, the presentation of the data can be very important and so should be specifiable by the programmer. If the 'renderer' was part of the form element, then combining form elements would have to just concatenate their renderers, which in general would not do the right thing. The renderer is therefore specified separately.

4.2.2 Two-level modules

The `Material.Make` functor (and similarly, the `Gradable.Make` functor) exposes the traditional CRUD operations: create, lookup (i.e. retrieve), update, and delete. It also exposes a functor for building the GUI. At first, I only had one functor that exposed both GUI and model functions and took in both model and rendering data as arguments.

However, this prevents the custom GUI from easily interacting with the model. The `Material.Make` functor is responsible for creating the table that actually holds all the instances as well as the functions for creating/retrieving/updating/deleting instances, and so any function defined prior to the functor's instantiation will not have access to these functions.

As an example for why custom GUIs might be desirable, consider a gradable where any non-empty submission receives full credit; this could be used in order to solicit feedback on the course. 6.858 uses a similar mechanism, where each quiz has three two-point questions about the course itself; any answer at all, even 'I don't care', receives full credit. The obvious way to do this would be to expose an `updateGrade`

handler that would trigger a server-side recomputation of the student's grades; the submission form could then make a POST request to that URL after each save in order to trigger the grade recomputation. But the server-side handler for `updateGrade` needs to have access to the model functions so it can actually update the grade, since the grades table is defined in the functor itself. Similarly, the custom `OnSave` handler needs to have `updateGrade` in scope. This is difficult with only one functor since simple cross-module mutual recursion is impossible. Variant-based recursion doesn't help here, since I wanted the `Make` functor to perform the knot-tying, but `updateGrade` would need the output of the functor.

Therefore, I separated model generation and GUI generation; the functor that generates the model also returns a functor that can be used to generate the GUI itself. In this approach, the `updateGrade` function can then be declared with the result of the model-building functor in scope, and the GUI parameter to the second functor will then have `updateGrade` in scope and can invoke it. This could also be used to create multiple views for the same model, although I did not explore that option in any detail.

4.2.3 Validation

Data validation is an important part of any application that allows any kind of write access to a database. While the server obviously must validate all incoming data in the case of a malicious client, client-side validation is useful so that users can get immediate feedback on whether their input is valid. For example, when a teacher is editing an assignment, they shouldn't be able to save it if the title or the assignment description are empty. Ideally, although validation must be done on both the client and the server, the code for this validation would only be written once. Fortunately, since Ur/Web runs on both the client and the server, we can do this; any validator that doesn't perform client-only or server-only things like use a `signal` value or access a database will run on both the client and the server.

In the Galactic GUI framework, a validator for a value of type `t` has type `t -> option string`; if the value is valid, it returns `None`, and if it's invalid it returns

`Some err`, where `err` is a string indicating the reason for invalidity that should be displayed to the user. Parsing can also cause a validation error; the type of a parser is `string -> result t`, where `result` is an algebraic data type defined as

```
datatype result t = Error of string | Success of t
```

In other words, a `result t` is either a `t` or an error message.

If client-side validation fails, we would like not only to tell the user that it has failed but also dim out the save button so that they cannot even try to save invalid data. This dynamic updating of the page based on changing data input is done using Ur/Web's functional reactive programming model. The GUI framework uses this to implement the save button, which extracts the data from the form's signal into a `transaction (result t)` and pattern-matches on the `result` to determine whether to actually execute the save handler.

4.2.4 Tabular data

Table editing is a fairly common use case; the `Gradable` module needs a table for editing students' grades and a table for editing the list of components for a gradable. I focused on these two as the main motivators for constructing a generic combinator, named `tableEl`, for constructing editable tables.

There are two main differences between the two examples of student grades and the components list. First, while components can be added or removed, the list of students is essentially fixed; currently, that list is obtained from the authentication module, and even if it were editable on the site itself, the grades list would not be the appropriate place to do so. Second, the list of columns for the component list is known ahead of time: there is one column for the description, and one column for the maximum grade. (Other columns such as the amount of extra credit available could be added later, of course.) But for the student grades table, there is one column per component, and the number of components varies at runtime! This is inherent to the nature of gradable components and cannot be solved by simply requiring the

components to be specified at compile-time via a record; different instances of a given gradable type can have different numbers of components.

