Deklarative Konstruktion Web-basierter Benutzerschnittstellen

(Diplomarbeit)

Christof Kluß

Oktober 2008

Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Betreut durch:
Prof. Dr. Michael Hanus

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

....

Christof Kluß

Inhaltsverzeichnis

1 Einleitung **1**

2 Die Programmiersprache Curry **4**
2.1 Grundlagen 4
2.2 Zustandsabhängige Berechnungen 6
2.3 Verwendete Entwurfsmuster 8

3 Graphische Benutzerschnittstellen **9**
3.1 Grundlagen 9
3.2 Tcl/Tk 10
3.3 Die GUI-Bibliothek 11

4 Webanwendungen **18**
4.1 Grundlagen 18
4.2 Die HTML-Bibliothek 22
4.3 JavaScript 30

5 Deklarative Konstruktion von allgemeinen Benutzerschnittstellen **36**
5.1 Benutzung der UI-Bibliothek 36
5.2 Trennung der Darstellung und Funktionalität 39
5.3 Änderungen von UIs zur Laufzeit 41
5.4 Beispiele 42

6 Implementierung der UI-Bibliotheken **44**
6.1 Das Interface (UI) 44
6.2 Generierung von Desktopanwendungen (UI2GUI) 46
6.3 Generierung von Desktop-ähnlichen Webanwendungen (UI2HTML) . . 48
6.3.1 Verwaltung der Event-Handler 49
6.3.2 Generierung neuer Webseiten 54
6.3.3 Änderung der Darstellung von UIs 56
6.3.4 Weitere Ideen zur Implementierung 58
6.4 Die GUI2HTML-Bibliothek 62

7 Typsichere Benutzerschnittstellen **64**

7.1 Die WUI-Bibliothek 64
7.2 Typsichere UIs 69
7.3 Beispiele 72

8 Weitere Anwendungen der UI2HTML-Bibliothek 76
8.1 Einbettung von UIs in HTML 76
8.2 Mehr Dynamik in Webanwendungen 77
8.2.1 SpicyWeb 77
8.2.2 SpicyWeb für UIs 81

9 Ähnliche Konzepte 83
9.1 Webanwendungen ohne Zusatzsoftware 83
9.2 Webanwendungen mit Zusatzsoftware 87

10 Zusammenfassung 90

A Inhalt der CD 92

B UI Beispiele 93

C Das Google Web Toolkit 96

1 Einleitung

Graphische Benutzerschnittstellen (GUIs) vereinfachen in vielen Bereichen die Bedienung von Anwendungen. Neben den normalen Desktopanwendungen werden immer häufiger Webanwendungen als GUIs genutzt.

Desktopanwendungen werden lokal auf einem Arbeitsplatzrechner installiert. Der Nachteil dieser Anwendungen ist, dass eine lokale Installation in größeren Unternehmen oft technisch problematisch und teuer ist. Gegenüber Webanwendungen haben sie jedoch ein besseres Laufzeitverhalten und einen höheren Bedienkomfort.

Webanwendungen werden auf einem Webserver ausgeführt. Wenn die Interaktion mit dem Benutzer über einen Webbrowser erfolgt, ist auf dem Computer des Benutzers keine weitere Installation von Software notwendig. Änderungen an einer solchen Anwendung erfolgen demnach nur auf dem Webserver, was die Wartung erheblich vereinfacht.

Benutzerschnittstellen können unabhängig von der genauen Art der Anwendung sehr ähnliche Aufgaben erfüllen. Es bietet sich also an, aus der gleichen Beschreibung verschiedene Arten von Anwendungen zu erzeugen. Für die Beschreibung von GUIs gibt es verschiedene Konzepte, die teilweise stark von der benutzten Programmiersprache abhängig sind. Programmiersprachen lassen sich in zwei Programmierparadigmen einordnen: die imperative und die deklarative Programmierung.

In *imperativen Programmiersprachen* beschreibt der Programmierer wie ein Algorithmus abläuft. Dazu gibt er die Schritte an, die nacheinander abgearbeitet werden sollen. Ein imperatives Programm ist eine Folge von Zustandsänderungen und das Ergebnis der Programmausführung hängt stark von der Auswertungsreihenfolge ab. Dabei gibt es viele Fehlerquellen wie unerwünschte Seiteneffekte und fehlende Initialisierungen.

Deklarative Programmiersprachen dagegen bieten ein höheres Abstraktionsniveau. Der Programmierer beschreibt durch Regeln wie das Programm mit der Eingabe umgehen soll. Dabei wird zwischen funktionalen und logischen Programmiersprachen unterschieden. In einer funktionalen Programmiersprache ist ein Programm eine Menge von Funktionsdefinitionen. Dabei ist der Wert eines Ausdrucks nur von seiner Umgebung abhängig und nicht vom Zeitpunkt seiner Auswertung. In einer logischen Programmiersprache ist ein Programm eine Menge von Relationen, anhand der eine

Lösungsaussage berechnet wird. Vorteile logischer Programmierung sind freie Variablen und Nichtdeterminismus.

Um die Vorteile der deklarativen Programmierung aufzuzeigen, werden immer mehr Bibliotheken entwickelt, die den Programmierer beim Lösen von alltäglichen Problemen unterstützen. In dieser Diplomarbeit wird eine allgemeine Möglichkeit zur Beschreibung von graphischen Benutzerschnittstellen vorgestellt. Aus dieser allgemeinen Beschreibung können dann verschiedene Arten konkreter GUIs generiert werden. Dazu werden Bibliotheken implementiert, die aus einer Beschreibung einerseits Desktopanwendungen und andererseits Desktop-ähnliche Webanwendungen erzeugen können. Als Programmiersprache wird Curry [10] verwendet. Curry bietet sowohl die Möglichkeit der funktionalen als auch der logischen Programmierung, wodurch alle Vorteile der deklarativen Programmierung genutzt werden können.

Um einen ersten Eindruck einer UI-Beschreibung zu bekommen, beschreibt Listing 1.1 einen interaktiven Zähler. Abbildung 1.1 zeigt zwei aus dieser Beschreibung generierte Anwendungstypen. In dem Textfeld kann der Benutzer Zahlen eingeben. Wenn er den „Increment“-Button drückt, wird die Zahl inkrementiert, durch Drücken auf „Reset“ wird der Wert auf 0 zurückgesetzt und mit dem „Stop“-Button kann die Anwendung beendet werden.

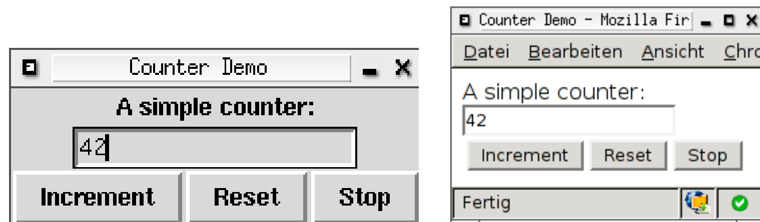

Abbildung 1.1: Interaktiver Zähler ausgeführt als Desktopanwendung (links) und als Webanwendung (rechts)

```
counter = col [
  label "A simple counter:",
  entry val "0",
  row [button inc "Increment",
       button reset "Reset",
       button exitUI "Stop"]]
  where
    val free
    reset env = setValue val "0" env
    inc env = do v <- getValue val env
                 setValue val (show (readInt v + 1)) env
```

Listing 1.1: Beschreibung eines interaktiven Zählers

In den nächsten 3 Kapiteln werden die Grundlagen und vorhandene Konzepte vorgestellt:

- die Programmiersprache Curry (Kapitel 2)
- graphische Benutzerschnittstellen (Kapitel 3)
- Webanwendungen (Kapitel 4)

Auf diesen Konzepten aufbauend wird in Kapitel 5 eine neue einheitliche deklarative Beschreibung von Benutzerschnittstellen eingeführt, deren Implementierung in Kapitel 6 beschrieben wird. Um Desktop-ähnliche Webanwendungen zu realisieren, wird eine vorhandene Curry-Bibliothek für Webanwendungen erweitert. In Kapitel 7 werden typsichere Widgets vorgestellt und schließlich in Kapitel 8 zusätzliche Möglichkeiten von Web-basierten Benutzerschnittstellen eingeführt.

In Kapitel 9 werden einige Frameworks vorgestellt, die eine ähnliche Funktionalität wie die UI-Bibliotheken haben.

2 Die Programmiersprache Curry

2.1 Grundlagen

Die deklarative Programmiersprache Curry wird im *Report on Curry* [10] beschrieben. Ein Curry-Programm besteht aus Datentyp- und Funktionsdefinitionen. Datentypen dienen in Funktionen als Grundlage für die Berechnung. Die Syntax eines Curry-Programms erinnert an Haskell[1]; allerdings ist in Curry auch logische Programmierung möglich.

Datentypen

Datentypdefinitionen bestehen aus dem Schlüssenwort `data`, einem Namen, Typparametern für polymorphe Typen und den durch `|` getrennten Konstruktoren. Vordefiniert sind die elementaren Datentypen `Int`, `Float` und `Char`.

```
data Bool = True | False
data Maybe a = Nothing | Just a
data List a = [] | (:) a (List a)
```

Listing 2.1: Beispiele für Datentypdefinitionen

`Bool` führt einen Typ für Wahrheitswerte ein. Mit dem polymorphen Typ „`Maybe a`“ lassen sich optionale Werte darstellen. „`List a`“[2] ist ein rekursiver Typ, mit dem Listen dargestellt werden können, deren Elemente vom Typ `a` sind. Der Datenkonstruktor `[]` steht für die leere Liste und der Konstruktor `:` fügt ein Element als Kopf in eine Liste ein. Da Listen in der funktionalen Programmierung häufig verwendet werden, führt Curry einige Vereinfachungen für Listen ein. Statt „`List a`“ wird der Listentyp als `[a]` geschrieben und mit `(x:xs)` wird eine Liste konstruiert, die aus dem Element `x` gefolgt von der Restliste `xs` besteht. Für eine Liste der Form `1:(2:(3:[]))` gibt es in Curry die Notation `[1,2,3]`.

[1] `http://www.haskell.org/`

[2] Die Definition des Datentyps „`List a`“ in Listing 2.1 ist keine gültige Curry-Syntax, sondern soll die Spezialsyntax für Liste verdeutlichen.

Mit dem Schlüsselwort `type` können Typ-Synonyme definiert werden. Für Zeichenketten gibt es in Curry das Typ-Synonym `String`.

```
type String = [Char]
```

Funktionsdefinitionen

Eine Funktion besteht aus einer optionalen Typdefinition gefolgt von definierenden Gleichungen. Eine Typdefinition für eine n-stellige Funktion f hat die Form

f `::` τ_1 `->` τ_2 `->` $\cdots$ `->` τ_n `->` τ

wobei $\tau_1, \ldots, \tau_n$ Typausdrücke sind und τ der Ergebnistyp ist. Die einfachste Form einer definierenden Gleichungen ist

$f\ t_1\ t_2\ \ldots\ t_n$ `=` e

wobei $t_1, \ldots t_n$ Datenterme sind und die rechte Seite e ein Ausdruck ist. Curry ist streng getypt. Die Typdefinitionen müssen im Programm jedoch nicht angegeben werden, da ein Typsystem sie automatisch herleitet. Funktionen können auch Bedingungen haben, eine solche Funktion hat die Form

$f\ t_1\ t_2\ \ldots\ t_n$ `|` c `=` e

Als Beispiele eine Implementierung der Fakultätsfunktion (`fac`) und der Funktion `append` die zwei Listen konkateniert:

```
fac :: Int -> Int
fac n | n == 0 = 1
      | True   = n * fac (n-1)

append :: [a] -> [a] -> [a]
append [] ys = ys
append (x:xs) ys = x:append xs ys
```

`append` ist in Curry als Infixoperation `(++)` vordefiniert. `[1,2] ++ [3,4]` wird also zu `[1,2,3,4]` ausgewertet.

Nichtdeterminismus

In Curry ist es möglich, Funktionen mit nichtdeterministischem Verhalten zu definieren, indem man überlappende linke Regelseiten oder freie Variablen verwendet. Die folgende nichtdeterministische Funktion fügt ein Element in eine Liste ein:

```
insert x [] = [x]
insert x (y:ys) = x:y:ys
insert x (y:ys) = y:insert x ys
```

Der Aufruf „`insert 3 [1,2]`“ hat die Ergebnismenge

```
{[3,1,2], [1,3,2], [1,2,3]}.
```

Freie Variablen

Mit den Regeln

```
mother John = Christine
mother Alice = Christine
mother Andrew = Alice
```

für Verwandtschaftsbeziehungen, kann ein Kind von `Alice` berechnet werden, indem die Gleichung „`mother x =:= Alice`“ gelöst wird. Hierbei ist `x` eine freie Variable, welche an `Andrew` gebunden wird. Auf ähnliche Weise können die Enkel von `Christine` berechnet werden, indem die Gleichung „`mother (mother x) =:= Christine`“ gelöst wird. In Curry muss jede freie Variable `x` in der Form „`x free`“ lokal deklariert werden.

```
last l | append xs [x] =:= l = x where x,xs free
```

`last` liefert das letzte Element einer Liste. Bei der Auswertung von „`last [1,2,3]`“ wird `x` an das Element `3` gebunden und somit als Ergebnis `3` zurückgegeben.

Funktionen höherer Ordnung

In Curry können Funktionen höherer Ordnung definiert werden. Zum Beispiel wendet die Funktion `map` eine Funktion auf alle Elemente einer Liste an. Die Funktion `maybe` wird als Abkürzung für Fallunterscheidungen über Werte vom Typ `Maybe a` verwendet.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

maybe :: b -> (a -> b) -> Maybe a -> b
maybe n _ Nothing  = n
maybe _ f (Just x) = f x
```

2.2 Zustandsabhängige Berechnungen

Ein- und Ausgabe

Wie in Haskell gibt es in Curry monadische Ein- und Ausgabe. Ein- und Ausgabefunktionen haben Seiteneffekte. Wenn eine solche Funktion einen Wert vom Typ `a`

liefern soll, dann muss sie den Ergebnistypen `IO a` haben. Bei der Auswertung wird eine Sequenz von IO-Aktionen ausgeführt. Vordefiniert gibt es Funktionen wie

```
getChar :: IO Char
putChar :: Char -> IO ()
```

`getChar` liest ein Zeichen von der Standardeingabe, `putChar` gibt ein Zeichen auf der Standardausgabe aus. Mit den Operationen

```
(>>) :: IO -> IO a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b
```

können zwei Aktionen kombiniert werden. „`a >>= fa`" führt eine Aktion `a` aus, die ein Ergebnis `res` liefert und das Ergebnis von „`fa res`" ist schließlich das Gesamtergebnis von `(>>=)`. Die Operation `(>>)` nimmt zwei Aktionen und liefert die Aktion, die beide Aktionen hintereinander ausführt.

```
putStr :: String -> IO ()
putStr []     = done
putStr (c:cs) = putChar c >> putStr cs
```

Schließlich ist „`return exp`" die Aktion, die `exp` liefert. Die IO-Aktion `return ()` gibt nichts zurück. Als Abkürzung für die Kombination von IO-Aktionen gibt es in Curry die `do`-Notation.

`do`-Notation	äquivalent zu
`do expr`	`expr`
`do expr` `   stmts`	`expr >> do stmts`
`do p <- expr` `   stmts`	`expr >>= \p -> do stmts`
`do let decls` `   stmts`	`let decls` `in  do stmts`

Polymorphe Speicherzellen

Das Konzept polymorpher Speicherzellen gibt es in der Sprachdefinition von Curry nicht. Es ist jedoch in den meisten Curry-Implementierungen vorhanden. In der Bibliothek IOExts, der PAKCS-Distribution [9], wird der abstrakte Datentyp „`IORef a`" definiert, welcher Speicherzellen beliebigen Typs repräsentiert.

```
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

Die Funktion `newIORef` generiert eine neue `IORef`, `readIORef` liest den aktuellen Wert einer `IORef` und `writeIORef` schreibt einen neuen Wert in eine `IORef`. Dieses Konzept sollte jedoch möglichst sparsam eingesetzt werden, da sequentielle Programme in funktionalen Sprachen durch Seiteneffekte oft schwer verständlich sind.

2.3 Verwendete Entwurfsmuster

Entwurfsmuster (*Design Pattern*) sind wiederverwendbare Vorlagen, um Programmieraufgaben zu lösen. Funktional logische Programmiersprachen ermöglichen einige neue Entwurfsmuster [1]. Für diese Arbeit interessant sind die folgenden Muster:

Locally Defined Global Identifier: Durch dieses Entwurfsmuster sind lokale Variablen auch im globalen Kontext eindeutig. Es werden lokal keine konkreten Werte, sondern freie Variablen benutzt. So können Datenstrukturen vereinigt werden, ohne dass es zu Namenskonflikten kommt. Für die Darstellung werden die freien Variablen eindeutig instanziiert. Dieses Muster kann für Referenzen (IDs) eingesetzt werden, die global eindeutig sein müssen.

Opaque Type: Für das Entwurfsmuster *Locally Defined Global Identifier* ist es wichtig, dass freie Variablen verwendet werden. Die Idee des Musters *Opaque Type* ist, die Konstruktoren eines Datentyps zu verstecken und damit dem Benutzer keine andere Möglichkeit zu geben, als freie Variablen zu verwenden. Dieses kann in Curry dadurch realisiert werden, dass die Bibliothek zwar den Datentyp, aber nicht seine Konstruktoren exportiert.

3 Graphische Benutzerschnittstellen

3.1 Grundlagen

Bei einer graphische Benutzerschnittstelle (*Graphical User Interface*, GUI) handelt es sich um ein Hauptfenster (*window*), in welchem graphische Elemente platziert werden, die so genannten Widgets (*window/gadget*). Die wichtigsten Widgets sind Schaltflächen (*buttons*), Textfelder (*entries*) und einfache Texte (*labels*). Widgets können im Programm mit Funktionen verknüpft werden, die ausgeführt werden, wenn der Benutzer ein spezielles Ereignis (*event*) auslöst. Beispielsweise wird durch das Klicken auf einen „Increment“-Button eines interaktiven Zählers eine Funktion aufgerufen, die den Wert der Zähleranzeige erhöht.

In vielen Programmiersprachen gibt es Bibliotheken zum Programmieren von graphischen Benutzeroberflächen, welche meistens nicht direkt mit dem Fenstersystem kommunizieren, sondern graphische *Toolkits* verwenden. Diese stellen fertige Fensterelemente und Layoutmanager zur Verfügung und können somit als Schnittstelle für die Programmierung von graphischen Oberflächen benutzt werden. Bei der Definition von GUIs gibt es große Unterschiede. Vor allem imperativen Programmiersprachen weisen bezüglich der Konstruktion einige Nachteile auf:

- Die Definition der GUI wird mit Zeigerstrukturen aufgebaut und enthält oft eine lange Sequenz von Befehlen. Die tatsächliche Layoutstruktur der GUI ist dadurch nur noch schwer im Quelltext erkennbar, was auch die Wartung erschwert.
- Um Widgets zu referenzieren werden in Skriptsprachen meistens fest gewählte Strings als IDs benutzt. Dabei kann es zu Tippfehlern kommen, die erst zur Laufzeit erkannt werden können. Außerdem ist es dadurch schwierig verschiedene GUIs zu einer neuen zusammenzusetzen, da es zu Namenskonflikten kommen kann.

Unabhängig vom konkreten Konzept kann die Beschreibung einer graphische Benutzerschnittstelle in drei Bestandteile aufgeteilt werden.

Struktur: Einzelne Widgets werden im Fenster mit *Layoutmanagern* platziert. Diese ordnen beispielsweise mehrere Widgets in einer Spalte oder Reihe an.

Funktionalität: Der Benutzer kann durch Interaktion mit der graphischen Oberfläche Ereignisse auslösen, die im Programm mit Funktionen (Event-Handlern) verknüpft sind. Wenn der Anwender einen Button drückt, kann ein Event-Handler ausgeführt werden. Event-Handler können wiederum auf Widgets der GUI zugreifen, was durch Referenzen realisierbar ist. Eine GUI hat also auch eine logische Struktur.

Darstellung: Die Darstellung von Widgets kann geändert werden. Zum Beispiel können Fehlermeldungen farblich hervorgehoben werden. Außerdem ist es oft möglich, die Ausrichtung und den Schriftstil von Texten zu ändern.

Ein Ziel der Beschreibung von graphischen Benutzerschnittstellen ist es, diese Bereiche bei der Definition möglichst voneinander zu trennen und damit leichter wiederverwendbar zu machen.

Für die deklarative Beschreibung von Benutzerschnittstellen gibt es in Curry die GUI-Bibliothek. Sie stellt graphische Anwendungen mit Hilfe des Toolkits Tk [15] dar.

3.2 Tcl/Tk

Tcl[1] (*Tool command language*) ist eine Skriptsprache, die vor allem durch ihre GUI-Bibliothek Tk (*Toolkit*) bekannt wurde. Tk stellt Tcl-Befehle für die Programmierung von graphischen Oberflächen zur Verfügung. Zur Laufzeit kann Code an einen Tcl/Tk-Interpreter gesendet und somit das laufende Programme geändert werden.

Zur GUI-Beschreibung gibt es in Tk die üblichen Widgets wie `label`, `entry` und `button`. Die Namensgebung der Widgets beschreibt dabei gleichzeitig eine Hierarchie. Das Hauptfenster hat den Namen „`.`“. Jedes Kindwidget des Hauptfensters hat einen Namen der Form „`.foo`“ und ein Kind von „`.foo`“ hat den Namen „`.foo.bar`“ und so weiter. Alle Tk-Widgets haben eine `configure`-Operation, mit der ihre Eigenschaften festgelegt werden können. Mit dem Kommando „`-textvariable`“ kann einem Widget eine Referenz zugewiesen werden. Außerdem kann mit dem Kommando `bind` einem Widget ein Event zugeordnet werden. Listing 3.1 beschreibt einen interaktiven Zähler in Tcl/Tk. Dabei wird der Layoutmanager `grid` benutzt, der Widgets tabellarisch in Zellen positioniert.

[1] `http://tcl.tk/`

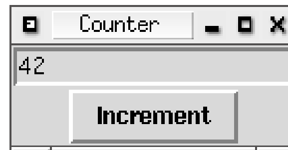

Abbildung 3.1: Interaktiver Zähler

```
#!/usr/bin/wish

wm title . "Counter"

entry .a
.a configure -textvariable _a
set _a "0"

button .b
.b configure -textvariable _b
bind .b <ButtonPress-1> { set _a [expr $_a+1] }
set _b "Increment"

grid .a -column 1 -row 1
grid .b -column 1 -row 2
```

Listing 3.1: Einfacher Zähler als Tcl/Tk Anwendung

3.3 Die GUI-Bibliothek

Mit der Curry-Bibliothek GUI [3] können Desktopanwendungen deklarativ beschrieben werden. Dazu wird aus einer GUI-Beschreibung ein Curry-Programm generiert, das mit dem Tcl/Tk-Interpreter `wish` (*windowing shell*) eine Tk-Oberfläche erstellt. Die weitere Kommunikation zwischen Curry-Programm und `wish`-Prozess findet mit Tcl-Kommandos über einen Kommunikationskanal statt (siehe Abbildung 3.2).

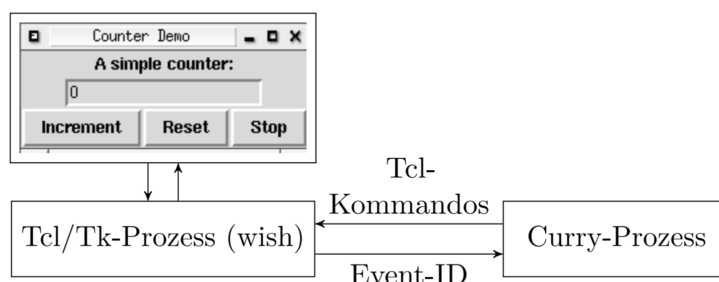

Abbildung 3.2: Kommunikation zwischen Tcl/Tk- und Curry-Programm

Die GUI-Bibliothek bietet ein hohes Abstraktionsniveau, um GUIs zu beschreiben. Sie leistet folgende Aufgaben:

- Übersetzen der GUI-Definitionen in Tcl/Tk-Code
- Verwaltung der Event-Handler
- Empfangen von Nachrichten (Events) vom Tcl/Tk-Interpreter und Aufruf entsprechender Event-Handler

Die Struktur einer GUI kann als Baum dargestellt werden. Abbildung 3.3 zeigt die Struktur eines interaktiven Zählers. Die gestrichelten Linien verdeutlichen, dass eine Verbindung bzw. Referenz zwischen den Buttons und dem Textfeld (Entry) besteht, denn die Handlerfunktionen der Buttons müssen auf den Wert des Textfeldes zugreifen können.

Der interaktive Zähler wird in der gesamten Diplomarbeit als Beispiel einer Benutzerschnittstelle eingesetzt. Die logische Struktur des Zählers verhält sich folgendermaßen:

- Die Handlerfunktion des „Increment“-Buttons liest den Wert aus dem Textfeld, inkrementiert ihn und schreibt ihn zurück in das Textfeld. Falls der Wert im Textfeld nicht in eine ganze Zahl konvertiert werden kann, wird er als 0 betrachtet.
- Die Handlerfunktion des „Reset“-Buttons setzt den Wert des Textfeldes auf 0.
- Der Benutzer kann jederzeit den Wert des Textfeldes per Eingabe mit der Tastatur ändern.

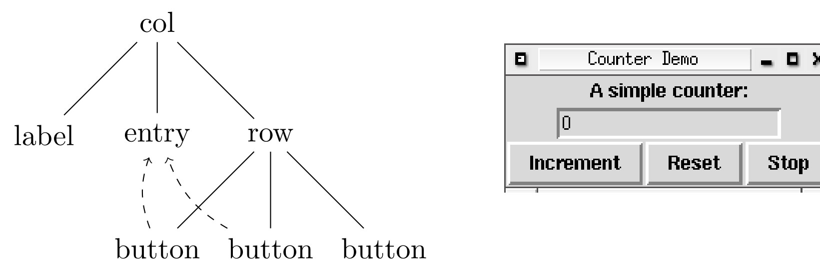

Abbildung 3.3: Struktur des Zählers als Baum

Eine hierarchische Struktur wie in Abbildung 3.3 wird in der GUI-Bibliothek mit Listen realisiert. Neben der Struktur werden in der Beschreibung Aktionen definiert, die bei bestimmten Benutzer-Events ausgeführt werden. Die GUI-Bibliothek verwendet für die Definition von Widgets den Datentypen `Widget`.

```
data Widget = Entry        [ConfItem]
            | PlainButton  [ConfItem]
            | Label        [ConfItem]
            | Col     [...] [Widget]
```

```
              | Row    [...] [Widget]
              ...

data ConfItem = Handler Event (GuiPort -> IO [...])
                | WRef WidgetRef
                | Text String
                ...

data Event = DefaultEvent | Return | MouseButton1 | ...
```

Zum einen gibt es die elementaren Widgets wie Texteingabefelder (`Entry`), Schaltflächen (`PlainButton`) und Texte (`Label`), aber auch Widgets wie Menüs und Listen. Zum anderen existieren Konstruktoren wie `Col` und `Row`, die eine Liste von Widgets als Argument haben, die in der graphischen Oberfläche als Spalte oder Zeile dargestellt werden. In der ersten Komponente von `Col` und `Row` kann die Ausrichtung der Widgets festgelegt werden.

