An Embedded Domain-Specific Language for Type-Safe Server-Side Web-Scripting

Peter Thiemann
Universität Freiburg, Germany
`http://www.informatik.uni-freiburg.de/~thiemann`

February 24, 2003

Abstract

WASH/CGI is an embedded domain-specific language for server-side Web-scripting. Due to its reliance on the strongly typed, purely functional programming language Haskell as a host language, it is highly flexible and —at the same time— it provides extensive guarantees due to its pervasive use of type information.

WASH/CGI can be structured into a number of sublanguages addressing different aspects of the application. The **document sublanguage** provides tools for the generation of parameterized XHTML documents and forms. Its typing guarantees that almost all generated documents are valid XHTML documents. The **session sublanguage** provides a session abstraction that provides a transparent notion of session state and allows the composition of documents and Web-forms to entire interactive scripts. Both are integrated with the **widget sublanguage** which describes the communication (parameter passing) between client and server. It imposes a simple type discipline on the parameters that guarantees that forms posted by the client are always understood by the server. That is, the server never asks for data not submitted by the client and the data submitted by the client has the type requested by the server. In addition, parameters are received in their typed internal representation, not as strings. Finally, the **persistence sublanguage** deals with managing shared state on the server side as well as individual state on the client side. It presents shared state as an abstract data type, where the script can control whether it wants to observe mutations due to concurrently executing scripts. It guarantees that states from different interaction threads cannot be confused.

1 Introduction

Since its inception in 1991, the Web has dramatically changed its face. What started as a hypertext system for distributing scientific documents has changed into a ubiquitous global network for exchanging all kinds of data. Along with the content, the presentation of information in the Web has changed. Originally, Web servers acted like file servers that produced stored documents on demand. Nowadays, most Web pages are computed on the fly according to user profiles, language and image preferences, or the latest news. Not only are those pages generated dynamically, they also support all sorts of interaction, from chatting, blogging, and browsing catalogs to buying groceries and booking entire travel arrangements online.

Clearly, the implementation of all those interactive services requires substantial effort for developing software that runs on the Web servers and/or associated application servers. Despite the fact that it is deceptively easy to program dynamic Web pages (using CGI [5], for instance), it turns out that programming reliable interactive Web services is very hard. This is due to a number of reasons.

First, the protocol underlying all Web services is HTTP, the Hypertext Transfer Protocol [8]. The problem with HTTP is that it only specifies a one-shot request-response model of communication. It does not support any notion of session or state management which is required to implement an interactive transaction reliably. This places a burden on the programmer who has to implement some notion of session management.

Second, the original "script-centric" approach to program a Web service is derived from the one-shot model. Each request starts a "script" on the server and this script computes the answer page and terminates. It is a non-trivial software engineering problem to get a number of scripts to cooperate even on a single server because their interdependencies are recorded in unchecked string references.

Third, all communication between browser and server is in the form of name-value pairs where the names are defined in one program (the script generating the currently displayed form) and used in another program (the script processing the submission of the form). This, again, gives rise to a non-trivial consistency problem between a set of programs. And, again, this problem is caused by the use of strings for external naming.

Fourth, the values communicated to the server are strings. For that reason each submitted value must be checked and erroneous submissions must be rejected before the actual processing can commence. To make the application reliable, a large fraction of application code must be dedicated to validate the input values [31].

Fifth, XHTML does not provide any means to structure forms into small interaction components. Hence, all form-related code is monolithic in the sense that even a form with many unrelated components (as for example on a typical portal site) must be generated and processed by one chunk of code.

Sixth, the dynamically generated documents should be guaranteed to adhere to the XHTML standard. This can be tricky, in particular, if documents are generated from highly parameterized parts.

Last but not least, the management of persistent state on the server is a difficult problem due to the nature of the browser interface that permits the use of the back button and cloning of windows.

Hence, there is a growing demand for languages and tools that support this kind of programming task. WASH/CGI addresses all problems listed above by providing sublanguages for document generation, session management, widget composition, and state management. Despite the very specific problems of the Web programming domain, it is not necessary to design and implement a language from scratch. Instead, it turns out that the purely functional programming language Haskell provides the means to fulfill all requirements. The key to make that work is the consistent use of abstract data types, type-safe interfaces, and compositionality in the design of the sublanguages.

Specifically, WASH/CGI offers the following contributions to Web programming.

1. WASH/CGI has a built-in notion of a session with transparently handled session state.

2. The session sublanguage allows the construction of complicated interactions from Web forms and other documents. Many interactions may be specified within one program and there is no need to assure the consistency of a set of programs.

3. Input values from a Web form are communicated via lexical binding. Since an external name is not required, this eliminates a common source of errors.

4. All input values are received by the script in their typed internal representation. There is no need to explicitly convert strings to values in a script. This conversion is performed and validated once and for all by the WASH/CGI implementation. Errors are flagged automatically, without further programming. Furthermore, there is an extension mechanism that allows programmers to define their own datatypes, validations, and conversions.

5. The widget sublanguage provides the means for specifying interaction fragments, where an arbitrary part of a form may be specified *together* with the code processing it. These interaction fragments may be composed and reused even across applications.

6. All documents generated by a WASH/CGI script are guaranteed to be well-formed (due to an internal tree representation) and to adhere to the XHTML standard to a large degree.

7. WASH/CGI provides abstract data types for persistent server-side and client-side state. Their interface is engineered to detect inconsistencies due to concurrent updates of the state information.

Meanwhile, a number of languages provide support for Web programming in one form or another (see the discussion in Section 10). However, the contributions 3, 4, 5, and 7 are unique to WASH/CGI.

Since many readers will not be familiar with the Haskell language, Section 2 provides a brief overview of its main features. The code examples in the running text are designed to be understandable with just this introduction. After these preliminaries, Section 3 introduces the document sublanguage. It first discusses the basic interface and then enriches the interface to guarantee quasi-validity of generated documents. Section 4 first reviews the requirements on implementing sessions in Web applications and discusses some implementation schemes. Then it presents the basic WASH/CGI operators for sessions. Section 5 describes the widget sublanguage, which provides the building blocks for constructing typed interactive services. Various notions of state make sense for Web applications. Besides the session state that is implicit part of every WASH/CGI application, Section 6 introduces the persistence sublanguage which supports two notions of persistent state, client-side and server-side state.

Section 7 puts all of the above to work and develops a complete example.

All implementation related discussion is gathered in Section 8. This section first explains the implementation ideas in general terms and concentrates the discussion of host language specifics to one subsection. Full appreciation of this subsection requires deeper knowledge of Haskell as can be obtained from Section 2. We include it to demonstrate the feasibility of the whole scheme and the simplicity and elegance of its implementation.

The rest of the paper performs a detailed assessment of the design decisions taken (Section 9), discusses related work (Section 10), and finally concludes.

2 Preliminaries

This section is designed to convey just enough background on the Haskell programming language so that readers can understand the examples in the text and that they can code simple examples by themselves. The Haskell language report [15] provides a complete reference.

2.1 Types and their values

Haskell is a strongly typed language with parametric polymorphism, overloading, and type inference. For each expression, the compiler computes a most general type and rejects programs that might cause type conflicts at runtime.

The Haskell types used in this paper are

- `Int`, fixed-length machine integers with constants written in the usual way and equipped with the usual infix operations,
- `Bool`, truth values with elements `True` and `False`,
- `Char`, Unicode characters with constants written `'x'`, `'y'`, `'z'`, ...,
- `String`, written in the usual string notation `"abc"`, and
- `()`, the *unit type* with the single element `()`.

Haskell provides type constructors to build new types from existing ones. There are a number of standard constructors available. If t, t_1, and t_2 are types, then so are

- `[`t`]`, the type of lists with elements of type t, a list is written as either `[`e_1`,` e_2`,` ...`]` or e_1 `:` e_2 `:` ...`:` `[]`, where `:` is the infix list construction operator and `[]` is the empty list;
- `Maybe` t, the type of optional values of type t, where each element is either `Nothing` or `Just` e for some e of type t;
- t_1 `->` t_2 `->` ... `->` t, the type of functions that map values of type t_1, t_2, ... to a result of type t;
- `IO` t, the type of I/O actions that yield results of type t.

Further types and type constructors may be defined by the programmer.

Types can be polymorphic. They may contain type variables which can be substituted for by any type. For example, the function `length` that determines the length of a list has type `[a] -> Int`. In that type, `a` is a type variable indicating that `length` yields a result no matter what type is substituted for `a`, *i.e.*, no matter what the type of the elements of the list is.

2.2 Syntax of declarations and expressions

The syntax

f `::` t

specifies a type signature for f, as in a C prototype or a Java interface. It declares that f has type t. This declaration is optional as the compiler can usually compute t from the definition of f. A definition for f has the form

f `=` e

or —if f is a function with formal parameters x_1, x_2, ...—

f x_1 x_2 ... `=` e

In either case, e is an expression formed from

- variables x_1, x_2, ...;
- constants;
- infix operators applied to expressions, *e.g.*, e_1 `+` e_2;
- functions applied to expressions, *e.g.*, f e_1 e_2 ...;
- `let` x_1 `=` e_1 ...`in` e, definition of values $x_1, \ldots$ for use in e (and $e_1, \ldots$);
- conditional expressions `if` e `then` e_1 `else` e_2;

- `case` expressions, which are used to decompose, *e.g.*, lists

```
length xs = case xs of
              []     -> 0
              x : xs -> 1 + length xs
```

  and optional values

```
fromMaybe x opt =
  case opt of
    Nothing -> x         -- returns x if opt == Nothing
    Just y  -> y         -- returns y if opt == Just y
```

A unique syntactic feature of Haskell is the *offside rule*. It specifies scope by indentation and is used with `let`, `do`, `case`, and other expressions. Without specifying it formally, the idea is to vertically line up items in the same scope. For example, the start of the second alternative in the above `case` expressions is indicated by indenting it to just below the respective first alternative.

2.3 Type classes and overloading

Haskell includes a notion of parametric overloading by restricting type variables to range over a proper subset of all types. These sets are called *type classes*. They are specified inductively and are associated to a set of overloaded member functions. For example, in

```
(==) :: (Eq a) => a -> a -> Bool
```

the type variable `a` is restricted by prepending the constraint `Eq a`. This declaration specifies the type of the (infix) equality function `==`. The definition of `==` for a particular type `t` (for the type variable `a`) is provided via a special declaration syntax, the *instance declaration*. The same declaration makes `t` a member of `Eq`.

The type class `Eq` is part of the Haskell standard. Further standard classes are `Read`, for types `a` that have a function `read :: String -> a`, and `Show`, for types `a` that have a function `show :: a -> String`. It means that each type `a` in class `Read` has a method `read` to convert a string into a value of type `a`. Similarly, the `show` method converts a value into a string.

2.4 IO

I/O operations and other side effects cannot be performed at arbitrary places in a program because Haskell is a pure functional language where evaluation can be performed in arbitrary order. To deal with I/O, there is an abstract datatype `IO a` for *I/O actions* that yield values of type `a`. For example, the action that reads a character from standard input is `getChar` and its type is `IO Char`. There are operators to create trivial actions (`return x` is an action that does not perform I/O but just returns the value `x`) and to combine actions by putting them in sequence (`action >>= nextAction` is a combined action that first performs `action` and then passes the returned value to function `nextAction` that computes the following action depending on the returned value). I/O actions are passive (like a data structure) until they are processed at the toplevel of the program. At that point, they are executed in their specified sequence.

