Web Server Programming in Curry
– Extended Abstract –

Michael Hanus*

Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstr. 40, D-24098 Kiel, Germany, `mh@informatik.uni-kiel.de`

Abstract. In this paper we propose a new approach to implement web services based on the Common Gateway Interface (CGI). Since we use the multi-paradigm declarative language Curry as an implementation language, many of the drawbacks and pitfalls of traditional CGI programming can be avoided. For instance, the syntactical details of HTML and passing values with CGI are hidden by a wrapper that executes abstract HTML forms by translating them into concrete HTML code. This leads to a high-level approach to server side web service programming where notions like event handlers, state variables and control of interactions are available. Thanks to the use of a functional logic language, we can structure our approach as an embedded domain specific language where the functional *and* logic programming features of the host language are exploited to abstract from details and frequent errors in standard CGI programming.

1 Motivation

In the early days of the World Wide Web (in the following called the web), most of the documents were static, i.e., stored in files which can be viewed in a nicely formatted layout. With the introduction of the Common Gateway Interface (CGI), more and more documents become dynamic, i.e., they are computed on the web server at the time they are requested from a client. In combination with input forms specified in HTML documents, more complex forms of interactions become possible so that clients can retrieve or store specific data via their web browsers.

An advantage of CGI is that it is supported by most web servers. Thus, the use of CGI does not need any special extensions on the server or the client side (e.g., no servlets or cookies), which is a requirement for our development. On the other hand, CGI offers only a very primitive form of interaction so that the programming of web services often becomes awkward. Although general scripting languages like Perl provide libraries for decoding input form data, they do not support the programmer in the construction of correct output data or to control a sequence of interactions with the client. This demands for specialized languages (e.g., MAWL [8], DynDoc [11]) or specialized libraries in existing languages (e.g., [2, 9, 13]). In this paper we take the latter approach. We show how the features of an integrated functional logic language (see [3] for a survey on these kind of languages) can be exploited to provide a flexible and high-level approach to programming web services without any language extensions. In particular, our approach offers the following features for implementing web services:

* This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-1 and by the DAAD under the PROCOPE programme.

- The HTML documents requested by the clients can be flexibly generated depending on the computed data (in contrast to MAWL [8] which uses fixed templates with a simple list iteration construct).
- The data filled in a form by the user can be easily retrieved by an environment model using logical variables as references.
- The different actions to be taken when a user has completed a form are specified by an event handler model.
- The sequence (or iterations) of interactions with the web server is described in one script and not distributed over a set of script files. In particular, a form is described together with the handler for this form which avoids typical CGI programming errors (e.g., undefined input fields).
- State variables which should persist between different interactions are directly supported.
- The CGI interaction (usually, by environment variables and value decoding) is hidden to the user and encapsulated in a wrapper that translates the high-level scripts into HTML code.

Our library is completely implemented in Curry [4, 7], a modern multi-paradigm declarative language which integrates functional, logic, and concurrent programming paradigms. Thus, we show that Curry is an appropriate language for writing web service scripts. On the other hand, Curry is a complete programming language offering static typing, higher-order functions, constraints, and features for concurrent and distributed programming, which are useful for building complex web applications.

This paper is structured as follows. The next section provides a short overview of the main features of Curry as relevant for this paper. Sections 3 and 4 introduce our approach for modeling basic HTML documents and interactive forms. Section 5 discusses the use of our CGI programming model by various examples before we conclude in Section 6.

2 Basic Elements of Curry

Since we assume familiarity with the basic structure of HTML and CGI programming, we review in this section only the basic elements of Curry as necessary to understand the ideas presented in this paper. More details about Curry's computation model and a complete description of all language features can be found in [4, 7].

Curry combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of expressions with synchronization on logical variables). Curry also provides additional features in comparison to the pure paradigms (compared to functional programming: search, computing with partial information and constraints; compared to logic programming: more efficient evaluation due to the deterministic and demand-driven evaluation of functions, more flexible search strategies) and supports programming-in-the-large with specific features (types, modules, encapsulated search). From a syntactic point of view, a Curry program is a functional program[1] extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Thus, a Curry program consists of the definition of functions and the data types on which the functions operate. Functions are evaluated in a lazy manner. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. The behavior of function calls with free variables depends on the evaluation annotations of functions which can be either *flexible* or *rigid*. Calls to rigid functions are suspended if a demanded argument, i.e., an argument whose value is necessary to decide the applicability of a rule, is uninstantiated ("*residuation*"). Calls to flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments to the required values in order to apply a rule ("*narrowing*").

