A library for quizzes

Christian Queinnec
Université Paris 6 — Pierre et Marie Curie
LIP6, 4 place Jussieu, 75252 Paris Cedex — France
Christian.Queinnec@lip6.fr

ABSTRACT
Programming web dialogs is already known to be well served by continuations; this paper presents a continuation-based library for a particular class of web dialogs: quizzes for students. The library is made of objects representing the individual questions and of functional combinators hiding the imperative aspects of page shipping over HTTP and management of continuations. Mixing these three styles provide an elegant framework that fulfills our initial goal. The description of that library is hoped to be helpful for quizzes designers.

1. INTRODUCTION
Last year, we designed a CD-ROM in order to support a college-level course named “Evaluation process” strongly based on the Scheme programming language [1]. This is the first computer science (CS) course delivered to young scientists (eighteen-year old) who still have to choose whether to specialize in maths, CS, mechanics or physics. The goal of the course is to introduce students to recursion, trees, grammars and language interpretation.

The CD-ROM was given to a special group of 45 computer-equipped students who were then able to work at home comfortably with the same means they have access to at the university. Therefore, besides our course material, the CD-ROM also contains copies of the DrScheme programming environment [3] along with some add-ons providing exercises and quizzes.

An exercise is an assignment that should be performed with the help of the programming environment. A student chooses an exercise (with an additional menu), reads the question (an HTML page displayed by the inner browser of DrScheme), writes the required function(s) (as well as the required testing function(s)), tests them then hit the “check” button which synthesizes a new HTML page with some comments and a mark ranking the provided solution (see Figure 1). Above a given threshold, teachers’ solutions are displayed and the student may proceed to the next question.

Quizzes are tightly bound to the written course. The course is chopped into a number of HTML pages, each centered on a single topic. For ease of use, these pages are accessed through a mainstream browser such as Explorer or Communicator (the inner browser of DrScheme 103 was not able to handle forms). After every topic, the system proposes various quizzes (as HTML links) checking various levels of understanding. We distinguish level-1 quizzes that are simple applications of the course: they mainly correspond to very simple Scheme questions that do not require the whole power of the DrScheme environment (see Figure 2). Level-2 quizzes strive the student to verbalize its understanding; these questions are not checked but links to appropriate answers are given back. Finally, level-3 quizzes help to understand how the topic contributes to the overall goal of the whole course.

Technically, links to quizzes are served by a web server running as a thread inside DrScheme. A quiz (and the average ten questions it contains) is entirely held in a single file that is simply evaluated by the web server. Continuations [6] are used

- to suspend the server after shipping a page to the student
- and to resume the server with student’s answers to the displayed questions.

In order to give a uniform look for the quizzes and to minimize code for the definition of the individual questions of quizzes, quizzes were defined with the help of a library of functions and macros. A question is represented by an object, a quiz is a combination of questions, and combinators embed (and hide) the imperative aspects of page shipping and continuations management.

The rest of the paper presents that library and some elements of the rationale behind it. Section 2 will describe the “question” object, Section 3 will present how questions are composed via appropriate combinators to form quizzes. Section 4 will detail the imperative implementation of combinators and their use of continuations. Finally, Section 5 will conclude.

2. QUESTIONS
A quiz is made of a succession of pages, each of them contains one or more questions. When a question is asked, its terms are generated into HTML. Answers are graded; this grading triggers the synthesis of a good or a bad answer (both in HTML). The grade is a number – positive if the answer is correct, negative otherwise. The HTML produced by a question is limited to the terms or to the good or bad answer without

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

Copyright © 2002 Christian Queinnec.
Figure 1: Screen capture of an exercise – The student hit the “Tester” (check) button and got a mark good enough to let him see a solution (more than one solution may appear).

Figure 2: Screen capture of a quiz – This quiz corresponds to a compound question where the student has to write a function that is, its type, its definition and some associated tests.
any adornment. The resulting HTML is one (or more) paragraph(s), not a complete HTML page.

Besides these characteristics, a question also knows how to be logged (when displayed, answered or graded): this is necessary in order to assess students’ progress. Of course, all questions are identified with a unique identifier. Logging is done via an HTTP Post request to a centralized logging service feeding a database from which students’ progress is deduced.

When a question is displayed again (for instance after a student answers incorrectly), the HTML fields are pre-filled with their former content. It is also possible to blank those fields if required.