Another way of composing actions is the `do` notation:

```
do x <- action1
   y <- action2
   ...
   finalAction
```

This expression builds an action that first performs `action1` and binds its result to x. Next it performs `action2` (which may depend on x) and binds its result to y. And so on until `finalAction`, which may depend on all previously computed results. The binding part x `<-` may also be omitted. In this case, the result is ignored.

The operations `return` and `>>=` as well as the `do` notation may be used for other kinds of actions, too. In fact, all these operations are overloaded using the standard type class `Monad`.

3 The Document Sublanguage

The document sublanguage supports the creation of XHTML documents [40]. As such, it has to provide the means to create XHTML document nodes, that is, element, attribute, text, and comment nodes. Our approach builds on our previous work [35] where each kind of node has its own typed construction function. For ease of use, the name of the construction function for an HTML element is the same as its element name. For better integration with the host language, we are not supporting XHTML syntax[1]. Instead, WASH/CGI provides, for each element tag, an element constructor function that takes as an argument a sequence of child elements, text nodes, and attributes (arbitrarily mixed) and returns a singleton sequence that contains just the newly constructed element. Similarly, the function `attr` takes an attribute name *`name`* and its value *`value`* (both strings) and constructs a singleton sequence with the attribute assignment *`attr=value`*[2]. Also, the function `text` constructs singleton sequence with a text node from a string.

The notion of a sequence of document nodes is a unique feature of the document sublanguage. Besides the element constructors, it provides operators for constructing empty sequences and for concatenating sequences.

[1] A preprocessor would complicate the use of WASH/CGI.

[2] Unfortunately, the commonly used symbol `@` is reserved in Haskell. However, it is possible to use the notation `(@@ name) value` with mandatory parentheses.

3.1 Composing Documents

For reasons explained below in Sec. 5, sequences are constructed using actions as explained for I/O in Sec. 2.4. Hence, the operators for composing actions also concatenate sequences of document nodes: The `do` notation as well as the binary infix operators `##`, `>>`, and `>>=`[3]. The operator `empty` constructs the empty sequence of document nodes. For example, the code fragment

```
  do p (text "This is my first CGI program!")
     p (do text "My hobbies are"
           ul (do li (text "swimming")
                  li (text "music")
                  li (text "skiing")))
```

creates a document with two paragraphs. The first one contains the text node *This is my first CGI program!* and the second one contains some text and an embedded itemized list. The XHTML output generated from this code looks as follows (pretty-printed for readability):

```
<p>This is my first CGI program!</p>
<p>My hobbies are
  <ul>
    <li>swimming</li>
    <li>music</li>
    <li>skiing</li>
  </ul>
</p>
```

A parameterized document is a Haskell function that constructs a sequence of document nodes. For example, WASH/CGI provides a standard document wrapper, `standardPage`, which takes two arguments, a title string `ttl` and a sequence of document nodes `nodes`, and returns a proper XHTML page. It is implemented as follows:

```
standardPage ttl nodes =
  html (do head (title (text ttl))
           body (h1 (text ttl) >> nodes))
```

This small example already demonstrates that our notation for documents easily scales to arbitrary parameterization. For this kind of document template to work satisfactorily, it is crucial that the `nodes` parameter can construct a *sequence* of document nodes, not just a single node. With WASH/CGI, `nodes` may provide attributes for the enclosing `body` element as well as an arbitrary number of element and text nodes which are to become children of `body`.

[3]The reason for having multiple concatenation operators will become clear later in Sec. 5.

3.2 Valid Documents

A frequently recurring problem with generated XHTML documents is their validity[4]. A valid document is well-formed and adheres to its DTD, in this case the XHTML DTD [40]. Well-formedness just means that opening and closing tags are properly nested. Beyond that, the DTD places restrictions on which child nodes (attributes, text, other elements) may be used with which elements. The child elements and text are also subject to ordering constraints imposed by regular expressions. While it is easy to check the validity of a static document using one of the existing checkers (*e.g.*, [37]), it is not easy to guarantee that a program generating XHTML will only produce valid documents.

Since the actions of the document sublanguage construct an internal tree representation of the document without exposing its string rendition to the programmer, well-formedness of the generated documents comes for free. To guarantee validity, it is possible to impose all constraints defined in a DTD, by exploiting Haskell's type class-based overloading mechanism [35]. In practice, it turns out to be too restrictive to program fully DTD-compliant document generators in that way. For that reason, the document sublanguage approximates validity with *quasi validity*. Quasi validity enforces the parent-child restrictions from the DTD, but it ignores the ordering constraints. See Sec. 9 for further discussion.

WASH/CGI enforces quasi validity through the types of the construction functions for document nodes:

```
text    :: (Monad m, AdmitChildCDATA e)
        => String           -> WithHTML e m ()
ul      :: (Monad m, AdmitChildUL e)
        => WithHTML UL m a -> WithHTML e m a
```

The type parameter `e` of the `WithHTML` data type determines the context (*i.e.*, the name of the parent element) in which a particular node constructor is used. To apply the `text` node constructor, the context must admit a child of type `CDATA` as indicated by `AdmitChildCDATA e`. Similarly, the `ul` function can only be used in a context that admits a `ul` child element. In addition, it opens a new context `WithHTML UL m a`, in which all constructors that produce admissible children for `UL` may be used. There is an analogous mechanism for checking attribute occurrences.

WASH provides, for each kind of document node, a type and a type class. For example, the type `CDATA` models a text node and the type class `AdmitChildCDATA` models the set of nodes that take a text node as a child, the type `UL` models an element of name `ul` and the type class `AdmitChildUL` models the set of nodes that take a `ul` element as a child, and so on. The values of these types are not interesting, the types are only present to make information about the document available to Haskell's type checker.

The use of types for document nodes scales nicely to parameterized documents. For example, the `standardPage` function from Section 3.1 has type

[4] Or rather lack thereof.

```
standardPage :: (Monad m, AdmitChildHTML context) =>
        String -> WithHTML BODY m a -> WithHTML context m a
standardPage ttl nodes =
  html (do head (title (text ttl))
           body (h1 (text ttl) >> nodes))
```

The type says that the `ttl` argument must be a string, that `nodes` must be a sequence of nodes admissible as children to a `body` element, and that the result must be used in a context that admits an `html` element. In this case, the type accurately reflects the requirements imposed by the XHTML DTD.

4 The Session Sublanguage

To proceed from the simple one-shot request-response model dictated by HTTP to a more session-oriented programming model, we first discuss what comprises a session for an interactive Web service. Once we have laid out the requirements and the implementation options for sessions, we explain the design of WASH/CGI's session sublanguage.

4.1 Sessions on the Web

For Web-scripting purposes, a session is an alternating sequence of Web-forms and form submissions starting with a form submission:

POST $\mathtt{url}_0$; FORM $\mathtt{f}_1$; POST $\mathtt{url}_1$; FORM $\mathtt{f}_2$; POST $\mathtt{url}_2$; ...

Each `POST` stands for an HTTP request sent by a client and each `FORM` stands for the corresponding response. Each $\mathtt{url}_i$ is a URL [2] that addresses a particular resource on a server. For simplicity, we consider data that accompanies `POST` requests part of the $\mathtt{url}_i$. Two conditions make such a sequence a session:

1. Each `FORM` response of the server is determined by the prefix of the sequence up to the response.
2. For each request POST $\mathtt{url}_i$, the $\mathtt{url}_i$ is drawn from a set of URLs determined by FORM $\mathtt{f}_i$, with $\mathtt{url}_0$ drawn from a set of external entry points.

The concept of a session implies the existence of a *session state* that is shared between the client and the server. Unfortunately, this is at odds with the underlying transport protocol of the Web, HTTP [8]. Due to its stateless nature, it is simple to implement, but it leaves the burden of implementing sessions to the application programmer. To understand the issues involved in this, we examine the computation of the `FORM` response from its request more closely.

Whenever the Web server receives a request for a particular kind of URL[5], it does not simply deliver the contents of the file addressed by the URL, but

[5]For example, referring to a file in a directory which is script-enabled or a file with suffix `.cgi`, depending on the configuration.

rather executes the file as an external program and passes the parameters of the request to that external program. A standard convention for passing these parameters is the CGI standard [5], hence the name "CGI scripts" for these programs. The CGI script computes the response `FORM`, returns it to the Web server, and terminates. Usually, the response is an XHTML document that contains XHTML `form` elements, so that the interaction can continue from that response.

To implement sessions under these conditions requires a CGI script to determine the current session state before it can do anything else[6]. The usual technique to recover the session state is to store a token in each `FORM` so that the token is passed along with each `POST url` operation continuing the session. Such a token may be implemented in a number of well-known ways.

1. With URL rewriting, the token gets incorporated directly in the URL, either as a path component or via the parameter mechanism.

2. Using cookies [19], the token gets stored on the client side and it is passed along with each `POST` operation for a qualifying URL.

3. Using hidden input fields [40], the token gets stored in the response document itself (without being displayed) and is passed to the server along with other parameters entered by the user.

In each case, the application maintains a mapping from tokens to current session states, to recover the session state from the token. Quite often, the token only identifies the session but not the position inside the session.

Unfortunately, the token-based approach leads to a number of problems which are due to the way that users can navigate the Web [12].

1. Users can hit the back button to go back in the history of the browser and resume the interaction from some earlier point.

2. Users can clone the session at any point and continue with each clone in different ways.

3. Users can store the current form in a file and may resume the interaction several months later. (Bookmarking)

Items 1 and 2 indicate that using a session token and one associated snapshot of the session state in the server is a bad idea. Using the back button will bring the server's state out of sync with what the user is actually looking at. Cloning is even worse because it mixes up the states from two or more interaction sequences.

Clearly, a new token must be assigned for each interaction step (*i.e.*, for each generated form). However, this creates another problem because the server will have to store very many token-state associations. As indicated in item 3, there

[6]The compile-time part of the session state may be provided via the name of the script, but remembering run-time values requires other techniques.

is no way for the server to figure out which of these tokens are still required and which can be discarded. Quite often, timeouts are used to address this problem, but they do not really solve the problem.

Hence, we are pursuing another option that relies on the observation that hidden fields may store large chunks of data whereas URLs and cookies are restricted in size. The idea is not to store *any* state information on the server but rather store the session information in the token itself. The server then puts the token into a hidden field of the response page returned to the client.

That way, the server need not maintain a token-state association table (hence, there are no problems with discarding entries of such a table), restoring the state amounts to reading the token, and the state can be regenerated at each interaction step. In consequence, our approach works with arbitrary combinations of going back, cloning, and bookmarking.

Programmers often specialize the compile-time part of the session state by implementing each interaction step in a different CGI script. Keeping this set of programs in sync requires

- that each script run for the request `POST` $\texttt{url}_i$ only generates forms `FORM` $\texttt{f}_{i+1}$ that are understood by $\texttt{url}_{i+1}$ and
- that all scripts agree on the session state that is passed along and the way it is represented.

This is a non-trivial software engineering problem and it is the cause of annoying errors in many existing Web applications [11]. These problems are not tied to CGI scripts but recur for other scripting techniques like Servlets and Java Server Pages.

