An Open System to Support Web-based Learning*
(Extended Abstract)

Michael Hanus Frank Huch

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh,fhu}@informatik.uni-kiel.de

Abstract. In this paper we present a system, called *CurryWeb*, to support web-based learning. Characteristic features of this system is openness and self-responsible use. Openness means that there is no strong distinction between instructors and students, i.e., every user can learn with the system or add new learning material to it. Self-responsible use means that every user is responsible for selecting the right material to obtain the desired knowledge. However, the system supports this selection process by structuring all learning material hierarchically and as a hypergraph whose nodes and edges are marked with educational objectives and educational units, respectively.
The complete system is implemented with the declarative multi-paradigm language Curry. In this paper we describe how the various features of Curry support the high-level implementation of this system. This shows the appropriateness of declarative multi-paradigm languages for the implementation of complex web-based systems.

Keywords: functional logic programming, WWW, applications

1 Overview of CurryWeb

CurryWeb is a system to support web-based learning that has been mainly implemented by students during a practical course on Internet programming. A key idea of this system was to support self-responsibility of the users (inspired by similar ideas in the development of open source software). Thus, every user can insert learning material on a particular topic. The quality of the learning material can be improved by providing feedback through critics and ranking, or putting new (hopefully better) material on the same topic.

In order to provide a structured access to the learning material, CurryWeb is based on the following notions:

Educational Objectives: An educational objective names a piece of knowledge that can be learned or is required to read and understand some learning material. Since it is not possible to give this notion a formal meaning,

* This research has been partially supported by the German Research Council (DFG) under grant Ha 2457/1-2 and by the DAAD under the programme Acciones Integradas Hispano-Alemanas.

each educational objective has a distinguished name. Its meaning can be informally described by a document.

Educational Unit: An educational unit is the smallest piece of learning material in the system. Each educational unit has *prerequisites* that are required to be able to understand this unit and *objectives* that can be learned by studying this unit. Both prerequisites and objectives are sets of educational objectives. Thus, the educational units together with their prerequisites and objectives form a hypergraph where educational objectives are the nodes and each educational unit is a hyperedge (a directed edge with $m \geq 0$ source and $n \geq 0$ target nodes).

Courses: A course is a sequence of elements where each element is an educational unit or a course. There is exactly one root course from which all other courses are reachable. Thus, the courses structure all educational units as a tree where a leaf (educational unit) can belong to more than one course.

The course and hypergraph structure also implies two basic methods to navigate through the system. The courses provide a hierarchical navigation as in Unix-like file systems. Often more useful is the navigation through the hypergraph of educational objectives: a user can select one or more educational objectives and search for educational units that can be studied based on this knowledge or which provide this knowledge. Moreover, the system also offers the usual string search functions on titles or contents of documents.

Finally, CurryWeb also manages *users*. All documents can be anonymously accessed but any change to the system's contents (e.g., critics and ranking of educational units, inserting new material) can only be done by registered users in order to provide some minimal safety. Registration as a new user is always possible and automatically performed (initially, a random password is sent by email to the new user). Additionally, there is also a super user who can change the rights of individual users which are classified into three levels:

- *Standard users* can insert new documents and update their own documents, write critics and rank other educational units.
- *Administrators* are users who can modify all documents.
- *Super users* can modify everything, i.e., they can modify all documents including user data (e.g., deleting users).

Each document in the system (e.g., educational unit, educational objective, course description) belongs to a user (the *author* of a document). Documents can only be changed by the author or his *co-authors*, i.e., other users who received from the author the right to change this document.

To give an impression of the use of CurryWeb, Fig. 1 shows a snapshot of a selected educational unit. The top frame contains the access to basic functions (login/logout, search, site map, etc.) of the system. The upper left frame is the navigation window showing the course structure or the educational objectives, depending on the navigation mode selectable in the lower left frame. The navigation mode shown in Fig. 1 is the course structure, i.e., the upper left frame shows the tree structure of courses. A user can open or close a course folder

Fig. 1. A snapshot of the CurryWeb interface

by clicking on the corresponding icons. By selecting the "Modify Mode", the visualization of the course structure is extended with icons to insert new courses or educational units in the tree. If the navigation mode is based on educational objectives, the upper left frame contains a list of educational objectives in which the user can select some of them in order to search for appropriate educational units.

