Continuations: What Have They Ever Done for Us? (Experience Report)

Marc Kaufmann
Central European University
Vienna, Austria
kaufmannm@ceu.edu

Bogdan Popa
Independent
Cluj-Napoca, Romania
bogdan@defn.io

Abstract

Surveys and experiments in economics involve stateful interactions: participants receive different messages based on earlier answers, choices, and performance, or trade across many rounds with other participants. In the design of Congame, a platform for running such economic studies, we decided to use delimited continuations to manage the common flow of participants through a study. Here we report on the positives of this approach, as well as some challenges of using continuations, such as persisting data across requests, working with dynamic variables, avoiding memory leaks, and the difficulty of debugging continuations.

CCS Concepts: • **Software and its engineering → Reusability**; **Control structures**; *Domain specific languages*; *Data flow architectures*; *Functional languages*.

Keywords: Continuations, Web, Racket/Scheme, State management

ACM Reference Format:
Marc Kaufmann and Bogdan Popa. 2024. Continuations: What Have They Ever Done for Us? (Experience Report). In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Software Architecture (FUNARCH '24), September 6, 2024, Milan, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3677998.3678223

1 Introduction

Continuations in a web context allow applications to be programmed in a direct style [5, 7]. In the design of Congame, a platform for running economics studies, we took advantage of this style of programming to implement a framework for specifying composable surveys in a declarative way that elides most of the details of day-to-day web programming from the study creator.

In particular, Congame automatically tracks and manages much of the state of study participants, which is a big boon, since Congame studies are inherently stateful applications. A participant's next step may depend on random treatments — as in A/B tests — or their own or other participants' actions: they may only be allowed to move on if they pass a comprehension test, and their payoff may be co-determined by other participants with whom they interact in market games. Congame thus frees the study creator from having to implement their own ad hoc bug-ridden state management system.[1]

We report on our experience using delimited continuations to implement Congame. In section 2 we show a minimal implementation of a system similar to Congame and demonstrate how natural it is to program web applications in this style. Then, in section 3 we note some positive benefits that follow from our design and compare our system to a popular platform for creating studies. In section 4 we describe some challenges of managing the data flow and of debugging in such a system. Finally, in section 5 we reflect on and broadly recommend our approach, analyze its pros and cons, and make suggestions targeted at other functional architects looking to implement similar systems.

2 Mini Congame

The core of Congame is a *study*, represented as a tree of *steps* and sub-studies. Each *step* in a study is a procedure that generates a web page used to display and possibly retrieve information to and from the participant being surveyed. Figure 1 implements a minimal harness for constructing and running these types of studies. A study creator uses the structures defined in figure 1 alongside *widgets* such as the one defined in figure 2 to put together a study. The study can then be run from within a Racket [3] web server servlet with `run-study`.

When a study is run, its steps are executed sequentially, and when a step uses a widget, the widget reifies the current continuation of the step and stores it in a hash table that maps URLs to continuations. The URL of that continuation is then linked in the resulting HTML. Once a continuation URL is visited, the continuation is restored so that `run-step` returns

[1]To riff on Greenspun's Tenth Rule.

```
(define current-embed (make-parameter #f))
(struct step (handler))
(struct study (steps))
(define (run-study the-study)
  (let loop ([steps (study-steps the-study)])
    (if (null? steps)
        '(continue)
        (match (begin0 (run-step (car steps))
                       (redirect/get/forget))
          ['(retry) (loop steps)]
          ['(continue) (loop (cdr steps))]))))
(define (run-step the-step)
  (match the-step
    [(step (? study? substudy))
     (run-study substudy)]
    [(step handler)
     (send/suspend/dispatch
      (lambda (embed)
        (parameterize ([current-embed embed])
          (response/xexpr (handler)))))]))
```

Figure 1. A mini Congame implementation.

```
(define (button label [action void])
  `(a
    ([href ,((current-embed)
             (lambda (req)
               (action)
               '(continue)))])
    ,label))
```

Figure 2. A button "widget".

and the study loop can continue to the next step. Following this visit, the continuation is removed from the hash table to prevent the participant from pressing the "Back" button in their browser and redoing previous steps. The Racket Web Server [5] provides `send/suspend/dispatch`, which takes care of all the continuation management. Figure 3 shows a basic study implemented using this framework.

3 Benefits of Continuations

Using continuations allows us to progress through the study by traversing the study tree using regular techniques without having to worry much about the fact that we are doing web programming. While performing the traversal, we keep track of the participant's position in the tree which allows us to store participant data in a way that imitates lexical scope, making it very natural for study writers to keep track of local data.