Die Konstruktoren der elementaren Widgets bekommen als Argument eine Liste mit Werten vom Typ `ConfItem` übergeben. Mit diesen Werten können ähnliche Angaben gemacht werden wie mit `configure` in Tcl/Tk, siehe Listing 3.1. Unter anderem gibt es Konstruktoren für Event-Handler (`Handler`), Referenzen (`WRef`) und Beschriftungen (`Text`). Mit einem Event-Handler kann angegeben werden wie ein Widget auf die Interaktion eines Benutzers reagiert. Hat ein Widget einen Handler der Form „`Handler event cmd`" bedeutet dies: Wenn das Ereignis `event` auftritt, soll die IO-Aktion `cmd` als Event-Handler ausgeführt werden. Dieser Event-Handler kann wiederum mit Hilfe von Referenzen (`WRef`) den Wert anderer Widgets abfragen und ändern. Bei der Definition einer Benutzerschnittstelle können für Referenzen nur freie Variablen verwendet werden, da der Konstruktor des Datentyps `WidgetRef` nicht von der Bibliothek exportiert wird. Dadurch können Tippfehler bei den Bezeichnern schon vom Compiler erkannt werden. Es wird also das funktional logische Entwurfsmuster *Opaque Type* [1] benutzt.

Die Event-Handler der Widgets werden durch Interaktion mit dem Benutzer ausgelöst, sind also Curry-Funktionen mit Ergebnistyp „`IO a`". In der Bibliothek gibt es einige vordefinierte IO-Aktionen zur Manipulation von Widgets. Diese können in selbstdefinierten Event-Handlern genutzt werden. Ein Wert vom Typ `GuiPort` dient in der GUI-Bibliothek als Kommunikationsschnittstelle zwischen Tcl/Tk- und Curry-Prozess. Jede Aktion, die auf ein Widget zugreift benutzt diese Kommunikationsschnittstelle, bzw. Umgebung, um den Inhalt der Widgets abzufragen und zu ändern. Mit vordefinierten IO-Aktionen in der GUI-Bibliothek kann auf Widgets zugegriffen werden:

```
getValue :: WidgetRef -> GuiPort -> IO String
setValue :: WidgetRef -> String -> GuiPort -> IO ()
```

Der Aufruf „`getValue ref gp`“ liefert den Wert des Widgets mit der Referenz `ref` und der Aufruf „`setValue ref str gp`“ ändert den Wert des Widgets mit der Referenz `ref`. Eine IO-Aktion `cmd`, die als Event-Handler dienen soll, wird einem Widget in der Form „`Handler event cmd`“ übergeben:

```
cmd :: GuiPort -> IO [ReconfigureItem]
```

Die Handler-Funktion für den „Increment“-Button des interaktiven Zählers kann nun wie in Listing 3.2 implementiert werden.

```
inc gp = do v <- getValue val gp
            setValue val (show (readInt v + 1)) gp
            return []
```

Listing 3.2: Handlerfunktion des „Incement“-Buttons

Die Funktion liest zunächst den Wert des Widgets mit der Referenz `val` ein und schreibt einen um 1 inkrementierten Wert in das Widget zurück. Der Wert aller Widgets ist vom Typ `String`. Für die Konvertierung des Wertes können die Funktionen

```
readInt :: String -> Int
show :: a -> String
```

eingesetzt werden. Durch den Aufruf „`readInt str`“ wird die Zeichenkette `str` in eine ganze Zahl konvertiert. Falls die Konvertierung nicht möglich ist, wird `0` zurückgegeben. Die Funktion `show` konvertiert einen Wert eines beliebigen Datentyps in seine Stringrepräsentation.

Eine mit dem Datentyp `Widget` definierte graphische Oberfläche wird mit „`runGUI titel widget`“ im Curry-Programm ausgeführt. Listing 3.3 stellt den vollständige Quelltext eines interaktiven Zählers mit der GUI-Bibliothek dar.

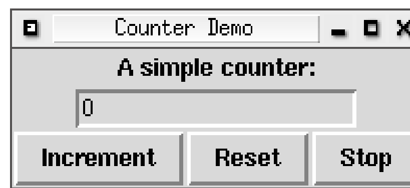

Abbildung 3.4: Zähler-GUI

```
import GUI
import Read

counterGUI = Col [] [
  Label [Text "A simple counter:"],
  Entry [WRef val, Text "0"],
  Row [] [
   PlainButton [Text "Increment", Handler DefaultEvent inc],
```

```
    PlainButton [Text "Reset", Handler DefaultEvent reset],
    PlainButton [Text "Stop", Handler DefaultEvent exit]]]

  where val free
        inc gp = do v <- getValue val gp
                    setValue val (show (readInt v + 1)) gp
                    return []
        reset gp = setValue val "0" gp >> return []
        exit gp = exitGUI gp >> return []

main = runGUI "Counter Demo" counterGUI
```

Listing 3.3: Interaktiver Zähler mit der GUI Bibliothek

In Listing 3.3 dient die freie Variable `val` als Referenz für das Textfeld (`Entry`). Die Funktionen `inc`, `reset` und `exit` sind die Event-Handler der Anwendung. Die Funktion `inc` erhöht den Wert, der gerade im Textfeld steht. Der Wert des Textfeldes kann vom Benutzer jederzeit geändert werden. Der Event-Handler `res` setzt den Wert des Textfeldes auf 0 und der Event-Handler `exit` beendet die Anwendung, indem er die vordefinierten Aktion `exitGUI` aufruft. Die Handlerfunktionen in Listing 3.3 geben alle eine leere Liste zurück. Diese Liste kann Elemente vom Typ `ReconfigureItem` enthalten. Mit diesem Datentyp ist es möglich bestehende Widget-Konfigurationen zur Laufzeit zu ändern. Mit Werten vom Typ `ReconfigureItem` kann zum Beispiel die Beschriftung und die Farbe eines Widgets geändert werden.

```
data ConfItem
  = Handler Event (GuiPort -> IO [ReconfigureItem]) | ...

data ReconfigureItem = WidgetConf WidgetRef ConfItem | ...
```

Da jedoch oft auf diese Möglichkeit verzichtet werden kann, werden einige Abkürzungen eingeführt, durch die die Definition von graphischen Oberflächen einfacher wird. Für das Zählerbeispiel können folgende Abkürzungen zur Vereinfachung genutzt werden:

```
col = Col []
row = Row []

updateValue upd wref gport = do
  val <- getValue wref gport
  setValue wref (upd val) gport

command c = Handler DefaultEvent c

cmd c = command (\gport -> c gport >> return [])

button c confs = PlainButton (cmd c : confs)
```

Die Aktion `updateValue` wendet eine Updatefunktion `upd` auf den Wert eines Widgets an. Die Funktion `command` erzeugt einen Handler, der bei dem `DefaultEvent` des Widgets ausgeführt wird. Bei einem Button ist das „DefaultEvent" der Klick mit der Maus auf den Button. Mit der Funktion `cmd` kann direkt eine Aktion mit Ergebnistyp „`IO ()`" angegeben werden. Mit diesen Abkürzungen ist eine noch übersichtlichere Implementierung des Zählers möglich:

```
counterGUI =
  col [Label [Text "A simple counter:"],
       Entry [WRef val, Text "0"],
       row [button inc      [Text "Increment"],
            button reset    [Text "Reset"],
            button exitGUI [Text "Stop"]]]
  where
    val free
    inc gp = updateValue (\ v -> show (readInt v + 1)) val gp
    reset gp = setValue val "0" gp
```

Listing 3.4: Vereinfachte Form des Zählers

Implementierung

Abbildung 3.5 skizziert das Konzept der GUI-Bibliothek. Neben dem Curry-Hauptprogramm existiert ein Tcl/Tk-Programm, das als Schnittstelle zur graphischen Darstellung der GUI mit dem Fenstersystem dient. Da die Handlerfunktionen aus einer Sequenz von IO-Aktionen bestehen, kann auch auf das Dateisystem zugegriffen werden.

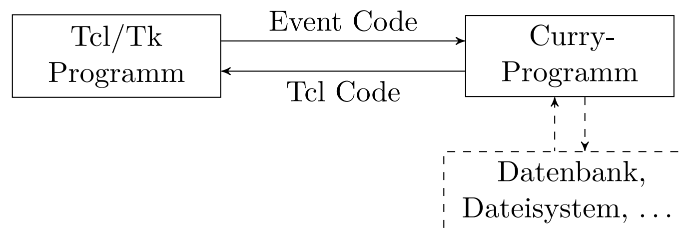

Abbildung 3.5: Konzept der GUI-Bibliothek

Mit der externen Funktion `connectToCommand` aus der Curry-Bibliothek IOExts wird der Shell-Prozess `wish` ausgeführt und der Ein- und Ausgabekanal dieses Prozesses als Handle zurückgegeben. Auf diesem Kanal findet die Kommunikation zwischen Tcl/Tk- und dem Curry-Programm statt.

Wird „`runGUI title widget`" ausgeführt, werden die Referenzen, die als freie Variablen angegeben wurden, instanziiert und aus der GUI Beschreibung ein Tcl/Tk-Programm erzeugt. Dieses Tcl/Tk-Programm stellt die graphische Oberfläche dar und

bietet Funktionen an, mit denen Curry auf den aktuellen Inhalt von Widgets zugreifen kann.

Die Übersetzung in Tcl/Tk-Code wird mit der Funktion `initSchedule` der GUI-Bibliothek gestartet. Die entstehenden Tcl-Kommandos werden als Strings über den Kommunikationskanal (`gport`) an den `wish` Prozess gesendet.

Eine wesentliche Aufgabe, die die GUI-Bibliothek leistet, ist die Verwaltung der Event-Handler. Auf Tcl/Tk Seite bekommen die Widgets, die Events auslösen sollen, eine spezielle Tcl-Funktion zugewiesen. Wenn der Benutzer das entsprechende Event auslöst, sendet diese Funktion eine ID über den Kanal (`gport`) an das Curry-Programm. Dieses sucht anhand der ID den passenden Handler und führt die Handlerfunktion aus.

Jeder Handler „`Handler evtype cmd`“ wird als Tupel `(label,evtype,cmd)` in einer Liste verwaltet. Die Variable `label` ist dabei die ID des Handlers, die von der Bibliothek eindeutig instanziiert wird. Die rekursive Funktion `scheduleTkEvents` verwaltet die Ereignisse, die vom Benutzer ausgelöst werden. Dabei wiederholt sich folgender Ablauf:

1. Wenn der Benutzer ein Event auslöst, sendet die generierte Tcl/Tk-Anwendung eine Event-ID an das Curry-Programm.
2. Das Curry-Programm wartet auf Nachrichten, die über den Kommunikationskanal (`gport`) vom Tcl/Tk Programm kommen. Anhand der Event-ID wird die entsprechende Handlerfunktion herausgesucht.
3. Der Event-Hander wird schließlich sequentiell ausgeführt. Dabei kann mit den vordefinierten Aktionen wie `getValue` und `setValue` über Referenzen auf den Inhalt von Widgets zugegriffen und dieser verändert werden. Diese Aktionen werden in Tcl/Tk-Kommandos übersetzt und ausgeführt.

Es ist auch möglich, externe Handler anzugeben. Die Funktion `hWaitForInputsOrMsg` aus der IO-Bibliothek wartet dabei auf eine Eingabe eines Handles oder eines externen Nachrichtenstroms.

4 Webanwendungen

In diesem Kapitel werden einige Konzepte zur Erstellung von dynamischen Webanwendungen vorgestellt. Nach den Grundlagen wird auf die Curry-Bibliothek HTML eingegangen, die ein hohes Abstraktionsniveau für die Entwicklung von Webanwendungen bietet. Schließlich wird die Programmiersprache JavaScript vorgestellt, mit der der Inhalt von Webseiten ohne Kommunikation mit dem Server geändert werden kann.

4.1 Grundlagen

Eine Webseite ist ein Dokument, das auf einem Webserver liegt und über ein Netzwerk von einem Webbrowser abgerufen werden kann. Für die Beschreibung von Webseiten wird die Seitenbeschreibungssprache HTML (*Hypertext Markup Language*) verwendet. Ein HTML-Dokument ist eine spezielle Art von Textdokument, in dem der Inhalt einer Webseite beschrieben wird. HTML-Elemente werden durch Tag-Paare markiert. Zum Beispiel zeichnet das Element `p` einen Abschnitt (*paragraph*) aus.

```
<html>
  <head>
    <title>Titel der Seite</title>
  </head>
  <body>
    <h1>Eine Überschrift</h1>
    <p>Ein Abschnitt</p>
  </body>
</html>
```

HTML beschreibt nur die Struktur einer HTML-Seite; wie sie dargestellt wird hängt vom Webbrowser ab. Es ist jedoch möglich, über *Cascading Style Sheets*[1] (CSS) die Darstellung von Webseiten zu ändern. Dazu werden externe CSS-Dateien mit Darstellungsinformationen eingebunden; alternativ wird direkt im HTML-Element das Attribut `style` gesetzt.

```
<p style="color:blue;">Ein blauer Abschnitt</p>
```

[1] `http://www.w3.org/TR/CSS21/`

Für dynamische Webseiten gibt es in HTML Formulare. Ein Formular besteht ähnlich wie eine GUI aus Widgets wie Eingabefeldern und Buttons. Der Inhalt dieser Elemente kann vom Benutzer geändert und durch einen Klick auf einen Button zum Webserver gesendet werden.

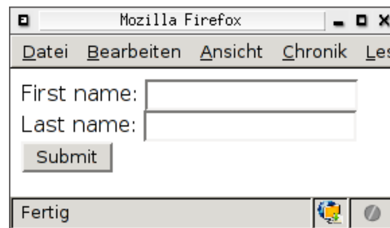

Abbildung 4.1: HTML-Formular

```
<form action="echo.cgi" method="post">
  First name: <input type="text" name="firstname"><br>
  Last name:  <input type="text" name="lastname" ><br>
  <input type="submit" value="Submit">
</form>
```

Listing 4.1: HTML-Quelltext des Formulars aus Abbildung 4.1

Das Formular in Listing 4.1 besteht aus zwei Eingabefeldern und einem Button. Die Widgets werden mit dem HTML-Element `input` beschrieben. Erst durch das Attribut `type` wird die Art des Widgets festgelegt. Weitere Attribute von Formularelementen sind `name`, die ID des Widgets, und `value`, für den Wert des Widgets. Beim Button ist dies die Beschriftung, beim Textfeld der Initialwert. Der Standardwert von `value` ist bei Textfeldern auf `""` gesetzt. Der Wert des `value`-Attributs ändert sich durch die Eingaben des Benutzers. Wenn der Benutzer schließlich auf eine Schaltfläche klickt, werden die Name/Value-Paare des Formulars an den Webserver gesendet. Dort wird das Programm, das im `action`-Attribut des `form` Elementes angegeben ist ausgeführt und kann auf die Formulardaten zugreifen.

Hypertext Transfer Protokol

Die Kommunikation zwischen Client und Webserver wird im *Hypertext Transfer Protokol*[2] (HTTP) definiert. HTTP ist ein zustandloses Protokoll, was bedeutet, dass die Verbindung zwischen Client und Server nach der Antwort auf die Anfrage nicht bestehen bleibt. HTTP kennt unter anderem die Anfragetypen

`GET file` – fordert die Datei `file` auf dem Webserver an.

[2] `http://tools.ietf.org/html/rfc2616`

`POST file cont` – fordert auch die Datei `file` an, wobei zusätzlich der Text `cont` übermittelt wird.

Die Antwort des Webservers beginnt mit einem Header, auf den nach einer Leerzeile die Nutzdaten folgen. Der Header enthält Informationen über die Nutzdaten. Er beginnt mit einer Statuszeile (z.B. 200 für erfolgreich), gefolgt von Variablen wie zum Beispiel `Content-Length`, was die Länge des Inhaltes in Bytes angibt. So kann festgestellt werden, wann die Übertragung abgeschlossen ist. Die Variablen `Content-Type` gibt an, von welchem Medientyp die gelieferten Daten sind. Diese Information wird in der Notation Typ/Untertyp angegeben, beispielsweise `text/html`, `text/plain`.

Fordert der Browser die Webseite `http://www.example.net/index.html` an, so wird zunächst eine TCP/IP-Verbindung zum HTTP-Server `www.example.net` aufgebaut und eine HTTP-Anfrage der Art

```
GET /index.html HTTP/1.1
Host: www.example.net
```

zum Server gesendet. Die Antwort enthält neben dem Header mit Statusinformationen als Body den HTML-Quelltext der Webseite. Dieser wird dann schließlich vom Browser angezeigt.

```
HTTP/1.1 200 OK
Content-Length: 25310
Content-Type: text/html

(Inhalt von index.html)
```

Common Gateway Interface

Webserver können nicht nur statische Dateien ausliefern, sondern auch, abhängig von der Anfrage, Webseiten durch Programme generieren. Diese Programme können mit dem Umgebungsmodell vom *Common Gateway Interface* (CGI) auf die Formulardaten, die der Benutzer eingegeben hat, zugreifen.

Im `action`-Attribut des HTML-Formulars kann ein solches CGI Programm angegeben werden. Sendet der Benutzer durch das Klicken eines „Submit“-Buttons seine Eingaben an den Server, wird dieses Programm ausgeführt. Der Webserver stellt dem Programm eine eigene Laufzeitumgebung zur Verfügung. In dieser kann das CGI-Programm die an den Server gesendeten Benutzereingaben aus Umgebungsvariablen abfragen und den POST-Inhalt der Benutzeranfrage vom `stdin` Kanal einlesen. Die Antwort, also das Ergebnis der Berechnung, gibt der Server auf dem `stdout` Kanal aus. Diese Ausgabe ist die Antwort des Webservers auf die Benutzeranfrage.

Listing 4.2 zeigt ein Curry-Programm, das als CGI-Skript verwendet werden kann. Der POST-Inhalt der Anfrage vom Client wird von der Funktion `getCgiContent` eingelesen, dazu wird die Umgebungsvariable `CONTENT_LENGTH` abgefragt und entsprechend viele Zeichen vom `stdin` Kanal mit der Funktion `getNChar` eingelesen.

```
main :: IO ()
main = do
  content <- getCgiContent
  putStrLn "Content-Type: text/plain"
  putStrLn ""
  putStrLn $ "Benutzereingaben: " ++ content

getCgiContent :: IO String
getCgiContent = do
  clen <- getEnviron "CONTENT_LENGTH"
  cont <- getNChar (maybe 0 fst (readNat clen))
  return cont
```

Listing 4.2: `echo.curry`

Das aus Listing 4.2 erzeugte CGI-Programm (`echo.cgi`) gibt die POST-Daten der Anfrage wieder als Antwort an den Client zurück.

```
<form action="echo.cgi" method="POST">
  First name:<input type="text" name="firstname"><br>
  Last name: <input type="text" name="lastname"><br>
  <input type="submit" value="Submit">
</form>
```

Listing 4.3: HTML-Formular aus `echo.html`

Wenn der Benutzer die HTML-Seite `echo.html` anfordert, wird vom Browser ein Formular angezeigt, in dem er Daten wie in Abbildung 4.2 eingeben kann.

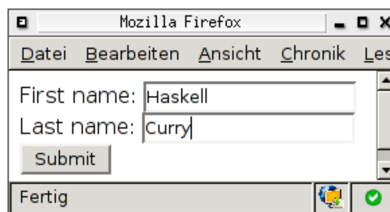

Abbildung 4.2: Darstellung des Browsers von Listing 4.3

Drückt er dann den „Submit“-Button, sendet der Browser eine Anfrage der Form:

```
POST /echo.cgi HTTP/1.1
Host: www.example.net
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 32

firstname=Haskell&lastname=Curry
```

an den Webserver. Durch diese Anfrage wird das Skript `echo.cgi`, aus Listing 4.2 generiert, auf dem Server ausgeführt. Das Ergebnis dieses Skripts wird als Antwort an den Client gesendet:

```
Content-Type: text/plain

Benutzereingaben: firstname=Haskell&lastname=Curry
```

4.2 Die HTML-Bibliothek

Benutzung

Die HTML-Bibliothek [4] der PAKCS-Distribution bietet ein hohes Abstraktionsniveau für die Entwicklung von Webanwendungen mit CGI.

- Die CGI-Interaktion wird von der HTML-Bibliothek intern abgewickelt, ist also für den Anwender versteckt.
- Die Formulardaten können auf dem Server mit einem einfachen Umgebungsmodell ausgelesen werden.
- Für die Bezeichnung der Formularfelder können nur freie Variablen, anstelle von Strings, benutzt werden. Damit werden Tippfehler bereits beim Compilieren erkannt. Außerdem werden so Namenskonflikte bei der Komposition mehrerer Webanwendungen vermieden.
- In der Beschreibung einer Webanwendung mit der HTML-Bibliothek wird in der konkreten Webanwendung jedem „Submit“-Button des Formulars eine Funktion auf dem Server zugeordnet. Diese Funktion kann eine neue HTML-Seite generieren.
- Das Session-Management wird intern von der HTML-Bibliothek abgewickelt.
- Die Sequenz der Interaktion mit dem Webserver kann in einem einzigen Dokument beschrieben werden.
- Eine HTML-Seite wird durch eine Datenstruktur repräsentiert, dadurch werden immer korrekt verschachtelte HTML-Dokumente erzeugt.

Um eine HTML-Seite zu definieren, gibt es in der Bibliothek Datenkonstruktoren des Typs `HtmlExp`.

```
data HtmlExp = HtmlText String
             | HtmlStruct String [(String,String)] [HtmlExp]
             | HtmlCref HtmlExp CgiRef
             | HtmlEvent HtmlExp HtmlHandler
```

„`HtmlText str`“ ist ein Textknoten mit dem Text `str`.

„`HtmlStruct tagname attrs hexps`“ repräsentiert ein HTML-Element mit dem Tagnamen `tag`, den Attributen `attrs` und einer Liste von Kindelementen.

„`HtmlCref hexp ref`“ weist einem Wert `hexp` vom Typ `HtmlExp` eine Referenz `ref` zu. Mit dieser Referenz kann ein Handler auf den Inhalt von `hexp` zugreifen.

„`HtmlEvent hexp handler`“ weist einer `hexp` einen `handler` zu. Dieser `handler` ist eine Funktion, die auf dem Server ausgeführt wird, wenn der Benutzer durch das Klicken eines Buttons die Formulardaten abgeschickt hat.

Eine vollständige Antwortseite wird mit dem Datentypen `HtmlForm` ausgedrückt.

```
data HtmlForm = HtmlForm String [FormParam] [HtmlExp]
              | HtmlAnswer String String
```

„`HtmlForm title params hexps`“ ist eine HTML-Seite mit Titel, einer Liste von Parametern für den Header der HTML-Seite und dem Inhalt als Liste von Werten mit Typ `HtmlExp`.

„`HtmlAnswer ctype cont`“ für eine Antwortseite beliebigen Typs `ctype`, mit dem Inhalt `cont` als String.

Üblicherweise werden in Skriptsprachen Eingabefelder mit Strings referenziert. In der HTML-Bibliothek werden jedoch freie Variablen verwendet. Dies hat einerseits den Vorteil, dass bereits beim Compilieren Schreibfehler in Namen von Referenzen erkannt werden und andererseits, dass das Kombinieren verschiedener Bestandteile einer Webanwendung zu einer neuen sehr leicht fällt, denn die HTML-Bibliothek instanziiert die freien Variablen global eindeutig und stellt damit sicher, dass nicht zwei verschiedene Variablen in der Webanwendung den gleichen Namen haben. Der Datentyp für die Referenzen

```
data CgiRef = CgiRef String
```

wird wie bei der GUI-Bibliothek von der HTML-Bibliothek nicht exportiert. Der Benutzer der Bibliothek wird dadurch gezwungen freie Variablen zu benutzen. Damit implementiert die HTML-Bibliothek das funktional logische Entwurfsmuster *Opaque Type* [1].

Um ein Formular an den Webserver zu senden werden „Submit“-Buttons verwendet. In der HTML-Bibliothek gibt es ein Eventhandling-Modell, das jeden „Submit“-Button

über eine ID mit einem Event-Handler auf dem Webserver verbindet. Ein Event-Handler ist eine IO-Aktion, die ein neues HTML-Formular (`HtmlForm`) liefert. Einem Event-Handler wird immer eine Umgebung übergeben, über die in der IO-Aktion auf die Eingaben des Formulars zugegriffen werden kann. Diese Umgebung ist eine Funktion vom Typ `CgiEnv`, die die Referenz eines Widgets auf dessen Wert abbildet.

```
type CgiEnv = CgiRef -> String
type HtmlHandler = CgiEnv -> IO HtmlForm
```

Die HTML-Bibliothek definiert einige Abkürzungen für Werte vom Typ `HtmlExp`. Zum Beispiel „`button str handler`" für einen Button mit Beschriftung und Handlerfunktion, „`htxt str`" für einen Textknoten und „`textfield ref str`" für ein Textfeld mit dem Initialwert `str` und der Referenz `ref`.

```
htxt      :: String -> HtmlExp
button    :: String -> HtmlHandler -> HtmlExp
textfield :: CgiRef -> String -> HtmlExp
```

Auch für den Datentyp `HtmlForm` gibt es Abkürzungen wie

```
form title hexps = HtmlForm title [] hexps
```

Beispiel

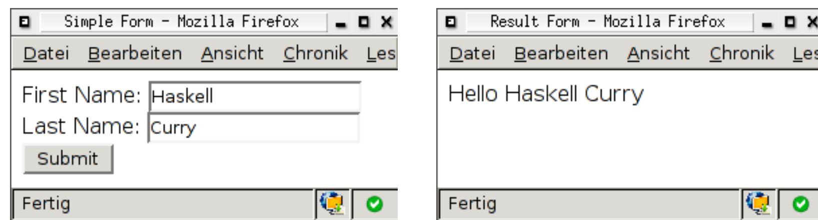

Abbildung 4.3: Die aus Listing 4.4 erzeugte Webanwendung vor und nach dem „Submit".