Example 1. The following Curry program defines the data types of Boolean values, polymorphic lists and trees (first three lines) and functions for computing the concatenation of lists and the last element of a list:

```
data Bool   = True | False
data List a = []   | a : List a
data Tree a = Leaf a | Node a (List (Tree a))

conc :: [a] -> [a] -> [a]
conc eval flex

conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last xs | conc ys [x] =:= xs    = x   where x,ys free
```

The data type declarations define `True` and `False` as the Boolean constants, `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types and the type `List a` is usually written as `[a]` for conformity with Haskell), and `Leaf` and `Node` as the constructors for trees.

The (optional) type declaration ("`::`") of the function `conc` specifies that `conc` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[2] Since `conc` is explicitly defined as flexible[3] (by "`eval flex`"), the equation "`conc ys [x] =:= xs`" can be solved by instantiating the first argument `ys` to the list `xs` without the last argument, i.e., the only solution to this equation satisfies that `x` is the last element of `xs`.

In general, functions are defined by (*conditional*) *rules* of the form "l `|` c `=` e `where` vs `free`" where l has the form $f\, t_1 \ldots t_n$ with f being a function, $t_1, \ldots, t_n$ data terms and each variable occurs only once, the *condition* c is a constraint, e is a well-formed *expression* which may also contain function calls, lambda

[1] Curry has a Haskell-like syntax [10], i.e., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

[2] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

[3] As a default, all functions except for constraints are rigid.

abstractions etc, and vs is the list of *free variables* that occur in c and e but not in l (the condition and the `where` parts can be omitted if c and vs are empty, respectively). The `where` part can also contain further local function definitions which are only visible in this rule. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable. A *constraint* is any expression of the built-in type `Success`[4]. Each Curry system provides at least equational constraints of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms (i.e., terms without defined function symbols). However, specific Curry systems can also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints for applications in operation research problems, as in the PAKCS implementation [6].

The operational semantics of Curry, as precisely described in [4, 7], is a conservative extension of lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Due to the use of an optimal evaluation strategy [1], Curry can be considered as a generalization of concurrent constraint programming [12] with a lazy (optimal) evaluation strategy. Due to this generalization, Curry supports a clear separation between the sequential (functional) parts of a program, which are evaluated with an efficient and optimal evaluation strategy, and the concurrent parts, based on the concurrent evaluation of constraints, to coordinate concurrent program units.

Monadic I/O: Since web service programs usually interact with their environment (e.g., retrieve or store information in files on the server), some knowledge about performing I/O in a declarative manner is required. The I/O concept of Curry is identical to the monadic I/O concept of Haskell [14], i.e., an interactive program is considered to compute a sequence of actions which are applied to the outside world. *Actions* have type "`IO` α" which means that they return a result of type α whenever they are applied to (and change) the outside world. For instance, `getChar` of type `IO Char` is an action which reads a character from the standard input whenever it is executed, i.e., applied to a world. Similarly, "`readFile f`" is an action which returns the contents of file `f` in the current world. Actions can only be sequentially composed. For instance, the action `getChar` can be composed with the action `putChar` (which has type `Char -> IO ()` and writes a character to the terminal) by the sequential composition operator `>>=` (which has type `IO` α `->` (α `-> IO` β) `-> IO` β), i.e., "`getChar >>= putChar`" is a composed action which prints the next character of the input stream on the screen. Furthermore, "`return` e" is the "empty" action which simply returns e (see [14] for more details).

3 Modeling Basic HTML

In order to avoid certain syntactical errors (e.g., unbalanced parenthesis) during the generation of HTML documents by a web server, the programmer should not be forced to generate the explicit text of HTML documents (as in CGI scripts written in Perl or with the Unix shell). A better approach is the introduction of an abstraction layer where HTML documents are modeled as terms of a specific data

[4] `Success` was called `Constraint` in previous versions of Curry

type together with a wrapper function which is responsible for the correct textual representation of this data type. Such an approach can be easily implemented in a language supporting algebraic data types. Similarly to the data type of trees in Example 1, we introduce the type of HTML expressions in Curry as follows:

```
data HtmlExp = HtmlText String
             | HtmlStruct String [(String,String)] [HtmlExp]
             | HtmlElem   String [(String,String)]
```

Thus, an HTML expression is either a plain string or a structure consisting of a tag (e.g., B,EM,H1,H2,...), a list of attributes, and a list of HTML expressions contained in this structure. The translation of such HTML expressions into their corresponding textual representation is straightforward: an HtmlText is represented by its argument, and a structure with tag t is enclosed in the brackets <*t*> and </*t*> (where the attributes are eventually added to the open bracket). Since there are a few HTML elements without a closing tag (like <HR> or
), we have included the alternative HtmElem to represent these elements.