Dynamic columns

Since I wanted the two kinds of tables to be able to share much of their underlying code, my generic `tableEl` element combinator takes in any form element that can be embedded inside a `<tr>` tag as well as a list of headers for the columns. Constructing such a form element for the case of a record type all of whose elements can be represented with textboxes is done with the `recordRow` function, which takes in information for each column such as how to turn values into strings, how to parse strings into values, and how to validate those values. In other words, the fundamental `tableEl` combinator is 'unsafe' since it relies on invariants not enforced via types.

This caused some pain; writing the `tableEl` combinator was one of the few cases where I actually ran into an unexpected runtime error during development. Since the columns are specified as a list, things like their headers must also be specified as a list (specifically, in the table element's metadata). Since in general the number of columns is not the same as the number of fields in the `fs` parameter of the underlying `formElement` (and cannot be if the number of columns is dynamic!), there is no way to ensure that this list is the same length as the number of columns. During one point during the development of `tableEl`, I stubbed out the code that would generate the list of headers in the location where I was calling `tableEl` and simply passed an empty list. The function I was using to iterate over pairs of lists would produce a fatal error if the lists were unequal length; since the number of columns was greater than the number of headers (i.e., 0), this produced a runtime error that was rather annoying to track down!

Row addition and removal

Handling the addition and removal of rows was the other significant challenge. Traditional JavaScript relies on an 'event listeners and mutation' style that Ur/Web strongly discourages, so taking the standard JavaScript approach of having the add

buttons create a new `<tr>` element and insert it into the appropriate location was out of the question. Instead, I used the `<dyn>` element, which allows for dynamic HTML; the element takes in a `signal` corresponding to an HTML fragment and continually updates as the signal changes. Since the `<dyn>` element updates by entirely replacing itself without trying to preserve any elements that remained the same, the value of the underlying `source` should only change when a new row is added or removed. Not only would changing it on each keystroke cause performance problems, but it would also defocus the textbox after the keystroke, since it would be immediately deleted and recreated!

The solution is simple, and obvious in retrospect: have the state be of type `source (list state)`, where `state` is the type of the state of an individual row. For example, if the row only has a single textbox and its state is `source string`, the type of the table's state would be `source (list (source string))`. Note that if `src` is a `source string`, the value of the underlying string changing does not change the value of `src` itself! (An analogy can be drawn with pointers in a language like C, or Haskell's `IORef`.) So the only time when the outer source changes is when a row needs to be added or deleted, and that can easily be done via the functions `get` and `set`, which mutate a `source` in the `transaction` monad.

Now implementing the add and remove buttons is easy; the add button in row `n` calls `add n`, where `add` is a function that adds a new row just before the nth row, and similarly the remove button calls `remove n`. Adding an element to the end can be done with a button located at the bottom.

4.2.5 Existential types

The GUI form element type is somewhat unfortunate; not only does it have five parameters, it also exposes the type of the state, which is an internal implementation detail. One potential way to solve this would be to use existential types, which are implemented in `top.ur` with the following signatures:

```
con ex :: K --> (K -> Type) -> Type
```

```
val ex_intro : K --> tf :: (K -> Type) -> choice :: K -> tf choice
                -> ex tf
val ex_elim : K --> tf ::: (K -> Type) -> ex tf -> res ::: Type ->
               (choice :: K -> tf choice -> res) -> res
```

The type `ex f` represents a value that is of type `f a` for some a; the only operation that it supports is application via a function that is polymorphic in the kind of `a`. For example, a value of type `ex list` represents a list that contains elements of an unknown type. `ex_intro` is the constructor for existentials, and `ex_elim` indicates how to work with them (sometimes known as an 'eliminator', hence the name).

In general, introducing an existential can make the types slightly simpler at the cost of limiting what can be done with the values; the function argument to `ex_elim` has a result type `res` that does not depend on the choice of `choice` (i.e., the 'missing' parameter to the existential), so writing a function to extract the first element of an `ex list` is impossible. But this is fine for the `formElement` case, since the only time when the internal state is created via `element.Init` is when it's passed to `Signals` and `Edit` later on. In a purely server-side context, this would be a sign that `Init` should instead just generate the signals and the editing HTML directly; however, the signal-generating code cannot be run on the server, since only clients can generate signals.