In Listing 4.4 wird eine Webanwendung mit der HTML-Bibliothek beschrieben.

```
import HTML

main = return $ form "Simple Form"
  [htxt "First Name: ", textfield firstname "", breakline,
   htxt "Last Name: ",  textfield lastname "",  breakline,
   button "Submit" handler]
  where
    firstname, lastname free

    handler env = return $ HtmlAnswer "text/plain"
```

```
("Hello " ++ env firstname ++ " " ++ env lastname)
```

Listing 4.4: Beispiel HTML Bibliothek

Sowohl das Startformular (`main`), als auch alle Event-Handler haben den Ergebnistyp „`IO HtmlForm`“. In dem Startformular wird ein Button definiert, für den als Event-Handler die Funktion `handler` vom Typ „`CgiEnv -> IO HtmlForm`“ angegeben ist. `CgiEnv` ist der Typ einer Funktion, die, angewendet auf eine Referenz, dessen Inhalt liefert. In Listing 4.4 liefert „`env firstname`“ den Inhalt, den der Benutzer in des Textfeld mit der Referenz `firstname` eingetragen hat.

Implementierung

Die HTML-Bibliothek instanziiert die freien Variablen für die Referenzen der Eingabefelder und verwaltet die Event-Handler. Jeder Handler kann auf alle bisherigen Benutzereingaben der Sitzung zugreifen.

- Die freien Variablen werden mit Strings der Form `"EDIT_..."` instanziiert. Diese Strings werden zu den `name`-Attributen der Eingabefelder. Aus einem Widget

  ```
  textfield ref "0"
  ```

  wird im Quelltext ein Eingabefeld wie

  ```
  <input type="text" name="EDIT_1" value="0">
  ```

- Zur Verwaltung der Event-Handler wird dem `name`-Attribut des „Submit“-Buttons im HTML-Quelltext eine ID zugewiesen. Durch diese ID wird der Button mit seiner Handlerfunktion auf dem Webserver verknüpft. Aus

  ```
  button "Absenden" handler
  ```

  wird im HTML-Quelltext dann ein Formularbutton der Form

  ```
  <input type="submit" name="EVENT_0" value="Absenden">
  ```

  wobei auf dem Server mit der ID `"EVENT_0"` auf den entsprechenden Event-Handler `handler` zugegriffen werden kann.

Eine mit Hilfe der HTML-Bibliothek beschriebene Webanwendung läuft folgendermaßen ab:

1. Der Benutzer drückt auf einen „Submit“-Button. Dadurch werden die ID des Buttons und alle anderen Formulardaten als Name/Value-Paare an ein CGI-Programm auf dem Webserver gesendet.

2. Auf dem Webserver werden die Formulardaten ausgewertet und die entsprechende Handlerfunktion anhand der gesendeten Event-ID herausgesucht. Die Handlerfunktion wird ausgeführt und liefert eine neue HTML-Seite.

Abbildung 4.4 stellt die Programme dar, die an einer Webanwendung mit der HTML-Bibliothek beteiligt sind. Clientseitig wird nur ein Browser benötigt. Wenn man eine Webanwendung mit der HTML-Bibliothek in der Datei `app.curry` beschreibt, wird diese mit dem Makeskript `makecurrycgi` in eine konkrete Webanwendung übersetzt. Dabei entstehen die ausführbaren Programme `app.cgi` und `app.cgi.server`.

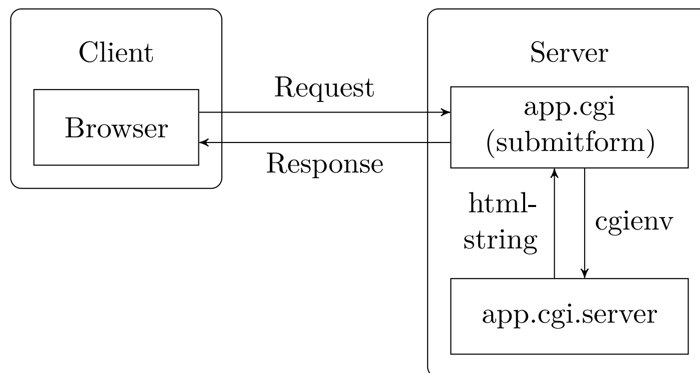

Abbildung 4.4: Skizze der Kommunikation eines Client mit dem Webserver

`app.cgi` ist das CGI-Skript, das vom Webbrowser angefragt und dadurch ausgeführt wird. Dieses Skript führt das Curry-Programm `submitform` aus. Das Programm `submitform` ist die Schnittstelle zwischen dem Client und dem Hauptprogramm auf dem Server. Die POST-Daten, die als String

$$name_1 = value_1\texttt{\&}name_2 = value_2\texttt{\&..\&}name_n = value_n$$

auf dem Server ankommen, werden mit der Funktion `parseCgiEnv` in eine Liste von Name/Value-Paaren vom Typ `[(String,String)]` zerlegt. Diese Liste wird an das Hauptprogramm der Anwendung `app.cgi.server` weitergegeben.

`app.cgi.server` ist das Hauptprogramm der Anwendung, das die Event-Handler verwaltet und ausführt und damit neue HTML-Seiten berechnet. Dazu muss es, im Gegensatz zum `app.cgi` Programm, dauerhaft auf dem Server laufen.

Sendet der Benutzer keine Event-ID, fordert er das erste Mal die `app.cgi` Anwendung an. Somit erzeugt das `app.cgi.server` Programm das Initialformular, also das Formular, das der Programmierer als Hauptfunktion (`main`) in der Beschreibung `app.curry` angegeben hat. Für die Event-Handler, die für die Startseite in `app.curry` definiert wurden, werden IDs erzeugt, die die HTML-Button als `name`-Attribut erhalten. Die generierte HTML-Seite bekommt der Client als Antwort zugeschickt. Sie ist die Start-

seite der Webanwendung. Wenn der Benutzer auf dieser Seite einen Button drückt, erkennt das Hauptprogramm auf dem Server anhand der ID (`name`-Attribut des Buttos) welcher Event-Handler ausgeführt werden soll.

Das Programm `app.cgi.server` wartet auf Anfragen. Sendet ein Benutzer Formulardaten, einschließlich einer Event-ID, wird der entsprechende Event-Handler ausgeführt. Mit Hilfe des momentanen Programmzustandes, der auch alte Formulardaten enthält, und den neuen Formulardaten wird die vom Event-Handler beschriebene HTML-Seite berechnet. Wie schon bei der Startseite werden die freien Variablen für die Sitzung global eindeutig instanziiert und die Event-Handler verwaltet.

Ist `handler` die Handlerfunktion, die ausgeführt werden soll, und `cenv` die Umgebung, die aus allen in dieser Sitzung vom Benutzer eingegebenen Daten besteht, dann wird auf dem Server der Handler folgendermaßen ausgeführt:

```
handler (cgiGetValue cenv)
```

`cgiGetValue` wird partiell auf `cenv` angewendet und liefert dadurch eine Funktion vom Typ „`CgiRef -> String`“, mit der jeder Referenz eines Widgets der Inhalt des Widgets zugeordnet wird.

```
cgiGetValue :: [(String,String)] -> CgiRef -> String
cgiGetValue ((n,v):cenv) (CgiRef ref) =
  if (n == ref) then v
              else cgiGetValue cenv (CgiRef ref)
```

Sessions

Oft müssen Benutzereingaben über einige Webseiten hinweg gespeichert werden. Eingabedaten können zum Beispiel direkt auf dem Client über Cookies oder unsichtbare Formularfelder (`type="hidden"`) gespeichert werden. Eine weitere Möglichkeit besteht darin, die Daten auf dem Webserver zu speichern. Dazu muss der Webserver einen bestimmten Benutzer identifizieren können.

Die HTML-Bibliothek speichert die Eingabedaten serverseitig ab, wobei der jeweilige Benutzer mit Hilfe der Event-ID der Buttons erkannt wird. Im Hauptprogramm auf dem Server steht unter einer Event-ID die alte Umgebung (`oldenv`) und ein Event-Handler. Die Umgebung `oldenv` enthält alle Benutzereingabedaten der Sitzung. Wenn ein Event ausgelöst wird, wird die entsprechende ID zum Server geschickt. Dort wird der mit der ID verknüpfte Handler ausgeführt und liefert ein neues HTML-Formular. Enthält dieses Formular wieder neue Handler, werden sie unter neuen Event-IDs jeweils mit der bisherigen und der aktuellen Umgebung, also „`newenv ++ oldenv`“, in den Serverzustand eingefügt. Damit stehen beim Ausführen eines Handlers nicht nur die

aktuellen Daten, die der Anwender zum Server geschickt hat, zur Verfügung, sondern auch alle in diesem Programmlauf bisher eingegebenen alten Daten.

Beispiel zur Sessionverwaltung

Listing 4.5 beschreibt eine Webanwendung, in der der Benutzer zunächst 3 Formulare ausfüllt und schließlich als Ergebnis alle seine Eingaben geliefert bekommt. Zur Übersichtlichkeit wird eine Funktion `myForm` definiert, die ein Formular mit einem Textfeld und einem Button darstellt. Abbildung 4.5 zeigt einen möglichen Sitzungsverlauf.

```
import HTML

main :: IO HtmlForm
main = myForm val1 handler1
  where
    val1, val2, val3 free

    handler1 env = myForm val2 handler2
    handler2 env = myForm val3 handler3
    handler3 env = return $ form "Result"
      [htxt $ (env val1)++(env val2)++(env val3)]

myForm :: CgiRef -> (CgiEnv -> IO HtmlForm) -> IO HtmlForm
myForm val handler = return $ form "Form"
  [textfield val "", button "Submit" handler]
```

Listing 4.5: `session.curry`

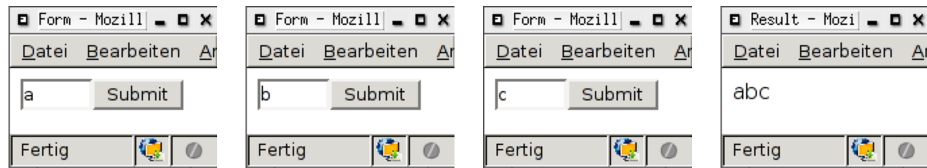

Abbildung 4.5: Möglicher Ablauf der Webanwendung aus Listing 4.5

Das Makeskript `makecurrycgi` erzeugt aus Listing 4.5 die Dateien `session.cgi` und `session.cgi.server`.

1. Der Benutzer sendet eine GET-Anfrage an das `session.cgi`-Skript. Dieses leitet die Eingabedaten weiter an das Hauptprogramm `session.cgi.server`, welches das `main`-Formular generiert und die Handler verwaltet. In diesem Fall wird `handler1` und die aktuelle Umgebung unter der ID `"EVENT_0"` in den Zustand des Hauptprogramms eingefügt. Im Hauptprogramm wird der Handler als Tupel der Art `("EVENT_0",[],handler1)` verwaltet.

2. Nun gibt der Benutzer in das neue Formular den Text `"a"` ein und schickt die Eingabedaten zum Server. Die Anwendung auf dem Webserver erhält durch die Anfrage die Umgebung `[("EDIT_1","a"),("EVENT_0","")]`. Also wird `"EVENT_0"` herausgesucht und der entsprechende Handler, `handler1`, ausgeführt.

   Der Handler `handler1` generiert als Ergebnis wieder ein HTML-Formular, dessen Handler im Zustand des Hauptprogramms eingefügt werden, also wird das Tupel `( "EVENT_1", [] ++ [("EDIT_1","a")],handler2)` aufgenommen und an den Client eine neue Seite ausgeliefert.

3. Jetzt gibt der Benutzer ein `"b"` ein und schickt seine Eingabedaten ab. Der Webserver erhält die Umgebung `[("EDIT_2","b"),("EVENT_1","")]`. Also sucht er den Event-Handler zur ID `"EVENT_1"`, also `handler2` und führt ihn aus. Das Ergebnis von `handler2` enthält wieder einen Event-Handler, `handler3`, welcher wiederum mit der vollständigen Umgebung und einer neuen ID in den Zustand des Hauptprogramms eingefügt wird.

   `("EVENT_2", [("EDIT_1","a"),("EDIT_2","b")],handler3)`

4. Wenn der Benutzer nun `"c"` in das neue Formular eingibt und abschickt, erhält das Hauptprogramm die Umgebung `[("EDIT_3","c"),("EVENT_2","")]`, also wird `handler3` ausgeführt. Dieser hat die Umgebung `[("EDIT_1","a"),("EDIT_2","b") ,("EDIT_3","c")]` zur Verfügung. Er gibt alle bisherigen Eingaben aus und hat keinen weiteren Handler.

Probleme

Identische Referenzen

Da die Referenzen erst zur Laufzeit instanziiert werden, meldet der Compiler keinen Fehler, wenn Widgets als Referenz die gleiche freie Variable haben. Bei Listing 4.6 kommt es auf dem Server beim Instanziieren der freien Variablen also zu einem Laufzeitfehler. Der Client bekommt eine Fehlerseite geliefert, auf der auf einen Laufzeitfehler im Programm hingewiesen wird.

```
main = return $ form "Test"
  [textfield ref "0", textfield ref "0", button "Submit" cmd]
  where
    ref free
    cmd env = return $ form "Result" [htxt (env ref)]
```

Listing 4.6: Zwei Textfelder mit gleicher Referenz

Fehler dieser Art sind für den Programmierer schwer zu finden, jedoch ist es möglich bei dem Skript `makecurrycgi` eine Debug-Option zu aktivieren, wodurch aussagekräftigere Hinweise auf den Fehler angezeigt werden wie

```
Failure due to irreducible expression: ('1' =:= '2')
```

4.3 JavaScript

Grundlagen

Alle modernen Webbrowser besitzen einen Interpreter für die imperative Programmiersprache JavaScript. Mit JavaScript können die Inhalte und das Aussehen von Webseiten verändert werden, ohne dass ein Webserver angefragt wird und die Seite neu geladen werden muss.

Datentypen

In JavaScript gibt es die Datentypen `number`, `string`, `boolean` und `object`, außerdem die trivialen Datentypen `undefined` und `null`.

Die Parameter von Funktionen werden bei den primitiven Datentypen *call-by-value* übergeben und Objekte werden per Referenz übergeben. Auch Arrays und Funktionen sind in JavaScript Objekte. Wird in einer Funktion eine Variable implizit definiert, d.h. ohne das Schlüsselwort `var`, so ist sie global. Auch Funktionen höherer Ordnung und anonyme Funktionen können definiert werden.

JavaScript in Webseiten

Mit JavaScript sind ereignisgesteuerte Berechnungen möglich. Ein Webbrowser erzeugt bei Benutzeraktivität Events (Ereignisse), an die man JavaScript-Code anhängen kann. Wenn ein Event auftritt, wird dieser JavaScript-Code ausgeführt. Events können als Attribut von HTML-Elementen angegeben werden. Einige wichtige Events sind in Tabelle 4.1 dargestellt.

Event-Handler	Ereignis
`onclick`	wenn ein Element angeklickt wird
`onfocus`	beim Aktivieren eines Elementes
`onblur`	beim Verlassen eines Elementes
`onsubmit`	das Formular wird versendet
`onload`	nachdem die HTML-Datei geladen wurde

Tabelle 4.1: JavaScript Events

Document Object Model (DOM)

Das *Dokument Objekt Model*[3] (DOM) beschreibt wie alle Elemente einer HTML-Seite in Beziehung zur obersten Struktur, dem `document`, stehen. Mit DOM können Elemente abgefragt und verändert werden. Die DOM-Struktur einer HTML-Seite kann als ein Baum, wie in Abbildung 4.6, dargestellt werden.

```
<html>
  <head>
    <title>Document title</title>
  </head>
  <body>
    <h1>An important heading</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

Listing 4.7: HTML-Dokument

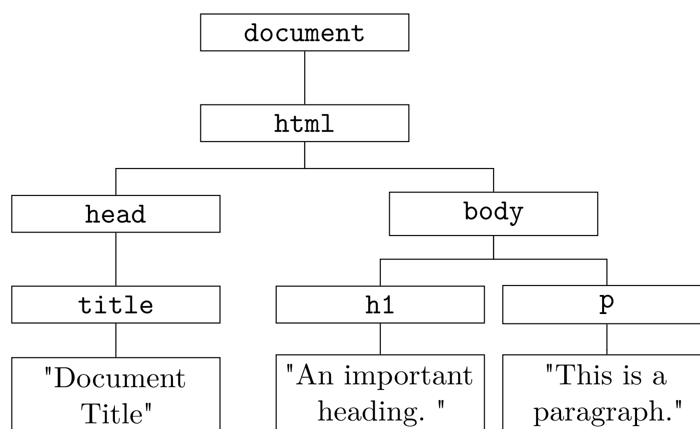

Abbildung 4.6: DOM-Baum von Listing 4.7

An der Wurzel eines DOM-Baumes steht der Dokumentknoten (`document`). Dieser enthält Elementknoten, die wiederum Elementknoten und Textknoten enthalten können. Außerdem hat jedes HTML-Element Attributknoten. Der Inhalt von Knoten kann mit JavaScript abgefragt und verändert werden. Um den DOM-Baum nach bestimmten Knoten zu durchsuchen, besitzt jeder Knoten Attribute wie `parentNode` und `childNodes`. Wesentlich einfacher kann man HTML-Elemente jedoch finden, wenn man JavaScript-Funktionen wie `getElementById()` und `getElementsByTagName()` benutzt.

[3] `http://www.w3.org/TR/DOM-Level-2-HTML/`

```
var elem  = document.getElementById("EDIT_1");
var elems = document.getElementsByTagName("input");
```

Listing 4.8: Zugriff auf HTML-Elemente per JavaScript

In Listing 4.8 wird `elem` das HTML-Element zugewiesen, dessen `id`-Attribut den Wert `"EDIT_1"` hat und `elems` ist nach dieser Zuweisung ein Array von HTML-Elementen, die den Tag Namen `input` haben. Wenn `elem` ein Formularelement ist, kann sein Inhalt mit dem Attribut `value` abgefragt werden.

Mit JavaScript kann der DOM-Baum geändert werden. Dazu gibt es Funktionen wie `appendChild()`, `removeChild()` und `replaceChild()`. Um neue Element- und Textknoten zu erstellen, gibt es die Funktionen `createElement()` und `createTextNode()`. In Listing 4.9 wird ein neuer Abschnitt erzeugt und diesem ein Textknoten zugewiesen. Dann wird der neue Abschnitt an den `body`-Knoten des DOM-Baumes gehängt.

```
var elem = document.createElement("p");
var text = document.createTextNode("new paragraph");

elem.appendChild(text);
document.body.appendChild(elem);
```

Listing 4.9: Änderung des DOM-Baumes

Eine Sonderrolle spielt die Eigenschaft `innerHTML` von HTML-Elementen. Diese Eigenschaft gibt es nicht in der W3C-DOM-Spezifikation, sie wird aber von allen modernen Browsern unterstützt. Mit `innerHTML` kann der HTML-Inhalt eines Elementes direkt abgefragt und verändert werden.

Ajax

In herkömmlichen Webanwendungen werden durch einen Klick auf einen „Submit“-Button die Formulardaten einer HTML-Seite an einen Webserver geschickt und dieser generiert eine komplett neue Seite, die er an den Client ausliefert.

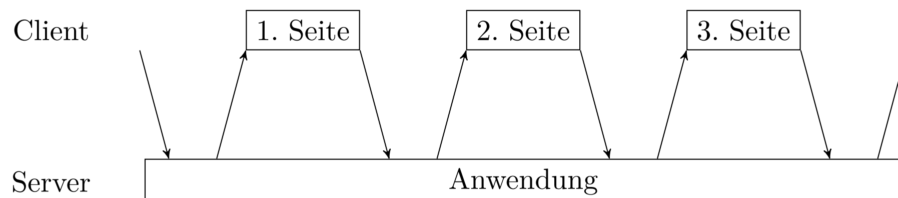

Abbildung 4.7: Ablauf einer herkömmlichen Webanwendung.

Mit dem JavaScript-Objekt `XMLHttpRequest`[4] ist es jedoch möglich, Daten von einem Server abzufragen, ohne eine Seite neu zu laden. Anhand der Antwort vom Webserver kann dann die DOM-Struktur der Seite geändert werden. Diese Technologie wurde durch den Begriff Ajax[5] (*Asynchronous JavaScript and XML*) bekannt.

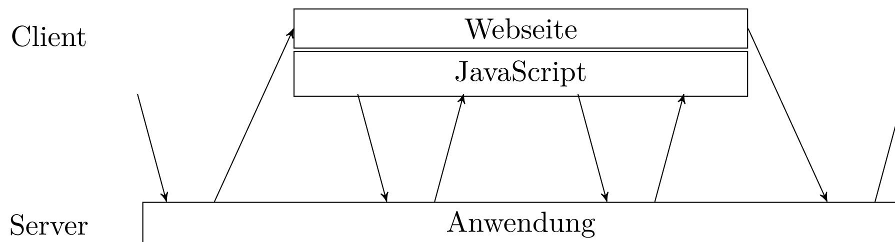

Abbildung 4.8: Ablauf einer Webanwendung mit Ajax

Unter asynchron versteht man, dass der Browser nicht auf die Antwort wartet, sondern auch andere JavaScript- und Benutzeraktionen erlaubt. Im Gegensatz dazu sind bei synchronen Anfragen keine anderen Aktionen nebenläufig möglich. Bei zeitaufwändigen Anfragen hat der Benutzer bei synchronen Anfragen den Eindruck, dass der Browser einfriert.

Das XMLHttpRequest-Objekt

Die wichtigsten Methoden und Attribute des `XMLHttpRequest`-Objektes sind:

`open(method,url,true)` – `method` ist eine Anfragemethode wie GET oder POST. Als `url` wird eine Datei übergeben, die auf dem Server geöffnet werden soll. Der letzte Parameter gibt an, ob die Anfrage asynchron erfolgen soll. Hier wird im Allgemeinen `true` verwendet.

`onreadystatechange` – Dieser Eigenschaft wird der Event-Handler zugewiesen. Der Event-Handler ist die JavaScript-Funktion, die ausgeführt wird, wenn die Daten vom Server zurückkommen

`send(data)` – Sendet die Anfrage an den Server. Bei der Anfragemethode POST wird als Parameter der POST-Inhalt übergeben, sonst der Wert `null`.

`readyState` – Wenn die Antwort des Webservers vollständig empfangen wurde hat diese Eigenschaft den Wert 4.

[4] http://www.w3.org/TR/XMLHttpRequest/
[5] http://www.adaptivepath.com/ideas/essays/archives/000385.php

`responseText` – Enthält die Antwort der Anfrage als String. Für XML-Antworten gibt es die Eigenschaft `responseXML` die ein DOM-Dokument liefert.

In Listing 4.10 wird eine JavaScript-Funktion `sendRequest(url,callback,postData)` angegeben. `url` ist die URL des Programms auf dem Server, das angefragt wird, beispielsweise ein CGI-Skript. `postData` sind die Daten, die zum Webserver gesendet werden und `callback(str)` ist die Funktion, die auf die Antwort des Servers angewendet wird. Durch diese Funktion kann zum Beispiel der Inhalt der Webseite abhängig von der Antwort des Servers geändert werden.

```
function sendRequest(url,postData,callback) {
  var req = new XMLHttpRequest();
  req.open('POST',url,true);
  req.setRequestHeader('Content-type',
                       'application/x-www-form-urlencoded');
  req.onreadystatechange = function(){
    if (req.readyState == 4 && req.status == 200) {
      callback(req.responseText);
    }
  };
  req.send(postData);
}
```

Listing 4.10: Anwendung des `XMLHttpRequest`-Objektes

JavaScript Object Notation

Mit der *JavaScript Object Notation* (JSON) können JavaScript-Objekte als String dargestellt werden. JSON wird zum Datenaustausch benutzt und kann in JavaScript direkt mit der `eval()`-Funktion in ein JavaScript-Objekt übersetzt werden. Die JSON-Syntax wird auf der JSON Webseite[6] beschrieben. Die Curry-Bibliothek Json repräsentiert ein Json-Objekt als Datenstruktur.

```
data Json
  = Object [(String, Json)]
  | Array [Json]
  | String String
  | Int Int
  | Bool Bool
  | Null
```

[6] `http://json.org/`

`Object nvs` – für beliebige JavaScript-Objekte. Ein Element der Liste ist jeweils ein Paar aus Eigenschaftsname und Wert der Eigenschaft. Die Ordnung der Liste spielt für JavaScript keine Rolle.

`Array elems` – Arrays werden als Liste dargestellt. Die Elemente können beliebige JSON-Werte sein.

Außerdem gibt es Konstruktoren für Strings (`String str`), ganze Zahlen (`Int n`), Wahrheitswerte (`Bool b`) und den `null`-Wert von JavaScript (`Null`). Mit der Funktion

```
showJson :: Json -> String
```

kann eine Json-Datenstruktur in einen JSON-String konvertiert werden.

Listing 4.11 beschreibt ein JavaScript-Objekt in JSON. Dieses Objekt hat als einzige Eigenschaft `content`. Der Wert von `content` ist ein Array. Beide Elemente dieses Arrays sind Objekte mit den Attributen `id` und `changes`. `id` ist ein String. `changes` wieder eine JSON-Objekt.