4.2 Sessions in WASH/CGI

The WASH/CGI approach addresses these problems by

- enabling the programmer to implement the entire application in one program and
- providing a transparent notion of session state to the programmer.

In fact, the programmer's perspective on the Web application is similar to writing a GUI program. The difference is that the GUI is specified in XHTML and that the structure of the interaction is limited due to the underlying request-response protocol.

Sessions are constructed from documents by using two operators.

```
ask  :: WithHTML XHTML_DOCUMENT CGI a -> CGI ()
tell :: (CGIOutput a) => a -> CGI ()
```

Some explanation of the types involved is in order before we can explain `ask` and `tell` themselves.

A value of type `CGI a` is a *CGI action* that computes a result of type `a`. CGI actions are the basic building blocks for sessions, they comprise the session sublanguage. CGI actions can be composed (*i.e.*, put in sequence) using the operator `>>=` and the `do` notation.

A value of type `WithHTML context CGI a` is an *HTML action* that creates a sequence of document nodes. In fact, the XHTML construction operations like `p`, `ul`, and `li` introduced in Section 3 all have types like

```
p :: (Monad m, AdmitChildP context) =>
        WithHTML P m a -> WithHTML context m a
```

The context `XHTML_DOCUMENT` in the type of `ask` indicates that the context expects a single `html` element to construct an XHTML document.

The `(Monad m)` constraint indicates that `m a` is the type of an *action returning a result of type* `a`. In the type `m a`, the type variable `m` ranges over *type operators* (constructors) whereas `a` ranges over types, as usual. That is, the implementation of `p` will work for all kinds of actions that we substitute for `m`. In many cases, `m` will be `CGI`, but sometimes other kinds of actions, like `IO`, are used.

Coming back to `ask` and `tell`, `ask` is a function that takes an HTML action and yields a CGI action. Conceptually, this CGI action constructs the document and then presents it to the browser.

`tell` is an overloaded function (as indicated by the `(CGIOutput a) =>` constraint of the type) that maps any value that can by converted to an HTTP response to a CGI action. The action delivers this response to the browser. For example, the response type `a` can be an HTML `Element`, a `String` (which is shown as `text/plain` document), a `FileReference` (which specifies a content type and a file name to be sent to the browser), a `StatusCode` (as defined in the HTTP standard [8]), or a `Location` (which specifies a redirection URL).

The `ask` operation is different from `tell` in that the XHTML document created by its HTML action argument may contain CGI actions that lead to further interaction steps. In contrast, a response returned by `tell` terminates an interaction sequence.

The typical script is specified as a single CGI action. Since Haskell only accepts an IO action as a main program, there is a function `run` that maps a CGI action to an IO action. Conversely, there is a function `io` that embeds an IO action into a CGI action.

```
run  :: CGI () -> IO ()
io   :: (Read a, Show a) => IO a -> CGI a
```

The constraint `(Read a, Show a) =>` in the type of `io` restricts the values that may be returned from IO actions embedded in CGI actions[7].

At this point, we can write our first script.

```
main =
```

[7]This is further discussed in Sec. 8.

```
  run (ask (standardPage "Hello World!" myinfo))

myinfo =
  do p (text "This is my first CGI program!")
     p (do text "My hobbies are"
           ul (do li (text "swimming")
                  li (text "music")
                  li (text "skiing")))
```

The argument of `run` is a CGI action that displays the result of `standardPage` applied to the title `"Hello World!"` and the document `myinfo`. Since the latter document does not specify further CGI actions, the interaction is finished after the document is returned.

5 The Widget Sublanguage

In this section, we introduce the means to create interactive applications. In the first step, we attach simple continuation actions to a Web form. In the further steps, we introduce input widgets and explain how their values are passed to callback actions.

5.1 Continuation Buttons

Many operations of the widget sublanguages are wrappers for creating `input` elements with suitable attributes. The declaration

```
type HTMLField context a =
  WithHTML INPUT CGI () -> WithHTML context CGI a
```

defines a type synonym `HTMLField` that captures the common type for these wrapper operations.

A value of type `HTMLField context a` expects a sequence of attributes for the `INPUT` field (of type `WithHTML INPUT CGI ()`) and returns a one-element sequence with the input element[8].

Creating a button and attaching a CGI action to it relies on the operation

```
submit0 :: (AdmitChildINPUT context) =>
           CGI () -> HTMLField context ()
```

If `action :: CGI ()` is a CGI action, then `submit0 action` constructs an `input` element with attribute `type="submit"`. In the browser, this element is rendered as a button and clicking the button submits the form to the server. This submission will trigger the `action` on the server. According to the XHTML specification, an input element is only legal inside of a `form` element. The latter should be created using the `makeForm` operation:

[8]Technically, the `context` type should be restricted to contexts admitting an `input` element, but this restriction is not expressible with a type synonym.

Figure 1: Rendering of "Hello World!"

```
makeForm :: (AdmitChildFORM context) =>
        WithHTML FORM CGI a -> WithHTML context CGI ()
```

This operation is not called `form` because it does more than just creating a `form` element: It also fills in the attributes `action` and `method` and stores the current session token as a hidden field in the form[9].

To extend the example, we might display the hobbies on a different page. See Fig. 1 for a rendering of the code below.

```
main =
  run page1

page1 =
  ask (standardPage "Hello World!" (makeForm myinfo1))

myinfo1 =
  do p (text "This is my second CGI program!")
     submit0 page2 (attr "value" "Click for my hobbies")

page2 =
  ask (standardPage "My hobbies are" (makeForm myinfo2))

myinfo2 =
  ul (do li (text "swimming")
         li (text "music")
         li (text "skiing"))
```

The main novelty is in the line

```
     submit0 page2 (attr "value" "Click for my hobbies")
```

It specifies a submission button which executes the CGI action `page2` when clicked. The button displays the text specified by the `value` attribute.

The above combination of `ask`, `standardPage`, and `makeForm` is so common that it has become part of the standard library:

[9]See Sec. 8.1 for more information what is actually stored in this field.

```
standardQuery :: String -> WithHTML BODY CGI a -> CGI ()
standardQuery ttl nodes =
  ask (standardPage ttl (makeForm nodes))
```

5.2 Callbacks With Parameters

Input widgets are HTML actions that actually compute something. In addition to creating a suitable XHTML element with the correct set of attributes, a widget constructor also returns an *input handle*. It is via the input handle that the code can later extract the value entered into the widget. For example, the widget constructor

```
textInputField :: (AdmitChildINPUT context) =>
        HTMLField context (InputField String INVALID)
```

creates a textual input field and returns its handle of type `InputField String INVALID`. Upon creation, this handle is *invalid* (as indicated by the `INVALID` in the type) because it has not received a value, yet. In contrast to other approaches, the programmer is never required to name the input widget, thus eliminating a common source of errors[10]. Instead, the handle is bound to a variable and passed along to those callback actions that wish to inspect the value.

Passing an input handle to a callback action requires the use of the `submit1` operation. Its type is

```
submit1 :: (AdmitChildINPUT context) =>
        InputField a INVALID
        -> (InputField a VALID -> CGI ())
        -> HTMLField context ()
```

That is, in `submit1 handle callback`, `submit1` takes an invalid input handle, `handle`, gets the value for the handle and turns it internally into a valid input handle (indicated by `VALID` in the type of `callback`), and then passes the latter as a parameter to `callback`. The result is again an HTMLField, which takes attributes for the button and constructs its XHTML representation, an `input` with attribute `type="submit"`. Inside the callback action, the operation `value` extracts the value from a valid handle:

```
value :: InputField a VALID -> a
```

To illustrate parameter passing, we extend the example further by asking for the Web surfer's name before showing the hobbies. Figure 2 shows the browser's rendition of the code below.

```
main =
  run page1
```

[10]In those cases where the name of the input field is externally mandated (as with ECML [7]), the `name` attribute can be used explicitly.

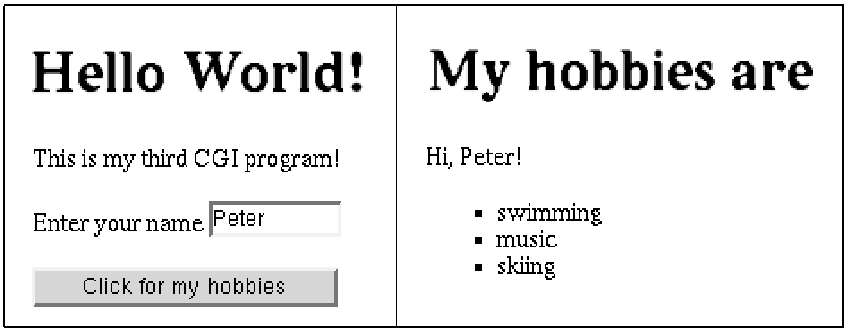

Figure 2: Rendering of parameterized "Hello World!"

```
page1 =
  standardQuery "Hello World!" $
  do p (text "This is my third CGI program!")
     inf <- p (do text "Enter your name "
                  textInputField (attr "size" "10"))
     submit1 inf page2 (attr "value" "Click for my hobbies")

page2 inf =
  standardQuery "My hobbies are" $
  do p (text "Hi, " ## text (value inf) ## text "!")
     ul (do li (text "swimming")
            li (text "music")
            li (text "skiing"))
```

The `$` infix operator used in the above code stands for function application. Using the infix operator (instead of just juxtaposition) avoids the need for putting the two `do` blocks in parentheses.

At this point, we also have to explain how the values computed by HTML actions are propagated. The notation

```
do ...aaa...
   x <- htmlAction
   ...bbb...
   finalAction
```

extracts the value computed by `htmlAction` and binds it to `x`. The variable `x` is visible inside the remainder of the action, *i.e.*, in `...bbb...` and `finalAction`. The result computed by the entire `do` expression is the result of `finalAction`.

Each XHTML constructor (*e.g.*, `p`, `ul`, `li`, ...) returns the value computed by its argument action, as evidenced by their type, *e.g.*

```
p :: (AdmitChildP context) =>
       WithHTML P CGI a -> WithHTML context CGI a
```

The operators `##` and `>>` differ in the way that they propagate the results of their operands. They can be easily explained in terms of the `do` notation:

```
act1 ## act2 =
  do x <- act1          -- result is returned
     act2               -- result is ignored
     return x

act1 >> act2 =
  do act1               -- result is ignored
     act2               -- result is returned
```

Hence, in

```
  do p (text "This is my third CGI program!")
     inf <- p (do text "Enter your name "
                  textInputField (attr "size" "10"))
     submit1 inf page2 (attr "value" "Click for my hobbies")
```

the result of the `do` block is the result of `textInputField ...`, that is, the input handle for the field. This handle is then passed through by the `p` constructor so that the variable `inf` is bound to the input handle.

5.3 Typed Input Fields

The `textInputField` of the previous subsection was only good for input of strings. However, there is a more general operation

```
inputField :: (AdmitChildINPUT context, Reason a, Read a) =>
        HTMLField context (InputField a INVALID)
```

that creates a *typed input field* for any type `a`, provided this type belongs to type classes `Reason` and `Read`. The constraint `Reason a` implies that the type `a` has an operation `reason :: a -> String` which provides an explanation of the input syntax for a value of type `a`. The constraint `Read a` implies that type `a` has an operation `read :: String -> a` which converts a string into a value of type `a`. The latter class `Read` is a standard type class of Haskell [15]. Of course, the explanation generated by `reason` should match the syntax parsed by `read`.

