Seminar Funktionale Programmiertechniken

SS 2007

Thema:

WASH - Web Authoring System Haskell

von Heiko Hoffmann

Betreuer: Michael Hanus

1. Einleitung

WASH, geschrieben von Peter Thiemann, umfasst eine Gruppe von in Haskell eingebetteten domänenspezifischen Programmiersprachen („embedded domain specific languages“ - EDSL oder DSEL) zur Erstellung von dynamischen Webseiten. Das Programmierinterface ist ähnlich wie bei der GUI-Programmierung. Dabei gibt es zwei verschiedene WASH-Implementierungen für eine Websession: Mittels Common Gateway Interface (CGI) oder mittels Servlets/Server Pages auf einer erweiterten Version von Simon Marlow's Webserver (hws). Durch den hohen Abstraktionsgrad sind nur wenige Codeänderungen vonnöten, um von einer CGI-Implementierung auf Wash Server Pages oder Servlets umzusteigen. Die gemessenen Laufzeiten sind auf dem HWS Server mit Servletimplementierung auch deutlich besser als die CGI Variante auf einem Apache Server. Im folgenden Dokument werden wir uns aber nur auf den WASH/CGI-Ansatz konzentrieren, weil er auch einfacher zu installieren ist. Zusätzlich gehören zu WASH u.a. noch verschiedene Module für Mailservices(WASH/Mail) und Datenbankanbindungen(WASH/DB, DBconnect). Die Gültigkeit des erzeugten XHTML-Codes und Zustandsspeicherung (sessions) sind durch WASH gegeben, was aber mittlerweile ja vielleicht nichts besonderes mehr ist. WASH hat aber auch noch einige Vorteile, die andere Sprachen nicht haben: Durch Verwendung von Haskell ist Typsicherheit garantiert, auch bei Eingabefeldern, außerdem wird speziell darauf geachtet, dass der Programmstatus immer konsistent bleibt.

2. WASH/CGI

2.1 Bestandteile

WASH/CGI kann in mehrere abstrakte Teilsprachen eingeteilt werden, die für unterschiedliche Aufgaben zuständig sind. Erstens eine Dokument-Sprache zur Erstellung von gültigen XHTML-Dokumenten (document sublanguage), zweitens eine Session-Sprache für Zustände, die die Dokumente zu interaktiven Skripten verbindet (session sublanguage). Drittens eine Kommunikations-Sprache für die Client-Server Kommunikation mit eingebauter Typsicherheit (widget sublanguage). Und zuletzt noch eine Persistenz-Sprache für die Steuerung von mehreren Threads mit gemeinsamen Zuständen (persistence sublanguage). Auf die zu Grunde liegende CGI-Implementierung wird hier nicht weiter eingegangen, die Funktionalität ist aber ähnlich wie bei *Meijer’s CGI library* aus der Haskell-Standardbibliothek.

Zuerst wollen wir uns nun ein einfaches Beispiel ansehen, um einen ersten Eindruck für die Sprache zu bekommen:

Hello

This is my first CGI program!

```
import CGI                                -- indicate it's using CGI
main =                                    -- main program (fixed)
  run $                                   -- starts a CGI script
  ask $                                   -- delivers a Web page
  standardPage "Hello" $                  -- constructs a Web page
  text "This is my first CGI program!" -- contents of page
```

kurze Erklärung:

- $ ist Funktionsanwendung;
- schreibe "f $ a" für "f (a)" oder "f a"
- *main* ist eine I/O - Aktion vom Typ "IO ()"
- *run* ist eine Funktion, die eine CGI-Aktion auf eine I/O-Aktion abbildet
- *ask* bildet ein Dokument auf eine CGI-Aktion ab
- *standardPage* baut aus einer Überschrift und zusätzlichem Inhalt eine Standardseite
- *text* konstruiert eine einzelne Sequenz mit einem Text-Knoten

Nun aber ein tieferer Einstieg in die einzelnen Teilsprachen von WASH/CGI:

