Modular Rollback through Control Logging[†]

Twin pearls in continuation, and a monadic third

Olin Shivers (shivers@ccs.neu.edu)
and Aaron Turon (turon@ccs.neu.edu)
Northeastern University

Conor McBride (conor.mcbride@strath.ac.uk)
University of Strathclyde

Abstract. We present a technique, based on the use of first-class control operators, enabling programs to maintain and invoke rollback logs for sequences of reversible effects. Our technique is modular, in that it provides complete separation between some library of effectful operations, and a client, "driver" program which invokes and rolls back sequences of these operations. In particular, the checkpoint mechanism, which is entirely encapsulated within the effect library, logs not only the library's effects, but also the client's control state. Thus, logging and rollback can be almost completely transparent to the client code.

This separation of concerns manifests itself nicely when we must implement software with sophisticated error handling. We illustrate with two examples that exploit the architecture to disentangle some core parsing task from its error management. The parser code is completely separate from the error-correction code, although the two components are deeply intertwined at run time.

Keywords: checkpoint, delimited control, functional programming, effect logging, error handling

Prologue

We all make mistakes. What counts is how we handle them.

There's little less pleasant in programming than watching beautiful code turn ugly. But our programs live in a messy world, full of erroneous input. To cope, they must all too often become messes themselves.

The problem is that error handling is not naturally modular. It is context-dependent, often relying on the details and current state of input processing. It is poorly structured, because it must make sense of the poorly-structured input it is given. Extricating it into a separate prepass means duplicating much of the processing intended for correct input—and when input is provided and processed interactively, checking and processing *must* be interleaved. For complicated artifacts like typecheckers, which are hard enough to get right in the first place, code

[†] This is a revised and extended version of an earlier conference paper (Shivers and Turon, 2011).

to produce friendly error messages can obscure or even dwarf the code to check types (Lerner et al., 2007).

We tell a story in three acts, the moral of which is: error handling can be separable, even beautiful. The hero of our tale is `call/cc`, for to err is human, but to forgive requires first-class control.

On further reflection (Filinski, 1994), it will become apparent that our hero's mission is one of infiltration and liberation: first-class control can stealthily invade an otherwise-pure function in order to free latent monads.

Part one takes place in an "eager" Scheme REPL, which parses complete s-expressions as the user types each closing parenthesis, long before a carriage return. Yet, despite advancing the state of the parser, the user can interactively alter erroneous text by "backing up" with the delete key and re-entering corrected text. So here the error-handling is done, in part, by the user, but done in a way that's invisible to the input-processing code.

Part two retells the tale in Standard ML, using a similar technique to automatically discover and repair the "true" source of a syntax error, perhaps far removed from the location where the parser got stuck. It's the functional programming equivalent to Burke-Fisher error repair (Burke and Fisher, 1987). Again, the key benefit is the complete separation between the parsing algorithm and the repair algorithm: you write a parser in any style you like, and send it through our `BurkeFisher` functor to get a new parser that intelligently repairs errors.

The first two parts use first-class control in two crucial ways:

- First, to provide rollback logging. We layer an interface on top of `call/cc`, constructed so that the very performance of a reversible side-effect has an additional effect: it logs the appropriate reversal. Control effects and other effects smoothly interleave, whether going forward or backward.

- Second, to provide modularity: we sneak checkpointing through an unsuspecting parser. Although parsing and error correction *functionality* is interleaved, the *code* is not. Parsing is done via standard recursive descent, with no concern for error correction; likewise, error correction needs only understand the structure of the input to the parser.

Part three pulls back the curtain, using Haskell to reveal the monadic side of the story that was there all along. The key is to think of input *acquisition* as monadic, even when there are no effects in sight as in, *e.g.*, reading lexemes from a functional stream. Once the monadic interface is made explicit (as a *free* monad), we can choose to interpret it in various ways. The "vanilla" interpretation yields the original parser/lexer

pairing, with no rollback or error handling. But we can instead choose an interpretation that checkpoints each use of monadic bind—just as `call/cc` had done previously—and allows jumping backwards in time to correct erroneous input. The downside is that the monadic signature is explicit, which requires changing the interface between the lexer and parser, and writing the latter in an explicitly-monadic style. The upside is that the monadic signature is explicit, which clarifies the code and makes it easier to swap in additional effect interpretations (*e.g.*, different repair strategies).