Using continuations further allows us to use regular control flow [5, 7], meaning that every step of a study can decide locally what the participant can do next. The actions in a step can close over the step's environment and use regular functional programming techniques. Figure 4 illustrates that we can write a step that tosses a coin and stores the outcome in the local variable `toss`, then offers a choice to the participant and, after the participant makes their choice, checks the answer against `toss`.

```
(define (hello)
  `(div
    (p "Welcome to the study.")
    ,(button "Continue")))
(define (done)
  `(p "Thank you for participating."))
(define example-study
  (study
   (list
    (step hello)
    (step done))))
```

Figure 3. An example study.

```
(defvar* ok?)
(defstep (intro)
  (html
   (h1 "Welcome to the study!")
   (button "Start")))
(defstep (heads-or-tails)
  (define toss (random-ref '(h t)))
  (html
   (button "Heads" (λ () (set! ok? (eq? toss 'h))))
   " or "
   (button "Tails" (λ () (set! ok? (eq? toss 't))))))
(defstep (result)
  (html
   (if ok?
       (p "You guessed right.")
       (p "You guessed wrong."))))
(defstudy choices
  [heads-or-tails --> ,(λ () done)])
(defstudy example
  [intro --> choices --> result --> ,(λ () done)])
```

Figure 4. A study showing lexical scope and data flow in Congame.

Since our approach is data-driven, changing our data structures requires only minor changes to our harness. For instance, adding support for view handlers — study-specific static pages — meant extending the `step` struct with another field and adding one more request handler to traverse the study tree and display those handlers as necessary. If instead we had opted for a design where we store a representation of steps in the database, then we would have had to update the schema.

More generally, our design is flexible to changes. Extending studies to be generated dynamically was as simple as adding one more case to `run-study` to handle callable study struct instances. Furthermore, the combination of continuations that can close over arbitrary Racket objects alongside the data-driven nature of the studies allows us to easily create and compose studies using the full suite of Racket's facilities, including higher-order studies, just as we would any other tree-like data structure.

To highlight that the above benefits are in no way obvious or automatic, let us illustrate how they are absent from oTree [1], a popular framework for economic experiments.[2] oTree represents studies as apps that are run in a linear sequence, with each app requiring its own folder with various files. This design makes it hard to combine and reuse apps, not least due to difficulties in sharing data between apps. For example, when `app2` should be run only for participants with a high score in `app1`, then `app1` has to store the score in a global namespace, then `app2` looks up this score and decides whether to run or hand over to `app3`. In Congame, `study1` can locally decide to transition to `study2` or `study3` depending on a high or low score. In figure 5 we replicate the Congame example from figure 4 using oTree, illustrating the problem of sharing data between apps.[3] So, while the ease of use of oTree makes developing simple studies easy, its limitations on composing studies and managing state make developing complex studies hard.

Of course, our design could be replicated without continuations, but continuations made this design natural and allowed us to stay flexible.

4 Challenges of Continuations

4.1 Too Few or Too Many Parameters

To allow participants to resume a study when necessary (e.g., when they close the browser tab and return to the website, after their continuations expire, or after a server re-deployment), Congame persists the participant's position: the fully-qualified path to the node they reached within the study tree as represented as a list of ids. In memory, this position is tracked using dynamic variables (*parameters* [4] in Racket parlance). In specific cases, continuations interact with parameters in surprising ways.

When a continuation URL is visited and the continuation is restored, it is run in a fresh Racket thread. Racket threads inherit the parameters of their parent threads. That is, if a parameter is set to one value in the parent, it will be set to the same value in the child thread. When `parameterize` is used to change the value of a parameter for a particular block of code, instead of storing each new parameter value in a thread cell individually, a parameterization object is extended[4] to include the new values of the changed parameters. As a consequence, when `parameterize` is used within the dynamic extent of a continuation and that continuation is later restored in a thread, more parameters than might be expected may end up being restored, because the aforementioned extended parameterization object is installed alongside it.

```
# In settings.py:
SESSION_CONFIGS = [{"app_sequence": [
  'Intro', 'Choices', 'Result'
]}]
PARTICIPANT_FIELDS = ['ok']

# In Intro/IntroPage.html:
# ...
{{ block content }}
  <h1>Welcome to the study!</h1>
  {{ next_button }}
{{ endblock }}

# In Choices/__init__.py:
# ...
class Player(BasePlayer):
    choice = models.StringField(label='Choice:')

class ChoicePage(Page):
    form_model = 'player'
    form_fields = ['choice']

    @staticmethod
    def before_next_page(player, timedout):
        player.participant.ok = player.choice ==
random.choice(['heads', 'tails'])
# ...

# In Result/ResultPage.html:
# ...
{{ block content }}
  {% if player.participant.ok %}
    <p>You chose right.</p>
  {% else %}
    <p>You chose wrong.</p>
  {% endif %}
{{ endblock }}
```

Figure 5. A heavily edited-for-space example of the coin toss study implemented in oTree.

