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, databases,...)

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)

\[
\begin{align*}
\text{data } \text{Bool} & = \text{True} \mid \text{False} \\
\text{data } \text{Nat} & = \text{Z} \mid \text{S} \text{ Nat} \\
\text{data } \text{List} \ a & = [] \mid a : \text{List} \ a \quad -- [a] \\
\text{data } \text{Tree} \ a & = \text{Leaf} \ a \mid \text{Node} \ [\text{Tree} \ a] \\
\text{data } \text{Int} & = 0 \mid 1 \mid -1 \mid 2 \mid -2 \mid \ldots
\end{align*}
\]

**Value** $\simeq$ **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)

\[ f \ t_1 \ldots t_n \mid c = r \]

---

\[
\begin{align*}
\text{conc} & \quad \text{[]} \quad \text{ys} = \text{ys} \\
\text{conc} & \quad (x:xs) \quad \text{ys} = x : \text{conc} \ x \ x \ s \ y s \\
\text{last} & \quad x s \mid \text{conc} \ y s \ [x] =:= x s \\
& \quad = x \\
\text{where} & \quad x, \ y s \ \text{free}
\end{align*}
\]

last \ [1,2] \ \mapsto \ 2
\[ e ::= \]

\[ c \quad \text{(constants)} \]

\[ x \quad \text{(variables } x \text{)} \]

\[ (e_0 \ e_1 \ldots e_n) \quad \text{(application)} \]

\[ \lambda x \to e \quad \text{(abstraction)} \]

\[ \text{if } b \text{ then } e_1 \text{ else } e_2 \quad \text{(conditional)} \]
\[ e ::= \]
\[ c \quad (\text{constants}) \]
\[ x \quad (\text{variables } x) \]
\[ (e_0 \ e_1 \ldots e_n) \quad (\text{application}) \]
\[ \lambda x \rightarrow e \quad (\text{abstraction}) \]
\[ \text{if } b \text{ then } e_1 \text{ else } e_2 \quad (\text{conditional}) \]
\[ e_1 := e_2 \quad (\text{equational constraint}) \]
\[ e_1 \& e_2 \quad (\text{concurrent conjunction}) \]
\[ \text{let } x_1, \ldots, x_n \text{ free in } e \quad (\text{existential quantification}) \]
Expressions

\[ e ::= \]

\[ c \quad \text{(constants)} \]

\[ x \quad \text{(variables } x) \]

\[ (e_0 \, e_1 \ldots e_n) \quad \text{(application)} \]

\[ \lambda x \rightarrow e \quad \text{(abstraction)} \]

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

\[ e_1 =:= e_2 \quad \text{(equational constraint)} \]

\[ e_1 \, \& \, e_2 \quad \text{(concurrent conjunction)} \]

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

Equational constraints over functional expressions:

\[ \text{conc } ys \, [x] =:= [1,2] \quad \sim \quad \{ y = [1], \, x = 2 \} \]

Further constraints: real arithmetic, finite domain, ports
FUNCTIONS

- 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 $\sim$ *logic programming*

Rigid functions $\sim$ *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:

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

\(f\) rigid:

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

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

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

\[
\leadsto \ \{x = 1, y = 3\}
\]

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

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

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

Expressible as HTML term:

```
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!
CGI references:

```haskell
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:

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

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

```haskell
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
\(\leadsto\) 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"
   [htxt "Enter your first name: ", textfield first ",
   button "Continue" fhandler]

where first free

   fhandler _ =
   return $ Form "Last Name Form"
   [htxt "Enter your last name: ", textfield last ",
   button "Continue" lhandler]

where last free

   lhandler env = return $ Form "Answer"
   [htxt ("Hi, " ++ env first ++ " " ++ 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:

```plaintext
mail_epilog =
    [hxtxt "Enter a name: ", textfield nref "",
    button "search email" lookup, hrule]

where nref free

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

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

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

```
runcgi :: String -> IO HtmlForm -> 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