We hope that while savoring the stories below, you will see that the rollback techniques transcend them.

1. Prompt reading and effect logging in Scheme

In our first example, we show how to recreate, with modern tools, a bit of "lost art" from classic Lisp input systems: how to intertwine input parsing with terminal-entry text editing without having to intertwine the code that carries out these separate tasks. As we do this, we'll discover a general technique for constructing "command" systems that allows us to transparently *log effects* in a way that permits them to be rolled back on demand, without having to compromise the simplicity of client code that issues sequences of these commands.

That's the big picture; let's begin by diving into the specifics of our example. S-expressions, the standard syntactic form for programs and data in Lisp and Scheme, have an interesting property: they are frequently self-delimiting. For example, we do not need to look past the terminating right parenthesis in the string "`(+ x y)`" to know that we have reached the end of the form. By way of contrast, this is not true of numerals: if an s-expression reader sees the string "`387`", it must look ahead one character to know if this string comprises an entire term, or if it is simply a prefix of a larger term. If the following character is a left parenthesis, then the string represents the integer three hundred eighty-seven (which happens to be followed by some list to be read at a later date); if the following character is another digit, then our string is simply the first three digits of some longer numeral.

Table I gives the breakdown for standard Scheme s-expressions, showing which kinds are self-terminating, and which aren't.

The details vary across dialects of Lisp and Scheme, but the general property remains.

Interactive systems that read commands or other structured input from a user frequently take their inputs in the form of an s-expression—the classic example is the Lisp read-eval-print loop. In such a system,

Table I. Termination of standard Scheme s-expressions

Self-terminating	Lists:	`()`, `(+ x 3)`, ...
	Vectors:	`#()`, `#(-2 7)`, ...
	Strings:	`"fred"`, `"dog"`, ...
	Booleans:	`#f`, `#t`
	Characters:	`#\z`, `#\q`, ...
Not self-terminating	Symbols:	`fred`, `dog`, ...
	Numerals:	`387`, `0`, ...

the human user enters some command, expression or definition at the terminal; the operating system passes this text to the Lisp process, which parses it into an s-expression, and then computes the command, expression, definition, *etc.* that it describes.

It turns out that old, 1970's- and 1980's-era Lisp systems (*e.g.*, Maclisp and Lisp Machine Lisp) took advantage of the self-delimiting property enjoyed by their s-expression-based interaction languages. When reading s-expressions, these interactive systems would read characters from the terminal eagerly—that is, they turned off the operating system's buffering machinery, so that each character was presented to the reader as soon as the user hit the key on the keyboard. Thus, the parse would complete the instant the terminating close-parenthesis was entered, without requiring the user to enter a redundant newline. It's a small but pleasant convenience that, in the post-Lisp-Machine age, has become lost from the text-interaction modality.

What makes providing this capability tricky is the task of error handling. Human typists make mistakes, which they'd like to correct as they enter their text. This is usually handled by the so-called "line discipline" code in the operating system's terminal device driver. This code buffers terminal input and provides a simple text editor for operating upon this buffered text: most characters are taken as input from the user, and echoed back to the terminal device, to make them visible on entry. A few keys, however, are reserved for this editing task. On a Unix system, for example:

– The delete key "backs up" one character: the OS removes the last character from the input buffer and outputs a backspace-space-backspace sequence to the terminal to erase it from the screen.

– The control-U character typically erases the entire line of text, erasing the entire buffer's worth of input.

– The characters control-C and control-Z direct the OS to send an asynchronous control signal to the reading process.

– Control-V turns off any special treatment of the following character.