I attempted rewriting the GUI library to use existentials so that the type of the state would be hidden, but I ran into problems while doing so. Existential types complicate the types and usage of functions such as `concatForm`, and the complication can often result in more difficult type inference. For example, compare the declarations of the normal and existential form element concatenation functions (the body is elided for brevity; it is identical in both):

```
fun concatForm [routes ::: {Type}] [meta ::: Type] [fs1 ::: {Type}]
               [state1 ::: Type] [fs2 ::: {Type}] [state2 ::: Type]
               [fs1 ~ fs2] [ctx ::: {Unit}]
               form1 form2 = { ... }
```

```
fun concatFormEx [routes ::: {Type}] [meta ::: Type] [fs1 ::: {Type}]
                 [fs2 ::: {Type}] [fs1 ~ fs2]
                 [ctx ::: {Unit}]
                 (ex1 : formElementEx routes meta fs1 (xml ctx [] []))
                 (ex2 : formElementEx routes meta fs2 (xml ctx [] []))
    = ex_elim ex1 (fn [state1 ::_]
    (form1 : formElement routes meta fs1 state1 (xml ctx [] [])) =>
    ex_elim ex2 (fn [state2 ::_] form2 =>
    ex_intro [fn state => formElement routes meta (fs1 ++ fs2) state
                                      (xml ctx [] [])]
             [state1 * state2] { ... }
```

Unlike in the normal form, the existential form concatenator requires the types of both of its arguments to be written out in full, as well as the type of `form1` in the outer existential eliminator. Similarly, using the existential GUIs can be difficult; I ran into situations where code of the form

```
fun handler () =
   x <- value;
   ex_elim ex (fn [t ::_] gui =>
      doSomethingWith x;
      useGui gui)
```

would not type-check without either explicitly specifying the type of `x` inside the `ex_elim` block (i.e., `doSomethingWith (x : int)`) or moving the binding of `x` inside the `ex_elim`.

Since type inference in Ur/Web is inherently undecidable, this does not necessarily reflect a compiler weakness, as any type inference engine must fail to infer types for some valid program. The small gain of having shorter types for all the form elements was more than outweighed by the substantial extra pain of type inference failure and extra boilerplate around creating and using form elements and GUIs. When state

types are passed to the `Gui` functor, they are passed as abstract types, meaning that the functor cannot actually 'look into' the state anyway. Overall, I judged existential types to be not worth the trouble.

4.3 The 'material' abstraction

A course management system will have many different types of objects: reading material, homework assignments, and quizzes. All of these have a few things in common: they are usually presented as lists; teachers can create, edit, and delete instances of them; and students can view them. I abstracted this functionality out into the `Material` module, which represents anything that can be viewed by students and edited by teachers.

The `Material.Make` functor generates the table, with one column for each field in the input as well as a primary key field called `Id` and a field called `Title` that will represent the title of the item when shown in a list view. It also provides convenience functions for the standard CRUD (create/retrieve/update/delete) operations.

Activities are looked up by their `id`, but the type `id` is opaque to the consumer of the module. In other words, it is impossible to generate a material ID directly; it must be obtained from an existing material or by creating a new one. (This trick is reminiscent of Haskell's 'newtypes'; a declaration such as `newtype Id = MkId Int` creates a new type named `Id` that is represented internally the same as an `Int`, but must be explicitly converted; if the `MkId` constructor is not exported, then this conversion becomes impossible.)

The first draft of the material abstraction used a type-level record with one field for each instance of the material, meaning that the number of materials would be known at compile time. This was ultimately discarded as I believed that while certain material can be known ahead of time, such as quizzes and homework assignments, others such as reading material might need to be modified on the fly. Nevertheless, I think that this approach could possibly be useful in some other project.

The structure returned from `Material.Make` in turn contains another functor,

named `Gui`, that will actually generate the GUI. The three inferrable value-level parameters (ignoring the folder, which can be inferred) are `viewMaterial`, which is used to render a material; `formEl`, which is the form element for editing the material; and `auxHandlers`, which is a record of auxiliary handlers that are needed by `viewMaterial` and `formEl`. For example, a gradable (which is covered in more detail in section 4.4) needs a page where the teacher can edit grades; since almost all the linking in Galactic is done through variant-based recursion, this would result in `auxHandlers` having type `{Grades : id}` (i.e., there is a route named 'Grades' that takes a parameter of type `id`).

`basicMaterialModule.ur`, reproduced in figure 4-8, gives a simple use of the `Material` module; it defines a material with a description, which is a string that must be non-empty and an 'awesomeness level', which must be between 0 and 100 inclusive. Figure 4-9 shows the editor generated by this module.

4.4 The 'gradable' abstraction

As the name implies, a gradable is a material that can be graded, such as reading questions, homework assignments, and quizzes. The common thread between these is that, in addition to being a material, students can also receive a grade; for some gradables, students also can submit submissions for the teacher to grade. The grade is linked to the gradable and not to the submission, since generally a course's final grade accounting will only have one grade per student per assignment.