2.2 Dokument-Sprache

Ein XHTML-Dokument besteht aus einer Menge von Element-, Attribut- und Text-Knoten, die als Baumstruktur aufgebaut sind. WASH/CGI stellt für jede Art von Elementknoten eigene Konstruktoren zur Verfügung, außerdem die Funktionen *attr* und *text* für Attribut- bzw. Text-Knoten. Der Obertyp für solch einen Knoten ist: *WithHTML CGI a ,* wobei die einzelnen CGI-Elemente noch eigene Untertypen haben. Ein Dokument wird mittels einer Sequenz solcher Dokumentknoten erzeugt, wobei ein Konstruktor als Parameter wiederum eine Sequenz von Knoten hat. e*mpty* erzeugt eine leere Sequenz. Sequenzen werden verknüpft mit der für Monaden üblichen do-Notation, oder den Operatoren '>>' und '##'. Der Unterschied zwischen den beiden Operatoren ist ihr Rückgaberesultat:

```
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
```

Anwendungsbeispiel:

```
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")))
```

Der obige WASH-Code liefert den folgenden XHTML-Code:

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

Desweiteren gibt es noch einen Standard-Wrapper, der das Grundgerüst für eine XHTML-Seite liefert und folgendermaßen implementiert ist:

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

Weiter wollen wir betrachten, inwieweit sicher gestellt ist, dass mit WASH gültiger XHTML-Code produziert wird. Dadurch, dass das Dokument mittels einer baumartigen Datenstruktur erstellt wird, die die eigentliche String-Repräsentation des erzeugten XHTML-Codes verbirgt, ist Wohlgeformtheit (richtige Reihenfolge von öffnenden und schließenden Tags) des Codes garantiert. Desweiteren müssen aber auch noch die Beschränkungen der XHTML-DTD erfüllt werden. WASH beschränkt sich hier auf eine Quasi-Gültigkeit, indem alle Restriktionen der DTD, die besagen, welche Kind-Knoten (Attribute, Textknoten, andere Elemente) bei welchen Elementen benutzt werden dürfen. Die Kind-Knoten unterliegen laut DTD aber auch noch bestimmten Regeln ihre Reihenfolge betreffend, dies wird in WASH aus praktischen Gründen jedoch nicht implementiert. Die Typen für die Knotenkonstruktoren sehen deshalb folgendermaßen aus:

```
text :: (Monad m, AdmitChildCDATA e)=> String -> WithHTML e m ()

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

Der Typparameter *e* steht hierbei für den Kontext, in dem der Kindknoten aufgerufen wird, also für den Namen des Eltern-Knotens. Der Knoten *ul* öffnet auch einen neuen Kontext, in dem gültige Kindelemente von UL benutzt werden können: *WithHTML UL m a*

WASH versieht jede Art von Dokument-Knoten mit einem Typ und einer Typ-Klasse. So steht der Typ *CDATA* für einen Textknoten, und *AdmitChildCDATA* für die Typklasse, die Textknoten als Kinder enthalten kann.
Kommen wir nochmal auf die Funktion *standardPage* zum Aufbau einer Standardseite zurück, sie hat nun folgenden Typ:

```
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))
```

Die im Parameter *nodes* vorkommenden Knoten müssen also im Kontext *BODY* stehen können, und das Dokument muss in einem Kontext vorkommen, der HTML-Elemente erlaubt (*AdmitChildHTML*).

2.3 Session-Sprache:

Eine Web-Session ist aus Serversicht eine Sequenz von HTTP-Anfragen und den zugehörigen Antworten: *POST* url0; *FORM* f1; *POST* url1; *FORM* f2; *POST* url2; ...
(Dabei steht *Post* für die Anfrage, *Form* für die Antwort.)
Jede url*i* adressiert eine spezielle Ressource auf dem Server. Der Einfachheit halber sei der zusätzlich übertragene Datenblock als teil der url*i* angesehen. Zwei Bedingungen müssen erfüllt sein, damit die Sequenz zu einer Session wird:

1. Die *Form*-Antwort wird durch die vorherige Sequenz eindeutig festgelegt.
2. Jede Anfrage *POST* url*i* bezieht ihre url*i* aus der vorherigen Antwort *FORM* f*i*. (Ausnahme: url0)

Das Konzept einer Session verlangt nach einem Session-Zustand, der durch den Anwendungsprogrammierer implementiert werden muss, das zu Grunde liegende HTTP-Protokoll ist ja zustandslos. Der übliche Weg einer Zustandsspeicherung besteht darin, ein Zustandskürzel mit jedem *FORM* bzw. *POST* irgendwie mit zu übergeben. Dazu gibt es verschiedene mögliche Ansätze:

1. Direkt in der URL als Pfadzusatz oder Parameter
2. Speichern mittels Cookies auf der Client-Seite
3. Speichern im Dokument selber in einem versteckten Feld (hidden input field)

Probleme dabei können in allen Fällen auftreten, wenn der Benutzer im Web-Browser den Zurück-Knopf betätigt, oder eine Kopie der Seite/Session aufmacht und mit beiden Fenstern unterschiedlich verfährt, oder aber er speichert die Seite und öffnet sie erst viel später wieder (bookmarking). Das Zustandskürzel ist dann nicht mehr synchron mit dem auf dem Server gespeicherten Zustand.
Die WASH-Lösung besteht deshalb darin, die gesamte Session in einem versteckten Feld zu speichern, und auf dem Server keine Zustandsinformationen zu speichern. Genauer gesagt werden im Log nur die Benutzeranfragen und IO-Operationen gespeichert, der gesamte Zustand wird dann jedesmal wieder neu berechnet. Dabei wird genutzt, dass in den versteckten Feldern eines HTML-Dokuments große Datenmengen passen. Durch diese Lösung hat das Programm keinerlei Probleme mit dem Rückwärts-Knopf des Browsers oder dem Klonen von Fenstern. Nachteil dabei ist, dass dadurch der Dokumentenumfang sehr stark wachsen kann, bis der größte Teil eines Dokuments der Zustandsspeicherung dient. Dadurch wird natürlich auch das Programm langsamer, da immer erst alle Schritte aus dem Log wiederholt weren müssen, bevor neue Aktionen gestartet werden können. Es gibt aber auch noch einige Funktionen, um die Log-Größe zu verringern, wie z.B. die Funktion *once,* die mehrere Logeinträge in einem zusammenfasst.

Eine weitere mögliche Fehlerquelle bei der Zustandskontrolle von Webanwendungen ist durch die vielen verschiedenen einzelnen Skripte gegeben, aus denen eine Webanwendung häufig besteht. Diese müssen alle demselben Konzept der Zustandsspeicherung und -übergabe folgen. Der WASH/CGI-Ansatz hilft dabei auf zweierlei Art:

- Es wird dem Programmierer ermöglicht, die gesamte Anwendung in einem einzigen Programm zu schreiben.
- Eine transparente Darstellung des Session-Zustandes für den Programmierer

Sessions werden aus Dokumenten mittels folgender Konstruktoren erzeugt:

```
ask :: WithHTML XHTML_DOCUMENT CGI a -> CGI ()