The right frame is the content frame where the different documents (educational units, course descriptions, descriptions of educational objectives, user settings, etc.) are shown and can be modified (if this is allowed).

An important feature of this web-based system is the hyperlink structure of all entities. For instance, the name of the author of a document is shown with a link to the author's description, or the prerequisites and objectives of an educational unit are linked to the description of the corresponding educational objective. If the author or co-author of a document is logged into the system, the document view is extended with an edit button which allows to load a web page for changing that document.

The rest of this paper is structured as follows. The next section provides a short overview of the main features of Curry as relevant for this paper. Section 3 sketches the model for web programming in Curry. Section 4 describes

the implementation of CurryWeb and highlights the features of Curry exploited to implement this system in a high-level manner. Finally, Section 5 contains our conclusions.

2 Basic Elements of Curry

In this section we review those elements of Curry which are necessary to understand the implementation of CurryWeb. More details about Curry's computation model and a complete description of all language features can be found in [4, 9].

Curry is a modern multi-paradigm declarative language combining in a seamless way features from functional, logic, and concurrent programming 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 extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Curry has a Haskell-like syntax [11], 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 ("$f\ e$").

A Curry *program* consists of the definition of functions and 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 mode of functions which can be either *flexible* or *rigid*. Calls to flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments (i.e., arguments whose values are necessary to decide the applicability of a rule) to the required values in order to apply a rule ("*narrowing*"). Calls to rigid functions are suspended if a demanded argument is uninstantiated ("*residuation*").

Example 1. The following Curry program defines the data types of Boolean values and polymorphic lists (first two 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

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

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

The data type declarations define `True` and `False` as the Boolean constants and `[]` (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).

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.[1] Since `conc` is flexible,[2] the equation "`conc ys [x] =:= xs`" is 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 "$f\,t_1 \ldots t_n$ `|` c `=` e `where` vs `free`" with f being a function, $t_1, \ldots, t_n$ *patterns* (i.e., expressions without defined functions) without multiple occurrences of a variable, the *condition* c is a constraint, e is a well-formed *expression* which may also contain function calls, lambda abstractions etc, and vs is the list of *free variables* that occur in c and e but not in $t_1, \ldots, t_n$. 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`. 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 patterns. However, specific Curry systems can also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints, as in the PAKCS implementation [7].

The operational semantics of Curry, precisely described in [4, 9], is based on an optimal evaluation strategy [1] and a conservative extension of lazy functional programming and (concurrent) logic programming. However, the application considered in this paper does not require all these features of Curry. Beyond the standard features of functional languages (e.g., laziness, higher-order functions, monadic I/O), logical variables are essential to describe the structure of dynamic web pages appropriately by exploiting the functional logic programming pattern [2] "locally defined global identifier".

3 Programming Dynamic Web Pages in Curry

This section surveys the model supported in Curry for programming dynamic web pages. This model exploits the functional and logic features of Curry and is available through the library `HTML` which is part of the PAKCS distribution [7]. The ideas of this library and its implementation are described in detail in [5].

If one wants to write a program that generates an HTML document, one must decide about the representation of such documents inside a program. A textual representation (as often used in CGI scripts written in Perl or with the Unix shell) is very poor since it does not avoid certain syntactical errors (e.g., unbalanced parenthesis) in the generated document. Thus, it is better to

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

[2] As a default, all functions except for I/O actions and external functions are flexible.

introduce an abstraction layer and model HTML documents as elements of a specific data type together with a wrapper function that is responsible for the correct textual representation of this data type. Since HTML documents have a tree-like structure, they can be represented in functional or logic languages in a straightforward way [3, 10, 12]. For instance, we can define 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 (name/value pairs), and a list of HTML expressions contained in this structure (because there are a few HTML elements without a closing tag, like `<hr>` or `<br>`, there is also the constructor `HtmElem` to represent these elements).

