High-Level Server Side Web Scripting in

Michael Hanus

Christian-Albrechts-Universität Kiel
Early days of the World Wide Web: web pages with static contents

Common Gateway Interface (CGI): web pages with dynamic contents

Retrieval of a dynamic page:

- server executes a program
- program computes an HTML string, writes it to stdout
- server sends result back to client

HTML with input elements (forms):

- client fills out input elements
- input values are sent to server
- server program decodes input values for computing its answer
CGI programs on the server can be written in any programming language

- access to environment variables (for input values)
- writes a string to stdout

**Scripting languages:** (Perl, Tcl,...)

- simple programming of single pages
- error-prone: correctness of HTML result not ensured
- difficult programming of interaction sequences

**Specialized languages:** (MAWL, DynDoc,...)

- HTML support (structure checking)
- interaction support (partially)
- restricted or connection to existing languages
Library in multi-paradigm language

Exploit functional and logic features for

- HTML support (data type for HTML structures)
- simple access to input values (free variables and environments)
- simple programming of interactions (event handlers)
- wrapper for hiding details

Exploit imperative features for

- environment access (files, data bases, . . .)

Domain-specific language for HTML/CGI programming
Curry

[Dagstuhl’96, POPL’97]

- multi-paradigm language
  (higher-order concurrent functional logic language, features for high-level distributed programming)
- extension of Haskell (non-strict functional language)
- developed by an international initiative
- provide a standard for functional logic languages
  (research, teaching, application)
- several implementations available
Values in imperative languages: basic types + pointer structures

Declarative languages: **algebraic data types** (Haskell-like syntax)

```haskell
data Bool   = True    | False
data Nat    = Z       | S Nat
data List a = []      | a : List a  -- [a]
data Tree a = Leaf a  | Node [Tree a]
data Int    = 0       | 1       | -1      | 2       | -2      | ... 
```

**Value** \(\approx\) **data term, constructor term**: well-formed expression containing variables and data type constructors

\[
(S \, Z) \quad 1:(2:[]) \quad [1,2] \quad \text{Node} \ [\text{Leaf} \ 3, \text{Node} \ [\text{Leaf} \ 4, \text{Leaf} \ 5]]
\]
Functions: operations on values defined by equations (or rules)

\[ f(t_1 \ldots t_n | c = r) \]

```
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys
last xs | conc ys [x] =:= xs
         = x

where x,ys free
```

```
last [1,2] \rightsquigarrow 2
```
\[ e ::= \\
\text{(constants)} \\
\text{(variables } x \text{)} \\
\text{(application)} \\
\text{(abstraction)} \\
\text{(conditional)} \]
EXPRESSIONS

\[ e ::= \]

\[ c \] (constants)

\[ x \] (variables \( x \))

\[ (e_0 \; e_1 \ldots e_n) \] (application)

\[ \langle x \rightarrow e \rangle \] (abstraction)

\[ \text{if } b \; \text{then } e_1 \; \text{else } e_2 \] (conditional)

\[ e_1 =:= e_2 \] (equational constraint)

\[ e_1 & e_2 \] (concurrent conjunction)

\[ \text{let } x_1, \ldots, x_n \; \text{free in } e \] (existential quantification)
expressions

e ::= 

c (constants)
x (variables x)
(e0 e1 ... en) (application)
\x -> e (abstraction)
if b then e1 else e2 (conditional)
e1 := e2 (equational constraint)
e1 & e2 (concurrent conjunction)
let x1, ..., xn free in e (existential quantification)

Equational constraints over functional expressions:

conc ys [x] := [1,2] ⇒ {ys=[1], x=2}

Further constraints: real arithmetic, finite domain, ports
• lazy evaluation (evaluate only *needed* redexes)
• support infinite data structures, modularity
• optimal evaluation (also for *logic programming*)