– Control-D represents end-of-file.

The operating system does not release the buffered input to a process attempting to read from the terminal device until the user enters a newline character: this commits the input sequence.

Now, in order to provide eager reading, we have to turn this OS-provided machinery off: the user process must now be responsible for echoing input and implementing the interactive editing that humans use to back up through mistaken input and make corrections. This is not so terrible; we can encapsulate the editing code in a small library and be done with it.

What's tricky is that the parsing computation is now *interleaved* with the text-editing computation. Formerly, these things were split into distinct phases. First, we enter a string, with no parsing going on at all; during this phase, we can perform editing on each line as we enter it. Once we strike the newline key, the final version of that line is shipped off to the parser phase (*i.e.*, the interpreter), which consumes the text, produces a parse tree (that is, an s-expression), and proceeds with the requested task.

In our new (well, 1970's) world, the application is carrying out the parse as text is entered by the user. If we change our minds and wish to erase some of the text we've entered, as we back up through the input text, we also have to *rewind the parse computation*. For example, an s-expression reader is typically a little recursive-descent parser. When it is reading the elements of a nested list and it encounters a close-parenthesis character, it returns the accumulated list to its recursive caller. If we were to subsequently decide to *delete* that close parenthesis, we'd have to back the parse computation back down into that formerly completed call.

What we need is the ability to take "snapshots" of the computation at various points as we parse. Roughly speaking, every time the parser calls out to the "read character" routine, that routine should first checkpoint the parse computation and save the checkpoint away on a list. If the user tries to delete some previously entered characters, the input routine can pull the appropriate checkpoint from its saved history and reset the parse computation to that previous character-reading state.

By this point, it should be clear that the right tool for this job is the Scheme `call-with-current-continuation` (or, `call/cc`) procedure: creating computational checkpoints is its *raison d'être*.

The rubout port and the reset protocol

In Scheme, we do I/O on *ports*; our task is to construct a new kind of "rubout port" for reading from a terminal. We'll construct our rubout port using a low-level mechanism in Scheme48 (Kelsey and Rees,) and scsh (Shivers et al., 2004) that permits programmers to define their own input ports, which can be passed to the system input procedures like any other input port. To define one of our custom rubout ports, we provide the extended-port constructor two things. The first is a fixed suite of "port methods," described by a record whose fields are procedures the input system will use to read a character, "peek" at a character, close the port, and a few other less-important operations. We also provide a data value which encapsulates the port state; this value will be passed to the method procedures when operations are performed on the port. Here is the definition of the rubout port's data record:

```
(define-record rubout-port-data
  ttyin               ; input tty port
  ttyout              ; output tty port, for echoing
  (prev-tty-info #f)  ; tty's original echo state
  (peek-char     #f)  ; "lookahead" peek cache
  (checkpoint    #f)) ; most recent checkpoint
```

This piece of Scheme code defines a procedure

`(make-rubout-port-data` *ttyin ttyout*`)`

which constructs a record from a pair of ports connected to the terminal device; the other three fields of the record are initialised to false, `#f`. We also get field-accessor procedures, such as

`(rubout-port-data:ttyin` *rpd*`)`
`(rubout-port-data:ttyout` *rpd*`)`

and so forth, and some analogous field-assignment procedures with names like `(set-rubout-port-data:peek-char!` *rpd char*`)`.

Note the `checkpoint` field. Our intention is that every time application code does a `(read-char` *rp*`)` operation that causes the rubout-port to actually get a character from the terminal, before returning the character to the caller, the code will also grab a continuation with `call/cc` and stow it away in the port's `checkpoint` field. If, during a later attempt to read, the rubout-port code reads, say, a "delete" character and decides to undo the previous read, it will be able to do so by fetching this saved value from the `checkpoint` field of the rubout-port's data record and

invoking it—this will reset the entire computation back to the previous read point.