We chose to represent questions as objects with fields and methods. Here are the signatures of the methods on the question objects. They are given in a Meroon [5] style (although the current implementation uses message passing rather than Meroon itself).

```scheme
(define-method (id question) returns the identifier (a string))
(define-method (author question) returns the author (a string))
(define-method (reset question) erases all already-filled fields)

(define-method (html-question question) interactive?)
  returns an HTML string: the question stem.
  if interactive? is true then generate also the
  HTML INPUT tags (textfield, textarea, checkbox, etc.)

(define-method (report-question question) nextUrl) logs that the question was asked

(define-method (report-answer question) request) logs the answer

(define-method (verify question) request) grades student’s answer (encoded in the HTTP request) and returns a number coding the grade

(define-method (html-good-answer question) request a)
  generates a positive comment (an HTML string)
  based on a grade a

(define-method (html-bad-answer question) request a)
  generates a negative comment (an HTML string)
  based on grade a

We adopt objects to structure behavior sharing. The hierarchy of questions is sketched on Figure 3 where indentation denotes the subclass relationship. The first two classes generate questions offering single or multiple choices. The terms of a question of the third class always display a box where the student types in his answer. The predicate field of the question analyzes this answer that is, a string. Some subclasses exist for instance, the question-regexp which imposes students’ answers to satisfy a given regexp.

Another, more important, subclass is the question-scheme that expects students’ answers to be legal Scheme expressions. The associated predicate then receives that Scheme expression instead of a string (of course, the Scheme expression might be a Scheme string). Among questions expecting a

```
3. COMBINATORS

Questions form the basic building blocks for HTML pages, therefore, they should be freely re-usable in various contexts. For instance, when building a quiz, one may want a simple question to be iterated until the student answers it correctly, repeat another question at most twice if badly answered and so on. Questions must be combined in order to form quizzes.

A quiz is a Scheme file that, when evaluated, builds pages with questions and ships them to the student. When the student answers (with an HTTP request), the quiz is resumed at the point where the page was shipped. This is the essence of web continuations [6]. When resumed, the quiz dispatches the request towards the asked questions, gathers the positive/negative comments along with some new or previous questions, packs these all in a new page and ships it to the student. Reaching the end of the file ends the quiz.

In order to be able to re-use questions in various contexts, we separate questions’ content from the way questions are asked. In a given context, a question may be mandatory while in another context, the same question may be grouped (and displayed) with three others among which two good answers may be sufficient to proceed past this group of four questions. We must be able to precisely state how the student is led through the quiz depending on his previous good or bad answers.

Here are our current combinators:

(ask-only-once question)
(loop-until-verified question)
(loop-at-most n question exhaustion)

(ask-multiple-questions-once questions...)
(ask-multiple-questions n questions...)

(mute-ask-only-once question)
(mute-ask-multiple-questions-once questions...)

We group them into three families. The first family just confers a behavior to questions that is, — ask a question only once and proceed to the rest of the quiz even if the answer is incorrect — ask a question until obtaining a correct answer (students complain against this behavior, even though we scarcely used it) — ask a question until obtaining a correct answer or at most n times. After n failures, the student may proceed to the next question but is given a notice generated by exhaustion n.

The second family just gathers questions to make them appear as a single one. This is not an easy point since the meaning of the correctness of a group of questions immediately occurs. There is no such problem with the ask-multiple-questions-once, it just gathers the comments for the group of questions. The second combinator generalizes the loop-at-most combinator with the following behavior: the group of questions is asked again and again but correctly answered questions are removed from the group until the maximal number of iterations is reached or all questions are correctly answered.

The last family corresponds to examination performed on computers. They are similar to the combinators with the same name less the mute- prefix. The differences are

- positive/negative comments are not displayed
- students are not allowed to submit more than one answer to any questions (more on that point later).

Here is a contorted example of a quiz that asks a question over and over until the student clicks the “Yes” button. A confirmation is asked for (only once) immediately after. The first two questions are roughly the same but they are defined with alternate means: the first uses a macro while the second uses a function instead. The macro makes available finer details and adopts a uniform keyword-value look and feel.