```
{"content":[{"id":"FIELD_2","changes":{"value":"48"}},
            {"id":"FIELD_3","changes":{"value":"282"}}]
}

Object [("content",
  Array [Object [("id",String "FIELD_2"),
                 ("changes",Object [("value",String "48")])],
         Object [("id",String "FIELD_3"),
                 ("changes",Object [("value",String "282")])
                    ]])]
```

Listing 4.11: Ein JavaScript-Objekt als JSON-String und als Curry-Datenstruktur mit der Json-Bibliothek

5 Deklarative Konstruktion von allgemeinen Benutzerschnittstellen

Wie in Kapitel 1 beschrieben ist es wünschenswert, aus einer UI-Beschreibung verschiedene konkrete Anwendungstypen generieren zu können. So kann ein Programmierer seine Anwendung nach Belieben als Desktopanwendung oder als Webanwendung zur Verfügung stellen, ohne an der Beschreibung etwas zu ändern. In diesem Kapitel wird eine solche allgemeine UI-Beschreibung vorgestellt. Diese basiert auf den Curry-Bibliotheken GUI und HTML, die in Abschnitt 3.3 und Abschnitt 4.2 vorgestellt wurden. Die neu entstandenen Konzepte wurden in folgenden Bibliotheken umgesetzt:

UI – definiert den Datentypen `Widget`, von dem sich alle Fensterelemente ableiten lassen. Außerdem werden Abkürzungen für die gebräuchlichen Widgets definiert wie `button`, `entry` und `label`.

UI2GUI – wird importiert, wenn aus der Beschreibung mit Hilfe der GUI-Bibliothek eine Tcl/Tk Desktop-Anwendung generiert werden soll.

UI2HTML – wird importiert, wenn aus der Beschreibung mit Hilfe einer leicht erweiterten HTML-Bibliothek Desktop-ähnliche Webanwendungen generiert werden sollen.

GUI2HTML – kann statt der Bibliothek GUI importiert werden, um aus Beschreibungen von Desktopanwendungen, mit den Datenstrukturen aus der GUI-Bibliothek direkt Webanwendungen zu erzeugen.

5.1 Benutzung der UI-Bibliothek

Im Gegensatz zur GUI-Bibliothek, in der es für jede Art von Widget einen eigenen Konstruktor gibt, haben alle Fensterelemente in der UI-Bibliothek den gleichen Konstruktor `Widget`:

```
data Widget r e =  Widget WidgetKind
                          (Maybe String)
                          (Maybe (Ref r))
                          [Handler r e]
```

```
                    [StyleClass]
                    [Widget r e]
```

Durch diesen Datentyp wird die Funktionalität und Darstellung eines Widgets festgelegt. Der Datentyp `Widget` ist sehr allgemein gehalten, da die einzelnen Widgets grundsätzlich viele Gemeinsamkeiten haben. Anders als in der GUI-Bibliothek werden Beschriftung, Referenz und Handler direkt als Komponente im Konstruktor angegeben und nicht als Liste von `ConfItem`. Bei einem Widget der Form

```
Widget kind mbcont mbref handlers styles childs
```

haben die Paramter die Bedeutung:

`kind` – Die Art des Widgets gibt an wie es konkret dargestellt werden soll, zum Beispiel als Schaltfläche (`Button`), Texteingabefeld (`Entry`) oder Text (`Label`).

`mbcont` – Der Inhalt eines Widgets ist die Beschriftung (bei `Button` und `Label`) oder der Initialwert (bei Textfeldern). Einige Widgets wie Reihen und Spalten haben keinen Inhalt (`Nothing`).

`mbref` – Mit Hilfe der Referenz kann der Inhalt des Widgets abgefragt und gesetzt werden. Die Referenz kann wie bei der GUI- und HTML-Bibliothek nur als freie Variable angegeben werden.

`handlers` – Eine Liste mit Handlern für jeweils verschiedene Eventtypen. Ein Handler wird erzeugt durch „`Handler eventtyp cmd`“. Die Handlerfunktion `cmd` wird auf dem Server ausgeführt, wenn der Benutzer das Event vom Typ `eventtyp` auslöst.

`styles` – Mit Stilklassen lässt sich die Darstellung von Widgets festlegen. Unter anderem kann die Farbe und die Schriftart geändert werden.

`childs` – Einige Widget-Arten enthalten eine Liste von Kind-Widgets, die sie zu einem neuen Widget kombinieren. Zum Beispiel sind Reihen (`Row`) und Spalten (`Col`) Container für andere Widgets.

Bei der Datentypdefinition „`data Widget r e = ...`“ werden Typvariablen verwendet, da die Referenzen und Handler je nachdem, ob eine Webanwendung oder eine Desktopanwendung erzeugt werden soll, verschiedene Typen haben. Dabei ist `r` der Typ der Referenz und `e` der Typ der Handlerfunktion. In den konkreten UI-Bibliotheken wie UI2HTML und UI2GUI wird ein Datentyp `UIEnv` eingeführt, so dass eine Handlerfunktion den Typ „`UIEnv -> IO()`“ hat. Der Wert vom Typ `UIEnv` ist eine Umgebung. Mit diesem Wert kann der Zustand der Benutzerschnittstelle abgefragt und geändert werden.

Was die Umgebung genau ist, hängt von der konkreten Anwendung ab. Soll eine Tcl/Tk-Anwendung generiert werden, ist die Umgebung ein Wert vom Typ `GuiPort`, also ein Kanal, auf dem das Hauptprogramm mit der Tcl/Tk-Anwendung kommuniziert. Wird eine Webanwendung erstellt, ist die Umgebung eine `IORef`, mit der auf die Eingabedaten des Nutzers zugegriffen werden kann. In dieser `IORef` werden auch die Zustandsänderungen an der graphischen Oberfläche gespeichert, die die IO-Aktionen des jeweiligen Event-Handlers bewirken. Was genau die Umgebung ist, ist für den Anwender der Bibliothek uninteressant; ihm werden vordefinierte Handler bereitgestellt, mit denen er Werte von Widgets abfragen und ändern kann.

```
getValue :: UIRef -> UIEnv -> IO String
setValue :: UIRef -> String -> UIEnv -> IO ()
```

„`getValue ref env`“ liefert den Wert des Widgets mit der Referenz `ref` und „`setValue ref str env`“ setzt den Wert eines Widgets auf `str`.

Um dem Anwender der Bibliothek lästige Arbeit zu ersparen, werden für die einzelnen Widgettypen Wrapper-Funktionen eingeführt, siehe Listing 5.1. Dies hat auch den Vorteil, dass der Datentyp `Widget` intern geändert werden kann, ohne dass sich das Interface ändert.

Zum Beispiel erzeugt der Aufruf „`button cmd str`“ eine Schaltfläche (`Button` als `WidgetKind`) mit einer Beschriftung `str` und einer Handlerfunktion `cmd`, die bei dem `DefaultEvent` des Buttons ausgeführt wird. Jedem `WidgetKind` wird in der Implementierung der Eventtyp `DefaultEvent` zugeordnet. Im Fall `Button` ist es das Event, das ausgelöst wird, wenn der Button gedrückt wird. Ein weiteres wichtiges Widget ist das Texteingabefeld, für das es die Wrapper-Funktion „`entry ref cont`“ gibt. Dieser wird die Referenz `ref` und der Initialwert `cont` übergeben. Außerdem gibt es unsichtbare Widgets wie Spalten und Zeilen. Zum Beispiel ordnet `col` eine Liste von Widgets in einer Spalte an.

```
col ws = Widget Col Nothing Nothing [] [] ws
row ws = Widget Row Nothing Nothing [] [] ws
label str = Widget Label (Just str) Nothing []  [] []
entry ref cont = Widget Entry  (Just cont) (Just ref) [] [] []
button cmd label =
  Widget Button (Just label) Nothing
                [Handler DefaultEvent (Cmd cmd)] [] []
```

Listing 5.1: Wrapper-Funktionen für UI Widgets

Listing 5.2 beschreibt einen interaktiven Zähler. Je nachdem, ob der Benutzer die Bibliothek UI2GUI oder UI2HTML importiert, kann aus der Beschreibung eine Desktopanwendung oder eine Webanwendung generiert werden. Die Beschreibung hat große Ähnlichkeiten mit der Beschreibung der GUI-Bibliothek in Listing 3.4. Auch die Handler und Referenzen werden wie in der GUI-Bibliothek verwendet.

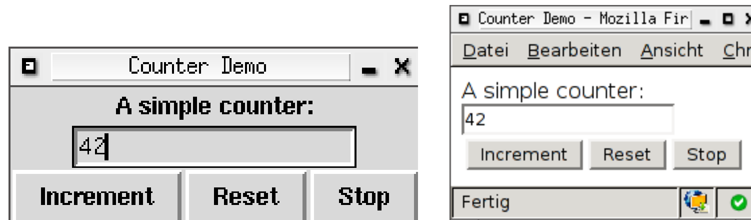

Abbildung 5.1: Interaktiver Zähler ausgeführt als GUI (links) und als Webanwendung (rechts).

```
import UI2GUI -- oder UI2HTML
import Read

counterUI = col [
  label "A simple counter:",
  entry val "0",
  row [button inc "Increment", button reset "Reset",
       button exitUI "Stop"]]
 where
    val free
    reset env = setValue val "0" env
    inc env   = do v <- getValue val env
                   setValue val (show (readInt v + 1)) env

main = runUI "Counter Demo" counterUI
```

Listing 5.2: UI-Spezifikation eines Zählers

5.2 Trennung der Darstellung und Funktionalität

Neben der Anordnung von Widgets, die mit den Widgets `row` und `col` realisiert werden, kann in der UI-Bibliothek auch die Darstellung von Fensterelementen festgelegt und zur Laufzeit geändert werden. Stilinformationen können dem `Widget`-Datentyp als Liste von `StyleClass`-Werten übergeben werden. Eine `StyleClass` besteht wiederum aus einer Liste von Styles.

```
data StyleClass = Class [Style]

data Style = Active Bool | Bg Color | Font FontStyle | ...
```

Ähnlich wie bei CSS für HTML-Dokumente (siehe Kapitel 4) können allgemeine Stile unabhängig von der UI-Beschreibung festgelegt werden. Mit der Funktion `setStyles` kann einem Widget eine Liste von Stilklassen zugewiesen werden. Um dem Anwender lästige Tipparbeit zu ersparen und die UI-Definitionen übersichtlicher zu machen,

werden für einige Widgets Wrapperfunktionen angeboten. Ihr Name endet auf `S` und sie erhalten als ersten Parameter eine Liste von Stilklassen. Alle anderen Parameter sind wie bei den normalen Wrapperfunktionen. Zum Beispiel

```
labelS styles str = label str `setStyles` styles
```

Außerdem wird die Handler-Funktion `changeStyles` angeboten, mit der auch zur Laufzeit die Gestaltung von Widgets geändert werden kann.

```
changeStyles :: UIRef -> [StyleClass] -> UIEnv -> IO ()
```

Listing 5.3 beschreibt eine UI, die aus mehreren Widgets mit Stilinformationen besteht. Gibt der Benutzer eine zu lange Zeichenkette ein, wird eine Fehlermeldung ausgegeben, wenn er den „Check“ drückt.

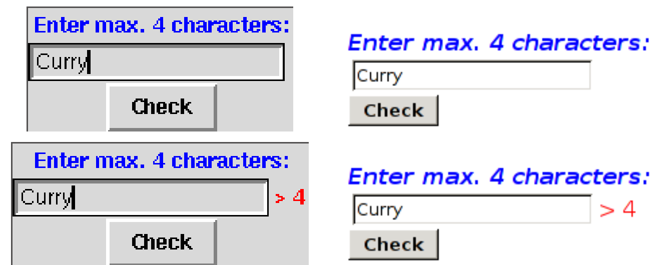

Abbildung 5.2: Die UI-Beschreibung aus Listing 5.3, als Desktopanwendung (links) und Webanwendung (rechts).

```
blueitalic = Class [TextColor Blue, Font Italic]
bold = Class [Font Bold]
red = Class [TextColor Red]
green = Class [TextColor Green]

checkUI = col [
    labelS [blueitalic, bold] "Enter max. 4 characters:",
    row [entry val "", message mref],
    buttonS [bold] check "Check"]
  where
    val, mref free
    check env = do v <- getValue val env
                   if length v <= 4
                     then do changeStyles mref [green] env
                             setValue mref "ok" env
                     else do changeStyles mref [red] env
                             setValue mref "> 4" env
```

Listing 5.3: Anwendungsbeispiel für die Gestaltung von UIs, inklusive Änderung zur Laufzeit

5.3 Änderungen von UIs zur Laufzeit

Neben den bereits vorgestellten IO-Aktionen `getValue` und `setValue` zum Abfragen und Setzen der Werte von Widgets, gibt es in der UI-Bibliothek weitere vordefinierte IO-Aktionen.

```
setDisabled ref bool env
```

(de)aktiviert das Widget mit der Referenz `ref` abhängig vom booleschen Wert (2. Parameter). Deaktivierte Widgets reagieren nicht auf Benutzerinteraktionen.

```
showPopup title widget env
```

zeigt das Widget `widget` in einem neuen Fenster an.

```
setHandler ref eventtype cmd env
```

weist dem Widget mit der Referenz `ref` eine neue Handlerfunktion `cmd` zu, die ausgeführt wird, wenn der Anwender für das Widget das Event `eventtype` auslöst.

```
changeStyles ref styles env
```

Wie in Abschnitt 5.2 beschrieben, weist diese Funktion einem Widget Stilklassen zu.

Listing 5.4 beschreibt wieder eine Benutzerschnittstelle eines interaktiven Zählers. Diesmal gibt es einen „Change“-Button, der aus einem „Increment“-Button einen „Decrement“-Button macht. Nach einem Klick auf den „Change“-Button wird dieser deaktiviert. Abbildung 5.3 zeigt mögliche Zustände der Anwendung.

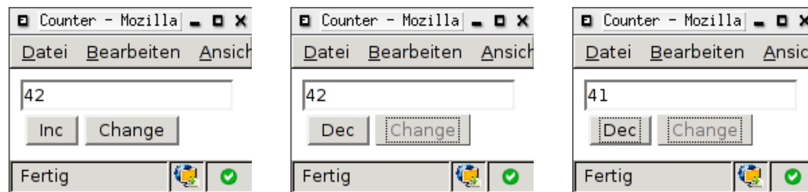

Abbildung 5.3: Nach dem Klick auf den „Change“-Button wird die Beschriftung und der Handler des „Increment“-Buttons geändert.

```
counterUI =
  col [entry val "0",
       row [button cmd "Inc" `setRef` b1,
            button change "Change" `setRef` b2 ]]
  where
    val, b1, b2 free
    cmd = inc 1
    inc n env  = updateValue
                   (\v -> show (readInt v + n)) val env
    change env = do setDisabled b2 True env
```

```
                      setValue b1 "Dec" env
                      setHandler b1 DefaultEvent (inc (-1)) env

main = runUI "Counter" counterUI
```

Listing 5.4: Änderungen des Eventhandlers zur Laufzeit

Da ein Button Widget standardmäßig keine Referenz hat, werden den Buttons in Listing 5.4 mit `setRef` Referenzen zugewiesen.

5.4 Beispiele

Vordefinierte Widgets

In der UI-Beschreibung in Listing 5.5 kommen einige vordefinierte Widgets zum Einsatz. Da es in HTML keine Elemente für Menüs und Schieberegler gibt, werden diese Widgets mit Auswahllisten dargestellt, siehe Abbildung 5.4.

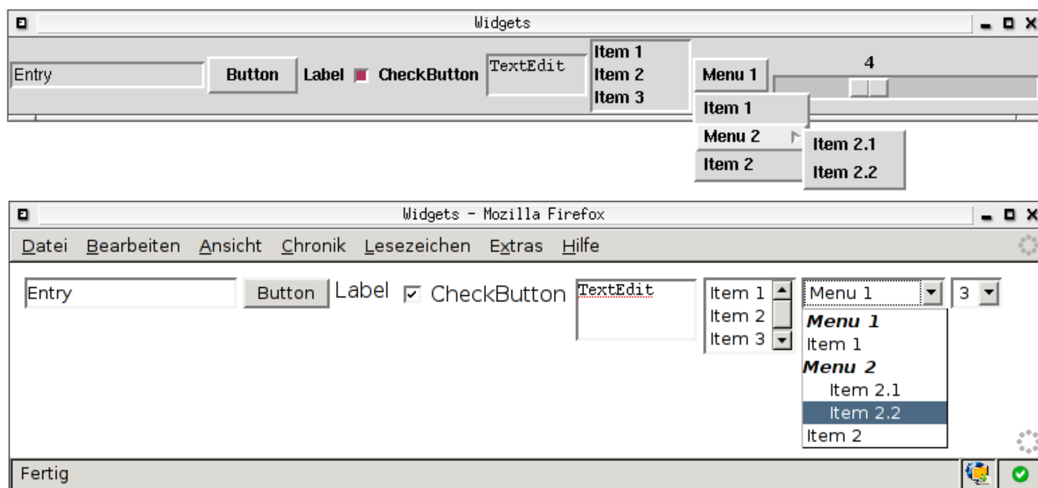

Abbildung 5.4: UI-Widgets als Desktopanwendung (oben) und Webanwendung (unten).

```
ui = row [
    entry r1 "Entry", button cmd "Button", label "Label",
    checkButton r2 cmd "CheckButton" True,
    textEdit r4 "TextEdit" 2 10,
    listBox 3 ["Item 1","Item 2","Item 3","Item 4"] r6 cmd,
    menuBar [
      menu "Menu 1" [
        menuItem cmd "Item 1",
```

```
            menu "Menu 2" [menuItem cmd "Item 2.1",
                           menuItem cmd "Item 2.2" ],
            menuItem cmd "Item 2" ]],
        scale r5 cmd 3 6
      ]

  where r1, r2, r3, r4, r5, r6, r7 free
        cmd = const done
```

Listing 5.5: UI-Widgets

Der CurryBrowser als Webanwendung

Mit der Desktopanwendung CurryBrowser [6] können Curry-Programme analysiert werden. Er kann den Quelltext eines Curry-Programms, einschließlich aller direkt und indirekt importierten Module, anzeigen und analysieren. Die Implementierung des CurryBrowsers benutzt für die Benutzerschnittstelle die GUI-Bibliothek. Daher kann aus der Beschreibung des CurryBrowsers eine Webanwendung generiert werden, indem statt der Bibliothek GUI die Bibliothek GUI2HTML importiert wird.

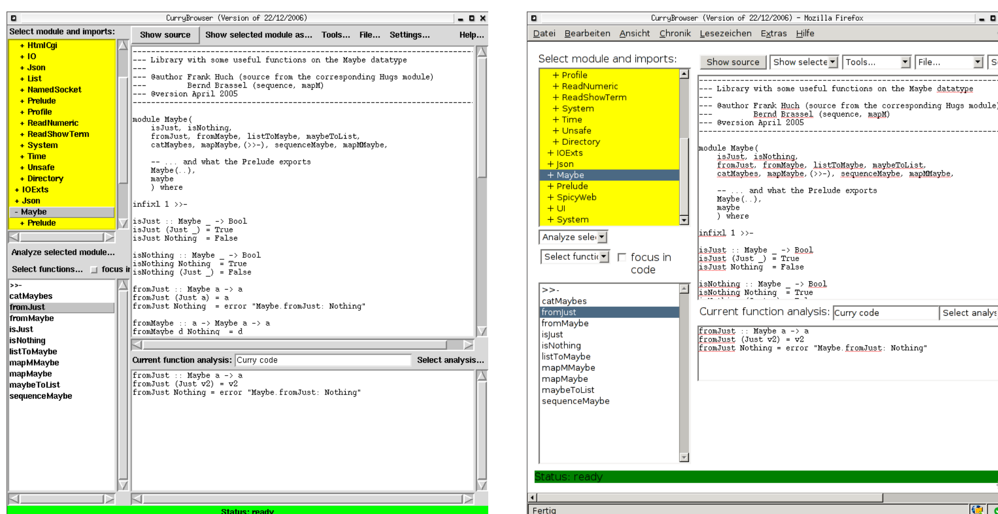

Abbildung 5.5: Der CurryBrowser als Desktopanwendung (links) und als Webanwendung (rechts)

6 Implementierung der UI-Bibliotheken

Die UI-Bibliothek definiert ein allgemeines Interface für die Beschreibung graphischer Benutzerschnittstellen. Die UI2GUI- und UI2HTML-Bibliothek sind konkrete Implementierung dieses Interfaces für Desktopanwendungen und Webanwendungen. Diese Bibliotheken basieren auf die GUI-Bibliothek und die leicht erweiterte HTML-Bibliothek der PAKCS-Distribution, die in Kapitel 3 und Kapitel 4 beschrieben wurden. Abbildung 6.1 zeigt die Abhängigkeiten der Bibliotheken, die in den folgenden Abschnitten beschrieben werden. Die GUI2HTML-Bibliothek kann anstatt der GUI-Bibliothek verwendet werden, um aus GUI-Beschreibungen Webanwendungen zu generieren.

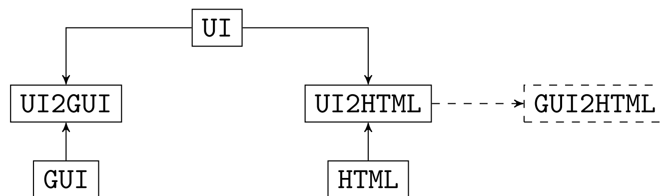

Abbildung 6.1: Abhängigkeiten der UI-Bibliotheken

6.1 Das Interface (UI)

Die Referenzen und Handler der Widgets werden in der UI-Bibliothek als polymorphe Datentypen definiert:

```
data Ref r = Ref r

data Handler act = Handler Event (Command act)
data Event       = DefaultEvent | Click | ...
data Command act = Cmd act

data WidgetKind =  Col | Row | Button | Entry | ...

data Widget =
  Widget
    WidgetKind
    (Maybe String) (Maybe (Ref r)) [Handler act]
```

```
      [StyleClass]
      [Widget r act]
```

Außerdem werden einige Abkürzungen für konkrete Widgets eingeführt, zum Beispiel:

```
entry ref cont = Widget Entry  (Just cont) (Just ref) [] [] []
label str = Widget Label (Just str) Nothing [] [] []
buttonS cmd text =
  Widget Button (Just text) Nothing
         [Handler DefaultEvent (Cmd cmd)] [] []
```

Erst in den Bibliotheken, die konkrete Benutzerschnittstellen erzeugen, wird der genaue Typ der Referenzen und der Umgebung festgelegt. Die UI2HTML- und UI2GUI-Bibliotheken führen jeweils die Typsynonyme `UIRef` und `UIWidget` ein. Dadurch können im Quelltext die gleichen Typen angegeben werden, unabhängig von der Art der zu generierenden Anwendung. In der UI2GUI-Bibliothek sind die Referenzen vom Typ `WidgetRef` und die Handler vom Typ „`GuiPort -> IO ()`“. Der konkrete Typ der Umgebung wird durch den Datentypen `UIEnv` versteckt, so dass der Programmierer für die Umgebung nur den Typ `UIEnv` angeben kann.

```
data UIEnv = ...
type UIRef = Ref WidgetRef
type UIWidget = Widget WidgetRef (UIEnv -> IO())
```

In der UI2HTML-Bibliothek werden entsprechende Typen definiert. Die Referenzen sind wie in der HTML-Bibliothek nun vom Typ `CgiRef`.

```
data UIEnv = ...
type UIRef = Ref CgiRef
type UIWidget = Widget CgiRef (UIEnv -> IO())
```

Somit können für UIs unabhängig von der konkreten Anwendung Typen angegeben werden. In Listing 6.1 wird ein interaktiver Zähler mit Typangaben definiert.

```
counterUI :: UIWidget
counterUI = col [
  label "A simple counter:",
  entry val "0",
  row [button inc "Increment", button reset "Reset",
       button exitUI "Stop"]]
 where
    val :: UIRef
    val free

    reset :: UIEnv -> IO ()
    reset env = setValue val "0" env

    inc :: UIEnv -> IO ()
```

```
inc env    = do v <- getValue val env
               setValue val (show (readInt v + 1)) env
```

Listing 6.1: Interaktiver Zähler mit Typangaben

Ein erheblicher Nachteil dieser Implementierung ist, dass keine Datentypen, sondern nur Typ-Synonyme für Referenzen und Widgets, benutzt werden können. Der Programmierer könnte also auch die konkreten Typen angeben. Dadurch wäre die Allgemeinheit der Beschreibung nicht mehr gegeben. Wenn statt `UIRef` zum Beispiel „`Ref CgiRef`" als Typ angegeben wird, kann aus der Beschreibung keine Desktopanwendung mehr generiert werden, sondern nur noch eine Webanwendung. Ein weiteres Problem bei der Verwendung von Typ-Synonymen ist, dass der Compiler von PAKCS [9] bei Fehlern nicht die Typ-Synonyme, sondern die konkreten Datentypen anzeigt.

Die Probleme mit den Typangeben können gelöst werden, indem man auf die UI-Bibliothek verzichtet und ihren gesamten Quelltext in den Bibliotheken UI2HTML und UI2GUI wiederholt. Dann können Datentypen statt des Typ-Synonyms für `UIRef` und `UIWidget` definiert werden:

```
data UIRef = Ref CgiRef
data UIWidget = Widget CgiRef (UIEnv -> IO())
```

Ein Nachteil dieser Vorgehensweise ist, dass ein Großteil des Quelltextes der UI2GUI- und UI2HTML-Bibliothek identisch wäre, wodurch sich die Wartung der Bibliotheken erheblich erschwert.

6.2 Generierung von Desktopanwendungen (UI2GUI)

Die UI2GUI-Bibliothek konvertiert die UI-Datenstrukturen in Datenstrukturen der GUI-Bibliothek. Dabei bildet die Funktion

```
lrh2confitems :: Maybe String -> Maybe (UIRef)
                 -> [Handler (UIEnv -> IO ())] -> [ConfItem]
```

Beschriftung, Referenz und Handlerliste eines UI-Widgets auf die entsprechenden Werte vom Typ `ConfItem` aus der GUI-Bibliothek ab. Die Referenzen der UI-Widgets, die als freie Variablen vorliegen, werden entsprechend an den Typ der Referenzen der GUI-Bibliothek (`WidgetRef`) angepasst.

In Abschnitt 5.3 werden einige Handlerfunktionen beschrieben. Außer `setHandler` können diese mit der Funktion `setConfig` in GUI-Aktionen konvertiert werden. Durch den Aufruf

```
setConfig ref confitem gp
```

wird dem Widget mit der Referenz `ref` eine neue Konfiguration, in Form eines Wertes vom Typ `ConfItem` zugewiesen.

Der Aufruf „`setHandler ref eventtype cmd env`“ weist einem Widget einen neuen Event-Handler zu. Die Änderung eines Handlers ist in der GUI-Bibliothek nicht mit der Funktion `setConfig` möglich, sondern wird über den Rückgabewert des Handlers, also über einen Wert vom Typ `ReconfigureItem`, gemacht. Die UI2GUI-Bibliothek speichert Werte vom Typ `ReconfigureItem` in einer `IORef` und gibt sie als Ergebnis des Handlers zurück. Listing 6.2 zeigt die Konvertierung einer UI-Handler-Funktion in eine GUI-Handler-Funktion.