Clearly, there is no guarantee that a Web surfer enters a parsable value into a type field. For that reason, the `submit1` function *validates* each handle before it passes it on to the callback action. A handle is validated only if its input is parsable to a value of the desired type. Hence, a valid handle guarantees the presence of a value. If `submit1` cannot validate a handle passed to it, it redisplays the last form with the erroneous input marked.

As an example, consider the code below which reads an integer an displays it again.

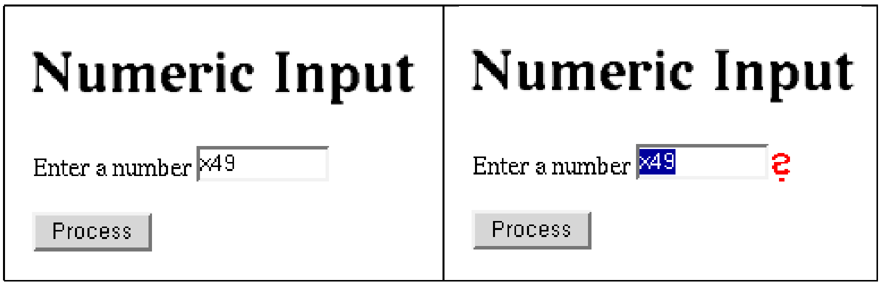

Figure 3: Capturing errors with typed input fields

```
main =
  run page1

page1 =
  standardQuery "Numeric Input" $
  do inf <- p (do text "Enter a number "
                  inputField (attr "size" "10"))
     submit inf page2 (attr "value" "Process")

page2 inf =
  let n :: Int
      n = value inf
  in
  standardQuery "Your Number" $
  p (text "Your number was " ## text (show n) ## text "!")
```

In `page1` the essential change is the use of `inputField` instead of `textInputField`. In `page2` the `n :: Int` in the `let` declares that `n` is an integer and then extracts the integer value from the input handle `inf`. The function `show` converts (in this example) an integer to a string.

Without any further programming, the program behaves as shown in Fig. 3. The left-hand snapshot shows the browser window after typing in an incorrect input, but before clicking the "Process" button. The right-hand snapshot shows the window after clicking the button. The erroneous input field is marked with a question mark and its contents is selected.

With this approach it is easy to create an input field with customized syntax. It amounts to defining a new type and providing a parser and a `reason` function for it. Doing so requires some skill in Haskell programming. The following example defines a type `NonEmpty` for non-empty strings and then provides a parser (`instance Read NonEmpty`) and a description of the input syntax (`instance Reason NonEmpty`).

```
newtype NonEmpty = NonEmpty { unNonEmpty :: String }
-- `NonEmpty str´ constructs a NonEmpty value from a string
```

```
-- `unNonEmpty n´ extracts the string from a NonEmpty value

instance Read NonEmpty where
  readsPrec i =
    do str <- many1 anyChar     -- regular expression: .+
       return (NonEmpty str)    -- construct value of type NonEmpty

instance Reason NonEmpty where
  reason _ = "non empty string"
```

To avoid an extended discussion of Haskell, we will not comment on it further at this point. However, the code snippet should make it clear that defining a new input field type is simple programming task.

Using the new `NonEmpty` type is as simple as using the numeric input in the example above:

```
page2 inf =
  let nonemptystring = unNonEmpty (value inf) in
  -- nonemptystring is guaranteed to be a non-empty string
```

5.4 Non-textual Input Fields

The remaining input widgets (buttons, check boxes, selection boxes, images, etc) are all be expressed in terms of typed input handles in WASH/CGI. For example, check boxes yield handles of type `InputField Bool INVALID`, buttons can be made to return any type `a` provided values of type `a` have a string representation, selection boxes can return any value whatsoever, and images yield a handle of type `InputField (Int, Int) INVALID`, *i.e.*, they return a pair of integers. Details about them may be found in the WASH/CGI documentation [36].

5.5 Combining Input Fields

Up to now, the examples have only dealt with callback actions that take a single handle as a parameter. To pass multiple handles to a callback requires a more general type than the one given for `submit1`. The function `submit` is this upwards-compatible generalization for `submit1` and its type is

```
submit :: (AdmitChildINPUT context, InputHandle h) =>
        h INVALID -> (h VALID -> CGI ()) -> HTMLField context ()
```

The type `h INVALID` generalizes the former `InputField a INVALID`. The type class `InputHandle` restricts the possible instantiations of `h`[11]. Creating a pair of handles requires the use of a special pairing data constructor,

```
F2 :: a x -> b x -> F2 a b x
```

[11] Note that `h` ranges over *type constructors* and that the type class `InputHandle` specifies a set of *type constructors*, not plain types. This is a standard feature of Haskell [15].

Furthermore, there is a rule (the instance declaration for class `InputHandle` and type `F2`) that states that `F2 h1 h2` belongs to `InputHandle` if both `h1` and `h2` are `InputHandle`s, too. For convenience, WASH/CGI defines types similar to `F2` for other tupling operators and for lists.

As an example, the following code may be used to query for a name and a password. The `check` function enforces that both name and password fields are non-empty. The function `passwordInputField` works just like `inputField` but generates an `input` element with `type="password"` so that the characters entered are not echoed.

```
login =
  standardQuery "Login" $ table $
  do nameF <- tr (td (text "Name ") >>
                  td (inputField (attr "size" "8")))
     passF <- tr (td (text "Password ") >>
                  td (passwordInputField (attr "size" "8")))
     tr (td (submit (F2 nameF passF) check (attr "value" "LOGIN"))))

check (F2 nameF passF) =
  let name = unNonEmpty (value nameF)
      pass = unNonEmpty (value passF)
  in
  standardQuery "Authenticate" ...
```

5.6 Virtual Input Fields

In some cases, it is useful to have input fields that do not have a screen rendition. An example for such a field arises with the use of the standard `radio` input type. An input element of type `radio` gives rise to a radio button. In HTML, all radio buttons of the same name form an implicit radio group, that is, the browser makes sure that at most one of the radio buttons of this group is pressed.

WASH/CGI makes this concept explicit by providing a function

```
radioGroup :: Read a =>
        WithHTML context CGI (RadioGroup a INVALID)
```

Its use fixes the value returned by a radio button that belongs to this group to have type `a`. Moreover, the returned handle of type `RadioGroup a INVALID` is used to attach buttons to the group:

```
radioButton :: (AdmitChildINPUT context, Show a) =>
        RadioGroup a INVALID -> a -> HTMLField context ()
```

This design makes sure that

- radio buttons are not grouped by accident, they always have to be created inside a specific radio group;
- all buttons of a group return values of the same type; and

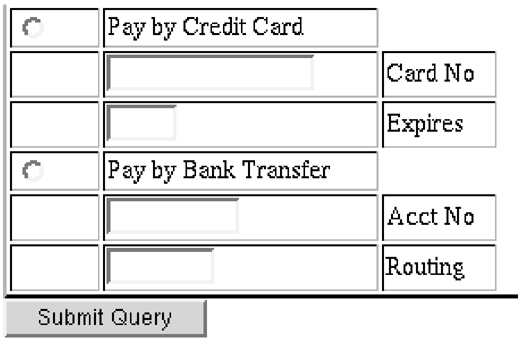

Figure 4: Rendition of the payment form

- error processing can be done once and for all.

The next subsection has an example of using radio buttons.

5.7 Decision Trees

The `submit` function explained above requires that each callback validates one fixed subset of the input fields present in the form. For some forms, this requirement is too restrictive. For example, consider a typical form fragment from the payment processing part of a shopping application rendered in Fig. 4:

```
data ModeOfPayment = PayCredit | PayTransfer
-- declares an enumerated data type ‘ModeOfPayment´
-- with two symbolic values ‘PayCredit´ and ‘PayTransfer´

do rg <- radioGroup empty
   tr (do td (radioButton rg PayCredit empty)
          td (text "Pay by Credit Card"))
   ccnrF <- tr ((td empty >> td (inputField (attr "size" "16")))
              ## td (text "Card No"))
   ccexF <- tr ((td empty >> td (inputField (attr "size" "5")))
              ## td (text "Expires"))
   tr (do td (radioButton rg PayTransfer empty)
          td (text "Pay by Bank Transfer"))
   acctF <- tr ((td empty >> td (inputField (attr "size" "10")))
              ## td (text "Acct No"))
   routF <- tr ((td empty >> td (inputField (attr "size" "8")))
              ## td (text "Routing"))
   tr (td (submit (F5 rg ccnrF ccexF acctF routF) payCGI empty))

payCGI (F5 rg ccnrF ccexF acctF routF) =
  let ccnr = unCreditCardNumber (value ccnrF)
```

```
      expMonth = cceMonth (value ccexF)
      acct = unAllDigits (value acctF)
      rout = unAllDigits (value routF)
  in
  standardQuery "Payment Confirmation"
  (text "Thanks for shopping at TinyShop.Com!")
```

It contains a group of two radio buttons and four textual input fields. The variable `rg` is the handle to the value representing the radio buttons' selection and the variables ending in `F` are handles to the values of the other input fields. The last line submits the handles of the two radio buttons, `rg`, the credit card number, `ccnrF`, its expiration date, `ccexF`, the account number, `acctF`, and its routing number, `routF`, to the callback `payCGI`. The form is fully typed and set up to reject unreasonable inputs without any further programming due to the way the handles are accessed in `payCGI`. For example, the credit card number and the account number are both submitted to the Luhn check[12], the expiration date must have the form as printed on the card, and so on. Before submitting the form, a customer would select one mode of payment by clicking one of the radio buttons and then fill in the two fields associated to it. However, the validation built into `submit` tries to validate *all five* input handles, thus insisting that both credit card data *and* the full account information are present!

To handle such cases requires introducing an additional data structure, a decision tree. A decision tree is either a leaf that contains an unconditional `CGI` action or it is a node that contains a group of input handles (of type `h INVALID`) and a function that maps their validated versions to the next decision tree. Hence, decision trees have two constructor functions:

```
dtleaf :: CGI () -> DTree x y
dtnode :: InputHandle h =>
        h INVALID -> (h VALID -> DTree x y) -> DTree x y
```

The first, `dtleaf`, creates a trivial decision tree from a CGI action. The `dtnode` constructor takes an invalid input handle, validates it, and (upon successful validation) passes it to the second argument that builds the rest of the decision tree. The function `submitx` transforms a decision tree into a submit button that can be put into a Web form:

```
submitx :: (AdmitChildINPUT context) =>
        DTree context y -> HTMLField context y ()
```

Submitting through the trivial decision tree, `submitx (dtleaf action)`, is equivalent to `submit0 action` and submitting through a one-stage decision tree, `submitx (dtnode handles (dtleaf .  callback))`, is equivalent to `submit handles callback`.[13]

[12] A standard checksum algorithm for credit card numbers and bank account numbers, see for example [23].

[13] The infix dot operation, `.`, stands for function composition.

