Seminar „Fortgeschrittene Techniken der funktionalen Programmierung“

Marion Müller

CGI-Programmierung mit Haskell:
Web Authoring System Haskell (WASH)

Betreuer: Prof. Michael Hanus

Literatur:

1. Peter Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms.
   In Practical Aspects of Declarative Languages (PADL'02), Volume 2257 of Lecture Notes in Computer Science, Portland, Oregon, USA, January 2002

2. `http://www.informatik.uni-freiburg.de/~thiemann/WASH`

1 Einleitung

WASH/CGI ist eine Haskell-Bibliothek für serverseitiges Web-Skripting, die von Peter Thiemann an der Universität Freiburg entwickelt wurde.
WASH/CGI ermöglicht die Programmierung von ganzen Interaktionen unter Verwendung eines Callback-Konzeptes, ähnlich wie aus der Programmierung von GUIs (graphischen Benutzerschnittstellen) bekannt, d.h. es muss nicht für jedes Formular ein eigenes CGI-Skript geschrieben werden, sondern nur ein Haskell-Programm. Weiterhin können Eingabeelemente getypt und dynamische Graphiken eingebunden werden.

WASH/CGI besteht aus den folgenden vier Teilsprachen:
Die Dokument-Sprache stellt Möglichkeiten für die Generierung von parametrisierten XHTML-Dokumenten und Formularen zur Verfügung. Durch die Typisierung wird sichergestellt, dass die generierten Dokumente wohlgeformt sind und es wird eine Quasi-Validität aller generierten Dokumente zur Compile-Zeit sichergestellt.
Die Session-Sprache ermöglicht es, interaktive Web-Seiten mit einem transparenten Session-Management zu erstellen.
Diese beiden Sprachen werden von der Widget-Sprache (Widget = Komponente einer Benutzeroberfläche) benutzt, welche die Kommunikation (Parameterübergabe) zwischen Client und Server beschreibt. Sie garantiert, dass vor jedem nächsten Schritt einer Interaktion alle Eingaben validiert werden, indem keine Strings verschickt werden, sondern eine interne getypte Repräsentation, d.h. alle Daten, die ein Client an einen Server schickt, kann der Server auch weiterverarbeiten.
Die Persistenz-Sprache ist zuständig für die Verwaltung von Zuständen sowohl auf der Server-Seite als auch auf der Client-Seite. Ein Zustand wird als abstrakter Datentyp repräsentiert.

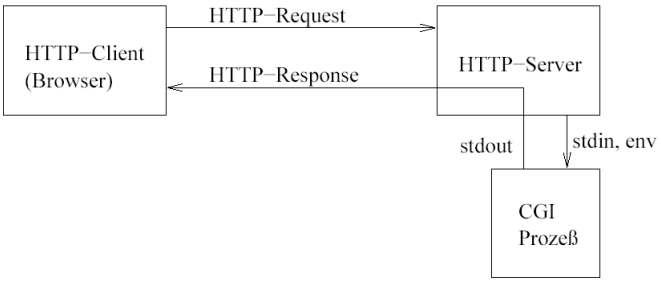
Die Implementierung basiert auf dem Common Gateway Interface (CGI), einem Standard für Datenaustausch zwischen Programmen auf Web-Servern (CGI-Skripte) und den sie über eine URL aufrufenden Web-Browsern. Die serverseitigen Programme können Daten vom Browser empfangen und generierte Daten an den Browser schicken. Dadurch ist das CGI eine Methode, um Web-Seiten dynamisch bzw. interaktiv zu machen.
Die CGI-Programmierung in der reinen Form ist sehr fehleranfällig und aufwendig. Das grösste Problem ist die Parameterübergabe zwischen HTML-Formularen und CGI-Skripten. Die Namen der Eingabeelemente im Formulare müssen mit den verwendeten Namen im CGI-Skript übereinstimmen, und die an das CGI-Skript gesendeten Daten müssen geprüft werden. Das zugrundeliegende Protokoll HTTP (Hypertext Transfer Protocol) ist zustandslos und unterstützt deshalb keine Sessions oder Zustandsverwaltung. Dies

wird jedoch benötigt, um eine interaktive Transaktion zuverlässig zu implementieren. Jeder Programmierer muss bei der Verwendung des reinen CGI selbst dafür sorgen.