```
ui2guicmd cmd gp = do
  stateref <- newIORef (State (gp,[]))
  cmd (UIEnv stateref)
  State (_,reconfigs) <- readIORef stateref
  return reconfigs
```

Listing 6.2: Konvertierung einer UI- in eine GUI-Handler-Funktion

Der Zustand enthält zu Beginn einen Wert vom Typ `GUIPort` und eine leere Liste. Diese Liste kann durch den Aufruf von UI-Handler-Funktionen geändert werden. In ihr werden Werte vom Typ `ReconfigureItem` gespeichert. Die UI-Aktion `setHandler` kann nun wie in Listing 6.3 in eine GUI-Aktion übersetzt werden:

```
setHandler :: UIRef -> Event -> (UIEnv -> IO _) -> UIEnv -> IO
    ()
setHandler (UI.Ref ref) event cmd (UIEnv env) = do
  State (gp,reconfigs) <- readIORef env
  writeIORef env (State (gp,reconfigs ++
      [GUI.WidgetConf ref
         (GUI.Handler (ui2guievent event) (ui2guicmd cmd))]))
```

Listing 6.3: Abbildung der UI-Aktion `setHandler` in einen Wert vom Typ `ReconfigureItem`

Da die Referenzen in der UI2GUI-Bibliothek vom Typ „`Ref WidgetRef`“ sind, können die vordefinierten Aktionen wie `setValue` und `getValue`, problemlos aufeinander abgebildet werden.

```
getValue :: UIRef -> UIEnv -> IO (String)
getValue (UI.Ref ref) (UIEnv env) = do
  State (gp,_) <- readIORef env
  GUI.getValue ref gp
```

6.3 Generierung von Desktop-ähnlichen Webanwendungen (UI2HTML)

Webanwendungen können Desktopanwendungen ersetzen. Jedoch gibt es auf der Seite des Client einige Einschränkungen. Webanwendungen dürfen auf das System des Anwenders nicht zugreifen, daher unterliegen JavaScript-Programme, die vom Browser ausgeführt werden, einigen Beschränkungen. Einige Aktionen sind verboten oder nur auf Nachfrage erlaubt und es kann nur auf die Objekte des Browsers zugegriffen werden. Daher ist es auf dem Client nicht möglich Dateien zu speichern oder zu laden.

Mit Hilfe der UI2HTML-Bibliothek können aus einer UI-Definition Webanwendungen generiert werden. Als Widgets werden HTML-Formularelemente mit JavaScript Event-Handlern benutzt. Ein Widget der Form

```
Widget kind mbcont mbref handlers styles childs
```

wird mit der Funktion `widget2hexp` in den Datentyp `HtmlExp` der HTML-Bibliothek konvertiert. Die Argumente haben folgende Bedeutungen:

`kind` – Die Art eines Widgets bestimmt welches HTML-Element zur Darstellung benutzt werden soll und welche zusätzlichen Attribute es bekommt.

`mbcont` – Die Beschriftung des Widgets wird bei `input`-Tags über des `value`-Attribut gesetzt. Bei den meisten anderen Elementen wird der Inhalt als Textknoten angegeben, zum Beispiel bei `textarea`.

`mbref` – Mit der Referenz des Widgets kann auf seinen Inhalt zugegriffen werden. Um einen eindeutigen Wert zu generieren, wird der Konstruktor `HtmlCRef` aus der HTML-Bibliothek benutzt. Der konkrete Wert wird dem `id`- und `name`-Attribut des HTML-Elementes zugewiesen. Für Ajax-Anwendungen ist das `id`-Attribut ausreichend. UI-Widgets können jedoch auch in Beschreibungen mit der HTML-Bibliothek eingebettet werden, daher wird auch das `name`-Attribut gesetzt. Dadurch werden die Eingabedaten, als Name/Value-Paare, auch bei einem konventionellen Submit ohne Ajax an den Server geschickt.

`handlers` – Für die Event-Handler der UIs gibt es in der erweiterten HTML-Bibliothek den neuen Konstruktor `AjaxEvent`, der prinzipiell die gleiche Aufgabe wie der `HtmlEvent`-Konstruktor hat. `AjaxEvent` fügt eine neue Handlerfunktion in den Serverzustand ein, auf die dann über eine ID (`"EVENT_..."`) zugegriffen werden kann.

`styles` – Die Stilklassen, siehe Abschnitt 5.2, werden nach CSS übersetzt und dem `style`-Attribut zugewiesen.

`childs` – Falls das Widget Kind-Widgets enthält, wie bei Reihen, Spalten und Menüs, werden diese rekursiv mit `widget2hexp` in HTML-Ausdrücke übersetzt.

Jedes Widget kann eine Referenz und verschiedene Event-Handler haben, daher ist die Funktion `widget2hexp` sehr allgemein gehalten. Auch ein Button kann eine Referenz haben, um seine Beschriftung zu ändern oder ihm einen neuen Event-Handler zuzuweisen. In Listing 6.4 wird einem Label ein Handler zugewiesen, so dass es auf einen Klick reagiert:

```
ui = col [
     entry val "0",
     label "Increment" `addHandler` (Handler Click (Cmd cmd))]
  where  val free
         cmd env = do v <- getValue val env
                      setValue val (show (readInt v + 1)) env
```

Listing 6.4: Zähler mit einem Label als „Increment"-Button

6.3.1 Verwaltung der Event-Handler

Nachdem ein Startformular generiert wurde, wiederholt sich in der HTML-Bibliothek folgender Ablauf:

- Durch das Klicken eines „Submit"-Buttons werden die Formulardaten, einschließlich der Event-ID, an den Webserver geschickt.
- Auf dem Webserver wird der zur Event-ID passende Event-Handler ausgeführt. Das Ergebnis dieser Funktion ist ein neues HTML-Formular, das als Antwort an den Client gesendet wird.

Die UI2HTML-Bibliothek beruht auf dem gleichen Konzept. Hierbei verwaltet jedoch das JavaScript-Objekt `XMLHttpRequest` (siehe Abschnitt 4.3) im Hintergrund die Kommunikation mit dem Webserver. Das Konzept der Implementierung soll wieder am interaktiven Zähler vorgestellt werden.

```
col [label "A simple counter:",
      entry val "0",
      row [button inc "Increment", button reset "Reset",
           button exitUI "Stop"]]
  where
    val free
    inc env = ...
    reset env = ...
```

Listing 6.5: `counter.curry`

Aus der UI-Beschreibung in Listing 6.5 generiert das Makeskript `makecurrycgi` die ausführbaren Programme `counter.cgi` und `counter.cgi.server`. Diese Programme haben prinzipiell die gleichen Aufgaben wie in der HTML-Bibliothek:

`counter.cgi` ist ein CGI-Skript, das der Benutzer über seinen Browser aufruft. Es ist die Schnittstelle zwischen dem Client und dem Hauptprogramm auf dem Server.

`counter.cgi.server` ist das Hauptprogramm. Es generiert den HTML-Quelltext für die Counter-Oberfläche den es an den Browser sendet. Außerdem stellt es in Form von Event-Handlern die Funktionalität des Zählers zur Verfügung. Auf die Event-Handler kann über Event-IDs zugegriffen werden.

```
<div><span>A simple counter:</span></div>
<div><input type="text" value="0"
            id="FIELD_1" name="FIELD_1"/></div>
<div><input type="button" value="Increment"
            onclick="ajaxRequest('EVENT_0');"/>
     <input type="button" value="Reset"
            onclick="ajaxRequest('EVENT_1');"/>
     <input type="button" value="Stop"
            onclick="ajaxRequest('EVENT_2');"/></div>
```

Listing 6.6: Ausschnitt aus dem HTML-Quelltext des Counters

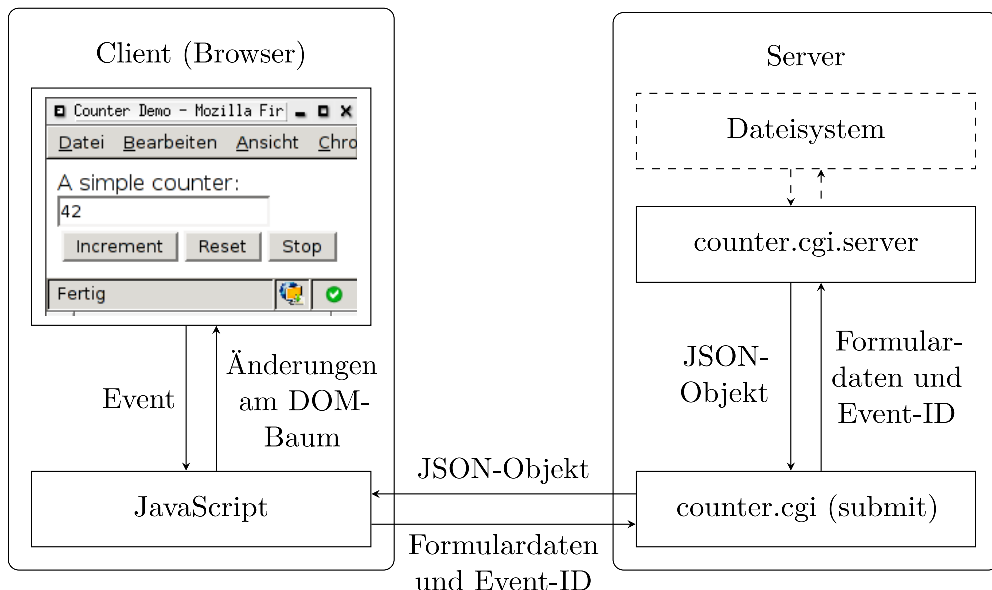

Abbildung 6.2: Skizze der Kommunikation zwischen Browser und Webserver, nachdem das Startformular Listing 6.6 vom Browser angefragt wurde.

Wenn der Client das `counter.cgi`-Skript anfragt, leitet dieses die Eingabedaten an das Hauptprogramm `counter.cgi.server` weiter. Das Hauptprogramm erzeugt anhand der UI-Beschreibung die Zähler-Oberfläche und verwaltet die angegebenen Event-Handler. Der Webserver sendet als Antwort eine HTML-Seite. Listing 6.6 zeigt einen Ausschnitt dieser HTML-Seite, die als graphische Oberfläche des Zählers im Browser angezeigt wird. Abbildung 6.2 skizziert den Ablauf der Kommunikation zwischen Browser und Webserver:

1. Drückt der Benutzer den „Increment"-Button, wird die im `onclick`-Attribut festgelegte JavaScript-Funktion `ajaxRequest('EVENT_0')` ausgeführt.

2. `ajaxRequest('EVENT_0')` durchsucht den DOM-Baum nach HTML-Elementen, für die eine Referenz angegeben wurde, also deren `id`-Attribut gesetzt ist. Alle Referenz/Wert-Paare und die Event-ID (`EVENT_0`) werden an das `counter.cgi`-Skript auf dem Webserver gesendet. In Abbildung 6.2 wird beim Klick auf den „Increment"- Button ein String der Form

   ```
   FIELD_1=0&EVENT_0=
   ```

   an den Webserver gesendet. Für die HTML-Bibliothek sieht die Ajax-Anfrage also genauso aus wie eine herkömmliche Anfrage, die durch einen „Submit"-Button ausgelöst wurde.

3. Auf dem Server wird das `counter.cgi`-Skript ausgeführt. Es zerlegt den Anfrage-String und sendet ihn als Liste von Name/Value-Paaren (`env`) weiter an das bereits laufende Programm `counter.cgi.server`.

4. Im Hauptprogramm `counter.cgi.server` wird der zur Event-ID gehörende Handler ausgeführt. In diesem Beispiel wird der Handler mit der ID `"EVENT_0"` herausgesucht und dementsprechend die Funktion `inc` ausgeführt. Die Handlerfunktion kann mit der entsprechenden Referenz auf die Werte der Widgets der Oberfläche zugreifen.

Um diesen Ablauf zu realisieren wurde die HTML-Bibliothek der PAKCS-Distribution erweitert, der Datentyp `HtmlForm` hat den neuen Konstruktor `AjaxEvent`. Mit „`AjaxEvent id handler`" wird eine freie Variable `id` mit einem Event-Handler `handler` verknüpft.

```
data HtmlExp =
   HtmlText String
 | HtmlStruct String [(String,String)] [HtmlExp]
 | HtmlCRef   HtmlExp CgiRef
 | HtmlEvent  HtmlExp HtmlHandler
 | AjaxEvent  String HtmlHandler
```

Durch die Konstruktion

```
[AjaxEvent id handler,
  (hexp `addAttr` ("onclick",
                   ajaxRequest"('EVENT_" ++ id ++ "');")]
```

werden die Attribute von `hexp` geändert. Dem JavaScript Event-Handler `onclick` wird eine JavaScript-Funktion zugewiesen, die die ID des Handlers `handler` als Parameter erhält. Der Event-Handler wird mit seiner ID im Hauptprogramm des Servers gespeichert. Die Beschreibung eines Zählers mit der UI-Bibliothek

```
main = runUI "Counter"
  (row [entry val "0", button inc "Increment"])
 where val free
       inc env = ...
```

Listing 6.7: Einfache Zähler UI

übersetzt die UI2HTML-Bibliothek in Datenstrukturen der HTML-Bibliothek, wie in Listing 6.8.

```
main = return $ ajaxForm "Counter"
  [HtmlCRef
    (HtmlStruct "input" [("type","text"),("value","0"),
                         ("id",idOfCgiRef val)] [])
   val,
   AjaxEvent id handler,
   HtmlStruct "input"
     [("type","button"),("value","Increment"),
      ("onclick","ajaxRequest('EVENT_" ++ id ++ "');")] []]
 where
    val, id free
    handler env = ... inc ...
```

Listing 6.8: Zähler UI in der HTML-Bibliothek

Wie in der HTML-Bibliothek wird dem Eingabefeld mit `HtmlCRef` eine ID (`val`) zugewiesen. Der Button wird über eine ID (`id`) mit dem Handler `handler` verknüpft. Der Handler hat den Ergebnistyp „`IO HtmlForm`“.

```
data HtmlForm = HtmlForm String [FormParam] [HtmlExp]
              | HtmlAnswer String String
              | AjaxAnswer Json
                           [([(String,String)],[HtmlExp])]
```

Für das Ergebnis der UI Event-Handler gibt es den neuen Konstruktor `AjaxAnswer`. Dieser enthält als erste Komponente Daten vom Typ Json, mit denen Änderungen des `value`- oder `style`-Attributes von HTML-Elementen ausgedrückt werden. Die zweite Komponente enthält Informationen für die Generierung von neuen HTML-Seiten. Spezialfälle davon sind die Generierung von Popups und das Ändern von Event-Handler zur Laufzeit.

Die Funktion `showAnswerFormInEnv` aus der HTML-Bibliothek erzeugt aus einem `AjaxAnswer`-Objekt ein Json-Objekt, das als Antwort an den Client gesendet wird. Dieses Json-Objekt wird von JavaScript interpretiert und die entsprechenden Änderungen an der graphischen Oberfläche vorgenommen.

Konvertierung der UI-Handler in HTML-Handler

Die Event-Handler in der HTML-Bibliothek haben den Typ „`CgiEnv -> IO HtmlForm`" und die UI Event-Handler sind vom Typ „`Env -> IO()`". In Listing 6.9 wird die Handler-Funktion `inc` in einen HTML-Handler `handler` „eingebettet".

```
handler :: CgiEnv -> IO HtmlForm
handler env = do stateref <- newIORef (env,...)
                 execCmdAndRespond inc stateref

inc :: UIEnv -> IO ()
inc env = do v <- getValue val env
             setValue val (show (readInt v + 1)) env
```

Listing 6.9: Konvertierung eines UI-Handlers in einen HTML-Handler

Wenn der „Increment"-Button gedrückt wird, wird der aktuelle Zustand der graphischen Oberfläche, also alle Formulardaten, an den Server gesendet. Der Event-Handler `inc` kann auf diesen Zustand zugreifen und ihn ändern, indem er eine Sequenz von IO-Aktionen ausführt. Da jedoch erst das Ergebnis der Berechnung an den Client geschickt wird, müssen die aktuellen Zustandsänderungen während der Ausführung der Handlerfunktion zwischengespeichert werden. Dieses geschieht über einen Wert vom Typ `IORef`. In der UI-Beschreibung kann mit den vordefinierten Funktionen `setValue` und `getValue` auf diese `IORef` zugegriffen werden.

Die Werte der Widgets liegen vor der Ausführung des Handlers wie in der HTML-Bibliothek üblich als Liste von Name/Value-Paaren vor. Außerdem gibt es eine Datenstruktur, die während der Ausführung des Handlers die einzelnen Veränderungen an den Widgets speichert. In Listing 6.10 muss `getValue` den Wert 42 liefern und nicht den Wert, den der Client für das Widget `val` zum Server gesendet hat:

```
cmd env = do setValue val "42" env
             v <- getValue val env
```

Listing 6.10: Event-Handler

Es müssen also die aktuellen Zustandsänderungen überprüft werden. Nur wenn der entsprechende Wert in diesen nicht vorhanden ist, wird der Wert genommen, den der Client geschickt hat. Die Funktion `getValue` kann wie in Listing 6.11 implementiert werden:

```
getValue :: Ref CgiRef -> UIEnv -> IO String
```

```
getValue ref (UIEnv stateref) = do
  State (oenv,nenv) <- readIORef stateref
  let value = findValue ref nenv
  case value of
    Just v  -> return v
    Nothing -> return $ env (ref2cgiRef ref)
```

Listing 6.11: `getValue`-Funktion der UI2HTML-Bibliothek

Um einen Wert in der Umgebung zu ändern, gibt es die Funktion `setValue`, siehe Listing 6.12. In dieser Funktion wird mit `readIORef` der aktuelle Zustand ausgelesen und geändert. Der veränderte Zustand wird mit `writeIORef` wieder zurückgeschrieben.

```
setValue :: Ref CgiRef -> String -> UIEnv -> IO ()
setValue ref value (UIEnv stateref) = do
  State (oenv,nenv) <- readIORef stateref
  let newenv = changeValue nenv ref value
  writeIORef stateref (State (oenv,newenv))
```

Listing 6.12: `setValue`-Funktion der UI2HTML-Bibliothek

Listing 6.13 zeigt die Funktion `execCmdAndRespond`. Diese führt einen UI-Handler aus und gibt als Ergebnis den Änderungszustand als `AjaxAnswer` zurück:

```
execCmdAndRespond :: (UIEnv -> IO()) -> IORef State ->
                                                    IO HtmlForm
execCmdAndRespond cmd stateref = do
  cmd (UIEnv stateref)
  State (_,nenv) <- readIORef stateref
  let ps = snd nenv
  return $ AjaxAnswer (state2json nenv) ps
```

Listing 6.13: Funktion zur Einbettung von UI-Handlern in HTML-Handlern

Die Änderungen, die der UI-Eventhandler an der Umgebung, also der graphischen Oberfläche macht, werden in einer `IORef` gespeichert. Nach dem Ausführen der Handlerfunktion werden sie mit `state2json` in einen JSON-String konvertiert. Dieser JSON-String, der vom JavaScript-Programm auf dem Client interpretiert wird, beschreibt die entsprechenden Änderungen an der graphischen Oberfläche. In den Zustandsänderungen können auch Informationen zu neuen HTML-Seiten und Änderungen von Event-Handlern stehen, in Listing 6.13 mit `ps` bezeichnet.

6.3.2 Generierung neuer Webseiten

Wie beschrieben, wird die Antwort auf eine Ajax-Anfrage mit dem Konstruktor `AjaxAnswer` dargestellt.

```
data HtmlForm
  = ...
  | AjaxAnswer Json [([(String,String)],[HtmlExp])]
```

Die erste Komponente enthält einen Wert vom Typ Json. Mit diesem Wert werden Zustandsänderungen wie Änderungen des Wertes oder der Darstellung von Widgets ausgedrückt. Die zweite Komponente enthält eine Liste mit Informationen, die zur Generierung neuer HTML-Seiten genutzt werden können. Die Listenelemente sind Paare der Form `(nvs,hexps)`, wobei `nvs` Name/Value-Paare sind, die Informationen enthalten, wie der Client mit dem aus `hexps` generierten HTML-Quelltext umgehen soll. Im Hauptprogramm auf dem Server werden die Werte vom Typ `HtmlExp` in konkreten HTML-Quelltext konvertiert. Dabei werden alle freien Variablen instanziiert und die neuen Handler als Liste von `(handler,key)` Paaren zurückgegeben.

In der HTML-Bibliothek wird die Funktion `showAnswerFormInEnv` auf einen Wert `(AjaxAnswer cont nvsAndhexps)` aufgerufen. Sie liefert neben dem JSON-String, der die Antwort der Benutzeranfrage darstellt, auch eine Liste von Event-Handlern, die mit der Funktion `storeEnvHandlers` in den Serverzustand aufgenommen werden.

Die zweite Komponente des `AjaxAnswer`-Konstruktors wird verwendet für Popups, für Änderungen an Event-Handlern zur Laufzeit und für die Generierung neuer HTML-Seiten. Im folgenden wird auf diese Möglichkeiten eingegangen.

Neue Fenster (Popups)

In einer UI-Beschreibung kann mit der Aktion „`showPopup title widget env`“ ein neues Fenster geöffnet werden, das wiederum mit einer UI-Beschreibung `widget` beschrieben wird. Um Popups zu generieren, erzeugt die UI2HTML-Bibliothek ein Paar der Form

```
([("type","Popup"),("title",title)],ui2hexps widget)
```

Die Liste in der ersten Komponente enthält Informationen, durch die das JavaScript-Programm auf dem Client erkennt, dass ein Popup dargestellt werden soll. Der Quelltext des Popups wird mit der HTML-Bibliothek aus der zweiten Komponente erzeugt. In JavaScript wird ein neues Fenster mit `open()` geöffnet und in dieses Fenster mit `write` der HTML-Quelltext geschrieben. Außerdem wird der Titel des Fensters gesetzt.

```
var newWindow = window.open();
newWindow.document.open();
newWindow.document.write(html);
newWindow.document.close();
newWindow.document.title = title;
```

Neue Event-Handler

Mit der Aktion „`setHandler ref event cmd env`“ wird einem Widget mit der Referenz `ref` eine neue Handlerfunktion `cmd` zugewiesen. Diese wird ausgeführt, wenn der Benutzer das Ereignis `event` auslöst.

```
([("type","changeevent"),("id",idOfRef ref),
  ("event",event),("eventid","EVENT_" ++ id)],
 [nhexp])
```

Die erste Komponente enthält wieder Informationen für das JavaScript-Programm. Diese Informationen enthalten eine ID, die Referenz des Elementes, für das der Handler gesetzt werden soll, den Eventtyp (`event`) und die Event-ID (`eventid`), unter der der Handler auf dem Server abgelegt wurde. Mit

```
elem.onclick = function () { ajaxRequest(eventid); };
```

wird einem Element ein neuer `onclick` Event-Handler zugewiesen.

Neue HTML-Seiten

Eine Aktion, die nicht zur allgemeinen UI-Beschreibung gehört, ist `nextHtmlForm`. Mit `nextHtmlForm (HtmlForm title params hexps) env`. wird direkt eine neue HTML-Seite im Browser dargestellt.

```
([("type","htmlsite"),("title",title)],hexps)
```

Auf dem Client wird mittels JavaScript der Inhalt des Formularelementes mit dem neuen Inhalt überschrieben.

```
document.title = title;
document.getElementsByTagName("form")[0].innerHTML = html;
```

6.3.3 Änderung der Darstellung von UIs

Wie in Abschnitt 5.2 beschrieben ist es möglich die Darstellung von Widgets zu ändern. Dafür wird in der UI2HTML-Bibliothek CSS eingesetzt. Die Datentypen für Darstellungsinformationen

```
data StyleClass = Class [Style]
data Style = Active Bool | Bg Color | Font FontStyle | ...
```

werden mit

```
styleClasses2String :: [StyleClass] -> String
```

in einen CSS String konvertiert. Dieser String wird dem `style`-Attribut des entsprechenden HTML-Elements zugewiesen.

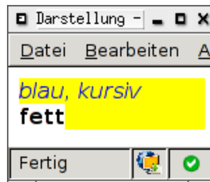

Abbildung 6.3: UI mit Stil

Die UI-Beschreibung in Listing 6.14 enthält Widgets mit Darstellungsinformationen. Aus ihr kann mit der UI2HTML-Bibliothek eine Webanwendung wie in Abbildung 6.3 generiert werden.

```
blueitalic  = Class [TextColor Blue, Font Italic]
bgyellow    = Class [Bg Yellow]
bgwhitebold = Class [Font Bold, Bg White]

ui = colS [bgyellow] [
  labelS [blueitalic] "blau, kursiv",
  labelS [bgwhitebold] "fett"]
```

Listing 6.14: UI mit Darstellungsinformationen

Die generierte HTML-Seite enthält in den `style`-Attributen Darstellungsinformationen:

```
<div style="background-color: yellow;"> ...
  <span style="color: blue; font-style: italic;">
    blau, kursiv</span> ...
  <span style="font-weight: bold; background-color: white;">
     fett</span>
</div>
```

Eine andere Möglichkeit wäre, vor dem eigentlichen Programmlauf diese Darstellungsinformationen aus dem Programm zu filtern und in eine externe CSS-Datei auszulagern. Dazu könnte die UI2HTML-Bibliothek IDs für die Stilklassen erzeugen, die dem `class`-Attribut der HTML-Elemente zugewiesen werden. Für Listing 6.14 kann eine CSS Datei mit folgendem Inhalt angelegt werden:

```
.cid1 {color: blue;}
.cid2 {font-style: italic;}
.cid3 {background-color: yellow;}
.cid4 {font-weight: bold;}
.cid5 {background-color: white;}
```

Im Dokument können dann die entsprechenden IDs (`cid...`) den `class`-Attributen zugewiesen werden.

```
<div class="cid3">
  <div><span class="cid1 cid2">blau, kursiv</span></div>
```

```
  <div><span class="cid4 cid5">fett</span></div>
</div>
```

Es ist also möglich, aus einem Programm die Stilinformationen zu extrahieren und im Programmcode durch Metaprogrammierung entsprechende IDs zu setzen.

In der UI2HTML-Bibliothek werden momentan jedoch nur die `style`-Attribute der HTML-Elemente genutzt.

6.3.4 Weitere Ideen zur Implementierung

Erweiterung der HTML-Bibliothek

Wie in Unterabschnitt 6.3.1 beschrieben, wurden in der HTML-Bibliothek einige neue Konstruktoren eingeführt, um besser mit Ajax umgehen zu können. Im folgenden werden einige Argumente gebracht, die für eine Änderung der HTML-Bibliothek sprechen.

In der HTML-Bibliothek wird der Konstruktor „`HtmlEvent hexp handler`“ benutzt, um einen HTML-Ausdruck (`hexp`) mit einem Event-Handler (`handler`) auf dem Server zu verknüpfen. Dazu wird dem `name`-Attribut des HTML-Elementes die Event-ID zugewiesen und diese dadurch bei einem „Submit“ an den Server gesendet. In der UI2HTML-Bibliothek kann das gleiche Konzept genutzt werden, indem dem JavaScript Event-Handler (z.B. `onclick`) die Funktion `ajaxRequest(this)` zugewiesen wird. Dann kann im JavaScript-Programm wie folgt die ID abgefragt werden:

```
<input type="button" value="Increment"
       name="EVENT_0" onclick="ajaxRequest(this);" >

function ajaxRequest(elem) {
  var eventid = elem.name;
  ...
}
```

Dieser Ansatz scheitert jedoch daran, dass in der UI2HTML-Bibliothek für ein Widget gleichzeitig eine Referenz und mehrere Event-Handler definiert sein können und es nicht ohne weiteres möglich ist, das `name`-Attribut für beides gleichzeitig zu nutzen. Zum Beispiel kann für einen Button gleichzeitig ein Event-Handler und eine Referenz definiert werden.