Since writing HTML documents in the form of this data type might be tedious, the `HTML` library defines several functions as useful abbreviations (`htmlQuote` transforms characters with a special meaning in HTML, like `<`, `>`, `&`, `&quot;`, 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
verbatim s   = HtmlStruct "pre" [] [htxt s] -- verbatim text
hrule        = HtmlElem "hr" []             -- horizontal rule
...
```

Thus, the following function defines a "Hello World" document consisting of a header and two words in italic and bold font, respectively:

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

A *dynamic web page* is an HTML document (with header information) that is computed by a program at the time when the page is requested by a client (usually, a web browser). For this purpose, there is a data type

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

to represent complete HTML documents, where the first argument to `HtmlForm` is the document's title, the second argument are some optional parameters (e.g., cookies, style sheets), and the third argument is the document's content. As before, there is also a useful abbreviation:

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

The intention of a dynamic web page is to represent some information that depends on the environment of the web server (e.g., stored in data bases). Therefore, a dynamic web page has always the type "`IO HtmlForm`", i.e., it is an I/O action[3] that retrieves some information from the environment and produces a web document. Thus, we can define the simplest dynamic web page as follows:

```
helloPage = return (form "Hello" hello)
```

This page can be easily installed on the web server by the command "`makecurrycgi`" of the PAKCS distribution [7] with the name "`helloPage`" as a parameter.

Dynamic web pages become more interesting by processing user input during the generation of a page. For this purpose, HTML provides various input elements (e.g., text fields, text areas, check boxes). A subtle point in HTML programming is the question how the values typed in by the user are transmitted to the program generating the answer page. This is the purpose of the Common Gateway Interface (CGI) but the details are completely hidden by the `HTML` library. The programming model supported by the `HTML` library can be characterized as programming with call-back functions.[4] A web page with user input and buttons for submitting the input to a web server is modeled by attaching an *event handler* to each submit button that is responsible to compute the answer document. In order to access the user input, the event handler is a function from a "CGI environment" (holding the user input) into an I/O action that returns an HTML document. A *CGI environment* is simply a mapping from CGI references, which identify particular input fields, into strings. Thus, each event handler has the following type:

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

What are the elements of type `CgiRef`, i.e., the *CGI references* to identify input fields? In traditional web programming (e.g., raw CGI, Perl, PHP), one uses strings to refer to input elements. However, this has the risk of programming errors due to typos and does not support abstraction facilities for composing documents (see [5] for a more detailed discussion). Here, logical variables are useful. Since it is not necessary to know the concrete representation of a CGI reference, the type `CgiRef` is abstract. Thus, we use logical variables as CGI references. Since each input element has a CGI reference as a parameter, the logical variables of type `CgiRef` are the links to connect the input elements with the use of their contents in the event handlers. For instance, the following program implements a (dangerous!) web page where a client can submit a file name. As a result, the contents of this file (stored on the web server) are shown in the answer document:[5]

[3] The I/O concept of Curry is identical to the monadic I/O concept of Haskell [14].

[4] This model has been adapted to Haskell in [13], but, due to the use of a purely functional language, it is less powerful than our model which exploits logical variables.

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