Diese Probleme werden durch die WASH/CGI Bibliothek gelöst.

2 Reine CGI-Programmierung

Das Commmon Gateway Interface (CGI) ist eine Methode zum Erstellen von dynamischen Web-Seiten, die auf serverseitigen Berechnungen basieren. Ein CGI-Skript wird also verwendet, um auf einem Server Dokumente dynamisch zu erzeugen. Eine typische Anwendung ist die Verarbeitung von Eingaben in einem Formular und das Erzeugen eines Antwortdokumentes in Abhängigkeit von den Eingaben.
Vorteile des CGI sind, dass es von den meisten Web-Servern unterstützt wird und keine spezielle Funktionalität für den Browser erforderlich ist. Die Kommunikation erfolgt über Standard Input/Output Streams und Umgebungsvariablen. Ausserdem ist das CGI nicht eingeschränkt auf eine bestimmte Architektur oder Programmiersprache.



Das lokale CGI-Skript auf dem Server wird behandelt wie eine ausführbare Datei. Das Skript empfängt Eingaben vom Standard Input Stream und von Umgebungsvariablen. Die Antwort wird an Standard Output geliefert und vom Browser des Clients angezeigt.
Der Nachteil der reinen CGI-Programmierung ist jedoch ihre Fehleranfälligkeit. Wenn der Submit-Button gedrückt wird, sendet der Client eine Liste von Eingabeelement Name-Wert-Paaren an das CGI-Skript. Es wird nicht garantiert, dass das Skript die gelieferten Namen erwartet. Das Parameterübergabeverfahren ist String-basiert, also vollständig ungetypt. Das zugrundeliegende Protokoll HTTP (Hypertext Transfer Protocol) ist zustandslos, d.h. es werden keine Sessions unterstützt und Session-Persistenz, d.h. Verfügbarkeit des Wertes einer Variablen während einer ganzen Session, gibt es nicht.

Das reine CGI wird in WASH/CGI in der Bibliothek `RawCGI` implementiert, mit der beispielsweise das folgende CGI-Skript geschrieben werden kann.

```
main :: IO ()
main = start cgi

cgi :: CGIInfo -> CGIParameters -> IO ()
cgi info parms =
  case assocParm "test" parms of
    Nothing -> cgiPut "Parameter 'test' not provided"
    Just x  -> cgiPut ("Value of test = " ++ x)
```

Die Funktion

```
start : (CGIInfo -> CGIParameters -> IO a) -> IO a
```

implementiert die Parameterübergabe, wobei `CGIInfo` ein Rekord ist, der Metainformation wie die URL des Skriptes, das verwendete Protokoll und den Namen und die Adresse des anfragenden Hosts enthält. `CGIParameters` ist eine Liste von Name-Wert-Paaren als Strings. In der Funktion `cgi` wird die Eingabe verarbeitet, ein Ausgabedokument erstellt und eine IO-Aktion zurückgegeben, die eine CGI Antwort an Standard Output schreibt. Es wird geprüft, ob der Parameter `test` geliefert wurde und eine entsprechende Nachricht generiert. Der Wert von `assocParm "test" parms` ist `Nothing`, wenn es keine Bindung für den Namen `test` gibt, und `Just x`, wenn der Wert von `test` der String x ist. Die Funktion `cgiPut` kann für verschiedene Typen (z.B. Strings, HTML) unterschiedlich instantiiert werden und sorgt dafür, dass die Ausgabe des CGI-Skripts ein spezielles vom Ausgabetyp abhängiges Format hat, das der Web-Server parsen kann.

3 HTML-Generierung

Die HTML-Generierung ist in WASH/CGI unterteilt in eine interne HTML-Repräsentation als Baumstruktur (Modul `HTMLBase`)

```
type Element   -- abstract
type Attribute -- abstract

element_  :: String -> [Attribute] -> [Element] -> Element

doctype_  :: [String] -> [Element] -> Element
```

```
attr_     :: String -> String -> Attribute

add_      :: Element -> Element   -> Element
add_attr_ :: Element -> Attribute -> Element
```

und eine Schnittstelle zu direkten Benutzung, die durch eine Monade parametrisiert wird (Modul `HTMLMonad`).

```
data WithHTML m a = WithHTML (Element -> m (a, Element))
```