```
button inc "0" `setRef` val
```

Ein weiteres Problem bei diesem Ansatz ist, dass einige HTML-Elemente, die als Widgets genutzt werden können, kein `name`-Attribut haben. Zum Beispiel können Label mit dem `span`-Tag beschrieben werden oder Reihen und Spalten mit Tabellen.

Diese Probleme können umgangen werden, indem versteckte HTML-Elemente (`type ="hidden"`) eingesetzt werden, um Event-IDs zu speichern. Diese können dann über eine ID mit dem jeweiligen Widget verknüpft werden. In Listing 6.15 verbindet eine ID (`ID_0`) einen Button mit seiner Event-ID (`EVENT_0`):

```
<input type="hidden" id="ID_0" name="EVENT_0" value="">

<input type="button" value="Increment"
      onclick="ajaxRequest('ID_0');" >
```

Listing 6.15: Versteckte HTML-Elemente zur Verwaltung von Event-IDs

Der Ansatz über versteckte Elemente die Event-IDs zu verwalten, wurde vor allem wegen Problemen bei der Instanziierung von freien Variablen verworfen. Außerdem wird die Implementierung und der erzeugte HTML-Quelltext dadurch unübersichtlich.

Auch aus folgenden anderen Gründen lassen sich Änderungen an der HTML-Bibliothek nur schwer vermeiden. Die UI2HTML-Bibliothek erlaubt Änderungen an der UI, wie das Öffnen von Popups und das Ändern von Handlern. Mit dem Datentyp `HtmlForm` der HTML-Bibliothek ist es schwierig solche Änderungen auszudrücken, denn es können beliebig viele solcher Änderungen in einer UI-Handler-Funktion vorkommen. Daher gibt es zusätzlich den Konstruktor `AjaxAnswer`, dem sowohl ein String, als auch HTML-Ausdrücke übergeben werden können. Prinzipiell ist er also eine Mischung aus den Konstruktoren `HtmlForm` und `HtmlAnswer`

```
data HtmlForm =
        HtmlForm String [FormParam] [HtmlExp]
      | HtmlAnswer String String
      | AjaxAnswer Json [([(String,String)],[HtmlExp])]
```

Änderungen an der graphischen Oberfläche (Umgebungsmodell)

Ein Event-Handler kann den Zustand der graphischen Oberfläche durch vordefinierte IO-Aktion wie „`getValue ref env`“ und „`setValue ref val env`“ abfragen und ändern. Auf dem Webserver wird ein Event-Handler abgearbeitet und die Änderungen an der graphischen Oberfläche zwischengespeichert. Als Antwort des Handlers werden die Zustandsänderungen an den Client gesendet und dort die Oberfläche entsprechend angepasst.

Ein UI-Event-Handler kann in einen HTML-Event-Handler konvertiert werden, indem in jedem Handler ein neuer Wert vom Typ `IORef` erzeugt wird. Diese `IORef` enthält einerseits den aktuellen Zustand der graphischen Oberfläche und andererseits eine Funktion `env` vom Typ „`CgiRef -> String`“, mit der auf die Daten zugegriffen werden kann, die die graphische Oberfläche vor der Ausführung des Handlers hatte. Bei dieser Implementierung gibt es jedoch Probleme mit der Umgebung, wenn eine Aktion neue Fenster (Popups) erzeugt wie in Listing 6.16.

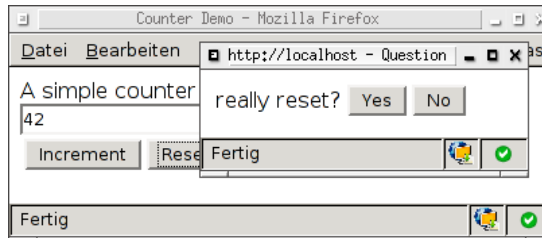

Abbildung 6.4: Zähler mit Sicherheitsabfrage beim Reset

```
counterUI =
  col [label "A simple counter:",
       entry val "0",
       row   [button inc "Increment", button reset "Reset",
              button exitUI "Stop"]]
 where
  val free
  inc menv = do v <- getValue val menv
                setValue val (show (readInt v + 1)) menv

  reset menv = do
    showPopup "Question"
      (col [label "Reset Counter?",
            row [button yes "Yes", button no  "No"]]) menv
   where
    yes penv = setValue val "0" menv >> exitUI penv
    no  penv = exitUI penv
```

Listing 6.16: UI mit Popup

Wenn der Benutzer auf den „Reset"-Button klickt, wird ein Popup wie in Abbildung 6.4 angezeigt. Dieses Popup ist im Programm eine eigenständige Benutzerschnittstelle und soll auf die Widgets im Hauptfenster zugreifen können. Da es verschiedene Benutzerschnittstellen sind, ist es sinnvoll, die Umgebungen im Quelltext zu unterscheiden. In Listing 6.16 gibt es zwei Umgebungen, die mit `menv` und `penv` bezeichnet werden.

`menv` ist die Umgebung des Hauptfenster. Mit ihr können die Widgets auf dem Hauptfenster abgefragt und geändert werden.

`penv` ist die Umgebung, die die Widgets des Popups verwaltet.

Der Popup Event-Handler `yes` muss auf die Umgebung des Hauptfensters zugreifen, um den Wert des Textfeldes zu ändern.

```
yes penv = do setValue val "0" menv
              exitUI penv
```

Wenn nun die Umgebung beim Ausführen von `yes` in einer lokalen `IORef` verwaltet wird, dann wird zwar auch die zu `menv` gehörende `IORef` geändert, aber es wird nur die `IORef`, die die Änderungen in `penv` enthält, zurückgegeben. Dieses Problem wird in der UI2HTML-Bibliothek durch eine globale `IORef` gelöst, die direkt beim Aufruf von `runUI` erzeugt wird und dann von allen Handlerfunktionen benutzt wird.

```
runUI title widget = do
   stateref <- newIORef (...)
   ...
```

Problematisch an diesem Ansatz ist, dass im Quelltext die gleiche Umgebung genutzt werden kann und das Programm als Webanwendung läuft. Aus einer solchen Beschreibung kann dann jedoch keine Desktopanwendung mehr erzeugt werden, denn in der GUI-Bibliothek müssen verschiedene Umgebungen, bzw. Kommunikationskanäle, benutzt werden.

Einige Probleme mit dem Umgebungsmodell könnten gelöst werden, indem Änderungen an den Widgets nicht mehr zwischengespeichert, sondern direkt ausgeführt werden wie es die GUI-Bibliothek macht. Dazu müssten jedoch Aktionen wie `setValue` oder `getValue` direkt beim Aufruf eine Anfrage an den Client stellen. Dieses wäre auch mit dem `XMLHttpRequest`-Objekt von JavaScript möglich. Für das Konzept von serverseitigen Anfragen an den Client gibt es Techniken, die unter Begriffen, wie *Comet*, *Ajax Push*, *Reverse Ajax* und *HTTP Streaming* bekannt sind. Auf diese Möglichkeit der Implementierung von Event-Handlern wird in dieser Arbeit jedoch nicht weiter eingegangen. Ein Nachteil dieses Ansatzes für den Einsatz in Event-Handlern ist, dass mehr Nachrichten über das Netzwerk verschickt werden müssen und dadurch die graphische Oberfläche langsamer reagieren könnte. Ein wesentlicher Vorteil wäre jedoch, dass wirklich nur die Daten vom Client zum Server übertragen werden, die die Handler-Funktionen benötigen.

Übermittlung der Eingabedaten

Über das `XMLHttpRequest`-Objekt sendet das JavaScript-Programm die Event-ID und den Zustand der graphischen Oberfläche, also alle Formulardaten, als Name/Value-Paare, an den Server. Der Client schickt dem Server also Daten, die von der Handler-Funktion auf dem Server nicht benötigt werden. Dies kann die Reaktionszeit der Oberfläche erheblich beeinflussen. An der Implementierung der UI2HTML gibt es also zwei Kritikpunkte.

- Alle Widgetdaten werden verschickt. Im schlimmsten Fall umfangreiche Texte aus Textfeldern (`textarea`). Diese Daten werden auf dem Server wieder eingelesen, auch wenn sie der Event-Handler nicht benötigt.

- Da erst am Ende der Berechnung eines Event-Handlers die Oberfläche entsprechend verändert wird, kann ein Event-Handler während der Berechnung keine Änderungen wie Statusmeldungen an der graphischen Oberfläche vornehmen.

Um das Problem mit der großen unnötigen Datenmenge zu lösen, könnten die Handlerfunktionen des Curry-Programms analysiert werden und alle Referenzen, die im Handler vorkommen, herausgesucht werden. Diese Referenzen könnten dann der JavaScript Handler-Funktion übergeben werden. JavaScript braucht dann nur noch die entsprechenden Name/Value-Paare zum Server schicken. Die UI-Beschreibung müsste also durch Metaprogrammierung geändert werden. In Listing 6.17 wird der JavaScript Funktion die Information übergeben, dass der Event-Handler nur die Werte der Widgets `EDIT_1` und `EDIT_5` benötigt.

```
<input type="button" value="Increment"
       onclick="ajaxRequest('EVENT_0','EDIT_1;EDIT_5');" >
```

Listing 6.17: JavaScript Event-Handler mit Zusatzinformationen

Da im Event-Handler auch Fallunterscheidungen möglich sind, werden zwar auch bei diesem Ansatz unter Umständen zu viele Daten übertragen, aber die Datenmenge insgesamt würde sich reduzieren.

Eine weitere Möglichkeit das Umgebungsmodell zu verbessern wäre im JavaScript-Programm die Änderungen an den Widgets zwischen zwei Serveranfragen zu protokollieren und nur diese Änderungen an den Webserver zu senden. Dazu müsste jedoch auch auf dem Server der aktuelle Zustand der Oberfläche gespeichert werden.

6.4 Die GUI2HTML-Bibliothek

Aus Beschreibungen mit der GUI-Bibliothek können mit der GUI2HTML-Bibliothek Desktop-ähnliche Webanwendungen generiert werden. Dazu muss im Curry-Programm nur die Importanweisung von GUI auf GUI2HTML geändert werden. Dabei gibt es jedoch einige Einschränkungen.

- Ausgaben auf der Standardausgabe sollte vermieden werden. Der Server fasst diese unter Umständen als Antwort auf die Anfrage des Client auf.
- In der GUI-Bibliothek können externe Event-Handler definiert werden. Diese Möglichkeit gibt es in der GUI2HTML-Bibliothek nicht.

In der GUI2HTML-Bibliothek werden die gleichen Datentypen definiert wie in der GUI-Bibliothek. Mit der Funktion `gui2ui` werden diese Datentypen auf die entsprechenden UI-Datentypen abgebildet und dann mit der UI2HTML-Bibliothek in eine

Webanwendung übersetzt. Die GUI-Event-Handler „`Handler event cmd`“ werden in UI-Event-Handler übersetzt.

```
Handler event cmd ->
  UI.Handler (gui2uievent event) (UI.Cmd cmd')
     where cmd' env = do wconfs <- cmd env
                         widgetconf2cmd wconfs env
```

Listing 6.18: Konvertierung der Event-Handler

In Listing 6.18 ist `wconfs` eine Liste von Werten mit dem Typ `ReconfigureItem`, durch die die Änderungen an der Konfiguration des Widgets ausgedrückt werden. Die Liste `wconfs` wird mit `widgetconf2cmd` in eine Sequenz von IO-Aktionen umgewandelt.

Die Referenzen werden so angepasst, dass die vordefinierten IO-Aktionen direkt aufeinander abgebildet werden können.

```
getValue = UI2HTML.getValue
setValue = UI2HTML.setValue
```

Auch die Typangaben in den GUI-Definitionen können beibehalten werden, da in der GUI2HTML-Bibliothek folgende Typ-Synonyme eingeführt werden.

```
type GuiPort = UI2HTML.UIEnv
type WidgetRef =  UI.Ref HTML.CgiRef
```

7 Typsichere Benutzerschnittstellen

Mit den bisher vorgestellten Bibliotheken ist die Beschreibung der Struktur und Funktionalität von Benutzerschnittstellen sehr übersichtlich und kompakt möglich. Die Eingabedaten des Benutzers müssen jedoch häufig eine Reihe von Anforderungen erfüllen, die der Programmierer bisher noch von Hand überprüfen muss.

- Sind alle erforderlichen Eingabefelder ausgefüllt?
- Hat der Benutzer wirklich eine ganze Zahl eingegeben und keine beliebige Zeichenkette?
- Ergibt eine Kombination von Eingabefeldern einen Sinn? Ist zum Beispiel mindestens eine der Checkboxen angekreuzt?

Es wäre also wünschenswert, mit einem einfachen Konzept komplexe Widgets zu erzeugen, für die beliebige Bedingungen angegeben werden können und die im Fehlerfall entsprechende Fehlerhinweise automatisch generieren. Dadurch würde dem Programmierer lästige Arbeit erspart; außerdem wären die Programme übersichtlicher. Die Bibliothek WUI (*Web User Interface*) setzt diese Idee für HTML-Formulare um.

7.1 Die WUI-Bibliothek

Mit der WUI-Bibliothek [7] aus der PAKCS-Distribution kann ein Programmierer auf einfache Weise sicherstellen, dass der Benutzer korrekte Daten in ein HTML-Formular eingegeben hat. Dazu überprüft die WUI-Bibliothek die vom Programmierer angegebenen Bedingungen für die Formulardaten und weist den Benutzer auf Eingabefehler hin. Die Formulardaten werden erst dann mit einem Event-Handler weiterverarbeitet, wenn alle Bedingungen erfüllt sind.

In der WUI-Bibliothek werden Funktionen definiert, mit denen kleine vordefinierte Widgets zu beliebig großen Widgets kombiniert werden können. Für jedes Widget können separat Darstellungsinformationen, eine Fehlermeldung und eine Bedingung angegeben werden. Falls die Bedingung nicht erfüllt ist, wird die entsprechende Fehlermeldung angezeigt. Bedingungen werden dabei von unten nach oben überprüft. Wenn bei einem Basiselement ein Fehler auftritt, wird die Bedingung einer Kombination

aus diesem Widget nicht mehr überprüft. Basiselemente sind Widgets wie `wString` und `wInt`. `wString` ist ein einfaches Texteingabefeld und `wInt` ein Texteingabefeld, in dem der Nutzer eine ganze Zahl eingeben soll. `wInt` ist typsicher, das heißt, falls der Benutzer in diesem Widget keine ganze Zahl eingegeben hat, wird automatisch zur Laufzeit eine Fehlermeldung generiert. Die Funktion `wPair` ist ein Kombinator, also eine Funktion die zwei Widgets zu einem neuen Widget kombiniert. Dieses neue Widget enthält die Funktionalität seiner Kindwidgets und kann wieder Bedingungen haben, die überprüft werden sollen. Ein kombiniertes Widget hat den kombinierten Typ der Kindwidgets. Widgets, die Werte vom Typ `a` enthalten, haben in der Bibliothek den Typ `WuiSpec a`.

```
wInt    :: WuiSpec Int
wString :: WuiSpec String

wPair wInt wString :: WuiSpec (Int,String)
```

Sendet der Benutzer seine Formulardaten an den Webserver, wird dort geprüft, ob die Eingaben den Bedingungen entsprechen, die vom Programmierer vorgegeben wurden. Sind alle Bedingungen erfüllt, werden die Daten vom Server verarbeitet und der Programmablauf fortgesetzt. Wenn die Bedingungen jedoch nicht erfüllt sind, bekommt der Benutzer ein ähnliches Formular vom Server geliefert, das neben den bisherigen Benutzereingaben auch Informationen enthält, die auf die Eingabefehler hinweisen. Dieser Vorgang wiederholt sich bis die Benutzereingaben alle Bedingungen erfüllen.

Um aus einem Widget bzw. WUI-Spezifikation eine konkrete Anwendung zu erzeugen, wird in der WUI-Bibliothek die Funktion `mainWUI` definiert. Diese wird auf ein Widget, einen Initialwert und eine „Updatefunktion“ angewendet. Durch den Aufruf

```
mainWUI wuispec val store
```

wird ein Webformular generiert, dessen Darstellung und Funktionalität mit dem Widget `wuispec` angegeben wird. Der Initialwert des Widgets ist der Wert `val`. Dieser legt die Anfangswerte des HTML-Formulars fest. Die `store`-Funktion wird bei korrekter Eingabe auf den durch den Benutzer veränderten Wert des Formulars `val'` angewendet und liefert die nächste Webseite.

Listing 7.1 ist ein Beispiel für eine WUI-Anwendung. Die Funktion `mainWUI` wird auf ein Widget `wInt` mit dem Initialwert `0` angewendet. Die Funktion `resultForm` in Listing 7.1 zeigt auf einer neuen HTML-Seite den Wert des Widgets an. Dazu wird der Wert mit der `show`-Funktion in einen String konvertiert. Abbildung 7.1 zeigt einige mögliche Zustände dieser Webanwendung. Dabei hat der Anwender zunächst eine Zeichenkette statt einer ganzen Zahl eingegeben und wird nach dem „Submit“ des Formulars auf seinen Fehler hingewiesen. Gibt er nun eine ganze Zahl ein, wird schließlich die Funktion `resultForm` auf seine Eingabe aufgerufen und eine Antwortseite generiert.

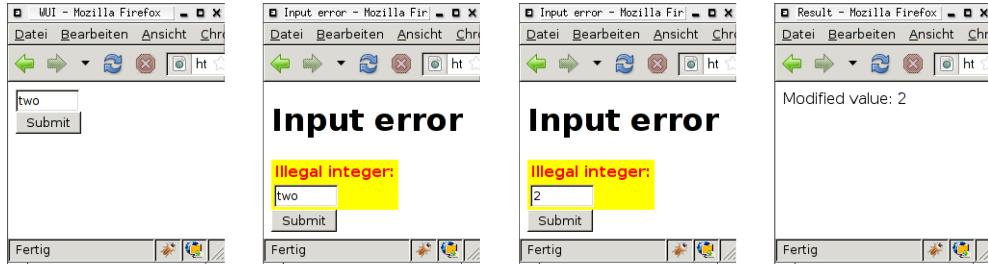

Abbildung 7.1: Zustände, der aus Listing 7.1 generierten Webanwendung.

```
main = mainWUI wInt 0 resultForm

resultForm :: a -> IO HtmlForm
resultForm v = return $ form
  "Result" [htxt ("Modified value: "++ show v)]
```

Listing 7.1: Einfache WUI für ganze Zahlen

In der WUI-Bibliothek sind einige Kombinatoren für Widgets vordefiniert. Es gibt zum Beispiel vordefinierte WUI-Kombinatoren für Tupel und Listen:

```
wPair   :: WuiSpec a -> WuiSpec b -> WuiSpec (a,b)
wTriple :: WuiSpec a -> WuiSpec b -> WuiSpec c ->
                                           WuiSpec (a,b,c)
wList :: WuiSpec a -> WuiSpec [a]
```

Abbildung 7.2 veranschaulicht das Kombinieren von Widgets beliebigen Typs zu einem Widget des kombinierten Typs.

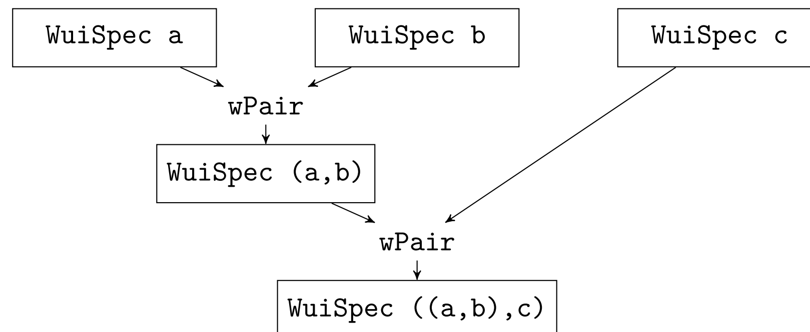

Abbildung 7.2: Kombination von Widgets, mit den Typen `a`, `b`, `c`, zu einem neuen Widget mit dem Typ `((a,b),c)`

Wie Abbildung 7.1 zeigt, stellt `wInt` automatisch sicher, dass der Benutzer eine ganze Zahl eingibt. Zusätzlich zu den Typprüfungen kann der Anwender der Bibliothek noch weitere Eigenschaften für jedes Widget angeben. Die Darstellung von Widgets kann

mit der Funktion `withRendering` geändert werden, die Fehlermeldung mit `withError` und die zu überprüfende Bedingung mit `withCondition`.

```
withRendering :: WuiSpec a -> Rendering   -> WuiSpec a
withError     :: WuiSpec a -> String      -> WuiSpec a
withCondition :: WuiSpec a -> (a -> Bool) -> WuiSpec a
```

Als Beispiel dazu zwei Widgets:

```
w1 = wInt `withCondition` (\n -> n < 10)
          `withError`     (">= 10!")

w2 = wPair wInt wInt
           `withCondition` (\ (n1,n2) -> n1 + n2 < 10)
```

Listing 7.2: Beispiele für WUI-Widgets

Widget `w1` in Listing 7.2 ist ein Textfeld, das sicherstellt, dass eine ganze Zahl eingegeben wird, die kleiner als 10 ist, ansonsten wird die Fehlermeldung `">= 10!"` angezeigt. Das Widget `w2` ist ein aus Basiselementen kombiniertes Widget. Bei einem kombinierten Widget werden zunächst die Kindwidgets überprüft und falls kein Fehler auftritt schließlich die Kombination. Allgemein werden Kombinationen wie folgt überprüft:

1. Zunächst werden die Basiswidgets getestet. Tritt ein Fehler auf, wird das komplette Widget noch einmal als Antwort angezeigt, wobei die Eingabefehler kenntlich gemacht werden.

2. Werden in den beiden Feldern ganze Zahlen eingegeben, wird die Bedingung der Kombination überprüft. Ist diese erfüllt, so wird die nächst höhere Kombination überprüft, ansonsten das gesamte Widget nochmals ausgegeben und der Benutzer auf seine Eingabefehler hingewiesen.

3. Wenn schließlich alle Kombinationen von unten nach oben abgearbeitet wurden und kein Fehler aufgetreten ist, werden die Daten des Benutzers als korrekt anerkannt und die `store`-Funktion auf den Wert, die die WUI-Spezifikation liefert, auf den Wert angewendet.

Implementierung

Ein WUI-Widget, das einen Wert vom Typ `a` enthalten soll, hat den Typ `WuiSepc a`.

```
data WuiSpec a =
  WuiSpec
    (WuiParams a)
    (WuiParams a -> a -> HtmlState)
    (WuiParams a -> CgiEnv -> WuiState ->
```

```
                                         (Maybe a,HtmlState))

type WuiParams a = (Rendering, String, a->Bool)

type Rendering = [HtmlExp] -> HtmlExp
type HtmlState = (HtmlExp,WuiState)
```

Die Eigenschaften eines Widgets werden in einem Tupel vom Typ `WuiParams a` gespeichert. Ein solches Tupel `(render,errmsg,legal)` besteht aus Informationen zur Darstellung, einer Fehlermeldung und der Bedingung, die für den Eingabewert überprüft werden soll. Vordefiniert ist beim Widget `wString` das Tupel

```
(head, "?", const True)
```

Die vordefinierte Bedingung eines `wString` Widgets ist also immer erfüllt. Die Bedingung kann jedoch mit `withCondition` geändert werden. Falls sie nicht erfüllt ist, wird dann als Fehlermeldung `"?"` benutzt. Diese Meldung kann wiederum mit der Funktion `withError` verändert werden. Ein konkretes Widget wird mit

```
WuiSpec wparams showhtml readval
```

definiert. Dabei haben die Parameter folgende Bedeutungen:

`wparams` – ist ein Tripel `(render,errmsg,legal)` aus Darstellungsfunktion, Fehlermeldung und Bedingung.

`showhtml` – generiert mit Hilfe der Informationen aus dem `wparams` Tupel und einem Initialwert einen Wert vom Typ `HtmlExp`. Aus diesem Wert wird mit der HTML-Bibliothek das HTML-Formular erzeugt.

`readval` – wird aufgerufen, wenn der Benutzer die Formulardaten an den Webserver gesendet hat. Wenn der Wert, den der Benutzer gesendet hat, nicht alle Bedingungen erfüllt, wird als Antwort ein neues Formular generiert, das die bisherigen Eingaben des Benutzers enthält und Fehlermeldungen anzeigt. Falls die Eingaben des Benutzers korrekt waren, liefert `readval` den kombinierten Eingabewert, welcher dann zur weiteren Verarbeitung einem Event-Handler übergeben wird. Die `readval`-Funktion kann mit Hilfe einer Funktion vom Typ `CgiEnv` auf die Feldinhalte zugreifen. Um die Referenzen der Eingabefelder zu verwalten, gibt es in der WUI-Bibliothek Werte vom Typ `WuiState`.

Bei einem Widget vom Typ `WuiSpec a` liefert `readval` entweder den Wert des Widgets mit dem Typ `Maybe a` oder `Nothing` und ein Fehlerformular.

7.2 Typsichere UIs

Mit einem ähnlichen Konzept wie dem der WUI-Bibliothek können auch Widgets der UI-Bibliothek kombiniert werden, um typsichere UI-Anwendungen zu erstellen. Im Unterschied zur WUI-Bibliothek können aus diesen Beschreibungen nach Belieben Webanwendungen oder Desktopanwendungen erzeugt werden. UI-Spezifikationen erfüllen grundsätzlich die gleichen Aufgaben wie WUI-Spezifikationen, siehe Abschnitt 7.1. Es gibt jedoch einige Unterschiede zur WUI-Bibliothek:

- Aus UI-Spezifikationen können nicht nur Webanwendungen, sondern auch Desktop-Anwendungen generiert werden. Dazu muss wieder nur eine Importanweisung im Programm geändert werden.
- Im Gegensatz zur WUI-Bibliothek werden bei Eingabefehlern keine neuen Oberflächen generiert, sondern dem Benutzer seine Eingabefehler direkt auf der ursprünglichen Oberfläche angezeigt. Für die Fehlermeldungen werden Labels benutzt, die entweder keine Beschriftung haben oder im Fehlerfall eine rote Fehlermeldung anzeigen.
- Es können nun auch mit `withConditionIO` Bedingungen vom Typ „`a -> IO Bool`“ angegeben werden.
- Die Updatefunktion (`store`) liefert keine neue UI, sondern ist eine IO-Aktion mit dem Typ „`a -> IO ()`“.

`typedui2ui` erzeugt aus einer UI-Spezifikation und einem Initialwert ein konkretes UI-Widget und Funktionen, um auf den Inhalt dieses Widgets typsicher zuzugreifen.