The key piece of design we must pin down is the protocol used to invoke the checkpointed continuation. Specifically, a checkpoint is a Scheme continuation that must be applied to a single boolean argument: `(checkpoint just-1?)`. If the `just-1?` argument is true, we wish to rewind the computation back just one step—that is, to the immediately prior read operation. If the argument is false, then we wish to rewind all the way back to the beginning of the entire read session.

The core effects: reading and echoing

We can now define the pair of low-level procedures that perform our system's two primitive side-effects: reading a character from the keyboard, and echoing an input character to the display. Since each of these procedures carries out an effect, it must log the effect as it performs it—that is, we must update the port's checkpoint to add a rollback handler that will undo the effect if we rewind back through this point in the computation.

Here is the code for echoing:

```
(define (echo c pd)
  (let* ((oport (rubout-port-data:ttyout pd))
         (reset (rubout-port-data:checkpoint pd)))
    (write-char c oport)                ; 1: Echo the character.

    (set-rubout-port-data:checkpoint pd  ; 2: Set a checkpoint
      (call/cc (λ (ret)                  ;    to undo the echo.
                 (let ((just-1? (call/cc ret)))
                   (print-rubout-sequence oport)
                   (reset just-1?)))))))

;;; Output backspace/space/backspace to the port.
(define (print-rubout-sequence oport)
  (write-string "\b \b" oport))
```

The rubout-port machinery calls `echo` whenever it needs to echo a character just read to the terminal. The procedure writes the character out on line 4, and then logs the effect in the last half of the procedure by updating the port's checkpoint. This is the first piece of serious continuation manipulation we've performed, so we will trace through its execution carefully. The first, outer `call/cc` creates a return point `ret`; applying `ret` to some value *cp* will cause *cp* to be installed as the port's new checkpoint. The second, inner `call/cc` creates the actual checkpoint

continuation; `call/cc` passes this continuation to `ret`, so, just as we described, this continuation gets installed as the current checkpoint.

Now, consider what happens when this checkpoint is invoked at some later time, by fetching it from the rubout port's data record and applying it to some boolean argument. We'll reset the computation back to *now*: the inner `call/cc` will return the boolean value, so it will be bound by the `let` form to the variable `just-1?`. Then we'll proceed into the body of the `let`, which contains the rollback action: we write out a backspace/space/backspace character sequence to the terminal, which will erase the character we previously echoed back on line 4 of the code, then we'll pass the `just-1?` boolean on to *our* checkpoint. We do this because we are in the process of rewinding back to some prior input—the echo action is an output effect, not an input effect, so we need to continue rewinding the computation.

Our next procedure (which appears in Figure 1) is the primitive character-input procedure. When the rubout-port machinery needs to actually read a character, it applies `%read-char` to the port's data record. This code is, essentially, our "line discipline" driver code, written in Scheme. The procedure sits in a loop (the named-let function `lp`), which does an input operation on the actual terminal, binding the variable `c` to the character entered by the user. Then we perform a variety of actions, depending on the character. Inducing rollback is easy: if the character is the delete character, we apply the port's current checkpoint continuation `rubout` to true, which will abort what we're doing, and reset to the previous read (and also undo any echoing we might have logged in-between). That previous read will then be able to get a new character from the user and return it to the parser in place of the original character it had input back when it first ran. (We'll see the code that does this—the code at the target end of our reset-continuation's non-local jump—in just a moment.)

If the character is the line-kill character, control-U, then we apply the checkpoint to false; by the rules of the checkpoint invocation protocol, this will induce a rewind all the way back to the beginning of the entire parse session, clearing previously echoed characters from the display as we rewind.

If the user entered control-C or control-Z, the code sends the current process the appropriate OS signal, and then loops by tail-calling `(lp)`, to continue trying to read a character. (The process signals itself in a context that resets the terminal's echoing and buffering state to its original settings, but this is a fine point we can skip.)