Figure 6 shows an example of this issue. When run, the program in figure 4 displays "a b"; but, since the continuation is captured up to a prompt that resides within the

[2]Here we highlight purposefully dimensions in which oTree is lacking, even though oTree is clearly successful and superior to Congame in many respects.

[3]Of course, such a simple study would not normally be split across multiple apps.

[4]https://github.com/racket/racket/issues/4216

outer `parameterize` form setting the parameter `a`, we had initially expected to see "#f b". Removing the inner use of `parameterize` causes the program to display "#f #f".

```
(define a (make-parameter #f))
(define b (make-parameter #f))
(define tag (make-continuation-prompt-tag))
(define k
  (parameterize ([a 'a])
    (call-with-continuation-prompt
     (lambda ()
       (parameterize ([b 'b])
         ((call-with-current-continuation
           (λ (k) (thunk k))
           tag))))
     tag)))
(call-with-continuation-prompt
 (lambda ()
   (k (lambda ()
        (printf "~s ~s~n" (a) (b)))))
 tag)
```

Figure 6. An example of the parameter revival issue.

On the opposite side of the coin, because the Racket web server restores continuations in fresh threads, it is also possible to "lose" changes to a parameter when using direct assignment. Directly assigning a parameter in a thread records the change to the parameter in a thread cell, without affecting the current parameterization. Originally, we had used direct assignment to update the participant's position, which caused the parameter to reset at the boundary between steps. Then, we switched to explicit uses of `parameterize`, which extends the parameterization such that the changes are available in the restored continuation, as in the previous example. However, this was not foolproof since whether or not a parameterization is extended depends on where the `parameterize` is situated in the dynamic extent of the delimited continuation: if it is before the prompt, the extension is not visible, otherwise it is. Finally, we settled on manually passing around the parameterization between steps to have full control over what values the parameters we depend on have at any time. We have not yet experimented with using continuation marks [2] directly.

Figure 7 demonstrates the parameter loss issue. When the continuation from the first thread is restored in the second, the direct assignment to the parameter is lost and the program displays "p1".

4.2 Debugging

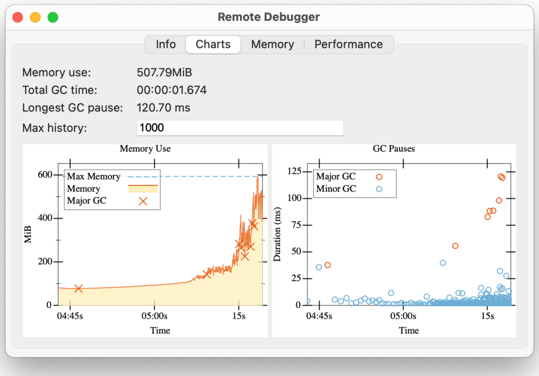
Debugging memory leaks in the presence of continuations is tricky. We had a set of small bugs in different areas of the system that were composing together to form a larger bug which led to massive memory leaks under load.

```
(define p (make-parameter #f))
(define tag (make-continuation-prompt-tag))
(define k-ch (make-channel))
(void
 (thread
  (lambda ()
    (call-with-continuation-prompt
     (lambda ()
       (parameterize ([p 'p1])
         (p 'p2)
         ((call-with-current-continuation
           (lambda (k)
             (thunk (channel-put k-ch k)))
           tag))))
     tag))))
(thread-wait
 (thread
  (lambda ()
    (define k (channel-get k-ch))
    (call-with-continuation-prompt
     (lambda ()
       (k (λ () (printf "~s~n" (p)))))
     tag))))
```

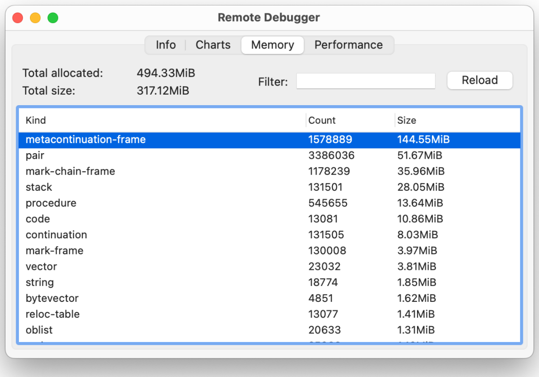
Figure 7. An example of the parameter loss issue.

First, our error reporting library was setting up exception handlers in its inner data collection loop, making the loop no longer tail-recursive. Second, our own middleware to configure the aforementioned error reporting library was accidentally creating a new instance of the error reporter per request, instead of reusing a single one, meaning that for every new request we would spin up a new data collection thread with a non-tail-recursive inner loop. Finally, we were using composable continuations to implement a special type of return from a sub-study to its parent, so when a participant continued a study at this boundary between parent and sub-study, we would see an increase in memory usage from stacking the composable continuations on top of each other.