```
(widget,getval,setval,updval) = typedui2ui uispec initval
```

Die erzeugten IO-Aktionen haben die gleichen Aufgaben wie `getValue`, `setValue` und `updateValue` aus der UI-Bibliothek, nur sind sie in diesem Fall typsicher.

```
getval :: UIEnv -> IO (Maybe a)
```

liefert den Wert des Widgets, wenn kein Fehler auftritt. Ansonsten wird `Nothing` zurückgegeben und Fehlermeldungen auf dem Widget angezeigt.

```
setval :: a -> UIEnv -> IO ()
```

setzt den Wert des Widgets neu.

```
updval :: (a -> a) -> UIEnv -> IO ()
```

wendet eine Updatefunktion auf den Wert des Widgets an, falls die Eingaben des Benutzers alle Bedingungen erfüllen.

In Listing 7.3 wird mit der UISpec-Bibliothek ein typsicheren Zähler beschrieben. Für die Anzeige des Wertes wird die UI-Spezifikation `wInt` benutzt und somit sichergestellt, dass der „Increment“-Handler nur ausgeführt wird, wenn der Benutzer in das Eingabefeld eine ganze Zahl eingegeben hat.

Abbildung 7.3: Zustand des typsicheren Zählers nach einer Eingabe, die keine ganze Zahl war.

```
counterUI = col [
    label "A simple counter:",
    widget,
    row [button inc "Increment", button reset "Reset",
         button exitUI "Stop"]]
  where
    (widget,getval,setval,updval) = typedui2ui wInt 0

    reset env = setval 0 env
    inc   env = updval (+1) env
```

Listing 7.3: typsicherer interaktiver Zähler

Durch die Typsicherheit sind in Listing 7.3 keine Konvertierungen der Typen mit `readInt` und `show` mehr nötig.

Implementierung

Im Fehlerfall werden keine neuen Seiten generiert, daher gibt es keine `read`-Funktion im Sinne der WUISpec-Implementierung. Die `show`-Funktion einer UI-Spezifikation liefert neben dem Widget zwei weitere Funktionen, mit denen der Wert des Widgets ausgelesen und neu gesetzt werden kann.

```
data UISpec a =
  UISpec (UIParams a)
         (UIParams a -> a -> (UIWidget, UIEnv -> IO (Maybe a),
                                        a -> UIEnv -> IO ()))

type UIParams a = (Rendering, String, a -> IO Bool)
```

```
type Rendering = [UIWidget] -> UIWidget
```

Dabei haben die Parameter eines konkreten Wertes

```
UISpec wparams show
```

folgende Bedeutungen:

wparams – ist wie bei den WUIs wieder ein Tupel `(render,errmsg,legal)` aus Informationen wie das Widget dargestellt werden soll, der Fehlermeldung und der Bedingung, die überprüft werden soll.

show – liefert nun, angewendet auf ein `wparams` Tupel und einen Initialwert, das Tupel `(widget,readval,setval)`. Dabei ist `widget` das UI-Widget, das mit Hilfe der `render`-Funktion konstruiert wurde. Die Aktion `readval` liefert angewendet auf eine Umgebung den Wert des Widgets oder zeigt dem Benutzer eine Fehlermeldung an. Mit `setval` kann der Wert des Widgets neu gesetzt werden.

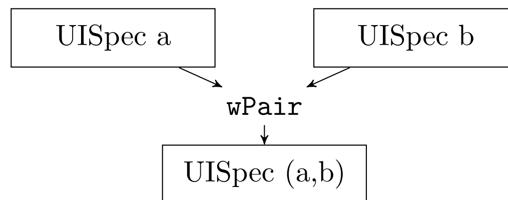

Abbildung 7.4: Kombination von Widgets mit den Typen `a`, `b` zu einem neuen Widget mit dem Typ `(a,b)`.

Abbildung 7.4 skizziert die Kombination zweier Widgets mit den Typen `UISpec a` und `UISpec b`. Wenn nun zwei konkrete Werte dieses Datentyps

```
UISpec uiparamsa showa
UISpec uiparamsb showb
```

sind, dann liefern die „show"-Funktionen Tripel der Form `(widgeta,reada,seta)` und `(widgetb,readb,setb)` mit:

```
reada :: UIEnv -> IO (Maybe a)
readb :: UIEnv -> IO (Maybe b)
```

Die beiden Widgets können mit der Funktion `wPair` zu einem neuen Widget kombiniert werden. Das kombinierte Widget hat eine neue `read`-Funktion, die auf `reada` und `readb` seiner Kindwidgets aufbaut. Zunächst prüft die `read`-Funktion, ob die Eingaben in den Kindwidgets korrekt sind, dann werden die Bedingungen für die Kombination getestet. Die `read`-Aktion eines mit `wPair` kombinierten Widgets kann wie in Listing 7.4 implementiert werden:

```
(render,errmsg,legal) = wparams

read env = do
  mba <- reada env
  mbb <- readb env
  if mba==Nothing || mbb==Nothing
    then do setValue errolabel "" env
            return Nothing
    else do
      let val = (fromJust mba,fromJust mbb)
      b <- legal val
      if b then do setValue errolabel "" env
                   return (Just val)
           else do setValue errolabel errmsg env
                   return Nothing
```

Listing 7.4: Kombination der `read`-Funktionen

Die `read`-Funktion versucht die beiden Werte der Kindwidgets einzulesen. Wenn einer der Werte `Nothing` ist, ist bei einem Kindwidget ein Fehler aufgetreten, also liefert auch `read` den Wert `Nothing`. Wenn jedoch die Bedingungen der Kindwidgets erfüllt waren, werden die Bedingungen des kombinierten Widgets überprüft. Dazu wird `legal` auf den kombinierten Wert der Kindwidgets angewendet. Ist die Bedingung nicht erfüllt, wird eine Fehlermeldung angezeigt, indem der Text eines Labels auf die Fehlermeldung `errmsg` gesetzt wird. Wenn die Bedingung erfüllt ist, wird der Wert des Widgets zurückgegeben. Nach diesem allgemeinen Konzept werden für alle Kombinatoren `read`-Funktionen erzeugt.

7.3 Beispiele

Listen von Widgets

Neben den Tupel-Kombinatoren gibt es in der UISpec-Bibliothek auch einen Kombinator, der Widgets gleichen Typs in einer Liste kombiniert. In Listing 7.5 wird eine Liste von `wInt`-Widgets zu einem neuen Widget kombiniert. Der Inhalt dieses Widgets ist eine Liste von ganzen Zahlen. Durch einen Klick auf den Button wird jedes Element dieser Liste inkrementiert.

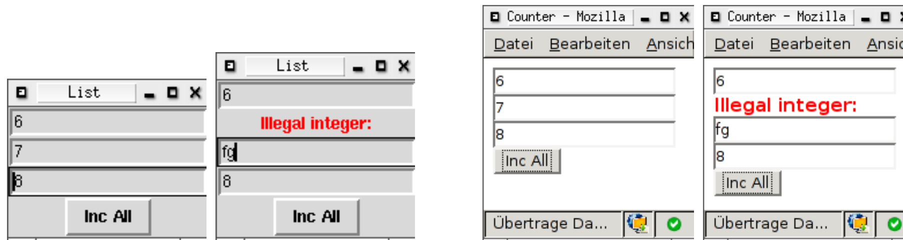

Abbildung 7.5: UI Anwendung mit einem `wList`-Widget als GUI und Webanwendung. Im Fehlerfall wird die Beschriftung eines Labels geändert.

```
ui2 = col [widget,button cmd "Inc All"]
  where
    (widget,_,_,updval) = uispec2ui (wList wInt) [1,2,3]

    cmd env =
      updval (\ (n1:n2:n3:[]) -> map (+1) [n1,n2,n3]) env
```

Listing 7.5: WUI Anwendung mit `wList`-Widget

Widgets mit Auswahlmöglichkeiten

In der UISpec-Bibliothek gibt es vordefinierte Widgets, mit denen der Benutzer Werte auswählen kann:

```
wSelect :: (a->String) -> [a] -> UISpec a}
```

stellt eine Liste mit Elementen vom Typ `a` als Auswahlliste dar. Die Einträge werden für die Darstellung mit der Funktion im ersten Argument in Strings konvertiert.

```
wMultiCheckSelect :: (a->[UIWidget]) -> [a] -> UISpec [a]
```

bietet die Möglichkeit einer Mehrfachauswahl durch Ckeckboxen. Die konkreten Werte werden auf Widgets abgebildet.

In Listing 7.6 wird ein Widget aus zwei Auswahllisten und einer Mehrfachauswahl kombiniert. Mit den Auswahllisten kann jeweils eine ganze Zahl ausgewählt werden und bei der Mehrfachauswahl durch Checkboxen können beliebig viele der angezeigten Zahlen ausgewählt werden. Damit die `store`-Funktion des Widgets ausgeführt wird, muss die Bedingung des Widgets erfüllt sein. Der Benutzer muss die Zahlen so wählen, dass die Summe der in der Auswahlliste angezeigten Zahlen gleich der Summe der mit den Checkboxen markierten Zahlen ist.

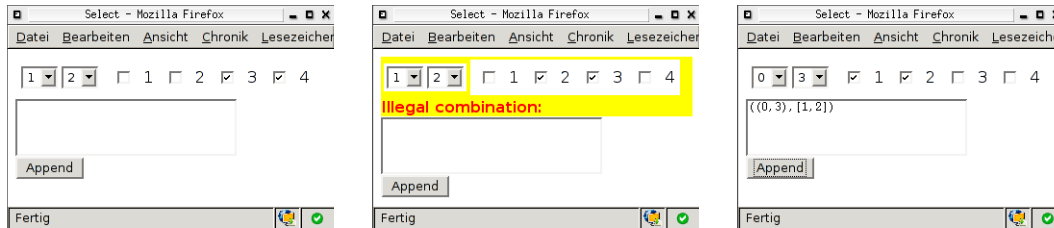

Abbildung 7.6: Widgets mit Auswahlmöglichkeiten

```
ui = col [widget,textEdit tref "" 3 30,button cmd "Append"]
  where
    tref free
    uispec = wPair
      (wPair (wSelect (\s -> show s) [0,1])
             (wSelect (\s -> show s) [2,3]))
      (wMultiCheckSelect (\s -> [label (show s)]) [1,2,3,4])
       `withCondition`
           (\((v1,v2),v3) -> v1+v2 == foldr (+) 0 v3)

    (widget,cmd) = typeduistore2ui uispec ((1,2),[3,4]) store

    store val env = appendValue tref (show val ++ "\n") env
```

Listing 7.6: UI mit Auswahlwidgets

`typeduistore2ui` liefert das Widget und einen Handler, der im Beispiel einem Button zugewiesen wird. Der Handler führt die `store`-Funktion nur aus, wenn alle Bedingungen erfüllt sind. Die `store`-Funktion gibt den Wert des Widgets im Textbereich aus.

Verwaltungssoftware

Ein Beispiel für eine größere typsichere Webanwendung ist eine Webanwendung, mit der Studiengänge verwaltet werden können. Diese Anwendung wurde ursprünglich mit der WUI-Bibliothek implementiert. Mit nur kleinen Änderungen im Quelltext kann aus ihr eine UI-Anwendung erstellt werden. Bei Fehleingaben wird dann keine neue Seite generiert, sondern die Fehler werden direkt angezeigt.

Modulverwaltung - Mozilla Firefox

Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe

Informatik I (Programmierung)

Anzeigen | Formatieren | Ändern | Semester hinzufügen | Semesterangaben ändern

Bitte auch die Hinweise zur Modulbeschreibung beachten!

Modul speichern | Abbrechen

Code: G1.1
Titel: Informatik I (Programmie
Verantwortlicher: Name: Prof. Dr. Michael Hanus Email: mh@informatik.uni-kiel.
Turnus: jedes Jahr Veranstaltungen: WS07/08 WS08/09 WS09/10
Präsenzzeiten (SWS): 4 Vorlesung 2 Übung 0 Praktikum 0 Seminar
ECTS-Punkte: 8
Dauer: 1 Semester
Modulkategorien: **Es muss mindestens eine Kategorie angegeben werden!**
G A WI A5 BA5 BA6 WA FD
Beschreibung: Lehrsprache:
Deutsch
Kurzfassung:
Dieses Modul gibt eine grundlegende Einführung in die Programmierung.
Hierbei werden verschiedene Abstraktionstechniken vorgestellt,
die es erlauben, die Komplexität großer Systeme unter Kontrolle
zu halten.

Fertig

Abbildung 7.7: Ausschnitt der Modulverwaltungsanwendung

8 Weitere Anwendungen der UI2HTML-Bibliothek

8.1 Einbettung von UIs in HTML

Mit `ui2hexps :: UIWidget -> [HtmlExp]` aus der UI2HTML-Bibliothek können UI-Beschreibungen in HTML-Ausdrücke konvertiert werden. Somit ist es möglich UIs in bestehende HTML-Anwendungen einzubetten, um diese für den Benutzer komfortabler zu machen.

Listing 8.1 beschreibt ein Bestellformular, in dem der Anwender seine Telefonnummer, die Bestellung und seine Adresse eingibt. Sobald das Feld mit der Telefonnummer verlassen wird, wird der Event-Handler `phoneHandler` ausgeführt. Dieser könnte beispielsweise anhand der Telefonnummer auf dem Server die zugehörige Adresse in einer Datenbank suchen und im Adressfeld anzeigen. Das Eingabefeld für die Telefonnummer ist ein Widget mit dem Handler

```
Handler FocusOut (Cmd phoneHandler)
```

Wenn der Anwender das Feld verlässt (`FocusOut`) wird auf dem Webserver die Funktion `phoneHandler` ausgeführt.

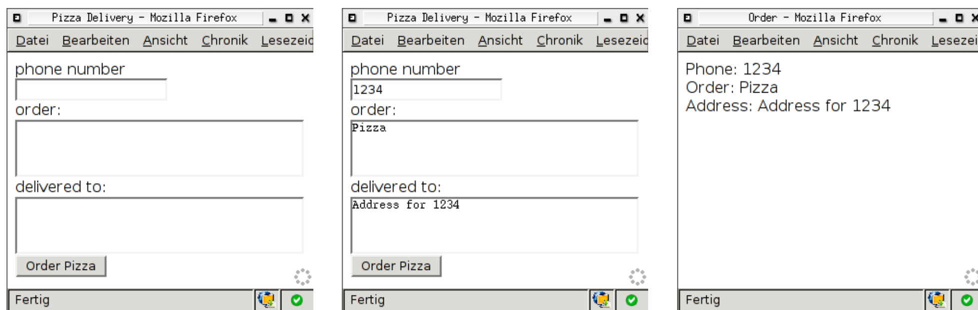

Abbildung 8.1: Sobald der Benutzer das Eingabefeld mit der Telefonnummer verlässt, wird nebenläufig mittels einer Serveranfrage das Adressfeld ausgefüllt.

```
col = map (\ x -> block [x])
```

```
main = return $ ajaxForm "Pizza Delivery" $ col [
  htxt "phone number", inline $ ui2hexps
    (entry phonenr ""
        `addHandler` (Handler FocusOut (Cmd phoneHandler))),
  htxt "order:", textarea order   (3,40) "",
  htxt "delivered to:", textarea address (3,40) "",
  HTML.button "Order Pizza" submitHandler]

 where
    phonenr, order, address free

    phoneHandler env = do
      p <- getValue phonenr env
      setValue (cgiRef2Ref address) ("Address for " ++ p) env

    submitHandler env = return $ ajaxForm "Order" $ col [
      htxt $ "Phone: "   ++ (env $ ref2cgiRef phonenr),
      htxt $ "Order: "   ++ (env order),
      htxt $ "Address: " ++ (env address) ]
```

Listing 8.1: Beschreibung eines Bestellformulars

Die Referenztypen der UI2HTML- und HTML-Bibliothek sind verschieden, daher werden zur Konvertierung von der UI2HTML-Bibliothek Funktionen angeboten.

```
ref2cgiRef :: UIRef -> CgiRef
cgiRef2Ref :: CgiRef -> UIRef
```

In Listing 8.1 wird zum Beispiel die Referenz des Adressfeldes mit „`cgiRef2Ref address`“ in eine UI-Referenz konvertiert.

8.2 Mehr Dynamik in Webanwendungen

Die bisher vorgestellten Event-Handler der UI2HTML-Bibliothek sind IO-Aktionen, die auf dem Server ausgeführt werden. Es ist jedoch auch möglich, andere Handler-Funktionen anzugeben, die beispielsweise direkt auf dem Client durch JavaScript ausgeführt werden. Dazu wird die SpicyWeb-Bibliothek benutzt.

8.2.1 SpicyWeb

Mit der SpicyWeb-Bibliothek [2] können in Curry clientseitig dynamische Webseiten definiert werden. Dazu wird in einem Curry-Programm der Aufbau eines HTML-Dokumentes und das Verhalten bei Benutzerevents beschrieben. Aus diesem Programm

kann eine konkrete HTML-Seite und JavaScript-Code erzeugt werden. Widgets werden ähnlich wie in der HTML-Bibliothek beschrieben und zusammengesetzt. Die Handler der SpicyWeb-Bibliothek sind jedoch keine Sequenz von IO-Aktionen, sondern werden mit Hilfe eines speziellen Datentyps dargestellt.

Für den Aufbau von Dokumenten gibt es in der SpicyWeb-Bibliothek den Datentyp `Doc`. Dieser repräsentiert ein HTML-Dokument.

```
data Doc = Elem String [(String,String)] Doc
         | Text String
         | WithID Doc ID
         | OnLoad Doc (Action ())
         | Group [Doc]
```

`Elem tag attrs doc` – für Elementknoten mit dem Tagnamen `tag`, Attribute `attrs` und einem Kinddokument `doc`.

`Text str` – für Textknoten.

`WithID doc id` – versieht ein Dokument mit einer Referenz.

`Group docs` – dient als Container für mehrere Dokumente

`OnLoad doc act` – weist einem Dokument eine Aktion zu. Diese wird in JavaScript-Code umgewandelt und direkt nach dem Laden der HTML-Seite ausgeführt.

SpicyWeb führt einige Abkürzungen für Dokumente ein. Den vordefinierten Elementen wird mit `iDoc` eine freie Variable als Referenz zugewiesen.

```
iDoc = (‘withID‘ freshID)
x <> y = iDoc (Group [x,y])
text s = iDoc (Text s)

textElem name attrs = iDoc . Elem name attrs . Text
para = iDoc . Elem "p" []
link = textElem "a" [("href",""),("onclick","return false;")]

yieldDoc d = yieldIDs [iD d]
```

Um die Aktionen unabhängig von JavaScript abstrakt zu definieren, gibt es in SpicyWeb den Datentyp `UntypedAction`:

```
data Action _ a = Action (UntypedAction a)

type UntypedEvent = String

data UntypedAction
  = Apply UntypedAction UntypedAction
```

```
    | Seq Bool UntypedAction UntypedAction
    | YieldID [ID]
    | Pure Json
    | Window
    | Event
    | Select UntypedAction String Bool
    | Curryfy Int UntypedAction
    | Observe Bool [ID] UntypedEvent UntypedAction
```

Jedem Dokument können Aktionen zugewiesen werden. Diese werden von der SpicyWeb-Bibliothek in JSON konvertiert und nach dem Laden der HTML-Seite von einer JavaScript-Funktion geparst und die entstehenden Aktionen werden ausgeführt. Grundsätzlich ist dieses Aktionenmodell unabhängig von JavaScript, jedoch ist es stark an JavaScript orientiert.

`Apply fun arg` – wendet die Funktion `fun` auf `arg` an.

`Seq first act1 act2` – führt zwei Aktionen hintereinander aus. Hat `first` den Wert `True` wird als Ergebnis der Wert von `act1` zurückgegeben, sonst der Wert von `act2`.

`YieldID ids` – liefert eine Liste von IDs.

`Pure json` – ist ein JSON-Objekt.

`Window` – entspricht dem `window`-Objekt von JavaScript.

`Event` – entspricht dem `event`-Objekt von JavaScript.

`Select obj prop bind` – wählt eine Eigenschaft (`prop`) eines JavaScript-Objektes (`obj`) aus. `bind` macht eine Aussage über den Gültigkeitsbereich.

`Curryfy arity fun` – ist eine partielle Applikation der Funktion `fun`.

`Observer once ids event act` – weist den HTML-Elementen mit den IDs `ids` für das Event (`event`) die Aktion (`act`) zu. Wird `once` auf `True` gesetzt, wird die Aktion nur beim ersten Auftreten des Events ausgeführt.

Um Aktionen übersichtlicher zu definieren, gibt es in SpicyWeb einige vordefinierte Abkürzungen.

```
Action f <*> Action a = Action (Apply f a)
Action a  *> Action b = Action (Seq False a b)
Action a <*  Action b = Action (Seq True a b)

displayAction = prim 1 "Prim.display"
```

```
hideAction    = prim 1 "Prim.hide"

display d = displayAction <*> yieldDoc d
hide    d = hideAction    <*> yieldDoc d

observeAll d = observe False [iD d]
onClick d a = d `onLoad` observeAll d "click" a
```

Listing 8.2 beschreibt eine dynamische HTML-Seite mit SpicyWeb. Wenn der Anwender auf den „hide“-Link klickt, wird der Abschnitt versteckt und ein „show“ -Link angezeigt. Wenn er auf diesen klickt, wird der Abschnitt wieder angezeigt.

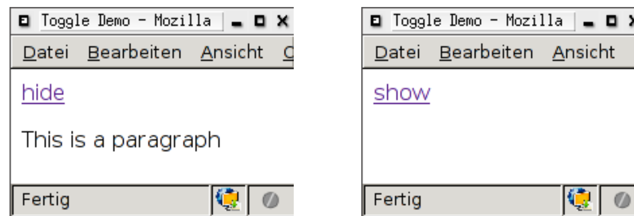

Abbildung 8.2: Zustände der aus Listing 8.2 entstandenen Webanwendung

```
doc = (more `onClick` display p *> hide more *> display less)
       <> (less `onClick` hide p *> display more *> hide less)
  where
    more = hidden (link "show")
    less = link "hide"
    p = para (text "This is a paragraph")
```

Listing 8.2: Toggle Beispiel in SpicyWeb

Den SpicyWeb-Dokumenten werden die Aktionen direkt zugewiesen wie in Listing 8.2 zu sehen ist. Wenn einer Gruppe von Dokumenten eine Aktion durch `onclick` zugewiesen wird, dann bedeutet dies, dass jedes Dokument der Gruppe bei einem Klick die gleiche Aktion ausführen soll. Um dies wirklich zu gewährleisten, werden die IDs entsprechend verwaltet.

```
txt = "Hide all blanks by clicking on one!"

doc = foldr1 (<>)
        (intersperse vanishingBlank (map text (words txt)))
 where
  vanishingBlank = text " " `onClick` hide vanishingBlank
```

Listing 8.3: Leerzeichen verstecken

In Listing 8.3 wird das gleiche Dokument (`vanishingBlank`) und damit die gleiche Referenz mehrmals im Dokument verwendet. Vor der Konvertierung nach HTML

bekommen die Dokumente daher neue Referenzen, denn HTML-Elemente müssen eindeutige IDs haben. Die JavaScript-Aktion enthält die Information, dass sie auf alle Elemente mit diesen IDs angewendet werden soll.

8.2.2 SpicyWeb für UIs

Der Konstruktor `Cmd` in der UI-Bibliothek wird für die Handlerfunktionen benutzt. Er enthält also Werte vom Typ „`UIEnv -> IO ()`“. Für Spicy-Aktionen wird nun ein weiterer Konstruktor `SpicyDoc` definiert.

```
data Command act1 act2 = Cmd act1
                       | SpicyDoc act2
```

Außerdem sind Dokumente, auf die die SpicyWeb-Aktionen angewendet werden in diesem Fall vom Typ `Doc r`, wobei `r` der Typ der Referenz ist.