These first few cases describe the text-editing functionality of our "device driver," where characters input by the user are not intended as data to be passed on to the process, but are rather interpreted directly

```
(define (%read-char port-data)
  (let ((rubout (rubout-port-data:checkpoint port-data))
        (iport  (rubout-port-data:ttyin port-data)))
    (let lp ()
      ;; Assert: (rubout-port-data:peek-data port-data) = #F.
      (let ((c (read-char iport))) ; real input!
        (cond
          ((rubout-char?   c)  (rubout #t)) ; DEL => rubout a char
          ((linekill-char? c)  (rubout #f)) ; ^U  => kill the sexp

          ((interrupt-char? c) ; ^C => interrupt
           ;; Undo the tty's raw/no-echo modes before signaling
           (with-rubout-port-restored port-data
             (signal-process 0 signal/int))
           (lp)) ; The process survived the signal => keep reading

          ((suspend-char? c) ; ^Z => suspend
           ;; Undo the tty's raw/no-echo modes before signaling
           (with-rubout-port-restored port-data
             (signal-process 0 signal/tstp))
           (lp))

          (else (let ((c (cond ; ^V => turn off special handling
                               ((knockdown-char? c) (read-char iport))
                               ; ^D => EOF
                               ((eof-char? c) *eof-object*)
                               ; Normal char
                               (else c))))

                  ;; Outstanding! We have read an actual character, C.
                  ;; Before returning it to the parser, set a new
                  ;; checkpoint, so that if we later wish to rub C out,
                  ;; we can come back here, get a new character, & resume.
                  (call/cc                      ; Therein lies the rubout.
                   (λ (ret)
                     (let ((just-1?
                            (call/cc (λ (ckpt) ; Set the checkpoint & return C:
                              (set-rubout-port-data:checkpoint port-data ckpt)
                              (ret c)))))
                       ;; On rewind, we'll bind JUST-1? & come here.
                       (if just-1?
                           (lp)           ; Either re-do the current read,
                           (rubout #f)))))))))))     ; or keep rewinding.
```

Figure 1. The checkpointing character reader

by the input-port system as requesting various actions. The final case (the `else` clause of the `cond`) is the actual-input case: we've read an ordinary character `c` (or we've encountered the end of file, or we've read the "knockdown" character, control-V, followed by any character at all). Since we can now be considered to have successfully accomplished an input side-effect, before we return `c` to our caller, we must first log the operation by updating the port's checkpoint. Just as with `echo`, this is done with a nested, double `call/cc` pattern. The first, outer `call/cc` simply captures the context we'll use to return `c` and proceed with the parse by returning from `%read-char`. The second, inner `call/cc` creates the checkpoint continuation, naming it `ckpt`. We then install `ckpt` into the port's data record, and apply `ret` to the character `c`, which will lead us to produce `c` as the return value of `%read-char`. What happens when, at some later time, the checkpoint continuation is invoked? What we want to accomplish, if this happens, is to rewind the computation back to *now*, then get a new character c' from the terminal and return that *instead* of the character `c` we just now produced.

That's exactly what happens. Suppose some future call to `%read-char` gets a delete character and so fetches the checkpoint continuation we just now created from the port's data record and applies it to true (*i.e.*, executes line 7 of the procedure). When the checkpoint continuation is applied to true, we'll rewind to "now," and the second, inner `call/cc` will return the true value—so it will be let-bound to the variable `just-1?`, and we'll proceed into the body of the `let` form, whose `if` conditional will jump back to the top of our original loop, `lp`, continuing our current read. Any character c' we get will be returned to our caller, so the net effect is that, after rewinding the computation to here, we'll proceed with c' instead of our originally-read character. In short, we undid the input effect and replaced it with a new one.

On the other hand, suppose at some point in the future the user enters the "line-kill" character, Control-U. Then the checkpoint continuation will be applied to false (by line 8 of that future invocation of `%read-char`). The computation will be reset to "now," and the inner `call/cc` will produce the false value, which will be let-bound to `just-1?`. So, instead of looping in our current read, we'll keep rewinding by applying *our* checkpoint to false, in the very last line of the procedure: `(rubout #f)`.