tell :: (CGIOutput a) => a -> CGI ()
```

Dabei ist ein Wert vom Typ *CGI a* eine *CGI action*, die ein Ergebnis vom typ *a* liefert. Der Kontext *XHTML_DOCUMENT* in der Typdeklaration von *ask* zeigt, dass der Kontext ein einzelnes HTML-Element erwartet für die Konstruktion des Dokuments. Die *CGI actions* sind der wichtigste Bestandteil der Session-Sprache. Mittels do-Notation und dem Operator '>>=' werden die *CGI actions* verbunden. Ein Wert vom Typ *WithHTML context CGI a* ist eine *HTML action*, die eine Sequenz von Dokument-Knoten zurückgibt. Die XHTML-Konstruktoren *p*, *li*, *ul* sind Beispiele dafür:

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

Die (*Monad m*)-Einschränkung zeigt, dass *m a* eine *action* vom Typ *a* ist. In den meisten Fällen wird dies eine *CGI-action* sein.
ask ist nun also eine Funktion, die eine *HTML action* nimmt und eine *CGI action* liefert.
tell ist eine überladene Funktion, die jeden Wert, der in eine HTTP-Antwort umgewandelt werden kann, auf eine *CGI action* abbildet. Die *action* liefert dann die Antwort an den Browser. Zum Beispiel kann der Antworttyp ein HTML-Element sein, aber auch eine Zeichenkette, eine Referenz auf eine Datei, ein Zustandscode, oder eine Adresse (redirection URL). Ein Unterschied zwischen ask und tell besteht darin, dass durch *das E*rzeugen des XHTML-Dokument in *ask* weitere *CGI actions* ausgelöst werden können, wohingegen *tell* eine Interaktionssequenz beendet.
Ein einfaches Skript besteht aus einer einzigen CGI-Aktion. Da Haskell eine IO-Aktion als Hauptprogramm erwartet, gibt es eine Funktion *run*:

```
run :: CGI () -> IO ()
```

Analog dazu gibt es noch die Funktion *io*, die eine IO-Aktion in eine CGI-Aktion einbettet.

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

Mit dem bisherigen Wissen gucken wir uns noch einmal ein erstes Skript an:

```
main = 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")))
```

Das Argument von *run* ist eine *CGI action (ask),* die das Ergebnis der Funktion *standardPage,* angewandt auf "Hello World" und *myinfo*, anzeigt. Da das weitere Dokument keine weiteren *CGI actions* enthält, wird anschließend die Interaktion beendet.

2.4 Kommunikations-Sprache

Viele Operationen der Kommunikations-Teilsprache sind Wrapper, um Eingabeelemente mit Attributen zu versehen. Die Deklaration

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

definiert ein Typsynonym HTMLField, der den Standardtyp für solche Operationen darstellt.
Ein Wert vom Typ *HTMLField context a* erwartet eine Sequenz von Attributen (vom Typ: *WithHTML INPUT CGI ()*) für das Eigabefeld und gibt eine einelementige Sequenz mit dem Eingabeelement zurück.
Ein Eingabe-Knopf mit der dazugehörigen *CGI action* wird mittels folgender Funktion erstellt:

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

submit0 action erstellt also ein Eingabeelement mit dem Attribut type="submit". Im Browser wird dies dann als Eingabeknopf (submit-button) dargestellt, dessen Betätigung das zugehörige Form-Element an den Server schickt und dort eine Aktion auslöst.

Das Form-Element wird mittels *makeForm* erstellt:

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

Diese Funktion erstellt nicht nur die Form, sondern füllt auch noch die dazugehörigen Attribute 'action' und 'method' aus und speichert die aktuelle Session als verstecktes Feld in der Form.

Wir erweitern unser Beispiel nun also um einen submit-button:


```
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"))
```

Die obige Kombination von ask, standardPage und makeForm kommt so häufig vor, dass es dafür in der WASH/CGI eine eigene Funktion gibt:

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

Eingabe-Widgets (widget: Zusammensetzung aus 'window' und 'gadget', ein Wort aus der GUI-Programmierung) sind HTML-Aktionen, die etwas berechnen. Ein Widget-Konstruktor erstellt ein XHTML-Element mit zugehörigen Attributen und zusätzlich gibt er noch ein Eingabe-Handle (*input handle)* zurück. Mittels dieses *input handles,* der meist auf eine Variable abgebildet wird, kann man später auf den Wert des Widgets zugreifen.

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

Zuerst ist das *input handle* ungültig, da es noch keinen Wert erhalten hat. Das Übergeben eines *input handles* an eine Rückfrage-Aktion (callback action) geschieht mittels der *submit1* funktion:

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

Der Aufruf *'submit1 handle callback'* nimmt also ein ungültiges *input handle,* bekommt den Wert für das *handle* und macht es zu einem gültigen *handle* und übergibt es dann als Parameter für *callback*. Das Ergebnis ist wiederum ein *HTMLField* mit Attributen für den *submit button,* mit der XHTML-Repräsentation eines Eingabefelds mit dem Attribut type="submit".
Innerhalb der callback action holt sich die Operation *value* den Wert von einem gültigen *handle*:

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

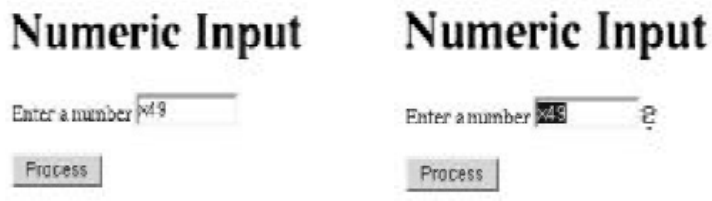
Wir erweitern nun das Beispiel um eine Namensabfrage:


```
main = run page1

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"))
```

Das *textInputField* aus dem obigen Beispiel ist nur für die Eingabe von Zeichenketten geeignet. Es gibt aber eine allgemeinere Funktion für ein Eingabefeld:

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

Natürlich gibt es keine Garantie, dass eine Benutzereingabe geparst werden kann, deshalb überprüft die *submit1* Funktion jedes *handle,* bevor es dieses weitergibt an die *callback action*. Ein *handle* ist nur gültig, wenn es geparst werden kann. Wenn eine Eingabe ungültig ist, wird eine Fehlermeldung daneben angezeigt.




```
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 "!")
```

Mit diesem Ansatz ist es nun möglich, die nötigen Haskell-Kenntnisse vorausgesetzt, Eingabefelder mit eigener Syntax zu kreieren: Man muss nur einen neuen Typ erstellen, einen Parser dazu schreiben und eine Reason-Funktion (*reason :: a -> String)*, die die Eingabe-Syntax des Typs erläutert.

Das folgende Beispiel kreiert ein Eingabefeld, was nicht-leere Strings erwartet:

```
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"
```

Der Einbau in das vorherige Beispiel sieht dann so aus:

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

Weitere Eingabe-Widgets (buttons, check boxes, selection boxes, images, etc) werden alle genauso als *typed input handles* behandelt. Z.B. liefert eine Check-Box ein *handle* vom Typ *InputField Bool INVALID* und ein Bild liefert eins mit Typ *InputField (Int, Int) INVALID.*

Bis jetzt hatten wir immer nur *callback actions* mit einer einem einzigen *handle* als Parameter behandelt. Um mehrere handles zu übergeben benutzt man die Funktion *submit*, von der *submit1* nur ein Spezialfall ist:

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

Der Typ *h INVALID* verallgemeinert das vorherige *InputField a INVALID*. Um ein Paar von *handles* zu kreieren benötigt man einen speziellen Konstruktor zum Zusammenfasen der beiden:

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

F2 h1 h2 gehört zur Typklasse *InputHandle,* wenn sowohl h1 als auch h2 dazugehören. Für andere Tupel oder Listen gibt es ähnliche Konstruktoren.
Zur Erläuterung schaue man sich das folgende Beispiel an, was nach Name und Passwort fragt:

```
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" ...
```

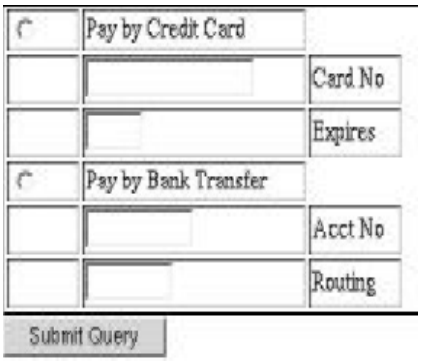
Die Funktion *check* überprüft, dass beide Felder nicht leer sind. *PasswordInputField* funktioniert wie *inputField,* aber generiert ein Eingabeelement mit type="password", damit die Zeichen nicht auf dem Bildschirm angezeigt werden.

Auch *radioButtons* sind in WASH implementiert, und zwar werden *radioButtons* gleichen Namens wie in HTML in einer *radioGroup* zusammengefasst.

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

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

Neue *radioButton* können also nur innerhalb einer *radioGroup* erzeugt werden, und es ist sichergestellt, dass alle denselben Typ haben.

Sehen wir uns nun das folgende Beispiel mit *radioButtons* an und das dazugehörige Code-Fragment:




```
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!")
```

Das Problem ist, dass für den Gültigkeitscheck alle 5 Felder (*rg, ccnrF, ccexF, acctF* und *routF)* gültig sein müssen, da in *submit* alle 5 geprüft werden. Für solche Fälle gibt es als Lösung eine neue Datenstruktur, den *DTree(decision tree)* mit den folgenden zwei Konstruktoren:

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

Die Funktion submitx wandelt einen *Dtree* dann in einen *submit button* um:

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

Eingesetzt im oberen Beispiel sieht es dann so aus:

```
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 amount (F2 ccnrF ccexF) =
  let ccnr = unCreditCardNumber (value ccnrF)
      expMonth = cceMonth (value ccexF)
  in
  standardQuery "Payment Confirmation"
            (text "Thanks for shopping at TinyShop.Com!")

payTransfer amount (F2 acctF routF) =
  let acct = unAllDigits (value acctF)
      rout = unAllDigits (value routF)
  in
  standardQuery "Payment Confirmation"
            (text "Thanks for shopping at TinyShop.Com!")
```

2.5 Persistenz-Sprache:

WASH bietet zwei verschieden Abstraktionen an (beide völlig typsicher), um eine bessere Persistenz der Session zu erreichen: den S*erver State* und den C*lient State*.

Der Server State ist global sichtbar für alle Programme auf dem Server und wird als abstrakter Datentyp *T a* definiert:

```
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))
```

Die Operation *init name initialValue* initialisiert einen neuen *Server State* mit dem Namen *name* und dem Wert *initialValue,* wenn es noch keinen *Server State* dieses Namen gibt.
Die Operation *get handle* gibt den Wert des *handle* zurück. Dieser Wert muss nicht aktuell sein, das *handle* kann veraltet sein, wenn der *Server State* sich nach der Erstellung des *handle* geändert hat.
Die Operation *set handle newValue* überschreibt den Wert des Server State Eintrags, auf den *handle* zeigt mit *newValue,* aber nur wenn das *handle* aktuell ist.
Die Operation *add handle newValue* ist für den Fall, dass der Server State Eintrag eine Menge (repräsentiert durch eine Liste) ist. Dies klappt immer, auch wenn das *handle* nicht aktuell ist.
Die Operation *current handle* liefert das aktuelle handle für den Server State Eintrag, auf den *handle* zeigt.

Damit der Speicher nicht zu voll wird, gibt es auch eine Garbage Collection für lange nicht mehr benutzte Werte.

Hier nun ein Beispiel, eine Highscoreliste:

```
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))
```

Das Client-side State Interface ist dem Server-seitigen sehr ähnlich, aber es gibt feine Unterschiede in seinem Verhalten. Implementiert ist es mittels Cookies.

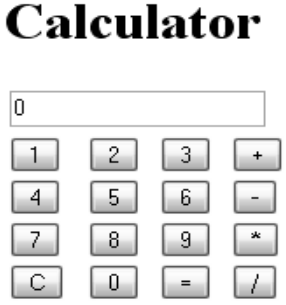
```
check :: (Read a, Show a, Types a) => String -> CGI (Maybe (T a))
init :: (Read a, Show a, Types a) => String -> a -> CGI (T a)
get :: (Read a, Show a, Types a) => T a -> CGI (Maybe a)
set :: (Read a, Show a, Types a) => T a -> a -> CGI (Maybe (T a))
current :: (Read a, Show a, Types a) => T a ->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