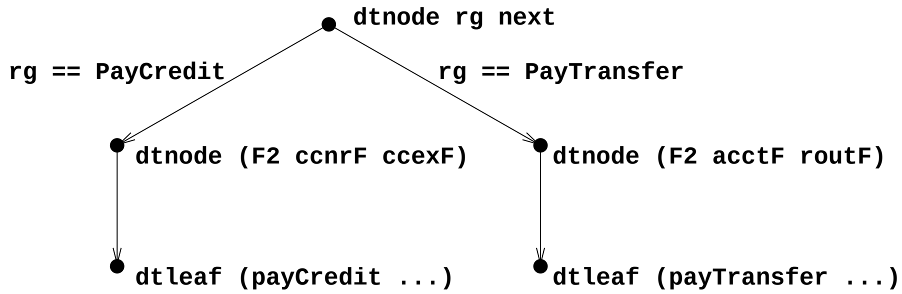

Figure 5: Decision tree for the payment example

The decision tree required in the example first dispatches on the payment mode extracted from the handle `rg` for the radio buttons and then validates either the credit card fields or the account information. The leaves of the tree just invoke the respective code for processing the payment, `payCredit` or `payTransfer`, where the computed price is assumed to be present in variable `price`. Figure 5 shows the tree constructed in the code below.

```
   let paymentOptions = dtnode rg next
       next paymodeF  =
         case value paymodeF of
           PayCredit   -> dtnode (F2 ccnrF ccexF)
                                (dtleaf . payCredit price)
           PayTransfer -> dtnode (F2 acctF routF)
                                (dtleaf . payTransfer price)
   in  tr (td (submitx paymentOptions empty))

payCredit price (F2 ccnrF ccexF) =
  let ccnr = unCreditCardNumber (value ccnrF)
      expMonth = cceMonth (value ccexF)
  in standardQuery "Confirm Credit Payment" $
     p $ text "Thanks for shopping with us!"
```

The `payTransfer` function is similar.

5.8 Summary

The examples above demonstrate that the widget sublanguage provides the means of specifying and combining input elements in almost arbitrary ways. Fragments of interactions can be specified and implemented separately, and later be combined on one Web page. Even in this combination, all interactions are still fully type safe and every submitted data is checked for validity without programmer intervention.

Beyond the facilities discussed, it is also possible to compose new widgets from existing ones. However, the techniques required to do so are out of the scope of this paper.

6 The Persistence Sublanguage

WASH/CGI provides two abstractions for persistent state beyond the builtin notion of session state: server state and client state. The main difference between these two is their scope. While the server state is globally visible to all programs running on the server, the client state is only visible to scripts started from one particular client. Hence, the global state acts as a database with which scripts may interact while the client state keeps information particular to one client (*i.e.*, cookies [19]).

Compared to the usual string-based interfaces to server and client state, the WASH/CGI interface is fully type-safe, even across separate programs. That is, any attempt to store data of one type and retrieve it as a value of another type will fail.

6.1 Server-side State

Server-side state is defined as a parameterized, abstract data type `T a` with the following signature.

```
init :: (Types a, Read a, Show a) => String -> a -> CGI (T a)
get  :: (Types a, Read a)         => T a         -> CGI a
set  :: (Types a, Read a, Show a) => T a -> a    -> CGI (Maybe (T a))
add  :: (Types a, Read a, Show a) => T [a] -> a  -> CGI (T [a])
current :: (Types a, Read a)      => T a         -> CGI (Maybe (T a))
```

A value of type `T a` is a handle to a *persistent entity* (*i.e.*, a value in the server-side state) of type `a`. These handles are the only means of accessing the state from the script. The idea is that each handle uniquely identifies a particular *value* of the persistent entity. Hence, getting a handle may be considered as getting a snapshot of the persistent entity. We call a handle *current* as long as its snapshot still reflects the current server-side state. In the implementation, currency of a handle is checked using version numbers. This design addresses two issues.

First, handles help keep the session state small: Since a handle identifies a value, the session state need not contain the value which may be very large.

Second, a handle allows the script to explicitly notice changes to the value by concurrently executing scripts. The script writer can choose to either continue to use the value denoted by the handle (which is sufficient as long as no `set` operation is attempted) or to observe the new value by discarding the handle and obtaining a current handle. The latter choice places the obligation on the programmer to check if the new value is still consistent with the assumptions of the program.

The constraints placed on the types of the interface are of technical nature. The constraint `Types a` indicates that there must be a term representation for the type `a` and the constraints `Read a` and `Show a` indicate that there must be conversion functions to and from strings for values of type `a`.

The operation `init name initialValue` attempts to create a new persistent entity with name `name` and initialized with `initialValue`. If it succeeds in creating a new entity, it returns a handle pointing to `initialValue`. If it cannot create a new entity because there is already a persistent entity of the same name and type, then the returned handle points to its current value.

The operation `get handle` returns the value of `handle`. This value may not be current because the server-side state may have evolved further after creation of the handle.

The operation `set handle newValue` attempts to overwrite the value of the persistent entity pointed to by `handle` with `newValue`. If `handle` is current, then it succeeds and returns a handle to the new value. Otherwise, it fails. In the latter case, it is up to the programmer to further deal with the situation.

The operation `add handle newValue` deals with the special case where the persistent value is a set, represented by a list where the order of elements does not matter. In that case, adding `newValue` to the set always makes sense, regardless whether the `handle` is current. Hence, `add handle newValue` never fails and the value of the handle returned from the `add` operation is guaranteed to contain `newValue`.

Finally, the operation `current handle` returns a current handle to the persistent entity pointed to by `handle`. This operation is potentially dangerous because it explicitly allows to observe changes to the persistent entity made by concurrently running scripts. After using `current`, it is the obligation of the programmer to check that the value identified by the handle is still the intended one.[14]

The problem in implementing this interface is that the server might be forced to keep all different values of a persistent entity until the end of time. However, unlike session state, users will expect that server-side state evolves. Hence, the server can reasonably garbage collect values that have not been used recently and discard them. A script that tries to `get` a handle that has been discarded receives an exception which can be translated into a user-level error message.

Here is a snippet from an example program that uses a persistent entity to store a list of high scores for a game.

```
gameover myScore =
  do initialHandle <- init ("hiscores") []
     currentHandle <- add initialHandle myScore
     hiScores      <- get currentHandle
     standardQuery "Hall of Fame" (ul (mapM_ showItem hiScores))
```

[14]The semantics of persistent entities and the `current` operation in particular has been misinterpreted by Graunke and others [11] who claim that the WASH/CGI implementation of server-side state is faulty.

```
showItem (name, score) =
  li (do text name
         text ": "
         text (show score))
```

In the second line, the variable `initialHandle` becomes a handle to the current value of the persistent entity named `hiscores`. It contains a list of high scores of the game. The next line adds the new score to the list of scores as explained above. The fourth line gets a list of scores that is guaranteed to contain `myScore`. The sixth line generates a page containing an unordered list where the individual list items are generated by applying the function `showItem` to each element of `hiScores` (that's what `mapM_` does). The function `showItem` takes a pair consisting of a string `name` and an integer `score` and generates a list item from it.

6.2 Client-side State

The interface for accessing persistent client-side state is very similar to the one for server-side state, but there are subtle differences in their behavior. The operations are also given in terms of a parameterized abstract data type `T a`[15]. They provide an abstraction layer on top of the implementation in terms of cookies [19].

```
check   :: (Read a, Show a, Types a) => String      -> CGI (Maybe (T a))
init    :: (Read a, Show a, Types a) => String -> a -> CGI (T a)
get     :: (Read a, Show a, Types a) => T a         -> CGI (Maybe a)
set     :: (Read a, Show a, Types a) => T a -> a    -> CGI (Maybe (T a))
current :: (Read a, Show a, Types a) => T a         -> CGI (Maybe (T a))
delete  :: (Types a)                 => T a         -> CGI ()
```

Although a handle still denotes a snapshot of the client-side state, the value corresponding to this snapshot may no longer be available by the time the script accesses the handle. This can happen if the user runs several scripts dealing with the same client-side value in different browser windows at the same time.

For that reason, `get handle` may fail! This is reflected in its returning a value of type `Maybe a`: if successful, it returns `Just theValue`, otherwise it returns `Nothing`. In the latter case, the script must obtain a new current handle to continue.

The operations, `init`, `set`, and `current`, work as for server-side state. In addition, there are operations `check`, to check for the existence of a cookie and return a handle on success, and `delete`, for removing a cookie. There is currently no way for a program to check if setting the cookie on a client is successful. If the client does not return the cookie with the next request an error message will be posted to the browser saying that cookies need to be enabled to run the script. Beyond the operations shown here, there are further operations to manipulate the expiration time of client-side state.

[15]A program that uses both, client-side and server-side state, must keep them apart using Haskell's module system.

WASH/CGI does not impose any restriction on where in the code the client-side state may be read or written. This is in contrast to other Web programming libraries that expose details of the cookie implementation by requiring that all cookie related computations must be completed before any part of the response page is sent to the browser. This restriction is due to the fact that cookies are part of the HTTP headers, which must be flushed before the contents of the page are sent.

7 A Complete Example

In this section, we give the code for a complete example, a guess-number game inspired by code from various tutorials on Servlets [32]. In the game, the script first generates a secret number. Then the user tries to guess the number guided by the answers "to small" or "to large". Guessing continues until the number has been found. To make the example more interesting, we are using a cookie to store the secret number (although session state would be sufficient) and we have added a high score list to demonstrate the use of persistent entities. The full code comprises 83 lines. The script is part of a suite of demo programs on the WASH/CGI web page [39].

In the code, the cookie (client-side state) module is accessed via name `C`, whereas the persistence (server-side state) module is named `P`. Hence, the operations as well as the types from either module are prefixed with either "`C.`" or "`P.`". In instance, `P.T` is the type of a server-side state handle whereas `C.t` is the type of a client-side state handle.

We start with code to initialize the high score list to the empty list of scores. The name of this persistent entity is `GuessNumber`. A score is pair of the number of guesses used and the name of the player.

```
type NumGuesses = Int
type PlayerName = String
type Score = (NumGuesses, PlayerName)

highScoreStore :: CGI (P.T [Score])
highScoreStore = P.init "GuessNumber" []
```

This code just defines `highScoreStore` as a CGI action that initializes a handle to the persistent store. The action is only executed when the action is used.

```
type SecretNumber = Int
setNumber :: SecretNumber -> CGI (C.T (NumGuesses, SecretNumber))
setNumber secret = C.create "theNumber" (0, secret)
```

The operation `setNumber` defines the initializer for the cookie. The cookie contains a pair of the current number of guesses and the secret number.

```
main :: IO ()
main = run mainCGI
```

```
message = text "I've thought of a number between 1 and 100."

mainCGI :: CGI ()
mainCGI = once
 (standardQuery "Guess a number"
   (do submit0 playTheGame (attr "value" "Play the game")
       submit0 admin (attr "value" "Check scores")))
  >> mainCGI
```

The first two lines are standard for starting a WASH/CGI program. The action `mainCGI` implements the start of the interaction. It presents the player to either run the program or check the high scores, which are both activated using a parameterless button (`submit0`). As soon as either of these choices terminates, the interaction is restarted (via the recursive invocation of `mainCGI` in the last line).