The `Gradable` module is mostly similar to the `Material` module in its design. The `Gradable.Make` functor takes in a description of an assignment as well as the submission for that assignment and outputs some helper functions as well as types (all of which are just passed through from the underlying `Material.Make` invocation). Similarly, the `Gui` functor takes in form elements and renderers for the gradable and the submission as well as a record of routing handlers and invokes `Linkage.Close` to bundle them all up into a record of handlers.

The code listing for the reading question module is given in figure 4-10; the data

```
open Util

fun validateAwesomeness x = if x < 0 || x > 100
                            then Some "Out of bounds"
                            else None

val formEl = Gui.concatForm
                 (Gui.textboxEl [#Description] "Description"
                                "Fill me in" Util.nonEmpty)
                 (Gui.intEl [#Awesomeness] "Awesomeness" 100
                            validateAwesomeness)

structure Mat = Material.Make(struct
    con cols = [Description = string, Awesomeness = int]
    val validate = formEl.Validate
  end)

structure Gui = Mat.Gui(struct
    val name = "Basic Material"
    val viewMaterial = fn _ _ material => return <xml>
      <h2>Description</h2>
      <div>{[material.Description]}</div>
      <div>The awesomeness level of this material is
        {[material.Awesomeness]}.</div>
    </xml>
    val auxHandlers = {}
    val formEl = formEl
  end)

val viewAll = Gui.routed.ViewAll
```

Figure 4-8: The source code listing for a basic material module

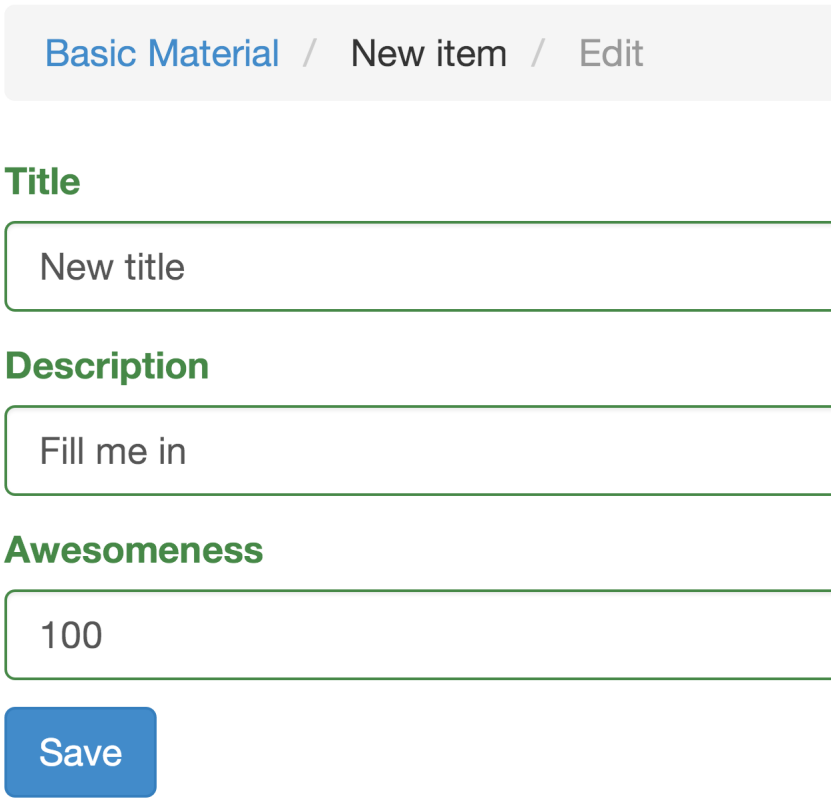

Figure 4-9: The editor created from figure 4-8

for a reading question is a question that must be answered, and the submission is both a response to the question and an in-class question that may be chosen for discussion (a model used in, e.g., 6.858).

The top structure, `Grad`, invokes the part of the `Gradable` module that constructs the tables. Note that only the type of the submission data is explicitly specified, even though this functor creates the tables for both the submission and the gradable itself; the type of the gradable can be inferred from the later invocation of the `Gui` module. This inference is very useful, since there are other hidden arguments to `Gradable.Make`, such as a record of 'witnesses' that each element of the submission can be inserted into a SQL table. In all, there are ten arguments to `Gradable.Make`, but only three of them need to be specified.