```
getFile = return $ form "Question"
          [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 " ++ env fileref)],
                     verbatim contents]
```

Since the locally defined name `fileref` (of type `CgiRef`) is visible in the right-hand side of the definition of `getFile` as well as in the definition of `handler`, it is not necessary to pass it explicitly to the event handler. Note the simplicity of retrieving values entered into the form: since the event handlers are called with the appropriate CGI environment containing these values, they can easily access these values by applying the environment to the appropriate CGI reference, like `(env fileref)`. This structure of CGI programming is made possible by the functional as well as logic programming features of Curry.

This programming model also supports the implementation of interaction sequences by nesting event handlers or recursive calls to event handlers. Since CGI references are locally defined without any concrete value, one can also compose different forms without name clashes for input elements. These advantages are discussed in [5] in more detail.

4 Implementation

In this section we describe some details of the implementation of CurryWeb and emphasize how the functional logic features of Curry have been exploited in this application.

CurryWeb, as described in Section 1, is completely implemented in Curry using various standard libraries of PAKCS [7], in particular, the `HTML` library introduced in the previous section. The source of the complete implementation is around 8000 lines of code. The implementation is based on the CGI protocol, i.e., it requires only a standard web server configured to execute CGI programs. All data is stored in XML format in files. The access to these files is hidden by access functions in the implementation, i.e., each fundamental data type of the application (e.g., educational unit, educational objective) has functions to store an entry or to read an entry (with a particular key). By changing the implementation of these functions, the file-based information retrieval can be easily replaced by a data base if this becomes necessary for efficiency reasons.[6]

The functionality of each fundamental notion of CurryWeb (educational unit, educational objective, course, user) is implemented in a separate module pro-

[6] We have not used a data base since the current implementation of PAKCS do not offer a data base interface and the hierarchical and semi-structured data format of XML is more appropriate for our application.

viding an abstract data type. Since CGI programs do not run permanently but are started for every request of a client, we separate all data in independent structures. For all structures we use unique identifiers to describe the connections between them. Hence, we get more efficient access to the data relevant for a single request.

Some common functionality is factorized by the notion of a *document*. Each document has an author, a modification date, and some contents. The latter can be a URL (reference to an externally stored document), a list of HTML expressions, or an XML expression (according to some specific format for representing documents). Documents are used at various places. For instance, an educational objective has a document to describe its meaning, a course has a document for the abstract, and an educational unit has two documents: one for the abstract and one for the contents.

Since the implementation of most parts of the system is not difficult (except for the algorithms on the hypergraph to select educational units for a given set of educational objectives), in the following paragraphs we mainly describe how the various features of Curry have helped to implement this system.

4.1 Types

As mentioned in Section 2, Curry is a polymorphically typed language. Types are not only useful to detect many programming errors at compile time, but they are also useful for documentation. The type of a function can be considered as a partial specification of its meaning. In this project, we have exploited this idea by specifying the type signatures of functions of the base modules before implementing them. This was useful to check the consistency of parts of the program immediately when they have been implemented. Moreover, a web-based documentation of these modules can be automatically generated by the tool `CurryDoc` [6]. `CurryDoc` is a documentation generator influenced by the popular tool `javadoc` but applied to Curry in an extended form. In order to apply this tool, the program must be compilable even it is not yet completely implemented. Therefore, we have initially defined all functions with their type signature and a rule with a call to the error function as its right-hand side. For instance, the preliminary code of a function `updateFile` could be as follows:

```
--- An action that updates the contents of a file.
--- @param f - the function to transform the contents
--- @param file - the name of the file
updateFile :: (String -> String) -> String -> IO ()
updateFile f file = error "updateFile: not yet implemented"
```

This is sufficient to generate the on-line documentation (documentation comments start with "`---`") or import it in other modules. Later, the `error` call in the right-hand side is replaced by the actual code.

The polymorphic type system was also useful to implement new higher abstractions for HTML programming. For instance, we have developed generic functions supporting the creation of index pages. They can be wrapped around

arbitrary lists of HTML expressions to categorize them by various criteria (e.g., put all items starting with the same letter in one category). Another module implements the visualization and manipulation of arbitrary trees as a CGI script (e.g., used for the course tree in the upper left frame in Fig. 1) according to a mapping of nodes and leaves into HTML expressions.

4.2 Logical Variables

We have already mentioned in Section 3 that logical variables are useful in web programming as links between the input fields and the event handlers processing the web page. Using logical variables, i.e., unknown values, instead of concrete values like strings or numbers improves compositionality: one can easily combine two independent HTML expressions with local input fields, CGI references and event handlers into one web page without the risk of name clashes between different CGI references. This is useful in our application. For instance, an educational unit has a document for the abstract and a document for the contents. Due to the use of logical variables, we can use the same program code for editing these two documents in a single web page for editing educational units. This will be described in more detail in the next section.

4.3 Functional Abstraction

As mentioned above, the document is a central structure that is used at various places in the implementation. Thus, it makes sense to reuse most of the functionality related to documents. For instance, the code for editing documents should be written only once even if it is used in different web pages or twice in one web page (like in educational units which have two separate documents). In the following we explain our solution to reach this goal.

The implementation of a web page for editing a document consists at least of two parts: a description of the edit form (containing various input fields) as an HTML expression and an event handler that stores the data modified by the user in the edit form. Both parts are linked by variables of type `CgiRef` to pass the information from the edit form to the handler. For the sake of modularity, we hide the concrete number and names of these variables by putting them into one data structure (e.g., a list) so that these parts can be expressed as functions of the following type (where `Doc` denotes the data type of documents):