Of the two choices the `admin` function is the simpler one.

```
admin =
  do highScoreList <- highScoreStore
     highScores <- P.get highScoreList
     standardQuery "GuessNumber - High Scores" $ do
       table $ do
         attr "border" "border"
         tr (th (text "Name") ## th (text "# Guesses"))
         mapM_ showScore (sort highScores)
       submit0 (return ()) (fieldVALUE "Continue")

showScore (guesses, name) =
  tr (td (text name) ## td (text (show guesses)))
```

`admin` first initializes a handle `highScoreList` to the high score store and reads its contents into `highScores`. Then it presents a high score table with two columns, one for the player's name and one for the number of guesses. The function `mapM_` applies function `showScore` to each element of the sorted list of high scores and pastes the resulting HTML actions together. The function `showScore` translates a score into one row of the enclosing HTML table. The `Continue` button at the end of the page takes the client back to the start.

The other choice in the menu is the `playTheGame` function.

```
playTheGame =
  do aNumber <- io (randomRIO (1,100))
     numHandle <- setNumber aNumber
     play numHandle message
```

It first chooses a random number between `1` and `100` (inclusive) using the predefined I/O action `randomRIO`. Then it stores the number (and the number of guesses so far) in a cookie and falls through to function `play`.

The function `play` is parameterized over the message `aMessage` to display. From the main menu it is called with `text "I've thought of a number ..."`. This message is not restricted to be a text, it may contain any document nodes that are appropriate at this place. The actual code fragment dealing with the interaction is placed in a paragraph after the message.

```
play aHandle aMessage =
  standardQuery "Guess a number" $
    do aMessage
       p (do text "Make a guess "
             guessF <- inputField (attr "size" "3")
             submit guessF (processGuess aHandle) empty)
```

`play` is straightforward. It displays the message, asks for a guess, extracts a handle to the guess from the input field and passes it to `submit`. The callback function `processGuess` performs the actual work.

```
processGuess aHandle guessF =
  let aGuess = value guessF in
  do mValue <- C.get aHandle
     case mValue of
       Just (nGuesses, aNumber) ->
         let nGuesses' = nGuesses + 1 in
         if aNumber == aGuess
         then do C.delete aHandle
                 youGotIt nGuesses' aNumber
         else do Just aHandle' <- C.set aHandle (nGuesses', aNumber)
                 play aHandle' (do text "Your guess "
                                   text (show aGuess)
                                   text " was too "
                                   if aGuess < aNumber
                                     then (b (text "small."))
                                     else (b (text "large.")))
       Nothing ->
         standardQuery "Don't do that!" $ do
           text "You are trying to outwit me by playing with "
           text "the back button and by cloning windows!"
           submit0 (return ()) (fieldVALUE "Restart")
```

Given the handle to the cookie and the next guess, `processGuess` first tries to read the current value of the cookie. If this fails for some reason (case `Nothing`), then it displays an appropriate error message. Otherwise, it increments the number of guesses and checks if the guess was correct. If it was, then the cookie is deleted and control is passed to `youGotIt` to display a winning message. Otherwise, the cookie is updated to the new number of guesses and `play` is invoked recursively with a message (some text nodes and a `b` element) announcing the result of comparing the two numbers.

```
youGotIt nGuesses aNumber =
  standardQuery "You got it!" $
  do text "CONGRATULATIONS!"
     br empty
     text ("It took you " ++ show nGuesses ++ " tries to find out.")
     br empty
     text "Enter your name for the hall of fame "
     nameF <- textInputField (attr "size" "20")
     submit nameF (addToHighScore nGuesses) empty
```

There is not much to say about `youGotIt`. It displays a parameterized HTML page and then falls through to `addToHighScore` as soon as a name is entered in the input field.

```
addToHighScore nGuesses nameF =
  let name = value nameF in
  if name == "" then return () else
  do highScoreList <- highScoreStore
     P.add highScoreList (nGuesses, name)
     return ()
```

The function first obtains the contents of the input field (unrestricted text). If the field is non-empty, then it gets a handle to the persistent `highScoreList` and adds the new score to it. In either case, the interaction wraps back to the start.

8 Implementation

The description of the implementation comes in two parts. The first subsections describe the techniques in general terms and the last subsection gives implementation details specific to Haskell. This last subsection is not essential for understanding the techniques used. However, it conveys that the techniques map to Haskell code in a natural way.

8.1 Sessions

In Section 4, we stated that each form sent to the browser contains a hidden token for the session state. Clearly, sheer size of the data does not permit to store a memory snapshot of the application in the form.

At this point, Haskell's purely functional nature becomes essential. Since all functions definable in Haskell are free of side effects, they deliver the same result whenever they are applied to the same arguments. I/O is achieved through an abstract data type `IO a` of I/O actions. Manipulation of these actions is still free of side effects, actions are only executed when they are returned from the main function. Interactive programs are possible because later I/O actions may depend on the results of earlier I/O actions.

Since all external influences to a Haskell program are mediated through I/O actions, it is possible to replay a Haskell program by keeping a log of the results of the first run of each I/O action. For this to work, the program must be denied direct access to I/O by interposing another abstract data type, in our case `CGI a`, that encapsulates the replay mechanism.

The replay mechanism maintains the log in the form of two lists, a *recorded list* and a *replay list*. The replay list is examined for the results of operations that have already been done. It is consumed during replay. The recorded list keeps all previous results and is extended with new results, once the replay list is depleted. To initiate a replay, both lists are initialized with the saved recorded list from a former run.

Concretely, here is how the operations `io` and `ask` of Section 4.2 work. The CGI action `io ioAction` first checks the replay list for a saved result. If the result is available, it is removed from the replay list and passed to the next CGI action (without executing `ioAction`). If the replay list is depleted, then `ioAction` is executed, its result is appended to the recorded list, and also passed to the next CGI action.

The CGI action `ask document` is slightly more complicated because its results are not immediately available. Instead, they must be extracted from the next request received from the browser, in another execution of the same program. Like `io`, `ask` first checks the replay list for a saved result. If the result is available, it is removed from the replay list and passed to the next CGI action, which is specified by `document` (details follow in Section 8.2). If the replay list is depleted, then `document` is converted to an HTML form and passed to the server as an HTTP response. At this point, the program terminates. However, the returned HTML page has a hidden input field which contains the current value of the recorded list, *i.e.*, all responses to `io` and `ask` actions leading up to that form. When the script receives the answer from the form, the request from the browser contains

1. the last value of the recorded list and

2. the input parameters for the last form.

From these two items, the script constructs a new log by appending the input parameters to the saved value of the recorded list. The resulting log is used to initialize both logs, the replay list as well as the recorded list. Then the script restarts from the first CGI action.

Figure 6 illustrates the mechanism by showing the state of the logs before executing the *current* action and before executing the *next* action that depends on the result `answer` of *current* [16]. The case where the replay list contains the answer works the same for `io` and for `ask`. In the second case, where the replay list is depleted, `ask` does *not* reach the *next* (`answer`) state directly. Instead, upon receipt of `answer` the whole program is replayed starting from the *first* action on the log `prms++[answer]`:

[16] The infix operators `:` and `++` are the cons and append operations on lists.

action	*current*	*next*(`answer`)	
replay list	`answer:rest`	`rest`	answer from log;
recorded list	`prms`	`prms`	*current* not executed
replay list	`[ ]`	`[ ]`	answer from executing
recorded list	`prms`	`prms++[answer]`	*current* action

Figure 6: The replay mechanism

action	*first*	...	*current*	*next*(`answer`)
replay list	`prms++[answer]`	...	`[answer]`	`[ ]`
recorded list	`prms++[answer]`	...	`prms++[answer]`	`prms++[answer]`

To get the replay mechanism initialized, the first invocation of the program sets both lists to the empty list.

8.1.1 Security

Storing the session state in the form sent back to the browser may have security implications if it reveals internal server information. In addition, the session state often contains sensible user input, like passwords. WASH/CGI addresses both concerns by providing a one-time pad encryption for the session state. This way, the required degree of security is entirely in the hands of the system administrator, who has to prime the encryption with data from an appropriate random source.

8.1.2 Limiting the Size of the Log

Another concern is that the size of the log grows without bounds during a long interaction. However, quite often interactions can be structured so that there is a main trunk of interaction, like returning to a main menu and starting subinteractions from the menu. There are two ways of dealing with this case.

First, the programmer may start a subinteraction in a new browser window by using the `target="_blank"` attribute for the submission button that activates the subinteraction. The subinteraction proceeds in the new window and terminates by presenting a page without buttons. Then, the user can close the window and is back at the main menu in the previous window. It is also possible to deal with both windows in an interleaved fashion. The point of this structuring is that the subinteractions never get incorporated in the main interaction trunk. Consequently, the main log does not grow unless some action updates the menu window.

Alternatively, the programmer may use the operator `once` to achieve a similar effect without having to worry about opening new windows. The type of `once` is

```
once :: (Read a, Show a) => CGI a -> CGI a
```

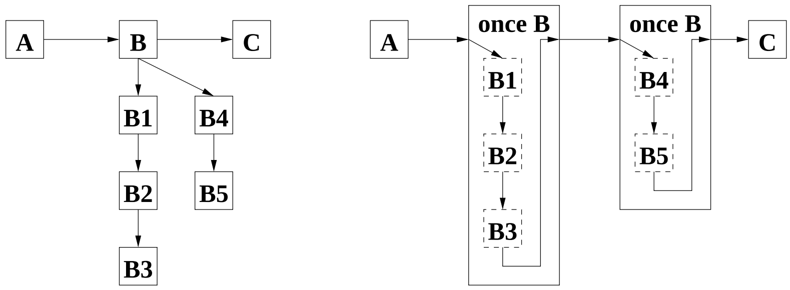

Figure 7: Spawning interactions vs. `once`

Its parameter is a CGI action that is once run to completion, involving any number of nested `io`, `ask`, and `once` operations. When the action returns its result, the log entries for the nested operations are replaced by a single log entry for the result. When `once` is replayed later on, it simply grabs the result from the log without passing control to the nested operations.

Figure 7 illustrates the difference between the two approaches. The left picture shows the interaction tree resulting from spawning subinteractions from a menu node **B**. Each node of the tree corresponds to an entry in the log. Hence, at point **C**, the log has length 3. The right picture shows a similar interaction using the `once` operator. The dashed boxes correspond to log entries that are removed by the `once` operator after that part of the interaction is complete. Hence, the log is temporarily extended (to length 4 at **B3** and **B5**) but then shrunk back again, so that the final log length at **C** is also 4.

8.2 Widgets

Earlier, in Section 5.3, we have seen that widgets are generated by functions of the following type:

```
type HTMLField context a =
  WithHTML INPUT CGI () -> WithHTML context CGI a
inputField :: (AdmitChildINPUT context, Reason a, Read a) =>
        HTMLField context (InputField a INVALID)
```

To understand how `inputField` works, we have to look a little bit inside the type `WithHTML INPUT CGI ()`. This abstract type threads the current state of the script throughout the whole construction of a Web form. The current state of the script contains all the information typically available in CGI scripts, as well as the replay and recorded lists. In addition, the `ask form` operation determines if there is already an answer to `form` in the log and passes this answer to the construction of `form`.

Hence, the widget operations work in two modes, depending on whether the input from the form is present. The first time a widget constructor (*e.g.*, `inputField`) is executed, the input is not yet present so that the constructor just generates its HTML representation. If the input for the form is present, then each widget constructor checks that the answer provides acceptable input for it and stores the input in its handle (after converting it from a string to a value of the expected type). This conversion may also fail, but this fact is only recorded in the handle data structure.