The `Grad` structure then itself contains a functor that we use to construct the actual GUI. `viewGradable` specifies how to view an instance of this specific gradable; the two unused arguments are the variant-based recursion router and the metadata, neither of which are necessary in this case. `gradableEl` and `submissionEl` are for the gradable itself and the submission; `submissionEl` will be used directly by `Gradable`, whereas `gradableEl` will have an element for the due date concatenated on to the end before it is passed to `Material`. Finally, the `subHandlers` value is a record of variant-based recursion handlers. In this case, since we do no variant-based recursion, the record can be empty, but a module that has file uploads will want to pass in handlers that load and serve those files so that `viewSubmission` can link to them.

The output of this `Gui` functor, which is assigned to a structure also called `Gui`, is a record of functions that take in whatever argument their corresponding variant-based recursion handler takes and produce a `transaction page`. `Gradable` itself defines a list of routes, including the ones `Material` defines; `routed` will also include routes that the submission itself needs, since these routes could not otherwise be obtained and it costs no code complexity to provide them.

```
val gradableEl =  Gui.textboxEl [#Question] "Question"
                                "What is the meaning of life?"
                                Util.nonEmpty
val submissionEl = let
      val responseEl = Gui.textboxEl [#Response] "Response"
                                     "" Util.nonEmpty
      val classQuestionEl = Gui.textboxEl [#ClassQuestion]
                                          "Class question" ""
                                          Util.nonEmpty
    in
      Gui.concatForm responseEl classQuestionEl
    end

structure Grad = Gradable.Make(struct
    con submissionData = [Response = string, ClassQuestion = string]
    val validateGradable = gradableEl.Validate
    val validateSubmission = submissionEl.Validate
  end)

structure Gui = Grad.Gui(struct
    val name = "Reading Question"
    val viewGradable = fn _ _ question => return <xml>
      <h2>Question</h2>
      <div>{[question.Question]}</div>
    </xml>
    val formEls = gradableEl
    val auxHandlers = {}
    val submissionEl = submissionEl
  end)

val viewOne = Gui.routed.ViewOne
val viewAll = Gui.routed.ViewAll
```

Figure 4-10: The source code listing for the reading question module

4.4.1 Components

Many gradables, such as quizzes, are not graded in one large block. Instead, the gradable has multiple parts (i.e., questions on a quiz or a homework assignment), each of which has its own maximum grade, and the grade on the gradable is simply the sum of the grades on the parts. The maximum grades are not normalized, since in many classes quizzes are scored out of some number other than 100; however, the total grade of the components of a gradable is not related to its weight. In other words, if one quiz is graded out of 50 and another quiz is graded out of 60, then the second quiz is not necessarily worth more than the first quiz. My motivation for this weighting scheme was that many classes that give out problems from a book for the weekly assignments give different numbers of problems on different problem sets, and requiring the maximum grade on each problem to vary on different problem sets would be very inconvenient for the course staff.

4.4.2 Non-submission gradables

Being able to submit an answer is not actually part of the requirements for a gradable; students can receive grades on things such as quizzes and class participation despite never actually submitting anything through the course interface. Whether or not a gradable should have a user submission is determined by whether or not the supplied record indicating the submission data has any fields; if there are no fields, then there's obviously nothing for a student to submit! This can be computed via a function of type `hasElements : [fs ::: {Type}] -> folder fs -> bool`, since Ur/Web's folders do not actually require a value of the type to be folded over in order to operate. Moreover, in the code segment

```
con myRecord = [Foo = int]
val x = if hasElements [myRecord] then "Elements."
        else "No elements."
```

the declaration of `x` will simplify to just

```
val x = "Elements."
```

since the fold will be simplified at compile time, meaning the conditional in the `if` statement will be `True` and so the `if` can be optimized away.

4.5 Templating

Any nontrivial Web application is going to have some common HTML reused throughout. Most modern Web applications use a templating framework, whether it's server-side such as Python's Jinja2, Haskell's Hamlet, PHP's built-in templating functionality, or Ruby's Jade, or client-side such as Underscore, Handlebars, or Mustache. Ur/Web has built-in support for HTML literals, so it doesn't need any sort of library for building the HTML itself, but I still had to write code to do the templating.