```
data Doc r = Doc r
           | OnLoad (Doc r) (Action () r)
```

„`Doc ref`“ ist eine einfache Referenz und „`OnLoad doc act`“ weist einem Dokument eine Aktion zu.

Mit der Funktion `onClickSpicy` kann einem UI2HTML-Widget eine SpicyWeb-Aktion zugewiesen werden. Wenn das Widget keine Referenz besitzt, bekommt es als Referenz eine neue freie Variable zugewiesen. Mit Hilfe der Referenzen der Widgets können dann, ähnlich wie in der ursprünglichen SpicyWeb-Bibliothek, Aktionen definiert werden, die beim Klicken des Elementes ausgeführt werden sollen.

Die Funktion `widget2docs` filtert alle Spicy-Aktionen aus der UI-Beschreibung. Die Aktionen werden von der SpicyWeb-Bibliothek in einen JSON-String konvertiert. Dieser JSON-String wird von JavaScript direkt nach dem Laden der HTML-Seite interpretiert und ausgeführt. Im Gegensatz zur SpicyWeb-Bibliothek wird keine externe JavaScript-Datei erzeugt, sondern die Aktionen werden direkt dem `onload`-Attribut der HTML-Seite zugewiesen. Listing 8.4 zeigt eine UI2HTML-Anwendung, die SpicyWeb benutzt:

Abbildung 8.3: UI mit SpicyWeb

```
ui = col [
 simpleButton b1 "hide"
  ‘onClickSpicy‘
    (hide (Doc lref) *> display (Doc b2) *> hide (Doc b1)),
 simpleButton b2 "show" ‘addStyle‘ (Class [Display False])
  ‘onClickSpicy‘
    (display (Doc lref) *> hide (Doc b2) *> display (Doc b1)),
 label "This is a label" ‘setRef‘ lref]

 where val, incbutton, lref, b1, b2 free
```

Listing 8.4: „Toggle“-Beispiel als UI-Beschreibung mit SpicyWeb

9 Ähnliche Konzepte

In Abschnitt 9.1 werden Frameworks vorgestellt, die wie die UI2HTML-Bibliothek auf der Clientseite nur einen normalen Webbrowser benötigen. In Abschnitt 9.2 wird ein Überblick über Frameworks gegeben, die auf der Clientseite zusätzliche Laufzeitumgebungen benötigen.

9.1 Webanwendungen ohne Zusatzsoftware

Die Rich Ajax Platform

Mit dem *Standard Widget Toolkit*[1] (*SWT*) können in Java grafische Oberflächen beschrieben werden. Mit der *Rich AJAX Platform*[2] (*RAP*) können aus diesen Beschreibungen auch Webanwendungen generiert werden. Dazu werden die SWT-Widgets auf Widgets aus dem JavaScript-Framework *Qooxdoo*[3] abgebildet. Wie bei der UI2HTML-Bibliothek wird clientseitig HTML und JavaScript eingesetzt. Die Programmlogik läuft auf dem Webserver als Servlet.

Wie bei UI-Beschreibungen kann also aus der gleichen SWT-Beschreibung einerseits eine normale Desktopanwendung und andererseits eine Desktop-ähnliche Webanwendungen generiert werden. Wie jedoch bereits der einfache interaktive Zähler in Listing 9.1 zeigt, ist die SWT-Beschreibung unübersichtlich und die Beschreibung der Benutzerschnittstelle nicht so klar von der Programmlogik getrennt wie in der UI-Bibliothek. Ein Vorteil auf Java basierender Frameworks ist, dass es viele Hilfsmittel für die Entwicklung von Anwendungen gibt.

[1] `www.eclipse.org/swt/`
[2] `http://www.eclipse.org/rap/`
[3] `http://qooxdoo.org/`

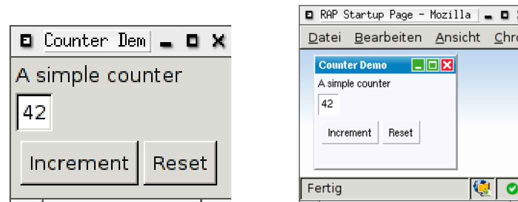

Abbildung 9.1: Interaktiver Zähler als Desktopanwendung (links) und als Webanwendung (rechts)

```
private Text edit1;

public int createUI() {
  Display display = new Display();
  Shell shell = new Shell(display);
  shell.setText("Counter Demo");
  shell.setLayout(new RowLayout(SWT.VERTICAL));

  Label label = new Label(shell, SWT.NONE);
  label.setText("A simple counter");

  edit1 = new Text(shell, SWT.BORDER);
  edit1.setText("42");

  Composite row = new Composite(shell,SWT.NONE);
  row.setLayout(new RowLayout(SWT.HORIZONTAL));

  Button incButton = new Button(row, SWT.PUSH);
  incButton.setText("Increment");

  Button resetButton = new Button(row, SWT.PUSH);
  resetButton.setText("Reset");

  incButton.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
      String str = edit1.getText();
      edit1.setText("" + (Integer.parseInt(str) + 1)); } });

  resetButton.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
      edit1.setText("0"); } });

  shell.setSize(150, 120);
  shell.open();
  while (!shell.isDisposed()) {
    if (!display.readAndDispatch())
      display.sleep(); }
  display.dispose();

  return 0;
}
```

Listing 9.1: Beschreibung eines interaktiven Zählers mit SWT

Mit RAP vergleichbar sind die Java-Frameworks Echo2[4] und wingS[5]. Beide Frameworks haben eine ähnliche API wie die Java Grafikbibliothek Swing. Für die Clientseite wird HTML und JavaScript generiert, so dass auf dem Client ein normaler Webbrowser ausreicht, um die Anwendung zu benutzen. Für die Serverseite werden Java-Servlets benutzt. Aus Echo2 und wingS Anwendungen können jedoch nicht direkt Swing-Anwendungen erzeugt werden, da die API nicht vollständig identisch ist.

Das Google Web Toolkit

Mit dem *Google Web Toolkit*[6] (*GWT*) können Benutzerschnittstellen und die Anwendungslogik mit Java implementiert und in eine Webanwendung übersetzt werden. Dabei wird JavaScript-Code für den Client generiert. Für die Serverseite wird Java-Code erzeugt, der als Servlet ausgeführt wird.

Mit dem GWT ist es auch möglich, clientseitig dynamische Webanwendungen zu generieren. Wenn nicht auf Daten auf dem Server zugegriffen werden muss, kann mit dem GWT aus Java-Code auch direkt JavaScript-Code erzeugt werden. Ein interaktiver Zähler kann daher mit dem GWT wie in Listing 9.2 beschrieben werden. Diese clientseitige Dynamik ist teilweise auch in Curry mit der in Abschnitt 8.2 vorgestellten SpicyWeb-Bibliothek möglich. Außerdem können mit dem Curry2JS-Compiler [8] aus Curry-Funktionen JavaScript-Funktionen generiert werden, jedoch ist dieses Konzept noch nicht in die UI2HTML-Bibliothek integriert worden.

```
public class Counter implements EntryPoint {
  public void onModuleLoad() {
    final Button incButton = new Button("Increment");
    final Button resetButton = new Button("Reset");
    final Label label = new Label("A Simple Counter");
    final TextBox textbox = new TextBox();
    textbox.setText("0");

    final VerticalPanel col = new VerticalPanel();
    final HorizontalPanel row = new HorizontalPanel();
    col.add(label); col.add(textbox); col.add(row);
    row.add(incButton); row.add(resetButton);

    incButton.addClickListener(new ClickListener() {
      public void onClick(Widget sender) {
        String str = textbox.getText();
        textbox.setText("" + (Integer.parseInt(str) + 1)); } });

    resetButton.addClickListener(new ClickListener() {
      public void onClick(Widget sender) {
        textbox.setText("0"); } });
```

[4] http://echo.nextapp.com/site/
[5] http://wingsframework.org
[6] http://code.google.com/webtoolkit/

```
    RootPanel.get("slot1").add(col);
  }
}
```

Listing 9.2: Interaktiver Zähler ohne Serveranfragen

Üblicherweise kommunizieren Webanwendungen mit einem Webserver, um beispielsweise Datenbanken anzufragen und zu ändern. In Anhang C wird ein interaktiver Zähler als GWT-Anwendung implementiert, dessen Handlerfunktionen auf dem Server ausgeführt werden.

ZK

ZK[7] ist ein Java-Framework zur Beschreibung von Webanwendungen. Benutzerschnittstellen können mit der Auszeichnungssprache *ZUML* (*ZK User Interface Markup Language*) beschrieben werden. Für die Clientseite wird HTML- und JavaScript-Code generiert und auf der Serverseite Servlets eingesetzt. Das ZK-Framework hat damit eine ähnliche Funktionalität wie die UI2HTML-Bibliothek. Die Beschreibung der Benutzerschnittstelle wird klar von der Programmlogik getrennt. Listing 9.3 beschreibt einen interaktiven Zähler mit dem ZK-Framework.

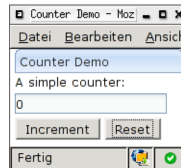

Abbildung 9.2: Interaktiver Zähler mit ZK

```
<?page title="Counter Demo"?>
<window title="Counter Demo">
<zscript>
  inc() { edit1.value = "" + (Integer.parseInt(edit1.value) + 1); }
  reset() { edit1.value = "0"; }
</zscript>
<vbox>
  <label value="A simple counter:" />
  <textbox id="edit1" value="0" />
  <hbox>
    <button label="Increment" onClick="inc()" />
    <button label="Reset" onClick="reset()" />
  </hbox>
</vbox>
</window>
```

Listing 9.3: ZK-Beschreibung eines interaktiven Zählers

[7] http://www.zkoss.org/

9.2 Webanwendungen mit Zusatzsoftware

Bei allen in diesem Abschnitt beschriebenen Frameworks läuft die Anwendungslogik hauptsächlich auf der Clientseite ab. Neben einem Browser werden Laufzeitumgebungen zum Beispiel in Form von Browser-Plugins benötigt. Die Kommunikation mit dem Webserver wird von den Frameworks vereinfacht, aber nicht vollkommen abstrahiert. Die vorgestellten Frameworks haben somit nur wenig Ähnlichkeit mit der UI-Bibliothek. Eine wesentliche Gemeinsamkeit ist jedoch, dass die Anwendungslogik von der GUI-Beschreibung getrennt wird.

Adobe Integrated Runtime

Mit *AIR*[8] (*Adobe Integrated Runtime*) möchte Adobe die Vorteile von Desktop- und Webanwendungen vereinigen. AIR-Anwendungen unterstützen viele Adobe-Standards wie Flash, Flex und PDF. Listing 9.4 beschreibt einen clientseitigen interaktiven Zähler mit Adobe Flex. Aus dieser Beschreibung kann eine „Adobe“-Flash Anwendung wie in Abbildung 9.3 generiert werden.

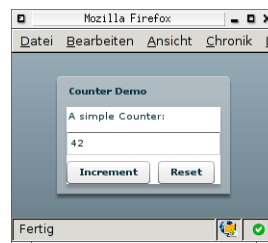

Abbildung 9.3: Interaktiver Zähler als Flash-Anwendung

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  viewSourceURL="src/HandlingEventsEventHandler/index.html" >
<mx:Script>
  import flash.events.MouseEvent;

  private function inc(event:MouseEvent):void {
    edit1.text = "" + (parseInt(edit1.text) + 1);  }

  private function reset(event:MouseEvent):void { edit1.text = "0"; }
</mx:Script>
<mx:Panel title="Counter Demo">
  <mx:VBox>
    <mx:Label text="A simple Counter:" />
    <mx:TextInput id="edit1" text="0" />
    <mx:HBox>
```

[8] http://www.adobe.com/de/products/air/

```
        <mx:Button label="Increment" click="inc(event);" />
        <mx:Button label="Reset" click="reset(event);" />
      </mx:HBox>
    </mx:VBox>
  </mx:Panel>
</mx:Application>
```

Listing 9.4: Adobe Flex-Beschreibung eines interaktiven Zählers

Die Windows Presentation Platform

Die *Windows Presentation Platform*[9] (*WPF*) ist Teil des .NET 3.0 Frameworks von Microsoft. Grafische Oberflächen können in WPF-Anwendungen mit der Auszeichnungssprache XAML (*eXtensible Application Markup Language*) beschrieben werden. Die Anwendungslogik wird in einer .NET kompatiblen Programmiersprache wie C# geschrieben.

XBAPs (*XAML Browser Applications*) sind WPF-Programme, die scheinbar im Browser ausgeführt werden. Der Anwender klickt auf einen Link zu einer `*.xbap`-Datei. Die Datei wird heruntergeladen und gestartet. Gegenüber WPF-Anwendungen haben XBAPs aus Sicherheitsgründen einige Einschränkungen. Direkte Dateizugriffe oder andere kritische Operationen sind nicht erlaubt.

„Microsoft Silverlight"-Anwendungen sind WPF-Programme, die im Gegensatz zu XBAPs direkt im Browser ablaufen. Es ist jedoch, wie bei Adobe Flash, ein extra Plugin erforderlich.

JavaFX

JavaFX[10] ist ein Java-Framework von Sun Microsystems, das ähnliche Ziele wie Adobe Flash und Microsoft Silverlight hat. Als Beschreibungssprache für grafische Oberflächen benutzt JavaFX die Skriptsprache *JavaFX Script*. Listing 9.5 beschreibt eine Swing-Anwendung mit JavaFX Script.

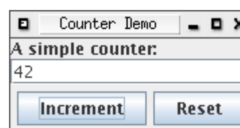

Abbildung 9.4: Interaktiver Zähler als Swing-Anwendung aus der JavaFX-Beschreibung

[9] http://windowsclient.net/
[10] http://www.javafx.com/

```
var edit1 = TextField { text: "0" }

var incButton = Button {
  text: 'Increment'
  action: function(): Void {
    edit1.text = "" + (Integer.parseInt(edit1.text ) + 1); }}

var resetButton = Button {
  text: 'Reset'
  action: function(): Void {
    edit1.text = "0"; } }

SwingFrame {
  title: 'Counter Demo'
  visible: true
  content:  BorderPanel {
    top: Label {text: "A simple counter:"};
    center: edit1;
    bottom: FlowPanel { content: [incButton, resetButton] }}}
```

Listing 9.5: JavaFX-Beschreibung eines interaktiven Zählers

10 Zusammenfassung

Die in dieser Diplomarbeit vorgestellten Konzepte ermöglichen eine allgemeine deklarative Konstruktion von Benutzerschnittstellen. In der vorgestellten Beschreibung von UIs können Struktur, Funktionalität und Darstellung getrennt voneinander angegeben werden. Dadurch wird der Quelltext übersichtlicher und die einzelnen Komponenten leichter wiederverwendbar. In den Handler-Funktionen werden logische Variablen verwendet, um auf den Inhalt von Widgets zugreifen zu können. Dadurch werden einerseits Tippfehler bereits beim Compilieren erkannt und andererseits Namenskonflikte beim Zusammenfügen mehrerer UIs vermieden. Weiterhin können in UI-Beschreibungen typsichere Widgets benutzt werden und mit Kombinatoren zu komplexen Widgets zusammengesetzt werden.

Es wurden Bibliotheken bereitgestellt, mit denen aus dieser allgemeinen Beschreibung sowohl Desktopanwendungen als auch Webanwendungen generiert werden können. Außerdem ist es möglich, direkt aus bestehende GUI-Beschreibungen Webanwendungen zu erzeugen.

Akronyme

Ajax – Asynchronous JavaScript and XML

CSS – Cascading Style Sheets

DOM – Document Object Model

GUI – Graphical User Interface

HTML – Hypertext Markup Language

JSON – JavaScript Object Notation

PAKCS – Portland Aachen Kiel Curry System

UI – User Interface

W3C – World Wide Web Consortium

XML – Extensible Markup Language

A Inhalt der CD

UI-Bibliotheken

`UI.curry`	Interface für UI-Beschreibungen
`UI2GUI.curry`	UIs als Desktopanwendung
`UI2HTML.curry`	UIs als Webanwendung
`ajaxrequest.js`	Ajax-Anfragen und DOM-Manipulation für die UI2HTML-Bibliothek
`GUI2HTML.curry`	GUIs als Webanwendung
`TypedUI2HTML.curry`	typsichere UIs als Webanwendung
`TypedUI2GUI.curry`	typsichere UIs als Desktopanwendung
`GUI.curry`	leicht veränderte GUI-Bibliothek aus PAKCS
`HTML.curry`	um Ajax-Funktionalität erweiterte HTML-Bibliothek aus PAKCS
`SpicyWeb.curry`	leicht veränderte SpicyWeb-Bibliothek
`Json.curry`	JSON-Bibliothek

Beispiele

`examples/ui2gui`	Beispiele für UIs als Desktopanwendungen
`examples/ui2html`	Beispiele für UIs als Webanwendungen
`examples/gui2html`	GUI-Beispiele aus PAKCS als Webanwendungen
`examples/browser`	der CurryBrowser als Webanwendung
`examples/masterdb`	die Mastermodulverwaltung mit Ajax Unterstützung
`examples/spicy`	UIs mit SpicyWeb-Funktionalität
`examples/typed`	typsichere UIs
`examples/dipl`	einige Beispiele aus der Diplomarbeit

B UI Beispiele

Taschenrechner

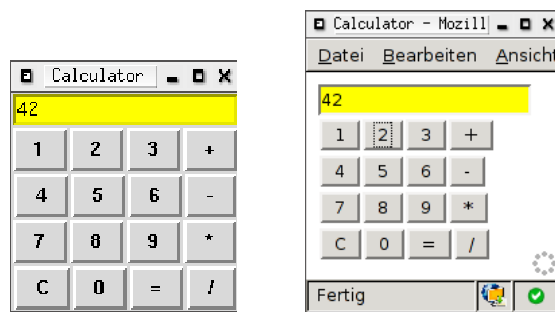

Abbildung B.1: Taschenrechner als Desktopanwendung (links) und Webanwendung (rechts)

```
import UI2HTML
import Char
import IOExts -- use IORefs for the GUI state

-- the GUI needs a reference to the calculator state
calc_GUI stateref =
  col [
    entryS [Class [Bg Yellow]] display "0",
    row (map cbutton ['1','2','3','+']),
    row (map cbutton ['4','5','6','-']),
    row (map cbutton ['7','8','9','*']),
    row (map cbutton ['C','0','=','/'])]
  where
    display free

    cbutton c = button (button_pressed c) [c]

    button_pressed c gp = do
       state <- readIORef stateref
       let (d,f) = processButton c state
       writeIORef stateref (d,f)
```

```
        setValue display (show d) gp

-- compute new state when a button is pressed:
processButton b (d,f)
 | isDigit b = (10*d + ord b - ord '0', f)
 | b=='+' = (0,((f d) + ))
 | b=='-' = (0,((f d) - ))
 | b=='*' = (0,((f d) * ))
 | b=='/' = (0,((f d) `div` ))
 | b=='=' = (f d, id)
 | b=='C' = (0, id)

main = do
  stateref <- newIORef (0,id)
  runUI "Calculator" (calc_GUI stateref)
```

Listing B.1: UI-Beschreibung eines Taschenrechners

Menü

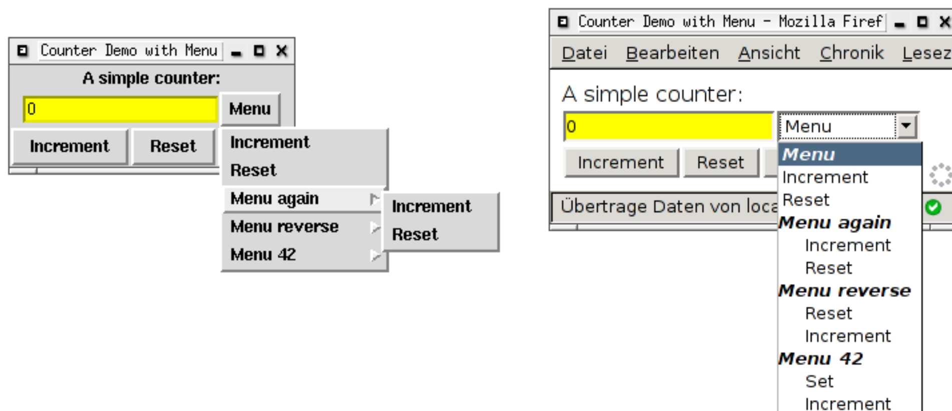

Abbildung B.2: Interaktiver Zähler mit Menü als Desktopanwendung (links) und Webanwendung (rechts)

```
import UI2HTML
import Read

counterGUI =
 col [
   label "A simple counter:",
   row
```

```
    [entryS [Class [Bg Yellow]] val "0",
     menuBar [
       menu "Menu"
         [menuItem increment "Increment",
          menuItem (setTo 0) "Reset",
          menu "Menu again"
            [menuItem increment "Increment",
             menuItem (setTo 0) "Reset"],
          menu "Menu reverse"
            [menuItem (setTo 0) "Reset",
             menuItem increment "Increment"],
          menu "Menu 42"
            [menuItem (setTo 42) "Set",
             menuItem increment "Increment"]]]],
   row [button increment "Increment",
        button (setTo 0) "Reset",
        button exitUI "Stop"]]
 where
   val free

   increment :: UIEnv -> IO ()
   increment gport = do
     updateValue incrText val gport

   setTo n gport = setValue val (show n) gport

incrText s = show (readInt s + 1)

main = runUI "Counter Demo with Menu" counterGUI
```

Listing B.2: UI-Beschreibung mit Menü

C Das Google Web Toolkit

In Abschnitt 9.1 wurde eine clientseitige Anwendung mit GWT vorgestellt. In diesem Abschnitt wird ein interaktiver Zähler serverseitig implementiert.

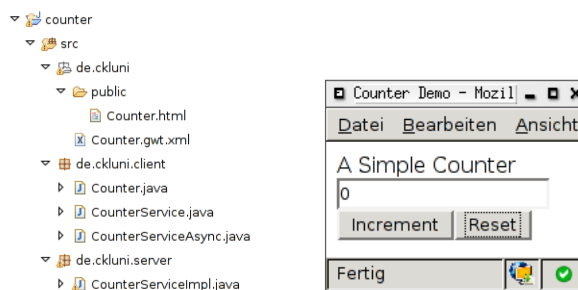

Abbildung C.1: Dateien für die Zähleranwendung in Eclipse

GWT stellt einige Programme bereit, mit der die Struktur einer GWT-Anwendung automatisch generiert wird. Mit der Option `eclipse` können auch Eclipse[1]-Projekte erzeugt werden.

```
projectCreator -eclipse counter
applicationCreator -eclipse counter de.ckluni.client.Counter
```

In Listing C.1 wird eine Webseite beschrieben, auf der das von GWT erzeugte Widget eingebettet wird. Die Oberfläche des interaktiven Zählers wird in das `span`-Element mit der ID `slot1` eingefügt.

In Listing C.2 wird die Clientseite der Anwendung definiert. Dabei werden die Widgets definiert und mit Layoutmanagern kombiniert. Dem „Increment“- und „Reset“-Button wird mit der Methode `addClickListener` jeweils einen Event-Handler zugewiesen.

```
<html>
  <head>
    <title>Counter Demo</title>
    <script language='javascript' src='de.ckluni.Counter.nocache.js'>
    </script>
  </head>
  <body><span id="slot1"></span></body>
```

[1] www.eclipse.org

```
</html>
```

Listing C.1: Counter.html

```
public class Counter implements EntryPoint {

  public void onModuleLoad() {
    final Button incButton = new Button("Increment");
    final Button resetButton = new Button("Reset");
    final Label label = new Label("A Simple Counter");
    final TextBox textbox = new TextBox();
    textbox.setText("0");
    final VerticalPanel col = new VerticalPanel();
    final HorizontalPanel row = new HorizontalPanel();

    row.add(incButton); row.add(resetButton);
    col.add(label); col.add(textbox); col.add(row);

    final CounterServiceAsync counterService =
                                      (CounterServiceAsync) GWT
    create(CounterService.class);
    ServiceDefTarget target = (ServiceDefTarget) counterService;
    String relativUrl = GWT.getModuleBaseURL() + "counter";
    target.setServiceEntryPoint(relativUrl);

    incButton.addClickListener(new ClickListener() {
      public void onClick(Widget sender) {
        String str = textbox.getText();
        counterService.increment(str, new AsyncCallback() {
          public void onFailure(Throwable caught) { }
          public void onSuccess(Object result) {
            textbox.setText((String) result); } }); } });

    resetButton.addClickListener(new ClickListener() {
      public void onClick(Widget sender) {
        counterService.reset(new AsyncCallback() {
          public void onFailure(Throwable caught) {}
          public void onSuccess(Object result) {
            textbox.setText((String) result); } }); } });

    RootPanel.get("slot1").add(col);
  }
}
```

Listing C.2: Interaktiver Zähler, dessen Handler Serveranfragen stellen

Auf der Serverseite wird eine Unterklasse von `RemoteServiceServlet` erstellt, die die Funktionalität des interaktiven Zählers, also die Event-Handler, implementiert.

```
public class CounterServiceImpl extends RemoteServiceServlet
                                implements CounterService    {
  public String increment(String str) {
    int n = Integer.parseInt(str);
    return "" + (n+1); }

  public String reset() { return "0"; }
```

```
}
```

Listing C.3: CounterServiceImpl.java

Um die Client-Klasse mit der Server-Klasse zu verbinden, müssen weitere Schnittstellen definiert werden.

```
public interface CounterService extends RemoteService {
  public String increment(String n);
  public String reset();
}
```

Listing C.4: CounterService.java

Außerdem muss eine asynchrone Version davon definiert werden. Der Klassenname ist der gleiche, mit der Endung `Async`. Für jede Version des originalen Interfaces gibt es jetzt eine Version, die den Rückgabetyp `void` hat und eine zusätzlichen Parameter vom Typ `AsyncCallback`. Der Client benutzt `AsyncCallback`-Objekte, um auf die Antwort des Servers zu reagieren.

```
public interface CounterServiceAsync {
  public void increment(String n,AsyncCallback callback);
  public void reset(AsyncCallback callback);
}
```

Listing C.5: CounterServiceAsync.java

Für das Servlet muss auf der Serverseite noch eine `web.xml` Datei mit einigen Konfigurationen angelegt werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
 <servlet>
  <servlet-name>CounterService</servlet-name>
  <servlet-class>de.ckluni.server.CounterServiceImpl</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>CounterService</servlet-name>
  <url-pattern>/counter</url-pattern>
 </servlet-mapping>
</web-app>
```

Listing C.6: web.xml

Literaturverzeichnis

[1] S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 67–87. Springer LNCS 2441, 2002.

[2] S. Fischer. Spicy Web.
`http://wiki.curry-language.org/libraries/spicy_coffee`, 2008.

[3] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *Proc. of the Second International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.

[4] M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

[5] M. Hanus. CurryDoc: A Documentation Tool for Declarative Programs. In *Proc. of the 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, pages 225–228, Grado (Italy), 2002. Research Report UDMI/18/2002/RR, Università degli Studi di Udine.

[6] M. Hanus. Currybrowser: A generic analysis environment for curry programs. In *Proc. of the 16th Workshop on Logic-based Methods in Programming Environments (WLPE'06)*, pages 61–74, 2006.

[7] M. Hanus. Type-Oriented Construction of Web User Interfaces. In *Proc. of the 8th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.

[8] M. Hanus. Putting Declarative Programming into the Web: Translating Curry to JavaScript. In *Proc. of the 9th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'07)*, pages 155–166. ACM Press, 2007.

[9] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2008.

[10] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry/report.html`, 2006.

[11] P.P. Koch. *PPK on JavaScript*. New Riders, 2007.

[12] B. McLaughlin. *Head Rush Ajax*. O'Reilly Media, Inc., 2006.

[13] S. Münz. SELFHTML. `http://de.selfhtml.org/`.

[14] John K. Ousterhout. Tcl: an embeddable command language. In *USENIX Winter Conference*, pages 183–192, January 1990.

[15] John K. Ousterhout. An X11 toolkit based on the Tcl Language. In *USENIX Winter Conference*, pages 105–115, January 1991.

[16] B.B. Welch, J. Hobbs, and K. Jones. *Practical programming in Tcl and Tk*. Prentice Hall Upper Saddle River, NJ, 2000.