Since writing HTML documents in this form might be tedious, we define several functions as useful abbreviations (the function htmlQuote transforms characters with a special meaning in HTML, like <, >, &, ", into their HTML quoted form):

```
htxt   s     = HtmlText (htmlQuote s)    -- plain string
h1     hexps = HtmlStruct "H1" [] hexps  -- main header
...
bold   hexps = HtmlStruct "B"  [] hexps  -- bold font
italic hexps = HtmlStruct "I"  [] hexps  -- italic font
...
hrule        = HtmlElem "HR" []          -- horizontal rule
...
```

As a simple example, the following expression defines a "Hello World" document consisting of a header and two words in italic and bold font, respectively:

```
[h1 [htxt "Hello World"],
 italic [htxt "Hello"], bold [htxt "world!"]]
```

Note that we do not check the validity of the attributes for each structure. This can be done by defining HTML expressions with a reacher type structure, as shown in [13]. The wrapper function to convert these HTML expressions into valid HTML documents will be shown in the next section where we discuss the modeling of input forms.

4 Input Forms

In order to enable more sophisticated interactions between clients using standard browsers and a web server, HTML defines so-called FORM elements which usually contains several input elements to be filled out by the client. When the client submits such a form, the data contained in the input elements is encoded and sent (on the standard input or with the URL) to the server which starts a CGI program to react to the submission. The activated program decodes the input data and performs some application-dependent processing before it returns an HTML document on the standard output which is then sent back to the client.

In principle, the type `HtmlExp` is sufficient to model all kinds of HTML documents including input elements like text fields, check buttons etc. For instance, an input field to be filled out with a text string can be modeled as

```
HtmlElem "INPUT" [("TYPE","TEXT"),("NAME",name),("VALUE",cont)]
```

where the string `contents` defines an initial contents of this field and the string `name` is used to identify this field when the data of the filled form is sent to the server. This direct approach is taken in CGI libraries for scripting languages like Perl or also in the CGI library for Haskell [9]. In this case, the program running on the web server is an I/O action that decodes the input data (contained in environment variables and the standard input stream) and puts the resulting HTML document on the output stream. Therefore, CGI programs can be implemented in any programming language supporting access to the system environment. However, this basic view results in an awkward programming style when sequences of interactions (i.e., HTML forms) must be modeled where state should be passed between different interactions. Therefore, we propose a higher abstraction level and we will show that the functional and logic features of the underlying language Curry can be exploited to provide an appropriate programming infrastructure.

There are two basic ideas of our CGI programming model:

1. The input fields are not referenced by strings but by elements of a specific abstract data type. This has the advantage that the names of references correspond to names of program variables so that the compiler can check inconsistencies in the naming of references.
2. The program that is activated when a form is submitted is implemented together with the program generating the form. This has the advantage that sequences of interactions can be simply implemented using the control abstractions of the underlying language and state can be easily passed between different interactions of a sequence using the references mentioned above.

For dealing with references to input fields, we use logical variables since it is well known that logical variables are a useful notion to express dependencies inside data structures [5, 15]. To be more precise, we introduce a data type

```
data CgiRef = CgiRef String
```

denoting the type of all references to input elements in HTML forms. This data type is abstract, i.e., its constructor `CgiRef` is not exported by the CGI library. This is essential since it avoids the construction of wrong references. The only way to introduce such references are logical variables, and the global wrapper function is responsible to instantiate these variables with appropriate references (i.e., instantiate each reference variable to a term of the form `CgiRef` n where n is a unique name).

To include references in HTML forms, we extend the definition of our data type for HTML expressions by the following alternative:

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

A term "`HtmlCref hexp cr`" denotes an HTML element `hexp` with a reference to it. Usually, `hexp` is one of the input elements defined for HTML, like text fields, text areas, check boxes etc. For instance, a text field is defined by the following abbreviation in our library:[5]

```
textfield :: CgiRef -> String -> HtmlExp
textfield eval flex
textfield (CgiRef ref) contents =
  HtmlCRef (HtmlElem "INPUT" [("TYPE","TEXT"),("NAME",ref),
                              ("VALUE",contents)])
           (CgiRef ref)
```

Note that `ref` is unbound when this function is applied but it will be bound to a unique name (string) by the wrapper function executing the form (see below).