To tell individual instances of the widgets apart, they are automatically numbered and named. This naming is also managed using the state threaded through by `WithHTML context CGI ()`.

A submit button is slightly more complicated because it also specifies a callback action, as evident from the type of its constructor:

```
submit :: (AdmitChildINPUT context, InputHandle h) =>
        h INVALID -> (h VALID -> CGI ()) -> HTMLField context ()
```

Without input (before submission of the form), each use of `submit handles callback` just generates XHTML like any other widget. If input to the form is present, then each `submit` button checks if it has been clicked (by checking if an input for its name is present). This check will be successful for at most one button. The thus selected, lucky button then tries to validate all input `handles` passed to it. If they are all valid, then the button's constructor composes a CGI action by applying `callback` to the validated input `handles` and stores the resulting action in the CGI state. Later on, when the construction and examination of the page is complete, the `ask` operation checks if an action has been stored in the CGI state and will invoke it.

If some of the handles do not validate, the button's constructor includes JavaScript code in the XHTML page that puts error indicators on the erroneous input fields. Since it does not store an action in the CGI state, the enclosing `ask` will just display the XHTML page with the error indicators.

With this approach, a callback function *never* receives invalid data and it is guaranteed that only those input fields required for a particular callback are validated.

8.3 Persistence

The main stepping stone for handling state effectively in WASH/CGI is the design of the type-safe interfaces shown in Section 6. Once the interfaces are in place, the actual implementation is relatively straightforward. A handle to either kind of state is represented by a pair of the name of the entity and a version number.

Server-side state is a type-indexed mapping from version numbers to values. Getting the value from a handle always returns the value corresponding to the version number in the handle. Consequently, the result of a `get` operation need not be stored in the log. Setting the value through a handle fails if the current version number on the server does not match the version number of the handle.

Client-side state is similar, but the mapping is only defined for the current version number. Hence, the results of all operations must be stored in the log.

A naive implementation might assign version numbers sequentially by just incrementing the version number at each `set` or `add` operation. However, it is safer to use random numbers as version numbers. To see this, consider the a scenario where some house keeping operation removes the entire history of server-side state. Using sequence numbers that start from 0 again, there might be a pending transaction stored on some browser that still refers to an old sequence number *before* the house keeping took place. Continuing that transaction will lead to inconsistencies. Hence, it is important that each version number is used at most once. A good random number generator can guarantee that with high probability [13].

8.4 Haskell Specifics

Internally, an XHTML document is represented by two (abstract) data types `ELEMENT_` and `ATTR_`. A document tree is built using the interface

```
element_  :: String -> [ATTR_] -> [ELEMENT_] -> ELEMENT_
attr_     :: String -> String -> ATTR_
add_      :: ELEMENT_ -> ELEMENT_ -> ELEMENT_
add_attr_ :: ELEMENT_ -> ATTR_ -> ELEMENT_
```

The functions `element_` and `attr_` are the obvious constructor functions. The function `add_ parent child` adds a child element and the function `add_attr_ parent attr` adds an attribute to a parent element. However, this interface is not accessible in a WASH/CGI program, it is only used to build the next level of abstraction.

The datatype that encapsulates generation of XHTML, `WithHTML c m a`, is a monad provided that `m` is a monad. That is, `WithHTML c` is a state monad transformer [22]. The state manipulated by `WithHTML c m` is an HTML element and the variable `c` is a phantom type variable that does not occur in the constructors of the type:

```
newtype WithHTML c m a =
  WithHTML (ELEMENT_ -> m (a, ELEMENT_))
```

The element passed as an argument is the currently open element, *i.e.*, in terms of the generated document it corresponds to the closest enclosing opening HTML tag. Each construction function for an element, attribute, or text node adds the node into this context.

The constructor for building a `<title>` element serves as an example.

```
title ma =
  WithHTML (\enclosingElem ->
    do (a, titleElem) <- runWithHTML ma (element_ "title" [] [])
       return (a, add_ enclosingElem titleElem))
```

The parameter `enclosingElem` contains the enclosing element. `runWithHTML ma` installs its argument, an empty `<title>` element, as the new enclosing element and inserts subelements according to action `ma`. This action takes place in

```
io :: (Read a, Show a) => IO a -> CGI a
io ioAction =
  CGI (\ cgistate ->
    case inparm cgistate of
      PAR_RESULT str : rest ->
        case reads str of
          (result, ""):_ ->
            return (result, cgistate { inparm = rest })
          _ ->
            reportError "Result unreadable"
                  (text "Cannot read " ## text (show code)) cgistate
      [] ->
        do result <- ioAction
           return (result,
                   cgistate { outparm = PAR_RESULT (show result) :
                                        outparm cgistate })
      _ ->
        reportError "Out of sync" empty cgistate)
```

Figure 8: Implementation of `io`

the underlying monad, hence the result is a pair from the result `a` of running `ma` and the transformed `<title>` element. To complete the constructor function, the result `a` is paired with the enclosing element after adding the transformed `<title>` element to it and then returned. For a user of a `WithHTML c m` monad, the internal management of the element data is completely invisible.

The abstract datatype `CGI a` that encapsulates a CGI action is a monad. More precisely, it is a state monad lifted over the `IO` monad:

```
newtype CGI a =
  CGI (CGIState -> IO (a, CGIState))
```

The `CGIState` is a record type that contains the replay and recorded lists, information derived from the CGI environment, information specific to the page currently generated, the current encoder for the state token (see Section 8.1.1), and data structures for dealing with cookies (see Section 6.2).

Figure 8 shows the code for the `io` operation (see Section 4). It serves as an example to see the log machinery in action. `CGIState` is a record type with fields `inparm` (for the replay list) and `outparm` (for the recorded list). The recorded list is stored in reverse order to make appending new results at the end a constant-time operation. Each element of the log can be an answer to an I/O operation of the form `PAR_RESULT str`, where `str` is the value encoded as a string, or an association list of CGI parameters (an answer to a Web form which is decoded by `ask`). The `case` expression examines the replay list. If its first element is the result of an I/O operation, then its string representation is converted back to a data value using `read` and returned as the value of the computation. The thus consumed answer is removed from the log as it is passed along. If the replay list is empty, then the code executes the `ioAction`, returns its `result` and extends the *recorded list* with the textual encoding of the result.[17] If the replay list starts with another kind of answer, then somebody has been tampering with the state tokens in the Web form or there is a version conflict between the code that has created the Web form and the presently running code. Either case is captured and displayed as an error.

9 Assessment

Since the creation of WASH/CGI in 2001, a number of applications have been developed with it. The applications range from demo programs of up to 50 lines through mini-applications (generic time table software and a mini shop, both in less than 200 lines) to applications like conference submission software (900 lines) and a mail reader (550 lines relying on a 900 line email processing library and a 2000 line mail store management system). The library source is roughly 3700 lines of Haskell code.

We have successfully integrated the WASH/CGI library with a Web server written in Haskell by Simon Marlow [24]. In this server, a WASH/CGI application can run like a servlet in its own thread. The integration clearly indicates that the ideas of WASH/CGI are not limited to CGI applications, but that they are applicable to Web programming in general.

We have taken every effort to keep the design as simple and intuitive as possible. One deliberate guideline was *not* to create a new language but to adhere to the Haskell98 standard [15]. Although it is tempting to rely on one of the many implemented extensions to the Haskell type system [34], there were satisfactory solutions to the document validation problem (see Section 3.2), the treatment of submission buttons with decision trees (see Section 5.7), and a problem with the lifetime of input handles completely within the Haskell98 language.

The document sublanguage guarantees well-formedness of all generated documents, but only quasi validity as detailed in Section 3.2. This design is motivated by the following considerations:

- Quasi validity guarantees validity for about 95% of the HTML elements. Out of 89 elements, it is fully adequate for 68 (which just restrict children to zero or more occurrences of certain elements) and mostly adequate for an additional 16 elements (which restrict to zero or one occurrences or one or many occurrences). Only the remaining 5 elements restrict the order of the child elements (`HMTL`, `HEAD`, `TABLE`, `FIELDSET`, `MAP`).

- A checker for full validity (including attribute occurrences) can be implemented in the Haskell type system [35]. However, programming in the

[17] This code relies on the assumption that `read (show x) == x`. This is recommended practice in the Haskell report [15] and is indeed the case for the builtin types and the automatically derived instances of the `Read` and `Show` classes.

presence of this checker gets very awkward and limiting, as described in that work.

- The checker for full validity requires some (well-known) extensions to the Haskell type system.

How to support the Web browser's back and clone buttons is a religious issue among Web programmers. While some welcome its additional power and flexibility and regard it as a new paradigm for structuring user interfaces [12], others regard it as intrinsically adverse to the concept of an interactive transaction [3]. The implementation suggested in the present work admits the use of the back button as well as cloning and bookmarking (see Section 4). This decision leaves the choice of the underlying interaction model to the application programmer. If Web-browser-style control makes sense for an application, then WASH/CGI can support it. If the application requires a strictly sequential transaction model, then that can also be enforced using client-side state (Section 6.2). However, the WASH/CGI programming model is *not* tied to the implementation. An alternative implementation that rules out the use of the back and clone buttons would be more efficient and at the same time it would essentially remove all restrictions currently placed on IO.

Another issue that is automatically addressed by our implementation of session state (Section 4) is scalability. Since all state information is present in the state token delivered to the browser, it does not matter which server processes the next submission. Hence, WASH/CGI scripts can be easily run on a farm of Web servers. The only caveat here lies in the accesses to server-side state, which must be shared among these Web servers[18].

10 Related Work

Meijer's CGI library [25] implements a low-level facility for accessing the input to a CGI script and for creating its output. While it supports a tree-based representation of documents, it does not encapsulate its manipulation. In addition, it still preserves the CGI-style script-centric programming style, it does have a notion of session.

Hughes [18] has devised the powerful concept of arrows, a generalization of monads. His motivating application is the design of a CGI library that implements sessions. Indeed, the functionality of his library was the major source of inspiration for our work. Our work indicates that monads are sufficient to implement sessions (Hughes also realized that [17]). Furthermore, WASH/CGI extends the functionality offered by the arrow CGI-library with a novel representation of HTML and typed compositional forms. Also, the callback-style of programming advocated here is not encouraged by the arrow library.

Hanus's library [14] for server-side scripting in the functional-logic language Curry comes close to the functionality that we offer. In particular, its design

[18]The present implementation does not support this.

inspired our investigation of a callback-style programming model. While his library uses logical variables to identify input fields in HTML forms, we are able to make do with a purely functional approach. Our approach only relies on the concept of a monad, which is fundamental for a real-world functional programmer.

Further approaches to Web programming using functional languages are using the Scheme language [10, 30]. The main idea is to use a continuation to take a snapshot of the state of the script after sending the form to the browser. This continuation is then stored on the server and the form contains a key for later retrieval of the continuation. Conceptually, this is similar to the log that we are using. Technically, there are two important differences. First, we have to reconstruct the current state from the log when the next user request arrives. Using a continuation, this reconstruction is immediate. Second, our approach relies only on CGI whereas the continuation approach implies that the script runs inside the Web server and hence that the Web server is implemented in Scheme, too.

