bashcc: Multi-prompt one-shot delimited continuations for bash

Spencer Baugh
University of Carcosa
104 Lost Beach
Carcaso, Hyades
first.last@gmail.com

Dougal Pugson
Pugson's C++ Crypt LLC
New York, USA
dougalpugson@gmail.com

Abstract

Among functional programmers, continuations are well known for the influence they have on the simplicity and understandability of a program. And among sysadmins, the bash programming language is renowned for the maintainability of programs written in it. Unfortunately, until this point, shell programmers have been denied the ability to use continuations in their programs. We provide an implementation of delimited continuations in GNU bash. This will provide a more familiar programming environment for functional programmers writing bash, and give bash programmers access to the advanced abstraction techniques of modern functional languages. We provide implementations of exceptions, early return, and coroutines as motivation, and outline areas for future work.

CCS Concepts •**Software and its engineering** →**Control structures; Scripting languages;** •**Social and professional topics** →Offshoring;

Keywords SIGBOVIK, effect handler, delimited continuations, bash, shells

ACM Reference format:
Spencer Baugh and Dougal Pugson. 2018. bashcc: Multi-prompt one-shot delimited continuations for bash. In *Proceedings of SIGBOVIK, Pittsburgh, PA USA, April 2018 (SIGBOVIK 2018),* 4 pages.
DOI: 10.475/123_4

1 Introduction

From the beginning[1], there has been a close relationship between the Unix shell and functional programming. The popular functional programming technique "currying" was first developed as part of the BSD Unix shell in 1983, as an implementation detail of the "rsh" monad. Indeed, the term "function" itself comes from the notation used to begin a subroutine in a Unix shell script, `function`.

As time has gone on, these two communities have grown in separate directions. The functional programming community and shell scripting community have each developed their own advanced techniques, with little cross-pollination.

We sought to rectify this by by porting a simple technique from functional programming languages to GNU bash. Delimited continuations are a straightforward mechanism, useful for implementing dynamic variables, going backwards in time, and in a pinch, providing a healthy meal of *continuatí al pomodoro*.

[1] the point at which our universe's continuation is delimited by the Prime Reset

SIGBOVIK 2018, Pittsburgh, PA USA

DOI: 10.475/123_4

Unix shells have long had "shift", one half of what is necessary for delimited continuations. Unfortunately, Ken Thompson apparently didn't read Oleg Kiselyov's website closely enough, so he didn't bother to implement "reset" as well.

Rather than build our continuation library on top of the existing "shift" infrastructure, we thought it was best to get a clean start. Hence we implemented "run" and "yield", following the naming convention of [1], but extending to work with multiple prompts.

For more background on delimited continuations, consult http://okmij.org/ftp/continuations/#tutorial or [2].

Our paper is organized in some number of sections. Section 1 contains an introduction and a description of the organization of the paper. Section 2 demonstrates several applications. Section 3 gives an overview of the implementation of delimited continuations in bash. Section 4 concludes the paper, discusses future work, and affirms that this was the right thing to do.

2 Applications

Our API is based on "run" and "yield", and "invoke" to resume continuations. For more details on the API, see 3.

2.1 Exceptions

We have implemented exceptions in bash using delimited continuations. The implementation of `raise` is simplicity itself:

```
function raise() {
    yield $exception_handler "$@"
}
```

We can raise whatever string we want as an exception by yielding it to the handler. `try` is implemented as follows.

```
function try() {
    local exception_handler=$(make_prompt)
    response=$(run $exception_handler "$@")
    while :;
    do
        if [[ $response =~ $yield_re ]]
        then
            continuation="${BASH_REMATCH[1]}"
            exception="${BASH_REMATCH[2]}"
            return 1
        elif [[ $response =~ $return_re ]]
        then
            file="${BASH_REMATCH[1]}"
            cat $file
            return 0
        fi
    done
}
```

Beautifully simple! We run an expression under a prompt, and case on the possible responses. If the expression returns without throwing, we propagate the result up (with `cat`). If the expression does throw, we'll see that as a yield, and we set "exception" and return non-zero.