```
editForm    :: Doc -> [CgiRef] -> HtmlExp

editHandler :: Doc -> [CgiRef] -> (CgiRef -> String) -> IO ()
```

The concrete CGI references are specified in the code of `editForm` by pattern matching (and similarly used in the code of `editHandler`):

```
editForm doc [cr1,cr2,...] =
                    ... textfield cr1 ... textfield cr2 ...
```

Note that the answer type of the associated handler is "`IO ()`" rather than "`IO HtmlForm`". This is necessary since one can only associate a single event

handler to each submit button of a web page. In order to combine the functionality of different handlers (like `editHandler`) in the handler of a submit button, we execute the individual (sub-)handlers and return a single answer page (see below).

To use the code of `editForm` and `editHandler`, no knowledge about the CGI references used in these functions is required since one can use a single logical variable for this purpose. For instance, a dynamic web page for editing a complete educational unit contains two different calls to `editForm` for the abstract and the contents document of this unit. Thus, the code implementing this page and its handler has the following structure:

```
eUnitEdit ... = ... return $ form "Edit Educational Unit" [
   ... editForm abstract arefs,... editForm contents crefs,...
   button "Submit changes" handler]
 where
   arefs,crefs free

   handler env = do ...
                    editHandler abstract arefs env
                    editHandler contents crefs env
                    ...
                    return $ answerpage
```

The logical variables `arefs` and `crefs` are used to pass the appropriate CGI references from the forms to the corresponding handlers that are called from the handler of this web page. Thanks to the modular structure, the implementation of the document editor can be implemented independently of its use in the editor for educational units (and similarly for the editors for courses, educational objectives, etc).

There are many other places in our application where this technique can be applied (e.g., changing co-authors of a document is a functionality implemented in the user management but used in the document editor). Since the different modules have been implemented by different persons, this technique was important for the independent implementation of these modules.

4.4 Laziness and Logic Programming

Due to the use of the `HTML` library, logical variables are heavily used to refer to input elements, like text fields, check boxes etc. In some situations, the number of these elements depends on the current data. For instance, if one wants to navigate through the educational objectives, the corresponding web page has a check button for each educational objective to select the desired subset. Thus, one needs an arbitrary but fixed number of logical variables as CGI references for these check buttons. For this purpose, we define a function

```
freeVarList = x : freeVarList   where x free
```

that evaluates to an infinite list of free variables.[7] Thanks to the lazy evaluation strategy of Curry, one can deal with infinite data structures as in non-strict functional languages like Haskell. Thus, the expression `(take` n `freeVarList)` evaluates to a list of n fresh logical variables.

In some situations one can also use another programming technique. The function `zipEqLength` combines two lists of elements of the same length into one list of pairs of corresponding elements:

```
zipEqLength :: [a] -> [b] -> [(a,b)]
zipEqLength [] [] = []
zipEqLength (x:xs) (y:ys) = (x,y) : zipEqLength xs ys
```

Due to the logic programming features of Curry, we can also evaluate this function with an unknown second argument. For instance, the expression `(zipEqLength [1,2,3] l)` instantiates the logical variable `l` to a list `[(1,x1),(2,x2),(3,x3)]` containing pairs with fresh logical variables `x`i. This provides a different method to generate CGI references for a list of elements.

Note that the concrete use of such lists of logical variables is usually encapsulated in functions generating HTML documents (with check boxes) and event handlers for processing user selections, as described in Section 4.3.

4.5 Partial Functions

In a larger system dealing with dynamic data, there are a number of situations where a function call might not have a well-defined result. For instance, if we look up a value associated to a key in an association list or tree, the result is undefined if there is no corresponding key. In purely functional languages, a partially defined function might lead to an unintended termination of the program. Therefore, such functions are often made total by using the `Maybe` type that represents undefined results by the constructor `Nothing` and defined results v by `(Just` v`)`. If we use such functions as arguments in other functions, we have to check the arguments for the occurrence of the constructor `Nothing` before passing the argument. This often leads to a monadic programming style which is still more complicated than a simple application of partially defined functions.

A functional logic language offers an appropriate alternative. Since dealing with failure is a standard feature of such languages, it is not necessary to "totalize" partial functions with the type `Maybe`. Instead, one can leave the definition of functions unchanged and catch undefined expressions at the appropriate points. For doing so, the following function is useful. It evaluates its argument and returns `Nothing` if the evaluation is not successful, or `(Just` s`)` if s is the first

[7] Note that `freeVarList` is not a cyclic structure but a function according to the definition of Curry. Since the logical variable `x` is locally defined in the body of `freeVarList` and each application of a program rule is performed with a variant containing fresh variables (as in logic programming), `freeVarList` evaluates to `x1:x2:x3:...` where `x1`,`x2`,`x3`,... are fresh logical variables.

solution of the evaluation (the definition is based on the operator `findall` to encapsulate non-deterministic computations, see [8]):

```
catchNoSolutions :: a -> Maybe a
catchNoSolutions e = let sols = findall (\x -> x =:= e) in
                     if sols==[] then Nothing else (Just (head sols))
```

From our experience, it is a good programming style to avoid the totalization of most functions and use the type `Maybe` (or a special error type) only for I/O actions since a failure of them immediately terminates the program.

5 Conclusions

We have presented CurryWeb, a web-based learning system that is intended to be organized by its users. The learning material is structured in courses and by educational objectives (prerequisites and objectives). The complete system is implemented with the declarative multi-paradigm language Curry. We have emphasized how the various features of Curry supported the development of this complex application. Most parts of the system were implemented by students without prior knowledge to Curry or multi-paradigm programming. Thus, this project has shown that multi-paradigm declarative languages are useful for the high-level implementation of non-trivial web-based systems.

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.
2. S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. 6th Int. Symp. on Functional and Logic Programming*, pp. 67–87. Springer LNCS 2441, 2002.
3. 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`.
4. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. 24th ACM Symp. on Principles of Programming Languages*, pp. 80–93, 1997.
5. 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)*, pp. 76–92. Springer LNCS 1990, 2001.
6. M. Hanus. CurryDoc: A Documentation Tool for Declarative Programs. In *Proc. 11th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2002)*, pp. 225–228. Research Report UDMI/18/2002/RR, University of Udine, 2002.
7. M. Hanus, S. Antoy, 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/, 2003.
8. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.

9. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at http://www.informatik.uni-kiel.de/ curry, 2003.
10. E. Meijer. Server Side Web Scripting in Haskell. *Journal of Functional Programming*, Vol. 10, No. 1, pp. 1–18, 2000.
11. S.L. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. http://www.haskell.org, 1999.
12. P. Thiemann. Modelling HTML in Haskell. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 263–277. Springer LNCS 1753, 2000.
13. P. Thiemann. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. In *4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002)*, pp. 192–208. Springer LNCS 2257, 2002.
14. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.