`WithHTML` ist eine spezielle Monade, deren Typkonstruktor wiederum parametrisiert über einem Monadenkonstruktor `m` ist und der so eine Kombination von Monaden ermöglicht. Diese Variante einer Monade nennt man Transformer-Monaden. Die Transformer-Monade `WithHTML` transformiert HTML-Elemente.
Die Verwendung einer Monade für die HTML-Generierung hat den Vorteil, dass Standard Monaden-Operationen und die `do`-Notation verwendet werden können.

```
(>>) :: Monad m => m a -> m b -> m b
(##) :: Monad m => m a -> m b -> m a
```

Beide Operatoren führen zwei Aktionen hintereinander aus, nur das Ergebnis ist unterschiedlich. Der `>>`-Operator gibt das Ergebnis der zweiten Aktion zurück, der `##`-Operator das Ergebnis der ersten Aktion. Hier werden mit diesen Operatoren Sequenzen von Dokument-Knoten konkateniert. Nur wenn der Rückgabewert der Berechnung weiterverwendet werden soll, muss man den entsprechenden Operator benutzen. Sonst sind beide Operatoren austauschbar, da beide Aktionen sequentialisieren.
Der Operator `empty` erzeugt eine leere Sequenz von Dokument-Knoten.
Die `do`-Notation

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

wird verwendet, um den Wert, der von der HTML-Aktion `htmlAction` berechnet wird, an eine Variable `x` zu binden, die in der weiteren Berechnung `...bbb...` und `finalAction` sichtbar ist. Das Ergebnis dieses `do`-Blocks ist das Ergebnis von `finalAction`.
Programme, die die Operatoren `>>` und `##` enthalten, lassen sich in die `do`-Notation umformen.

```
act1 >> act2 =
  do act1
     act2

act1 ## act2 =
  do x <- act1
     act2
     return x
```

Ein HTML-Dokument besteht aus Dokument-Knoten, die Element-, Text-, Attribut- oder Kommentar-Knoten sein können. Für jede Art von Knoten gibt es einen getypten Konstruktor. Konstruktoren für die Element-Knoten sind benannt wie die zugehörigen HTML-Tags, d.h. für jedes HTML-Tag `t` gibt es einen Konstruktor

```
t :: WithHTML CGI a -> WithHTML CGI a
```

der als Argument eine Sequenz von Kind-Elementen, Text-Knoten und Attributen nimmt und ein Element mit dem Tag `t` erzeugt.
Ein Attribut wird mit der Funktion

```
attr :: Monad m => String -> String -> WithHTML m ()
```

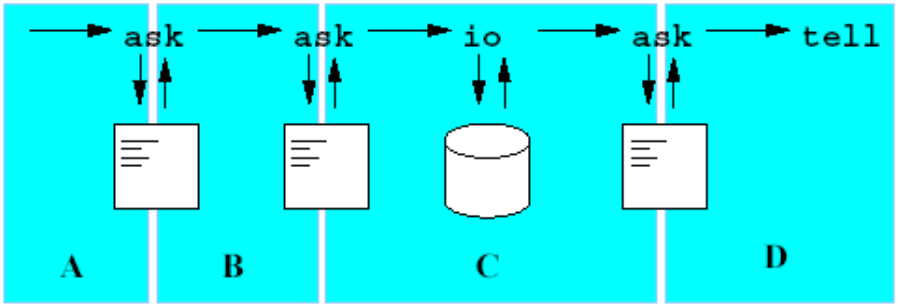
erzeugt, deren Argumente der Attributname und der Wert des Attributes als Strings sind. Die Funktionen

```
text    :: Monad m => String -> WithHTML m ()
comment :: Monad m => String -> WithHTML m ()
```

erzeugen eine Text- bzw. Kommentar-Knoten aus einem gegebenen String.
Nun kann ein parametrisiertes Dokument als Haskell-Funktion `standardPage` mit einen Titel und einer Sequenz von Dokument-Knoten als Parameter geschrieben werden.