Note that to achieve the dynamic scoping behavior of exception handlers, this uses bash's "local" keyword, which provides dynamic scope in bash.

Then usage is as simple as this:

```
despicable=12
function liker() {
    if [[ $1 -eq $despicable ]];
    then raise "I don't like $1!!!"
    else echo "I like $1" >&2
    fi
}
function evaluator() {
    seq 20 | while read number;
      do liker $number;
    done
}
function main() {
    try evaluator || {
        echo "exception was $exception"
        exit 1
    }
}
```

This will print numbers, until it gets to a despicable number (12), and then throw an exception and abort. Even such practical functionality as this is improved by delimited continuations!

2.2 Early return

Dealing with exceptions is hard and annoying, so we have implemented an "early return" functionality, which allows a function to immediately return a value at the enclosing prompt, without requiring a wrapper function or forcing the user to catch an exception.

Consider a simple recursive function which multiples all its arguments:

```
function multiply_args() {
    first=$1; shift
    if [[ $# -eq 0 ]]; then
        echo $first
    else
        rest=$(multiply_args "$@")
        echo $(( first * rest ))
    fi
}
multiply_args 1 2 0 4 5
```

As expected, this prints 0.

However, this function is very inefficient! It keeps iterating over its arguments even if it sees 0! We can improve it with continuations. We add this `early_return` functionality.

```
function early_return() {
    prompt=$1; shift
    file=$(name early_return.XXXX)
    echo "$@" > $file
    file_send_line $prompt "return: $file"
    exit 0
}
```

Then we can change our function to call `early_return` if it sees a 0.

```
function multiply_args() {
    prompt=$1; shift
    first=$1; shift
    if [[ $# -eq 0 ]]; then
        echo $first
    elif [[ $first -eq 0 ]]; then
        early_return $prompt 0
    else
        rest=$(multiply_args $prompt "$@")
        echo $(( first * rest ))
    fi
} #$
prompt=$(make_prompt)
run_with_prompt $prompt multiply_args \
  $prompt 1 2 0 4 5
```

Now it will be much more efficient, because it will only iterate down to the first 0 in the argument list, and then early return to print 0 immediately!

We omit the straightforward implementation of `run_with_prompt` as to not overly inflate our page count.

2.3 Coroutines

Coroutines are a recent programming technique invented by the "Golang" programming language. It allows several concurrent sequential processes to communicate by sending messages over a channel. While this sounds useless, it does have a few niche applications, and so we have implemented it in bash. As is traditional for coroutine implementations, we have not implemented actual parallel execution for coroutines.

Due to censorship by the squeamish weaklings on the SIGBOVIK review committee, we were not able to include our coroutine implementation in this paper.

But we wish to assure you that coroutines definitely work with this framework. The traditional primitives of "spawn", "send" and "recv" are all there.

We have received queries about whether we used the bitwise-or operator, `|`, in our coroutine implementation. While we are confused why anyone would ask such a thing, we would like to assure you that `|` is not used for any of the core functionality in our bash coroutine implementation.

Please view our implementation on Github for more: https://github.com/catern/bashcc.

3 Implementation

The implementation is guided by the insight of [3], which implemented multi-prompt one-shot delimited continuations using threads and synchronization primitives.

It occurred to the author that such thread-based continuations could be made multi-shot trivially, by simply switching the implementation to use processes and message passing instead, and then when invoking a continuation (by passing a message) using fork on the receive side to duplicate the continuation.

We discovered that continuations and processes have the follow correspondence:

Continuations	Processes
shift	blocking message send
reset	blocking message receive
invoking a continuation	replying to message

Understanding and formalizing this correspondence is left as an exercise for the reader.

Unfortunately, bash doesn't actually support fork, that is, returning twice. It only supports spawning new processes. This contrains us to implementing only one-shot continuations. As bash is a unityped language, supportingly only one type, we felt it was was thematically appropriate that its continuations be unicontinuations, supporting only one invocation.

Also, since bash does not have message passing functionality natively anyway, and it's a major pain to deal with pipes in bash, we implemented our message passing by reading and writing to files.