**Distinguish:**

*flexible* (generator) and *rigid* (consumer) functions

Flexible functions $\sim \text{logic programming}$

Rigid functions $\sim \text{concurrent programming}$
FLEXIBLE VS. RIGID FUNCTIONS

\[
\begin{align*}
f 0 &= 2 \\
f 1 &= 3
\end{align*}
\]

rigid/flexible status not relevant for ground calls:

\[
f 1 \leadsto 3
\]

\[f\] flexible:

\[
f x =:= y \leadsto \{x=0, y=2\} \mid \{x=1, y=3\}
\]

\[f\] rigid:

\[
f x =:= y \leadsto \text{suspend}
\]

\[
f x =:= y \land x =:= 1 \leadsto \{x=1\} f 1 =:= y \quad \text{(suspend } f \ x) \]

\[
\leadsto \{x=1\} 3 =:= y \quad \text{(evaluate } f \ 1) \\
\leadsto \{x=1, y=3\}
\]

Default in Curry: constraints are flexible, all others are rigid
Data type for representing HTML expressions:

```haskell
data HtmExp = HText String
            | HStruct String [(String,String)] HtmExp
```

Some useful abbreviations:

- `htxt s = HText (htmlQuote s)`       -- plain string
- `bold hexps = HStruct "B" [] hexps`   -- bold font
- `italic hexps = HStruct "I" [] hexps` -- italic font
- `h1 hexps = HStruct "H1" [] hexps`   -- main header

Example: `[h1 [htxt "1. Hello World"],
italic [htxt "Hello"], bold [htxt "world!"]]

∽ 1. Hello World
   Hello world!
Advantages:

- static checking of HTML structure (well-balanced parentheses)
- flexible dynamic documents
- functions for computing HTML documents

Converting tree structure (leaves contain strings) into nested HTML lists:

```haskell
data Tree a = Leaf a | Node [Tree a]

htmlTree :: Tree String -> [HtmlExp]
htmlTree (Leaf s) = [htxt s]
htmlTree (Node trees) = [ulist (map htmlTree trees)]

ulist :: [[HtmlExp]] -> HtmlExp
ulist items = HStruct "UL" [] (map litem items)
litem hexps = HStruct "LI" [] hexps
```
Specific HTML elements for dealing with user input

```html
<INPUT TYPE="TEXT" NAME="INPTEXT" VALUE="fill out!"> 
```

Form is submitted

```html
clients sends the current value of this field (identified by "INPTEXT") 
```

Expressible as HTML term:

```html
HStruct "INPUT" [("TYPE","TEXT"),("NAME","INPTEXT"),
("VALUE","fill out!")] []
```

Problems:

- server program must decode input values
- server program must know right names of field identifiers ("INPTEXT")
- error-prone
Solution:

- use free variables as references to input fields (CGI references)
- collect input values in CGI environments:
  - mapping from CGI references to strings
- associate event handlers to submit buttons
- event handlers take a CGI environment and produce an HTML form

Implementation:

straightforward in a functional logic language!
**Abstract Input Forms: Implementation**

**CGI references:**

```
data CgiRef = CgiRef String  -- data constructor not exported
```

→ no construction of wrong references
→ only free variables of type CgiRef
→ global wrapper function instantiates with the right strings

**HTML elements with CGI references:**

```
data HtmlExp = ... | HtmlCRef HtmlExp CgiRef
```

**Example:** Text fields with a CGI reference and initial contents

```
textfield :: CgiRef -> String -> HtmlExp
textfield (CgiRef ref) contents =
  HtmlCRef (HStruct "INPUT" [('TYPE","TEXT"),
                             ('NAME',ref),('VALUE',contents)])
  (CgiRef ref)
```
HTML form: title + list of HTML expressions
data HtmlForm = Form String [HtmlExp]

Example: simple form with a single input element (a text field)
Form "Form" [h1 [htxt "A Simple Form"],
    htxt "Enter a string:", textfield sref ""]

CGI environments: map CGI references to strings
type CgiEnv = CgiRef -> String

Event handlers have type CgiEnv -> IO Form
Event handlers are associated to submit buttons:
user presses a submit button
⇒ execute associated event handler with current environment
EXAMPLE: FORM TO REVERSE/DUPLICATE A STRING