The simplest templating engine is just a global function, `template : xbody -> transaction page` (returning a `transaction page` instead of a `page` because the template might want to access the database somehow) that simply puts its argument within a `<body>` tag, with some surrounding infrastructure: something like

```
<xml>
  <head>
    <title>Galactic</title>
    <link rel="stylesheet" href="//path/to/stylesheet"/>
  </head>
  <body>
    {inner}
  </body>
</xml>
```

This works well for simple applications. But one usability feature present in many modern Web applications is a list of 'breadcrumbs': links that traverse up the logical hierarchy of the application back to some defined root, such as can be seen in figure 4-9.

Clearly, the responsibility for generating the link hierarchy lies with the same portion of the logic that generates the inner template. But the responsibility for rendering the hierarchy into breadcrumbs is with the template engine.

Therefore, instead of simply taking an `xbody` parameter, it makes more sense for the template function to take both an `xbody` and a list of values of type `{Url : option url, Title : string}`; each element in the list corresponds to some breadcrumb, which is a link if the `Url` uses the `Some` constructor and plain text if it's `None`.

4.6 Datetime library

The calendar system in Galactic displays a list of 'items' on a calendar. Each event has a title, indicating what it should be displayed as; a datetime, indicating what date and time the event corresponds to; and a link, indicating what the item should lead to when clicked. In order to properly lay out the various events on a calendar, it's necessary to be able to compute things like the first day of the week of a given month and what day of the week a given event falls on. Initially, the only way to answer these sorts of questions was to take a value of type `time` and pass it to `timef`, which is a wrapper around the C `strftime` time formatting function; for example, when passed the `%w` format string, `timef` returns a string containing the day of the week as a number, which can then be parsed into a value of type `int` using `readError`. Similarly, computing the first day of the week of a given month can be done by creating a string representing that day and passing it to `readUtc : string -> option time` and passing the result through `unsafeGet` and the day-of-week-computing function; however, I thought this was unsatisfactory and would result in messy code.

Therefore, I created functions in `basis.urs`: one for creating a `time` from the year, month, day, hour, minute, and second, and one for extracting those values, as well as the day of week, from a `time` value. I then added a new module `datetime.ur`, which defines a record type `Datetime.t` with fields for the year, month, day, hour, minute, and second (for a module that only defines one 'main' type, the convention

is to call the type `t` to avoid redundancy in names like `Datetime.datetime`), and some utility functions for converting back and forth between a `Datetime.t` and a `time`. I also added a `normalize` function that converts 'invalid' datetimes such as the 0th of April into the 31st of March; this is done by converting to a `time` and back. Since the `mktime` function that I use to actually convert from a `Datetime.t` to a `time` is guaranteed to allow invalid values for the various components, this internal 'round-tripping' through the Ur/Web `time` type will always produce valid results.

4.7 Data attributes

I made extensive use of Twitter's Bootstrap 3 HTML/CSS/JS framework throughout this project, as it provides a clean, modern-looking interface with relatively minimal work. Ur/Web's first-class CSS support was useful here, as it is impossible to generate an invalid class name; if the value of a CSS attribute inside an HTML literal is a string, it is first split on whitespace and interpreted as a list of `css_class` values, which are combined using the `classes : css_class -> css_class -> css_class` operator.

However, when I began work on Galactic, Ur/Web did not have support for so-called 'data attributes'. Bootstrap uses these elements to provide data to the JavaScript that provides interactivity; for example, when one of the save buttons on Galactic's forms is clicked, it dims and the text is replaced with 'Saving...', only coming back to normal when the remote procedure call has finished. The replacement text is stored on the `<button>` element as a data attribute; i.e., `<button data-loading-text='Saving...'>`.

This would appear to pose a difficulty for the Ur/Web type system, since normally each tag has an associated record with one field for each legal attribute name. Obviously, the language should be able to support arbitrary data attributes, so simply picking some list and declaring them the 'valid data attributes' will not work.

An alternative way to fix this is to add a new field, `Data`, which encodes the values of all the data attributes on the element. Since there is no attribute named simply `data` on any HTML element, this does not introduce any naming conflicts (and if

one later appears, `Data` can be renamed to `DataAttrs` or similar). The problem then reduces to the encoding of the attributes; there are two obvious ways to do this.

4.7.1 Record approach

The first way is to encode the attributes as a record; the element `<div data-foo=bar data-baz=quux></div>` would then be represented as the literal `<div data={{Foo = "bar", Baz = "quux"}}></div>` (the outer set of curly braces indicates that the value is an Ur/Web value; the inner is part of record literal syntax). The record system then enforces some useful invariants: attribute names cannot be duplicated or empty, though they may be numeric. Data attribute names can contain hyphens, but this does not pose a significant obstacle; the HTML literal rewriter can interpret the field name `Foo_bar` as the attribute name `data-foo-bar` without difficulty. This does mean that underscores cannot be used in attribute names, but frameworks typically use hyphens as word separators so this is not an issue.