First off, we define creating new prompts as creating a new empty file.

```
function make_prompt() {
    prompt=$(name prompt.XXXX)
    touch $prompt
    echo $prompt
} #$
```

We pass the path of the file whenever we need to refer to the prompt.

Next, we define "run", and its return value. "run" takes a prompt and a command and runs the command under the prompt. If the command yields, then "run" returns the yielded value, along with the continuation, tagged with "yield:". If the command returns, then "run" returns the return value, tagged with "return:".

If the command is going to yield, it needs to take the prompt as an argument explicitly, because "run" does not pass the prompt in.

```
function run() {
    prompt=$1; shift
    prompt_size=$(file_size $prompt)
    stdout=$(name stdout.XXXX)
    { "$@";
      file_send_line $prompt \
        "return: $stdout";
    } >$stdout </dev/null &
    file_wait_for_one_line $prompt $prompt_size
}
```

"yield" takes an arbitrary line of data, sends it to the prompt (appropriately framed as a yield), then returns the line of data that the prompt replied with.

It's here that we create the continuation. The continuation, like the prompt, is a file. We send the filename of our continuation to the prompt along with the yield data. The prompt will write the resume data to our continuation file, and we'll resume.

```
function yield() {
    prompt=$1; shift
    continuation=$(name continuation.XXXX)
    touch $continuation
    file_send_line $prompt \
      "yield: $continuation message:" "$@"
    file_wait_for_one_line $continuation 0
} #$
```

Finally, we need to be able to invoke a continuation. "invoke" takes a prompt, a continuation, and an arbitrary line of data, and resumes that continuation with that data under that prompt. It returns the next YieldValue, just like "run".

```
function invoke() {
    prompt=$1; shift
    continuation=$1; shift
    prompt_size=$(file_size $prompt)
    file_send_line $continuation "$@"
    file_wait_for_one_line $prompt $prompt_size
}
```

For more details on the implementation, we have made our bash code available in the form of a Github repository, https://github.com/catern/bashcc.

4 Conclusion

We find that delimited continuations in bash are a great improvement in expressive power, and allow us to implement coroutines, early return, exceptions, and state, all in native, bare-metal 100% pure uncut Colombian GNU bash.

4.1 Future work

4.1.1 Multi-shot continuations

The most obvious direction for future work is to support multi-shot continuations. The missing piece is the ability to fork in bash. There are several ways this could be achieved.

We could provide a new bash builtin which exposes "fork" as a primitive. Unfortunately, that would need to be compiled against the end-user's bash system, an unacceptable deployment problem.

Instead, we can use gdb to attach to bash, force it to execute a "fork" syscall, and then detach. This is easy to deploy since most systems already have gdb installed, so this approach has absolutely no issues at all.

4.1.2 Type-and-effect system

We could provide a type-and-effect system for bash. Since bash does not have a type system, this would actually just be an effect system, no types.

This could likely be implemented through an additional compilation phase of bash, using a notion called "name inference". In the tradition of Scheme ending effectful functions in "!", we would rename functions to include their effects. For example, a function `f` which uses exceptions and IO would be automatically renamed to `f__effects:_exceptions_io`. Then it would be impossible for a programmer to use a function without knowing its effects, as it should be.

4.2 Conclusion conclusion

We conclude our conclusion with the hope that many new bash programs are written using these features. With these features, other languages such as Python and OCaml are permanently obsoleted, and bash is triumphant. We expect that the authors are likely to remembered forever by the maintenance programmers of the future.

References

[1] Roshan P. James and Amr Sabry. 1983. Yield: Mainstream Delimited Continuations.

[2] Oleg Kiselyov. 2012. Delimited control in OCaml, abstractly and concretely. *Theoretical Computer Science* 435 (2012), 56 – 76. DOI: http://dx.doi.org/https://doi.org/10.1016/j.tcs.2012.02.025 Functional and Logic Programming.

[3] Sanjeev Kumar, Carl Bruggeman, and R. Kent Dybvig. 1998. Threads Yield Continuations. *LISP and Symbolic Computation* 10, 3 (01 May 1998), 223–236. DOI: http://dx.doi.org/10.1023/A:1007782300874