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

4 Sessions



CGI-Session (aus [1])

Eine Session ist eine Folge von logisch zusammenhängenden Interaktionen zwischen einem Server und einem Client (Web-Browser). Dafür wird ein gemeinsamer Zustand von Client und Server benötigt.
Unter Verwendung von HTTP gibt es verschiedene Möglichkeiten, Sessions zu realisieren.
Der Server kann Informationen auf dem Client als Textdatei speichern (Cookies).
Der Zustand kann auch in der URL übergeben werden („URL Rewriting“), was problematisch ist, wenn der Benutzer die URL mit der Session als Bookmark speichert oder weitergibt, da die URL später eventuell nicht mehr gültig ist. Auch Missbrauch der URL wird so ermöglicht.
In WASH/CGI wird Session-Management auf der HTML-Ebene verwendet, da dies am einfachsten zu realisieren ist. Der Server schreibt die Zustandsdaten in versteckte Eingabe-Felder und fügt so jedem Formular für den Benutzer unsichtbare Informationen hinzu, die auch vom Server ausgewertet werden, wenn die Formulardaten vom Client zurückgesendet werden.
Der Zustand entspricht allen Antworten, die vom Benutzer oder von I/O-Operationen bis zu einem Zeitpunkt empfangen wurden. Daraus lassen sich alle Werte rekonstruieren, indem sie später mit dem gleichem Ergebnis neu berechnet werden können.
Für das Session-Management wird die `CGI`-Monade verwendet.

```
data CGI a = CGI { unCGI :: CGIState -> IO (Maybe a, CGIState) }
```

Der `CGIState` enthält den bisherigen Verlauf der Session.
Ein Wert des Typs `CGI a` ist eine CGI-Aktion, die ein Ergebnis des Typs `a` berechnet. CGI-Aktionen sind der grundlegende Baustein für Sessions, sie beinhalten die Session-Sprache. Mehrere CGI-Aktionen können mit dem Bind-Operator (`>>=`) oder der `do`-Notation hintereinander ausgeführt werden. Ein Wert des Typs `WithHTML CGI a` ist eine HTML-Aktion, die eine Sequenz

von Dokumentknoten erzeugt, welche dem Inhalt eines HTML-Dokuments entspricht.
Dokumente erzeugen Sessions durch die Operationen `tell` und `ask`.

```
tell :: (CGIOutput a) => a -> CGI ()
tell a = CGI (\cgistate -> cgiPut a >> exitWith ExitSuccess)

ask :: WithHTML CGI a -> CGI ()

run :: CGI () -> IO ()

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

Die Funktion `tell` nimmt einen beliebigen Wert, dessen Typ eine Instanz von `CGIOutput` ist, schickt ihn an den Browser und terminiert das Programm, also beendet `tell` eine Session. Beispiele sind HTML-`Element`e oder ein einfacher `String`, der als `text/plain`-Dokument dargestellt wird. Die Funktion `ask` nimmt einen monadischen Wert, der eine HTML-Seite erzeugt und schickt diese Seite an den Browser. Diese Seite kann Formulare und Eingabe-Widgets enthalten. Vor der Terminierung des Programms wird noch der Zustand gespeichert, um es später an dieser Stelle weiterlaufen zu lassen, d.h. `ask` macht weitere Interaktionsschritte möglich im Unterschied zu `tell`. Die Funktion `run` wandelt eine CGI-Aktion in eine IO-Aktion, da die main-Funktion in Haskell immer den Typ `IO ()` haben muss und ein typisches Skript eine CGI-Aktion ist. Die Funktion `io` modelliert die andere Richtung, d.h. die Umwandlung einer IO-Aktion in eine CGI-Aktion.

5 Formulare

Durch die Verwendung von Formularen werden interaktive Anwendungen erzeugt. Ein Formular kann Eingabe-Elemente wie Eingabefelder, Radio-Buttons und Submit-Buttons enthalten und ist als Typsynonym `HTMLField` definiert.

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

Am Beispiel für eine einfache Login-Seite werde ich diesen Teil von WASH/CGI erläutern.
Programmausschnitt:

```
login :: CGI ()
login = ask $ standardPage "LOGIN" $ makeForm $ table $
```

```
  do nameF <- tr (td (text "Enter your name ") >>
                  td (textInputField (attr "size" "10")))
     passF <- tr (td (text "Enter your password ") >>
                  td (textInputField (attr "size" "10")))
     submitField (check nameF passF) (attr "value" "LOGIN")

check :: InputField String -> InputField String -> CGI ()
check nameF passF =
  tell $ standardPage "LOGIN" $
    (text "You said " ## fromJust (value nameF) ##
     text " and " ## fromJust (value passF))
```

und das daraus generierte HTML-Formular:

```
<html>
  <head><title>LOGIN</title></head>
  <body>
    <h1>LOGIN</h1>
    <form enctype="application/x-www-form-urlencoded"
     name="f3"
          method="POST"
          action="http://localhost:80">
      <table>
        <tr>
          <td>Enter your name </td>
          <td><input size="10" name="f0" type="text"></td>
        </tr>
        <tr>
          <td>Enter your password </td>
          <td><input size="10" name="f1" type="text"></td>
        </tr>
        <input value="LOGIN" name="s2" type="submit">
      </table>
      <input value="%5B%5D" name="=CGI=parm=" type="hidden">
    </form>
  </body>
</html>
```

Der Konstruktor

```
makeForm :: HTMLField ()
```

erzeugt ein Formular mit Standardwerten für die Attribute `action`, `enctype` und `method`, und einem versteckten Eingabefeld für den Zustand der Session, das den Namen `=CGI=parm=` hat und als Wert eine URL-kodierte Liste der Parameter. In diesem Beispiel ist der Wert `%5B%5D`, also die leere Liste `[]`. Ein verstecktes Eingabefeld wird nicht vom Browser angezeigt, aber der Wert wird genau wie die Werte aller anderen Eingabefelder verschickt, wenn der Submit-Button gedrückt wird.
Als Abkürzung von `ask`, `standardPage` und `makeForm` gibt es die Funktion

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

Die Funktion

```
textInputField :: HTMLField (InputField String)
```

generiert ein normales Eingabefeld für Text, deren Länge durch ein Attribut festgelegt wird. Getypte Eingabefelder werden mit der Funktion

```
inputField :: Read a => HTMLField (InputField a)
```

generiert, wobei der Typ `a` eine Instanz von `Read` sein muss für die Unwandlung des Eingabe-Strings in den Typ `a`. Der Ergebniswert des Typs `InputField a` ist der Handler für das Eingabefeld, der an den Wert des Feldes gebunden ist. Dadurch ist es nicht notwendig, den Eingabefeldern Namen zu geben. Stattdessen kann auf den Wert durch zwei Funktionen zugegriffen werden.

```
value :: InputField a -> Maybe a
string :: InputField a -> Maybe String
```

Die Funktion `value` gibt `Just a` zurück, falls die Eingabe geparst werden kann, sonst `Nothing`. In diesem Fall kann man die Funktion `string` verwenden, die die Eingabe als `Just String` zurückgibt, falls es eine Eingabe gibt, sonst `Nothing`, um den Fehler zu analysieren.
Durch die Notation `nameF <- ...` im Beispiel wird der Handler für das erste Eingabefeld an die Variable `nameF` gebunden, denn die rechte Seite von `<-` wird folgendermaßen ausgewertet:
Wendet man einen Element-Konstruktor (hier `tr` und `td`) auf eine Sequenz von Knoten an, so ist das Ergebnis wieder die Sequenz. Da außerdem nur der `>>`-Operator verwendet wird, ist der Wert der rechten Seite also das Ergebnis der Funktion inputField.
Das zweite Eingabefeld wird analog an die Variable `passF` gebunden.
Diese Variablen können nun an die Callback-Funktion `check` weitergegeben werden, in der die Eingaben vom Benutzer weiterverarbeitet werden.
Als letzte verwendete WASH/CGI-Funktion im Beispiel fehlt noch `submitField`.