However, this causes issues with the type system. There is no simple way to indicate the type of a record with arbitrarily many fields all of which are strings; any such encoding would require a type variable. This would mean that the `tag` value-level function would need an additional type-level parameter, which could potentially slow down type checking since the `tag` function is used any time elements are nested within each other.

4.7.2 Association list approach

The second approach is to encode the attributes as an association list; i.e., a value of type `list (string * string)`. The element `<div data-foo=bar></div>` would then be represented as the literal `<div data={("foo", "bar") :: nil}></div>`. This is slightly more verbose than the record literal syntax, and loses the uniqueness and well-formedness guarantees; it's not clear what should be done if the key of some item in the association list contains a space, angle bracket, or other reserved character. On the other hand, modifying the existing type system to allow it is vastly

more straightforward than encoding the attributes in the type.

This is the approach that Adam ultimately wound up choosing, with a few caveats; instead of encoding the attributes directly as a list of elements of type `string * string`, the attributes are encoded as a value of type `data_attr`, which can be constructed either via an individual key-value pair or by concatenating two such values together.

Chapter 5

Related work

The Ur/Web standard library contains a spreadsheet demo, mentioned in [1] under the "Case studies" section, where the columns are specified using a record with fields like `A = editable [#A] "A" int`. Galactic's `recordRow` element constructor is similar, constructing a form element for a row given a field like

```
A = {Default = 0, Stringify = show, Parse = readResult,
     Validate = fn _ => None}
```

Obviously the Galactic version is more verbose, but it also specifies the default value (the demo's spreadsheet's default is specified elsewhere) and can specify how to validate a field. The Ur/Web GUI library contains an alternative approach to GUI-building, where a GUI component of type `gui t ctx` is actually a typeclass witness.

WebDSL [5] is, as the name implies, a domain-specific language for creating Web applications. The languages compiles to Java and has interoperability with native Java code. The original WebDSL paper mentions automatically creating view/edit pages for entities based on their model declarations, in contrast to the Galactic approach which effectively infers the model type from the declaration of the edit page. A similar idea could be implemented in Galactic through the 'GUI typeclass' approach mentioned above; the page-generating modules could then take in a record of typeclass instances and a record of field names. The current version of the WebDSL

manual deemphasizes this feature in favor of a form-specific DSL that looks somewhat like a Galactic form element declaration, which leads me to believe that the small amount of extra code necessary for the Galactic approach is more than outweighed by the customizability it provides. The WebDSL elements are also static, unlike the Galactic ones that can respond to changes in user input by, e.g., doing client-side validation.

The variant-based recursion routing technique is similar to that used in the Yesod Haskell framework. Yesod ensures generated URLs are valid by using a user-defined algebraic data type for routing in combination with a function to render that data type into a URL; the framework uses the Template Haskell [4] metaprogramming facility to generate both the data type and the renderer from the routing information. Yesod also allows string URLs, but unlike with Ur/Web, these URLs are not checked against a predefined whitelist before inclusion.

Template Haskell [4] is a compiler extension for the GHC Haskell compiler that adds metaprogramming capabilities more in line with those provided by Lisp or Scheme macros than Ur/Web's; the template expansion phase is even capable of performing IO, meaning that data types and functions can be generated from a plain text file on disk written in an arbitrary language. This is significantly more powerful than Ur/Web metaprogramming, but it creates dependency issues in the compilation graph and is more difficult to work with.

The GHC compiler also supports a metaprogramming extension called 'generic deriving' first outlined in [2] that more closely resembles Ur/Web style metaprogramming; only data types that are an instance of the `Generic` typeclass can be used, but this instance can be derived automatically (even in modules other than where the type was defined), so in practice this only amounts to a very small amount of overhead. Their approach also works over arbitrary Haskell ADTs (i.e., essentially all types); Ur/Web metaprogramming only supports record types, not ADTs.

iTask [3] is a DSL written in the Clean programming language for Web applications oriented around workflows: tasks are generated and assigned either to human users or to computers via a declarative combinator library. iTask is more suited for multi-user

workflow paradigms; right now Galactic assumes that all students are equivalent and all teachers are equivalent, and the modularity system is entirely oriented around materials and gradables. Future work might look to iTask for ideas on how to handle teachers wanting to break large tasks such as quiz grade entering into chunks that can be distributed among teaching assistants.