Form "Question" [htxt "Enter a string: ", textfield tref "", hr, button "Reverse string" revhandler, button "Duplicate string" duphandler]

where tref free

revhandler env = return $ Form "Answer"
    [h1 [htxt ("Reversed input: " ++ rev (env tref))]]

duphandler env = return $ Form "Answer"
    [h1 [htxt ("Duplicated input: " ++ env tref ++ env tref)]]
Form to show the contents of an arbitrary file stored at the server:

Form "Get File" [htxt "Enter local file name:",
    textfield fileref ",",
    button "Get file!" handler]

where fileref free

    handler env =
    do contents <- readFile (env fileref)
    return $ Form "Answer"
        [h1 [htxt ("Contents of file " ++ env fileref)],
         verbatim contents]
Sequence of forms to collect first and last name:

Form "First Name Form"

\[\text{htxt } "\text{Enter your first name: }", \text{textfield first } "", \text{button } "\text{Continue}" \ \text{fhandler}\]

where \textit{first} free

\textit{fhandler} _ = 
\text{return } \$ \text{ Form } "\text{Last Name Form}"

\[\text{htxt } "\text{Enter your last name: }", \text{textfield last } "", \text{button } "\text{Continue}" \ \text{lhandler}\]

where \textit{last} free

\textit{lhandler} \ \text{env} = \text{return } \$ \text{ Form } "\text{Answer}"

\[\text{htxt } ("\text{Hi, }" + \text{env first} + " " + \text{env last})\]
Programming arbitrary loops: a number guessing game:

guessform = return $ Form "Number Guessing" guessinput

guessinput =
  [htxt "Guess a number: ", textfield nref ",",
   button "Check" (guesshandler nref)] where nref free

guesshandler nref env =
  let nr = readInt (env nref)
in return $ Form "Answer"
  (if nr==42
    then [htxt "Right!"]
    else [htxt (if nr<42 then "Too small!" else "Too large!"),
         hrule] ++ guessinput)
Abstraction: HTML element for looking up email addresses:

```haskell
mail_epi = [hTxt "Enter a name: ", textfield nref ",",
button "search email" lookup, hrule]

where nref free

    lookup env = ... send (GetEmail (env nref)) ...
```

Now, `mail_epi` can be used as any other HTML element (without name conflicts with other form elements!):

```haskell
[... , textfield nref ",", hrule] ++ mail_epi ++ ...
```
The main form is executed by a wrapper function

\[
\text{run\text{\texttt{cgi}} :: String} \rightarrow \text{IO HtmlForm} \rightarrow \text{IO ()}
\]

- takes a title string and a form and transforms it into HTML text
- replaces all CGI references by unique strings
- decodes input values and invokes associated event handler

Event handlers return forms rather than HTML expressions
- sequences of interactions
- use control abstractions (branching, recursion) of underlying language
- state between interactions handled by CGI environments

Note: no language extension necessary (CGI library)
- multi-paradigm languages as scripting languages
IMPLEMENTATION

→ completely implemented in Curry
→ standard CGI programming features used
→ no server extension, usable with any standard web server, no cookies
→ available as library for
  PAKCS (Portland Aachen Kiel Curry System)
  http://www.informatik.uni-kiel.de/~pakcs
→ based on a Curry→Prolog compiler [Antoy/Hanus FroCoS’00]

Applications:
→ web pages for Curry
→ access to distributed address server [PPDP’99]
→ submission form for JFLP
  (Journal of Functional and Logic Programming)
→ questionnaires for students
→ testing home assignments of students
→ ...
CONCLUSIONS

Domain-specific language for HTML/CGI programming (CGI library)

Exploit functional and logic features for
- correct HTML coding (data type for HTML structures)
- simple access to input values (free variables and environments)
- simple programming of interactions (event handlers)
- wrapper for hiding details

Curry supports appropriate abstractions for software development

Other examples:
- GUI programming [PADL’00]
- FL parser combinators [Caballero/Lopez-Fraguas FLOPS’99]

More infos on Curry:
http://www.informatik.uni-kiel.de/~curry