```
submitField :: CGI () -> HTMLField ()
```

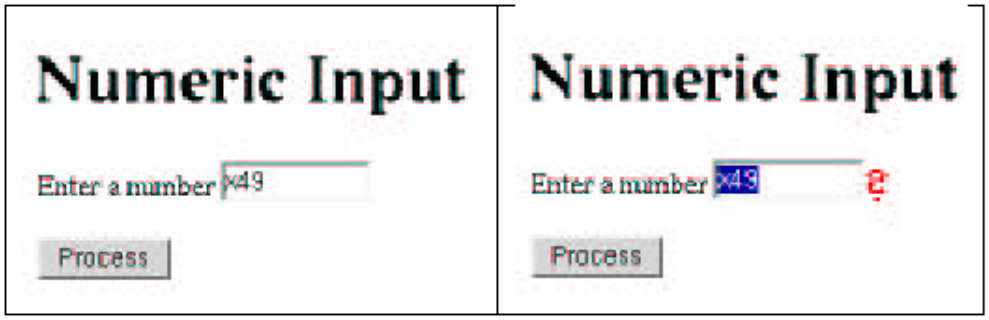
Diese Funktion nimmt eine CGI-Aktion und generiert einen Submit-Button in der HTML-Seite. Die CGI-Aktion wird beim Anklicken des Submit-Buttons ausgeführt und ist ähnlich wie eine Weiterleitung. Ein Button kann auf einer Seite nur unter den Eingabe-Elementen stehen, deren Eingaben für die CGI-Aktion benötigt werden. Das ist eine kleine Einschränkung der Gestaltungsmöglichkeiten. Ein Formular kann mehrere Submit-Buttons enthalten, die an unterschiedliche Aktionen gebunden sind, und somit auch aus mehreren Blöcken bestehen, die jeweils Eingabe-Elemente und Submit-Buttons enthalten.
Im generierten HTML-Formular haben alle Widgets Namen (`f0`, `f1`, `f3` und `s2`), die von WASH/CGI während der Konstruktion des Formulares festgesetzt werden. Dabei werden Eingabefelder (`fi`) und Submit-Felder (`si`) unterschieden. Jedes Feld bekommt einen eindeutigen Namen zugewiesen.
Weitere Eingabefelder werden durch die folgenden Funktionen generiert:

```
passwordInputField :: HTMLField (InputField String)
checkboxInputField :: HTMLField (InputField Bool)
fileInputField     :: HTMLField (InputField String)
resetField         :: HTMLField (InputField ())
```

Mit der Funktion `resetField` werden alle Eingaben gelöscht.
In der Login-Seite gibt es ein Eingabefeld für ein Passwort, das oben durch ein einfaches Eingabefeld realisiert ist. Eine Verbesserung des Beispiels wäre, dieses Feld durch ein `passwordInputField` zu ersetzen.

```
login :: CGI ()
login =
  standardQuery "LOGIN" $ table $
  do nameF <- tr (td (text "Enter your name ") >>
                  td (textInputField (attr "size" "10")))
     passF <- tr (td (text "Enter your password ") >>
                  td (passwordInputField (attr "size" "10")))
     submitField (check nameF passF) (attr "value" "LOGIN")
```

Die Funktion `passwordInputField` generiert ein Eingabefeld mit dem Attribut `type="password"`, damit die Eingabe nicht angezeigt wird.
Wird ein bestimmte Eingabe erwartet, ist es sinnvoll, ein getyptes Eingabefeld zu verwenden. Ein Beispiel dafür ist ein Eingabefeld, das einen Integer-Wert erwartet.




```
main :: IO ()
main =
  run page1

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

page2 :: InputField a -> CGI ()
page2 inf =
  let n :: Int
      n = value inf
  in
```

```
  standardQuery "Your Number" $
  p (text "Your number was " ## text (show n) ## text "!")
```

In diesem Beispiel wird für ein Eingabefeld die Funktion `inputField` benutzt. Klickt man den Submit-Button an, so wird die Callback-Funktion `page2` mit dem Eingabe-Handler aufgerufen. In `page2` mit einem `let` einer Variable `n` vom Typ `Int` der Wert aus dem Handler `inf` zugewiesen.
Der linke Teil der Abbildung zeigt die erste Seite des Beispiels mit einer falschen Eingabe „x49“. Der rechte Teil zeigt die Darstellung nach dem Anklicken des „Process“-Buttons. Alle Eingabefelder, die Typfehler enthalten, werden automatisch mit einem Fragezeichen gekennzeichnet und die Eingabe wird markiert.
Wird jetzt die Eingabe korrigiert und wieder der „Process“-Button angeklickt, dann wird der in `page2` konstruierte Text angezeigt, wobei die Funktion `show` den eingegebenen Wert in einen String konvertiert.

6 Persistenz

Ausser dem Session-Zustand gibt es in WASH/CGI auch persistente Zustände, d.h. Zustände, die unabhängig von jeder Session sind. Es gibt sowohl auf der Server-Seite als auch auf der Client-Seite persistente Zustände, die sich hauptsächlich durch ihre Reichweite unterscheiden.
Der Zustand auf der Client-Seite ist nur sichtbar für Skripte, die auf dem Client laufen und wird in Cookies gespeichert. Der Zustand auf der Server-Seite ist global sichtbar und wird typischerweise in einer Datenbank gespeichert.
Auch hier wird wieder Typsicherheit durch das Typsystem von Haskell garantiert, im Gegensatz zu üblicherweise verwendeten String-basierten Interfaces.
Im Folgenden werde ich nur auf den serverseitigen Zustand eingehen.
Eine Anwendungsmöglichkeit ist eine High-Score-Liste für ein Spiel, implementiert durch den folgenden Programmausschnitt:

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

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