Figure 8 shows what this type of issue looks like when visualized using dbg [6], a remote debugging tool for Racket. We can see memory use grow exponentially and that this stems from allocating a lot of "metacontinuation-frame" values. This drew our attention to our use of composable continuations, which we promptly changed to delimited-but-not-composable continuations, since we didn't actually need composable continuations for our purposes. Our use of composable continuations amplified the other two bugs, and this change seemed to fix the issue by drastically reducing the effect of the memory leak. In a way, this fix gave us a false sense of security, since the other two problems were still lurking, so we were surprised to later run into the same



Memory use: 507.79MiB
Total GC time: 00:00:01.674
Longest GC pause: 120.70 ms
Max history: 1000



Total allocated: 494.33MiB
Total size: 317.12MiB
Filter:

Kind	Count	Size
metacontinuation-frame	1578889	144.55MiB
pair	3386036	51.67MiB
mark-chain-frame	1178239	35.96MiB
stack	131501	28.05MiB
procedure	545655	13.64MiB
code	13081	10.86MiB
continuation	131505	8.03MiB
mark-frame	130008	3.97MiB
vector	23032	3.81MiB
string	18774	1.85MiB
bytevector	4851	1.62MiB
reloc-table	13077	1.41MiB
oblist	20633	1.31MiB

Figure 8. A remote debugger for Racket showing a spike in memory usage and the values that are taking up that memory.

problem again. Eventually, we were able to find the root problems and fix them.

5 Reflections & Recommendations

Continuations allow us to write web code in a direct style, simplifying the job of embedding domain specific languages in a web context. Without leveraging continuations, the study harness introduced in section 2 would have been a lot more cumbersome to implement than a straightforward depth-first traversal of a tree. The issues we encountered were primarily due to unexpected interactions between dynamic variables and delimited continuations. We recommend that functional architects avoid combining dynamic variables and continuations in their systems where possible, or do so with care, while taking into account the issues presented in section 4. If we were to rewrite Congame today, we would explicitly pass around a context object containing the data we need between steps instead of using dynamic variables. We also recommend that functional architects carefully consider whether they need composable *and* delimited continuations, or whether delimited alone will suffice.

To aid debugging, languages should provide tooling to allow continuations to be inspected at runtime. That way, when encountering issues such as the one presented in section 4.2, programmers would have an easier time finding the source of memory leaks. For example, in the case of Racket's "metacontinuation-frame" values, it would be helpful if those values were inspectable to determine their source location and what other objects they hold references to.

We believe continuations are the right abstraction for implementing interactive systems as targeted by Congame, such as surveys and market games, as well as any other system that requires some computation to be suspended until the user takes action (e.g. shopping carts, or simulations where part of the computation is delegated to another black box, etc.). For our use case, where backtracking via the browser's "Back" button is undesirable, multi-shot continuations are not required but, in other applications, they may be useful. In that sense, in a language without continuations, coroutines would likely provide us with the same benefits, but would not be as good a fit for use cases where the possibility of branching the user's progress through an interaction is a desirable feature (e.g. having the ability to open a separate tab to take a different path through a study tree). Other approaches, such as regular web programming with manual routing or a weaker form of "continuations" where URLs get mapped to closures, would not permit us to implement the core study harness in such a direct and simple way.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and suggestions. This work was partially funded by Central European University Private University.

References

[1] Daniel L. Chen, Martin Schonger, and Chris Wickens. oTree—An open-source platform for laboratory, online, and field experiments. *Journal of Behavioral and Experimental Finance* 9, pp. 88–97, 2016. doi:10.1016/j.jbef.2015.12.001

[2] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an Algebraic Stepper. *Lecture Notes in Computer Science* 2028, pp. 320–334, 2001. doi:10.1007/3-540-45309-1_21

[3] Matthew Flatt and PLT. Reference: Racket. PLT Design Inc., PLT-TR-2010-1, 2010. https://racket-lang.org/tr1/

[4] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. *ACM SIGPLAN Notices* 42, pp. 165–176, 2007. doi:10.1145/1291220.1291178

[5] Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT scheme Web server. *Higher-Order and Symbolic Computation* 20, pp. 431–460, 2007. doi:10.1007/s10990-007-9008-y

[6] Bogdan Popa. dbg. 2021. https://github.com/Bogdanp/racket-dbg

[7] Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *ACM SIGPLAN Notices* 38, pp. 57–64, 2003. doi:10.1145/772970.772977

Received 2024-06-03; accepted 2024-06-30