The third question asks for a Scheme expression returning a number (but at most 2 times). The question generator, named 7-77 (a local value) generates a question asking for a program whose value is a number between 7 and 77. If the answer is correct, the quiz ends with a final cul-de-sac combinator that displays a specific page telling the student that the quiz is over (this allows us to override the implicit call to cul-de-sac with a default message). If the answer is not correct (this is notified with an assignment to the local variable named success?) the same question generator exactly is called to create a new question that will be asked ad libitum.

;;; parameterless question generator
(define-question-generator (understood?)
type: qu
question with unique choice
id: "q-qnc-understood1" 
choices: ' (yes no) ; rendered as radio-buttons
correct: 'yes
author: "Christian.QUEINNEC@lip6.fr"
bad-answer: "Please think harder!"
text: "This is a quiz, i.e., a dialog where you get questions that you must answer." (p "Do you understand ?") )

(h1 "Welcome to a regular quiz") ; inter-title

(loop-until-verified ; 
question
(understood?) ) ; combinator

(ask-only-once ; 
question
(one-choice-question ; 
question "q-qnc-understood2"
'(yes no)
'(yes
div "Do you really understand ?") ) )

(h1 "Welcome to a less simplistic quiz"); inter-title

(let again ((success? #t))

;; another (hand-made) question generator:
(define (7-77)
(reverse-evaluation-question ; question
"q-qnc-7-77"
(+ 7 (random 70)) ) )

(loop-at-most ; combinator
2
(7-77)
(lambda (n)
(set! success? #f)
"Alas!" ) )

(if success?
(cul-de-sac ; combinator
"The quiz ends here!"

;; otherwise:
(again #t) ) )

So far we have a library of combinators over objects to define quizzes. Regular quizzes writers do not need further details, they just have to pick the right question generator, the
appropriate arguments and the right combinators (the first two questions of the example are examples of regular quizzes). Some of our colleagues even told us that they have the impression of writing Scheme data rather than Scheme code.

4. IMPERATIVE ASPECTS OF COMBINATORS

The combinators hide two very different aspects: they hide continuation management and HTML generation details. Since they manage continuations and HTTP, they require a deeper understanding to be written.

4.1 HTML generation details

Questions only generate fragments of HTML. Between combinator-expressions, there may be other HTML-generating expressions in the quiz (see, for instance, the \texttt{h1} function generating a \texttt{h1} tag in the previous quiz example: this tag will appear before the HTML stem of the next question). All these HTML fragments are sequentially (imperatively) accumulated in the communication channel.

All combinators force an interaction with the student. They gather all HTML fragments so far accumulated, wrap them in a \texttt{FORM} tag with a fresh URL bound to the continuation of the quiz (materialized as a “Submit” button), wraps again this form into a complete HTML page (then introducing standard headers, footers, logos, titles, styles, CSS, etc.) and ship it to the student.

Observe that it is up to the final wrapper (a mutable property of the communication channel) to decide how to arrange all these HTML fragments. This isolates questions from their appearance on students’ browsers. This also allows us to have a uniform presentation for all pages.

The combinators also solve another problem on the ergonomic side. To consider the quiz as made of a series of question/answer is rather abstract since the quiz has to deal with HTTP where server answers are only displayed when the user requests something. This is the usual inversion of control [4] which we name question/answer (from the view point of the server) or reply/request (from the point of view of the client’s browser) where the question is the reply while the answer is the request.

When the server receives an answer, there are various dialogical strategies, see Figure 4:

1. it may reply with a negative comment and a link directing the student back to the old question,
2. it may reply with a negative comment and the old question again (with pre-filled fields),
3. it may reply with a positive comment and the new question,
4. it may reply with a positive comment and the new question,

After some experiments, we chose options 2 and 4 since they minimize the number of clicks. Some of our colleagues do not like option 4 when the comment is too big since it refers to the previous question whose terms are gone and therefore pollutes the terms of the new following question. There again, combinators isolate questions from the way the dialog is chopped into pages.

The imperative side of the communication channel allows pages to share some information: the communication channel plays the role of a sort of shared “session object”, but limited to the quiz (as for servlets or ASP dynamic pages). For instance, to be less uniform, messages, button labels and titles are varied. Questions may also put some hints in the communication channel to suggest a title (recall the title of the page is chosen by the HTML wrapper that may pack more than one question on a single page).

For combinators that iterate over a question, the suggested title displays the current trial number and the maximal number of allowed trials.