A complete HTML form consists of a title and a list of HTML expressions to be displayed by the client's browser, i.e., we represent HTML forms as expressions of the following data type:

```
data HtmlForm = Form String [HtmlExp]
```

Thus, we define a form containing a single input element (a text field) by

```
Form "Form" [h1 [htxt "A Simple Form"],
             htxt "Enter a string:", textfield sref ""]
```

In order to submit a form to the web server, HTML supports "submit" buttons (we only discuss this submission method here although there are others). The actions to be taken are described by CGI programs that decode the submitted values of the form before they perform the appropriate actions. To simplify these actions and combine them with the program generating the form, we propose an event handling model for CGI programming. For this purpose, each submit button is associated with an event handler responsible to perform the appropriate actions. An *event handler* is a function from a CGI environment into an I/O action (in order to enable access to the server environment) that returns a new form to be sent back to the client. A *CGI environment* is simply a mapping from CGI references into strings. When an event handler is executed, it is supplied with a CGI environment containing the values entered by the client into the form. Thus, event handlers have the type

```
type EventHandler = (CgiRef -> String) -> IO HtmlForm
```

To attach an event handler to an HTML element, we finally extend the definition of our data type for HTML expressions by:

```
data HtmlExp = ... | HtmlEvent HtmlExp EventHandler
```

A term "`HtmlEvent hexp handler`" denotes an HTML element `hexp` (typically a submit button) with an associated event handler. Thus, submit buttons are defined as follows:

---

[5] Note that this function must be flexible so that the first argument, which can only be a logical variable, is instantiated by the application of this function.

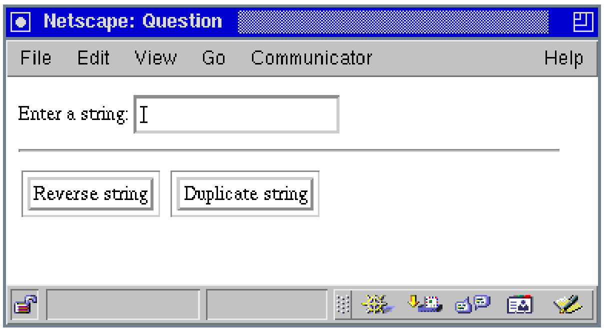

Fig. 1. A simple string reverse/duplication form

```
button :: String -> EventHandler -> HtmlExp
button txt handler =
  HtmlEvent (HtmlElem "INPUT" [("TYPE","SUBMIT"),
                               ("NAME","EVENT"),("VALUE",txt)])
            handler
```

The argument `label` is the text shown on the button and the attribute `NAME` is later used to identify the selected submit button (since several buttons can occur in one form).

To see a simple but complete example, we show the specification of a form where the user can enter a string and choose between two actions (reverse or duplicate the string, see Figure 1):[6]

```
revdup = return $ Form "Question"
           [htxt "Enter a string: ", textfield tref "",
            hrule,
            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)]]
```

Note the simplicity of retrieving values entered into the form: since the event handlers are called with the appropriate environment containing these values, they can easily access these values by applying the environment to the appropriate CGI reference, like `(env tref)`. This simple structure of CGI programming is made possible by the functional as well as logic programming features of the underlying language Curry.

[6] The predefined right-associative infix operator $f\ \$\ e$ denotes the application of f to the argument e.

Forms are executed by a special wrapper function that performs the translation into concrete HTML code, decoding the entered values and invoking the right event handler. This wrapper function has the following type:

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

It takes a string (the URL under which this CGI program is accessible on the server) and an I/O action returning a form and returns an I/O action which, when executed, returns the HTML code of the form. Thus, the above form is executed by the following main function

```
main = runcgi "revdup.cgi" revdup
```

provided that the executable of this program is stored in `revdup.cgi`.

5 Server Side Web Scripting

In this following section we will sketch that the components for CGI programming introduced so far (i.e., logical variables for CGI references, associated event handlers depending on CGI environments) are sufficient to solve typical problems in CGI programming in an appropriate way, like handling sequences of interactions or holding intermediate states between interactions.

From the previous example it might be unclear why the event handlers as well as the wrapper function assumes that the form is encapsulated in an I/O action. Although this is unnecessary for applications where the web server is used as a "computation server" (where the result depends only on the form inputs), in many applications the clients want to access or manipulate data stored on the server. In these cases, the web service program must be able to access the server environment which is easily enabled by running it in the I/O monad.

As a simple example for such kinds of applications, we show the definition of a (not recommendable) form to retrieve the contents of an arbitrary file stored at the server:

```
getfile = return $ Form "Question"
           [htxt "Enter local file name:", textfield fileref "",
            button "Get file!" handler]

 where
    fileref free

    handler env = readFile (env fileref) >>= \contents ->
                  return $ Form "Answer"
                    [h1 [htxt ("Contents of " ++ env fileref)],
                     verbatim contents]
```

Here it is essential that the event handler is executed in the I/O monad, otherwise it has no possibility to access the contents of the local file via the I/O action `readFile` before computing the contents of the returned form. In a similar way, arbitrary data can be retrieved or stored by the web server while executing CGI programs.