```
        text (show score))
```

Ein Zustand wird im Modul `Persistent` als parametrisierter Abstrakter Datentyp `T a` definiert und ist ein Handler (mit einem Timestamp) für einen persistenten Wert des Typs `a`, für den die folgenden Zugriffsfunktionen definiert sind:

```
init :: (Types a, Read a, Show a) => String -> a -> CGI (Maybe (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 (T a)
```

Aus den Typbeschränkungen `Read a` und `Show a` folgt, dass die Daten in einer Textdatei gespeichert werden und `Types a` bedeutet, dass es eine Term-Darstellung für den Typ `a` geben muss.
Die Operation `init` nimmt den Namen des persistenten Wertes und einen initialen Wert des Typs `a` und erzeugt im Beispiel eine neue persistente Entität mit dem Namen `"hiscores"` und dem initialen Wert `[]`. Existiert bereits eine Entität mit diesem Namen und Typ, so wird bei passendem Typ der gespeicherte Wert zurückgegeben, sonst `Nothing`. Andernfalls wird ein neuer Handler erzeugt und zurückgegeben.
Die Operation `add` fügt einen Wert zum persistenten Zustand hinzu, falls dieser eine Liste ist. Hier wird ein Punktestand (`myScore`) zum Handler `initialHandle` hinzugefügt und als `currentHandle` gespeichert.
Mit der Operation `get` kann man nun den Wert eines Handlers holen, der den gerade mit `add` hinzugefügten Punktestand enthält.
Nun werden die Namen mit Punkten als Auflistung in einer HTML-Seite zurückgegeben, indem die Funktion `showItem` auf jedes Element der High-Score-Liste mittels `mapM_` angewendet wird.
Die Operation `set` nimmt einen Handler und einen Wert und ermöglicht es, den Wert eines Handlers zu verändern, falls dieser aktuell ist. Die Operation `current` holt die aktuelle Version eines gegebenen Handlers.
Da persistente Werte nicht an eine Session gebunden sind, werden sie aus der Textdatei gelöscht, wenn sie lange nicht mehr abgerufen wurden, damit die Datei nicht zu groß wird. Versucht nun ein Skript, auf einen gelöschten Wert zuzugreifen, so gibt es eine Fehlermeldung. Um zu verhindern, dass mehrere Skripte gleichzeitig auf einen persistenten Wert zugreifen, wird die Textdatei gelockt, wenn sie benutzt wird.

7 Beispiel

Ein größeres Beispiel ist ein Spiel, bei dem es darum geht, eine Zahl zu raten.[1]
Startet man das CGI-Skript[2], so hat man die Möglichkeit, die persistent gespeicherte High-Score-Liste anzusehen, oder ein Spiel zu beginnen. Eine Zufallszahl wird generiert und im Spiel wird der Benutzer aufgefordert, diese zu erraten. Der Benutzer hat so viele Versuche, bis er die Zahl richtig geraten hat. Nach jedem Versuch wird angegeben, ob die geratene Zahl zu groß oder zu klein war. Wurde die Zahl gefunden, so kann man sich in die High-Score-Liste eintragen.

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

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

Die CGI-Aktion `highScoreStore` initialisiert eine persistente Liste von Spielergebnissen mit der leeren Liste. Wie in Abschnitt 6 beschrieben, wird eine neue persistente Liste angelegt oder die schon vorhandene zurückgegeben. Ein Ergebnis ist ein Paar aus einer Anzahl von Versuchen und einem Namen eines Spielers.

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

mainCGI :: CGI ()
mainCGI =
  io (randomRIO (1,100)) >>= \ aNumber ->
  standardQuery "Guess a number" $
    do submit F0
               (play 0 (aNumber :: Int)
                     "I've thought of a number between 1 and 100.")
               (fieldVALUE "Play the game")
       submit F0 admin (fieldVALUE "Check scores")
```