4.2 Continuation management

Following previous work [6], continuations are mainly put to use via the \texttt{show} function that receives an HTML page generator, captures the current continuation, binds it with a fresh URL, feeds the HTML page generator with that URL, ships the obtained HTML page and waits for an answer, that is, an HTTP request that will become the value of the invocation of the \texttt{show} function.

Combinators wrap a call to the \texttt{show} function with specific management of continuations. These continuations are obtained through the regular \texttt{call/cc} however some hackery specific to DrScheme was required since continuations cannot be called out of their birth thread.

On Figure 5 left, the student hits the “submit” button (labeled \texttt{go}), resumes the quiz server that decides whether to reply with a positive comment and the new question or to reply with a negative comment and the old question. This latter page is not the same as the first one since the second one contains, in addition, the negative comment. However the continuation of the “submit” button is the same.

This situation must be contrasted with the \texttt{mute-} combinators that prevent students from re-submitting to an already answered question. On Figure 5 right, the student answers question 1 then answers question 2 and obtains the terms of question 3, the student then instructs the browser to go back and back to question 1 and tries to change the answer. The combinator detects that and forces the student back to the last
unanswered question that is, question 3. A fine point is that
the invoked continuation leads to the point right before show-
ing question 3 and not the continuation bound to the “submit”
button of question 2 (since the answer of question 2 is already
graded).

Still playing with continuations we also introduce a mode
where a teacher may see a quiz at once that is in a single page.
This is, of course, only possible if the quiz is static enough and
linear. The trick is to transform the show operator to simply
accumulate HTML fragments rather than shipping them. The
concatenation of all these fragments is performed at the end of
the quiz file.

These various modes are well served by the separation of
methods on questions. Answers may be not graded (when the
teacher wants to have a global look to the entire quiz or wants a
paper copy to circulate), answers may be graded without emit-
ting any comment (this is the examination mode).

5. CONCLUSIONS AND PERSPECTIVES
Concerning web continuations, the paper does not present
new results. It only shows how they may be put to work for
quizzes. Only the task concern the continuations just be-
fore or after the shipping of a page in the implementation of
the mute-combinators is new.

Therefore, the paper is centered on the main features of the
quiz library that had several goals:

1. separation of concerns: A question writer just has to
understand how to build questions. These questions may
then be put in a big database (correctly indexed to let
them be easily retrieved); this is future work!

A quiz writer just has to understand combinators in or-
der to assemble questions into dialogs. A quiz program-
mer may dynamically builds thematic quizzes extracted
from the previous database. A special quiz may be de-
signed to build quizzes interactively.

An HTML designer just has to change the HTML genera-
tion part of questions and combinators to alter the look.

A web-dialog designer (just) has to understand continu-
tions to implement other kinds of dialog. For instance,
students asked us in the examination mode (the mute-
combinators) to be able to see all questions in advance
that is, to only prevent submitting more than once to any
given question.

2. nice multi-paradigmatic fit: Programmation requires
mastering various programming styles making some tasks
easier. Refining questions is well served by objects and
classes. Combinators are nice means to assemble ques-
tions to form dialogs. The sequentiality of web interac-
tions via HTTP forces an imperative view for continu-
tions and HTML fragments accumulation.

This is the third version of that library, each version has im-
proved the separation of concerns and adopted the most appro-
priate framework to deal with the new concerns. The current
library has been stable for the last year. Quizzes may have
very reactive behaviors and are far more easier to define and
manage compared to the very static tools of generic authoring
systems. In such systems, a quiz is usually defined with a num-
ber of boxes, radio-buttons, menus to fill, click or unroll. The
resulting quizzes are, most of the time, sequential and made of
independent questions that are syntactically graded (syntacti-
cally since there is no relationship between the label of a radio-
button and the fact that this radio-button should be pressed for
a correct answer).

In our system and since we are teaching a language with an
easy to use evaluator, questions may be specified in a more
semantical way. Since the quiz is a program, it may use the full
power of the underlying language and use conditional or recursion as shown in the previous quiz example where students with good answers may terminate quickly whereas others are provided fresh exercises until they got one right.

6. ACKNOWLEDGMENTS

Many thanks to the numerous (and anonymous as well) reviewers whose comments terrificly improves the paper.

7. REFERENCES