Many modern client-heavy Web frameworks such as EmberJS that treat the back-end as nothing more than a data store make extensive use of reactive programming in their HTML templating; fragments of HTML can be 'bound' in a reusable manner to an attribute of some object, and those fragments will update when the object's properties are updated, much like a `<dyn>` tag in Ur/Web. Unlike those frameworks, Ur/Web cannot perform I/O in the body of a `<dyn>` tag since the body must lie in the `signal` monad, but in practice I did not find this to be a restriction and I suspect making such a thing possible would be difficult and/or violate the language's design goals. On the other hand, an EmberJS `Model` object's properties must be accessed and mutated through `get` and `set` methods as opposed to the standard JavaScript `object.property` syntax; Ur/Web `signal` values can be worked with like any other monadic value.

Chapter 6

Further work

6.1 Nested variant-based recursion

Suppose that I have two gradables, A and B. I want A to be able to link to locations in B and vice-versa. This seems like it would be an excellent candidate for variant-based recursion: the only modification that would need to be made is to rely on the `Gui` functor's caller to do the linkage as opposed to doing the linkage in the functor itself. However, there is a problem: A's routes and B's routes will overlap since both of them will expose routes named, e.g., `ViewOne`. The obvious solution of simply changing the names of one of the module's routes would work but would produce ugly, repetitive code: while Ur/Web's type system would support a record that maps old route names to new route names, there is no way to 'apply' that renaming to a record, since such an application would require a proof that all the `Name` 'values' are distinct, which is not a notion that can be expressed in Ur/Web's type system.

The Yesod Haskell Web framework, which uses a type-safe URL routing mechanism similar to Galactic's, uses a concept called 'subsites' to handle this problem; each subsite has a constructor in the parent site's algebraic data type that takes as its argument a route in the subsite and returns a route for the parent site. Multiple subsites can then have different subsite constructors and will not interfere with each other.

Translated into variant-based recursion terms, this would mean generating URLs

that look like `r (make [#A] (make [#View] itemId))`, where the 'outer' variant construction represents the A subsite and the inner one represents a route within that subsite. However, implementing this is a bit tricky: under the most obvious approach, the kind of a handler's requirements would then have to be `{{Type}}`, meaning that every route would have to be contained within a subsite; in this case, the most sensible thing to do would be to have one 'catch-all' 'subsite' that contains all the routes that don't logically belong to any other subsite.

Variant-based recursion is also unable to handle mixed GET and POST requests. The Ur/Web compiler enforces the invariant that the handler for a GET request must be free of side effects, but all routes in a variant-based router use the same router. One solution would perhaps be to have two separate routers, a GET and a POST router.

6.2 Computer-generated modules

While the modules defining gradables and activities are fairly simple, one of the initial goals of this project was for Galactic instances to be creatable without ever touching code. This could be done via a website where a teacher could input the definition of their course requirements into a form; the site would then generate some Ur/Web code, compile it, and serve the Ur/Web source code, the generated C code, and the compiled binary to the teacher, who can then run it on their own server; the C code is included in case some form of library difference between the compiling server and the teacher's server prevents the binary from running as-is.

Chapter 7

Results

Ultimately, I believe Galactic was a success. The site, while not completely usable in its current state due to its lack of authentication system, is close enough to usable that it could definitely be brought up to full usability in time for the fall 2014 MIT semester.

This thesis makes four main contributions to the field of Ur/Web application programming:

1. I built Galactic, a generic course management system that can easily be customized even by users without any knowledge of Ur/Web; the syntax for defining new gradables or materials is simple and involves very little actual programming.

2. Refined the 'variant-based recursion' technique, which was the breakthrough that allowed cross-module linking. Without it, handling file uploads, breadcrumbs, and similar instances of recursive page handlers would be significantly more difficult, if not outright impossible.

3. I constructed a form element framework that supported parsing of non-string values from string inputs, client and server validation of those values, and responsive feedback regarding the validity that updates as the user types in the textbox.

Bibliography

[1] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 122–133, New York, NY, USA, 2010. ACM.

[2] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM.

[3] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: Executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 141–152, New York, NY, USA, 2007. ACM.

[4] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM.

[5] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II*, pages 291–373. Springer, 2008.

[6] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer Berlin Heidelberg, 1995.

[7] Nicholas C. Zakas. How many users have JavaScript disabled? `https://developer.yahoo.com/blogs/ydnfourblog/many-users-javascript-disabled-14121.html`, October 2010.