Mit den ersten beiden Zeilen beginnt jedes WASH/CGI-Programm. Die Funktion `mainCGI` implementiert den Beginn der Interaktion, d.h. sie berechnet

[1] `http://www.informatik.uni-freiburg.de/ thiemann/haskell/WASH/GuessNumber.hs|`
[2] `http://nakalele.informatik.uni-freiburg.de/cgi/WASH/GuessNumber.cgi|`

eine Zufallszahl zwischen 1 und 100, die an die Variable `aNumber` gebunden wird und konstruiert die Startseite, auf der der Benutzer einen der beiden Buttons anklicken kann, die durch die Funktion `submit` konstruiert werden, und an die die Callback-Funktionen `play` zum Starten des Spiels und `admin` zum Anzeigen der High-Score-Liste angehängt sind.

```
admin F0 =
  do highScoreList <- highScoreStore
     highScores <- P.get highScoreList
     standardQuery "GuessNumber - High Scores" $ table_T $
       (tr_S (th_S (text_S "Name") ## th_S (text_S "# Guesses")) ##
        foldr g empty (sort highScores) ##
attr_SS "border" "border")
  where
    g (Score name guesses) elems =
      tr_T (td_S (text name) ## td_S (text (show guesses))) ## elems
```

Die Funktion `admin` initialisiert den Handler `highScoreList` mit dem persistenten High-Score und holt dann daraus den Wert, der an die Variable `highScores` gebunden wird. Diese Liste wird dann sortiert und als Tabelle angezeigt.

```
play nGuesses aNumber aMessage F0 =
  standardQuery "Guess a number" $
    do text aMessage
       text_T " Make a guess "
       activeInputField (processGuess (nGuesses + 1) aNumber) empty

processGuess nGuesses aNumber aGuess =
  if aNumber == aGuess then
    youGotIt nGuesses aNumber
  else if aGuess < aNumber then
    play nGuesses aNumber
         ("Your guess " ++ show aGuess ++ " was too small.") F0
  else
    play nGuesses aNumber
         ("Your guess " ++ show aGuess ++ " was too large.") F0

youGotIt nGuesses aNumber =
  standardQuery "You got it!" $
  do text_S "CONGRATULATIONS!"
     br_S empty
```

```
      text_S "It took you "
      text (show nGuesses)
      text_S " tries to find out."
      br_S empty
      text_S "Enter your name for the hall of fame "
      nameF <- textInputField empty
      br_S empty
      defaultSubmit nameF (addToHighScore nGuesses) (fieldVALUE "ENTER")

addToHighScore nGuesses nameF =
  let name = value nameF in
  if name == "" then admin F0 else
  do highScoreList <- highScoreStore
     P.add highScoreList (Score name nGuesses)
     admin F0
```

Der Spielstart wird durch die Funktion `play` implementiert, die den Benutzer auffordert, eine Eingabe zu machen, und dann für die Auswertung der geratenen Zahl die Funktion `processGuess` aufruft. Wurde die Zahl noch nicht erraten, so wird wieder `play` aufgerufen mit einem entsprechenden Hinweis. Sonst bekommt der Benutzer durch den Aufruf der Funktion `youGotIt` die Möglichkeit, seinen Namen für die High-Score-Liste anzugeben. Klickt er nun noch den Button, so wird sein Name zusammen mit der Anzahl der benötigten Versuche zu der Liste hinzugefügt und diese ausgegeben.

8 Zusammenfassung

WASH/CGI ermöglicht die einfache Generierung komplizierter interaktiver Web-Seiten, wobei die Programmierung ähnlich ist wie die Programmierung graphischer Benutzerschnittstellen.
Die Generierung und Interpretation von HTML-Formularen ist konsistent.
Das reine CGI wird durch Session- und Zustandsverwaltung erweitert, und für die Konstruktion von HTML-Dokumenten werden Funktionen bereitgestellt.
Haskell stellt grundlegende Typen zur Verfügung und garantiert die Typsicherheit. Kenntnisse von Haskell und Monaden sind notwendig für die Benutzung von WASH/CGI, aber durch umfangreiche Beispiele, die auf der Web-Seite zu WASH/CGI bereitgestellt werden, wird der Einstieg erleichtert.
Die Ideen von WASH/CGI sind nicht an die CGI-Programmierung gebunden, so dass sie auch für andere serverseitigen Skriptsprachen anwendbar sind.