Die Operationen *init, set* und *current* funktionieren genau wie auf der Serverseite, allerdings gibt es bei der *set* Funktion keine Möglichkeit zu sehen, ob das Setzen des Cookie erfolgreich war. Bei der *get* Funktion muss man damit rechnen, dass sie fehlschlägt, weil das Cookie nicht mehr vorhanden ist. Deshalb gibt es noch die Funktion *check*, um zu überprüfen, ob ein Cookie noch vorhanden ist, sowie die Funktion *delete,* um einen Cookie zu löschen.

3. Weitere Beispiele

Das Skript ruft sich rekursiv wieder auf mit einem Integer-Parameter als Zähler.

```
module Main where

import Prelude hiding (head, div, span, map)
import HTMLMonad
import CGI

main =
  run mainCGI

mainCGI =
  counter 0

counter :: Int -> CGI ()
counter n =
  standardQuery "UpDownCounter" $
  do text_S "Current counter value"
     cnt <- activeInputField counter (fieldVALUE (show n))
     submit0 (counter (n + 1)) (fieldVALUE "++")
     submit0 (counter (n - 1)) (fieldVALUE "--")
```



Ein einfacher Taschenrechner, der sich immer nur die Anzeige und einen Wert merkt.

```
module Main where

import Char
import Prelude hiding (head, span, map)
import CGI hiding (div)

main =
  run mainCGI

mainCGI =
  calc "0" id
```

```
calc dstr f =
  ask $ standardPage "Calculator" $ makeForm $ table $
  do dsp <- tr_T (td_S (textInputField (fieldVALUE dstr)
                        ## attr_SS "colspan" "4"))
     let button c = td_T (submit dsp (calcAction c f) (fieldVALUE [c]))
     tr_T (button '1' ## button '2' ## button '3' ## button '+')
     tr_T (button '4' ## button '5' ## button '6' ## button '-')
     tr_T (button '7' ## button '8' ## button '9' ## button '*')
     tr_T (button 'C' ## button '0' ## button '=' ## button '/')

calcAction c f dsp
  | isDigit c = calc (dstr ++ [c]) f
  | c == 'C'  = mainCGI
  | c == '='  = calc (show (f (read dstr :: Integer))) id
  | otherwise = calc "0" (optable c (read dstr :: Integer))
  where dstr = value dsp
        optable '+' = (+)
        optable '-' = (-)
        optable '*' = (*)
        optable '/' = div
```

Einfaches Ratespiel, der Rater kann sich hinterher in einer globalen Highscore verewigen, die auf dem Server gespeichert wird:

```
module Main where

import Random
import Prelude hiding (head, div, span, map)
import List hiding (head, span, map)

import HTMLMonad
import CGI
import qualified Persistent2 as P

import Score

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

main :: IO ()
main =
  run mainCGI

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")

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

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
```

4. Zusammenfassung

Wir haben nun gesehen, wie man mit WASH einfache Text-Webseiten erstellt, wie getypte Eingabefelder funktionieren, und wie die Zustände der Webseiten verwaltet werden können. Alles in allem ist WASH für Haskell-Programmierer eine interessante Alternative zu den gängigen Websprachen, auch wenn es noch weiterentwickelbar ist, und wohl nie große Verbreitung unter Webentwicklern finden wird.

5. Literaturquellen

- Peter Thiemann: WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), pp. 192-208, Springer LNCS 2257 PDF
- Peter Thiemann: An embedded domain-specific language for type-safe server-side Web-scripting. ACM Transactions on Internet Technology, 5(1):1-46, 2005. PDF
- Peter Thiemann: WASH Server Pages. Proc. of the 8th International Symposium on Functional and Logic Programming (FLOPS 2006), pp. 277-293, Springer LNCS 3945 PDF