A recent revision of the continuation-based approach overcomes the tight coupling of scripts to the server. Graunke et al [9] present an approach to transform arbitrary interactive programs into CGI scripts. The transformation consists of three steps, a transformation to continuation-passing style, lambda lifting, and defunctionalization. The resulting program consists of a number of dispatch functions (arising from defunctionalization) and code pieces implementing interaction fragments. Compared to our approach, they reconstruct the state from a marshalled closure whereas we replay the interaction. Furthermore, they implement mutable state using cookies stored on the client side. It is not clear whether such functionality is needed for WASH/CGI scripts since the host language Haskell discourages the use of mutable state.

Bigwig [4] is a system for writing Web applications. It provides a number of domain specific customizable languages for composing dynamic documents, specifying interactions, accessing databases, etc. It compiles these languages into a combination of standard Web technologies, like HTML, CGI, applets, and JavaScript. Like our library, Bigwig provides a session abstraction. However, the Bigwig notion is more restrictive in that sessions may neither be backtracked nor forked. Each Bigwig session has a notion of a current state, which cannot be subverted. Moreover, the implementation of sessions is different and relies on a special runtime system that keeps one session thread alive for the entire duration of the session [3]. Bigwig includes a sophisticated static analysis that guarantees that all generated output is a valid HTML document. The same analysis also finds types for forms. Since WASH/CGI relies on Haskell's type system it only approximates validity. A special type system for forms is not required since (typed) field values are directly passed to (typed) callback-actions. Hence, all necessary type checking is done by the Haskell compiler.

Bigwig has been further developed into JWIG [6]. JWIG is an extension of Java with features of Bigwig, in particular the notion of sessions, the guarantee about valid XHTML output, and the guarantee that fields demanded by the program are present in the input submitted by the browser. JWIG adds a number of new statements and a special XML data type to the Java language. Beyond Bigwig, JWIG has an integrated notion of shared server-side state, but relies on the Java primitives for concurrency control. It does not have a special abstraction for server-side state that catches inconsistencies due to the (mis-) use of browser features as explained in Section 6.1. It is implemented by translation into Java. The translator performs extensive static program analysis to make the above guarantees. Since the analysis only covers the JWIG classes, it uses a number of conservative assumptions about library classes and the rest of the application. In contrast, since WASH/CGI relies solely on Haskell typing, no such conservative assumptions are required and all library and application code can be used in a WASH script without restriction.

MAWL [20, 1] is a domain specific language for specifying form-based interactions. It was the first language to offer a typing of forms against the code that received its input from the form. It provides a subset of Bigwig's functionality, for example, the facilities for document templates are much more limited.

Guide [21] is a rule-based language for CGI programming. It supports a simple notion of document templates, similar in style to that offered by MAWL. It provides only rudimentary control structure: it sequentially processes a list of predicates and takes the action (that is, it displays a HTML page) associated to the first succeeding predicate. It supports the concept of a session, a limited form of concurrency control, as well as session-wide and global variables. However, it neither supports typed input, nor composition of input fields, nor facilities to ensure that the variables used in a script match the variables defined in a form.

In previous work[33, 35], we have investigated the use of Haskell for generating valid HTML and XML documents, where the type system enforces adherence to the DTD. That work has been simplified and integrated into WASH/CGI as explained in Sec. 3.2.

The LAML system [26] is an approach to authoring XHTML documents based on the Scheme programming language. LAML provides construction functions for document nodes (called mirror functions), similar in style to our construction functions. It performs dynamic checks at runtime to guarantee that the generated output adheres to a DTD, but it cannot give static guarantees. LAML relies on Scheme's functional abstraction to provide parameterized documents. WASH relies on Haskell's functional abstraction but can give static guarantees due to its reliance on the type system. Finally, while there are various extensions for authoring documents that span multiple URLs, there is neither a session concept nor a concept for accessing persistent state built into LAML. Hence, it appears that LAML is primarily targeted for conveniently producing static Web pages.

Java Server Pages [27] and Servlets [32] are popular approaches to implementing Web services. They also provide a notion of session persistence and encapsulate its implementation. Unfortunately, they do not guarantee type safety and they do not provide the advanced support for generating HTML and forms as we do. More fundamentally, both approaches still support the page-centric or script-centric approaches to Web programming, whereas WASH/CGI supports a session-centric view of the application. Similar argumentation applies to other frameworks like ASP and ASP.NET [16], PHP [28], Python Server Pages [29], ColdFusion, and Zope.

11 Conclusion

WASH/CGI provides a high-level declarative approach to programming Web services. It is based on a session-centric programming model that hides away the details of the underlying protocols. This approach makes programming a Web service similar to programming an HTML-based GUI application.

WASH/CGI provides extensive guarantees ranging from well-formedness and quasi validity for all generated documents through type-safe input fields to type-safe server-side and client-side state. These guarantees are intrinsic in using WASH/CGI, no application code is required to achieve them. They are obtained by consistent use of abstract data types, type-safe interfaces, and compositionality.

Due to the host language Haskell, each aspect of WASH/CGI can be abstracted and parameterized. For example, a response page may be built from any number of fragment where each fragment comes with its own functionality. Again, the integration of these fragments does not require coding beyond assembling the fragments into one response page. The high potential for parameterization is unique to WASH/CGI.

While the efficiency of the implementation of server-side state and of the replay mechanism could certainly be improved by (*e.g.*) compiling WASH/CGI or by abolishing the use of the back button and cloning in the browser, we feel that its current state presents a good checkpoint. The fundamental ideas are proven to work, the implementation is stable, and many convenience features are integrated that alleviate the construction of Web services.

The WASH/CGI implementation is available for downloading at `http://www.informatik.uni-freiburg.de/~thiemann/WASH/`.

References

[1] D. Atkinson, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Conference on Domain-Specific Languages*, Santa Barbara, CA, Oct. 1997. USENIX.

[2] T. Berners-Lee, R. T. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. `http://www.faqs.org/ftp/rfc/rfc2396.txt`, Aug. 1998.

[3] C. Brabrand, A. Møller, A. Sandholm, and M. I. Schwartzbach. A runtime system for interactive web services. *Journal of Computer Networks*, 1999.

[4] C. Brabrand, A. Møller, and M. Schwartzbach. The `<bigwig>` Project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[5] CGI: Common gateway interface. `http://www.w3.org/CGI/`.

[6] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending java for high-level Web service construction. *toplas*, 2003. To appear.

[7] D. E. Eastlake, 3rd and T. Goldstein. ECML v1.1: Field specifications for E-commerce. `http://www.faqs.org/rfcs/rfc3106.html`, Apr. 2001.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol. `http://www.faqs.org/rfcs/rfc2616.html`, June 1999.

[9] P. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the Web. In *Proceedings of ASE-2001: The 16th IEEE International Conference on Automated Software Engineering*, San Diego, USA, Nov. 2001. IEEE CS Press.

[10] P. Graunke, S. Krishnamurthi, S. V. D. Hoeven, and M. Felleisen. Programming the Web with high-level programming languages. In D. Sands, editor, *Proceedings of the 2001 European Symposium on Programming*, Lecture Notes in Computer Science, pages 122–136, Genova, Italy, Apr. 2001. Springer-Verlag.

[11] P. T. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Modeling Web interactions. In *Proc. 12th European Symposium on Programming*, Lecture Notes in Computer Science, Warsaw, Poland, Apr. 2003. Springer-Verlag.

[12] P. T. Graunke and S. Krishnamurthi. Advanced control flows for flexible graphical user interfaces: or, growing GUIs on trees or, bookmarking GUIs. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 277–290, New York, May 19–25 2002. ACM Press.

[13] P. Gutmann. Software generation of practically strong random numbers. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, USA, Jan. 1998. USENIX.

[14] M. Hanus. High-level server side Web scripting in Curry. In *Practical Aspects of Declarative Languages, Proceedings of the Third International Workshop, PADL'01*, Lecture Notes in Computer Science, Las Vegas, NV, USA, 2001. Springer-Verlag.

[15] Haskell 98, a non-strict, purely functional language. `http://www.haskell.org/definition`, Dec. 1998.

[16] A. Homer and D. Sussman. *Professional ASP.NET 1.0.* Wrox Press Inc, 2 edition, 2002.

[17] J. Hughes. Private communication, Sept. 2000.

[18] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[19] D. M. Kristol. HTTP Cookies: Standard, privacy, and politics. *ACM Transactions on Internet Technology*, 1(2):151–198, Nov. 2001.

[20] D. A. Ladd and J. C. Ramming. Programming the Web: An application-oriented language for hypermedia service programming. pages 567–586. World Wide Web Consortium, Dec. 1995.

[21] M. R. Levy. Web programming in Guide. *Software—Practice & Experience*, 28(15):1581–1603, Dec. 1998.

[22] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, CA, Jan. 1995. ACM Press.

[23] Luhn formula. `http://www.webopedia.com/TERM/L/Luhn_formula.html`, Dec. 1999.

[24] S. Marlow. Developing a high-performance Web server in Concurrent Haskell. *Journal of Functional Programming*, 12(4&5):359–374, July 2002.

[25] E. Meijer. Server-side web scripting with Haskell. *Journal of Functional Programming*, 10(1):1–18, Jan. 2000.

[26] K. Nørmark. Programmatic WWW authoring using Scheme and LAML. In *Web Engineering Track of The Eleventh International World Wide Web Conferene - WWW2002*, pages 296–, Mar. 2002.

[27] E. Peligrí-Llopart and L. Cable. Java Server Pages Specification. `http://java.sun.com/products/jsp/index.html`, 1999.

[28] Php: Hypertext processor. `http://www.php.net/`, Feb. 2003.

[29] Python server pages. `http://www.ciobriefings.com/psp/`, June 1999.

[30] C. Queinnec. The influence of browsers on evaluators or, continuations to program Web servers. In Wadler [38], pages 23–33.

[31] D. Scott and R. Sharp. Abstracting application-level Web security. In *Proceedings of 11th ACM International World Wide Web Conference*, Hawaii, USA, 2002.

[32] Sun Microsystems. Java Servlet Specification, Version 2.4. `http://java.sun.com/products/servlet/`, Feb. 2003.

[33] P. Thiemann. Modeling HTML in Haskell. In *Practical Aspects of Declarative Languages, Proceedings of the Second International Workshop, PADL'00*, number 1753 in Lecture Notes in Computer Science, pages 263–277, Boston, Massachusetts, USA, Jan. 2000.

[34] P. Thiemann. Programmable type systems for domain specific languages. In M. Comini and M. Falaschi, editors, *Selected Papers of Workshop on Functional and Logic Programming, WFLP 2002*, volume 76 of *ENTCS*, Grado, Italy, 2002. Elsevier Science Publishers.

[35] P. Thiemann. A typed representation for HTML and XML in Haskell. *Journal of Functional Programming*, 12(4-5):435–468, July 2002.

[36] P. Thiemann. WASH/CGI reference manual. `http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH`, 2003. To appear.

[37] Markup validation service. `http://validator.w3.org/`, Dec. 2002.

[38] P. Wadler, editor. *International Conference on Functional Programming*, Montreal, Canada, Sept. 2000. ACM Press, New York.

[39] Web authoring system in Haskell (WASH). `http://www.informatik.uni-freiburg.de/~thiemann/haskell/WASH`, Mar. 2001.

[40] XHTML 1.0: The extensible hypertext markup language. `http://www.w3.org/TR/xhtml1`, Jan. 2000.