In the previous examples the interaction between the client and the web server is quite simple: the client sends a request by filling a form which is answered by the server with an HTML document containing the requested information. In realistic applications it is often the case that the interaction is not finished by sending back the requested information but the client requests further (e.g., more detailed) information based on the received results. Thus, one has to deal with sequences of longer interactions between the client and the server. Our CGI programming model provides a direct support for interaction sequences. Since the answer provided by the event handler is an HTML form rather than an HTML expression, this answer can also contain further input elements and associated event handlers. By nesting event handlers, it is straightforward to implement bounded sequences of interactions. Furthermore, arbitrary iterations or branches in interactions sequences can be implemented by the corresponding control constructs of the underlying language, e.g., recursion or if-then-else. This also offers an elegant solution to the nasty problem of handling intermediate states between different interactions (note that HTTP is a stateless protocol). Since Curry is a language with lexical scoping, event handlers can be nested and one can directly refer to input elements in previous forms. Although there are many other proposals to overcome the stateless nature of HTTP, it should be noted that our library is completely implemented in Curry and does not require any extension to web servers but uses only the standard features of CGI. Since these are supported by most web servers, our library can be used with most web servers (where a Curry system is also installed).

6 Conclusions

In this paper we have presented a new model for programming web services based on the standard Common Gateway Interface. Since this model is put on top of the multi-paradigm language Curry, we could exploit functional as well as logic programming techniques to provide a high abstraction level for our programming model. We have used functional abstractions for specifying HTML forms as expressions of a specific data type so that only well-formed HTML structures can be written. Furthermore, higher-order functional abstractions are used to attach event handlers to particular HTML elements like buttons and to provide a straightforward access to input values via an environment model. Since event handlers can be nested, we have a direct support to define sequences (or sessions) of interactions between the client and the server where states or input values of previous forms are available in subsequent interactions. This overcomes the stateless nature of HTTP. On the other hand, the logical features of Curry are used to deal with references to input values in HTML forms. Since a form can have an arbitrary number of input values, we consider them as "holes" in an HTML expression which are filled by the user so that event handlers can access these values through an environment. Using logical variables to refer to input values seems more appropriate than the use of strings as in raw HTML since some errors (e.g., mispelled names) are detected at compile time.

Since the programming model proposed in this paper needs no specific extension to Curry, it provides appropriate support to implement web-based interfaces to existing Curry applications. Moreover, it can be considered as a domain-specific language for writing CGI scripts. Thus, this demonstrates that a multi-paradigm declarative language like Curry can also be used as a scripting language for server side web applications. We have shown that the functional as well as the logic features provides a good infrastructure to design such a domain-specific language. The implementation of this library is freely available with our Curry development system PAKCS [6]. All examples in this paper have been tested with this implementation. Furthermore, the library is currently used to dynamically create parts of the web pages for Curry[7] and to handle the submission information for the Journal of Functional and Logic Programming[8].

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *To appear in Journal of the ACM*, 2000. Previous version in *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, 1994.
2. D. Cabeza and M. Hermenegildo. Internet and WWW Programming using Computational Logic Systems. In *Workshop on Logic Programming and the Internet*, 1996. See also `http://www.clip.dia.fi.upm.es/miscdocs/pillow/pillow.html`.
3. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
4. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
5. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
6. M. Hanus, S. Antoy, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PACS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2000.
7. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.7.1). Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.
8. D.A. Ladd and J.C. Ramming. Programming the Web: An Application-Oriented Language for Hypermedia Services. In *4th International World Wide Web Conference*, 1995.
9. E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, Vol. 10, No. 1, pp. 1–18, 2000.
10. J. Peterson et al. Haskell: A Non-strict, Purely Functional Language (Version 1.4). Technical Report, Yale University, 1997.
11. A. Sandholm and M.I. Schwartzbach. A Type System for Dynamic Web Documents. In *Proc. of the 27th ACM Symposium on Principles of Programming Languages*, pp. 290–301, 2000.
12. V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.
13. P. Thiemann. Modelling HTML in Haskell. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 263–277. Springer LNCS 1753, 2000.
14. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.
15. D.H.D. Warren. Logic Programming and Compiler Writing. *Software - Practice and Experience*, Vol. 10, pp. 97–125, 1980.

[7] `http://www.informatik.uni-kiel.de/~curry`

[8] `http://www-i2.informatik.rwth-aachen.de/~hanus/jflp.cgi`