In other words, we don't have to keep an explicit stack of checkpoints. The "stack" is implicit in the *environment structure* of the various checkpoints: each checkpoint continuation has access to the next checkpoint back in the timeline: it is bound to the variable `rubout` in the current checkpoint's lexical scope. (Similarly, it is bound to the variable `reset`, in the checkpoint created by the `echo` procedure.)

Peeking and reading

All the heavy lifting in our system is accomplished by the `echo` and `%read-char` primitives. The rubout-port system calls them from its peek-character and read-character "method" procedures. Note a policy decision encoded in these procedures: a character is *not* echoed as soon as we get it from the keyboard: peeks don't count. This code doesn't echo a character to the display until the parser actually consumes it with a `read-char` operation.

```
(define (rubout-peek-char port-data)
  ;; A "subsequent" peek: don't checkpoint
  (or (rubout-port-data:peek-char port-data)
      ;; A "first" peek -- checkpoint it
      (let ((c (%read-char port-data)))
        ;; then set the peek cache:
        (set-rubout-port-data:peek-char port-data c)
        c))) ; then return the character.

(define (rubout-read-char port-data)
  ;; If the peek does input, it will be logged:
  (let ((c (rubout-peek-char port-data)))

    ;; If we echo, it will be logged:
    (if (not (eof-object? c)) (echo c port-data))

    ;; Clear the peek cache.
    (set-rubout-port-data:peek-char port-data #f)
    c))
```

Note also that these two higher-level procedures do not concern themselves with setting checkpoints or logging effects. One of the nice properties of our design is that the effect-logging code is tightly bound to the code that commits the effects. Thus, doing a side effect and logging its rollback handler are welded together. Client code, like the two procedures above, or the application's parser, just commit side-effects at will; it's impossible for them to break the pairing of effects and rollbacks. (This nice property is only extended to the side-effects that we included in our design, of course. Parser clients must be aware that they cannot perform other side-effects and expect them to be undone during rollback.)

All that remains is to define `with-rubout-session*`, the procedure used to delimit a single parse session; it appears in Figure 2. We can perform a rubout-enabled parse with something like:

```
(with-rubout-session* rubout-port
  (λ () (read rubout-port)))
```

The central body of this procedure executes in a dynamic context, established by the R5RS Scheme `dynamic-wind` procedure, that saves and restores the rubout port's checkpoint if we should throw out and then back into the session's dynamic extent by invoking saved continuations. It also serves another purpose: the `dynamic-wind`'s exit thunk clears the checkpoint from the rubout port, which permits the checkpoint to be garbage collected even if the rubout port itself continues to remain alive.

Note that the rubout-session's thunk executes in an exception-handler context that treats parser-syntax errors in a clever way: we refuse to accept input from the terminal that triggers a syntax-error exception. If the user should enter a character that causes the parser to raise this error, the exception handler "rings the bell" (in a visual manner), then discards the bad character by invoking the rubout port's current checkpoint, passing it true for its `just-1?` parameter. This rewinds the parse computation back to the read operation where the user entered the offending character (erasing the character as we rewind, if it had been echoed); we then resume the parse by reading an alternate character from the user.

For example, if the user tries to type in an ill-formed dotted-pair that has two dots, *e.g.*, `(a . b . c)`, the parser will stubbornly refuse to accept the second dot, ringing the bell every time the user attempts to enter it to signal that we have departed from the syntactic straight and narrow. Note that this facility is completely independent of the grammar we are parsing, or the details of the parser: the parser is just a piece of code that raises an error exception as soon as it encounters an illegal character.

The final task the rubout-session procedure must perform before invoking the client's parser computation `thunk` is to set the rubout port's initial checkpoint. Again, we see the nested, double `call/cc` pattern. Tracing through the `restart` loop shows that each time we rewind back to the initial checkpoint, we recreate and reinstall it into the port, then begin the parse (that is, call `thunk`) all over again. Thus, invoking the initial checkpoint has the effect of restarting the whole parse.

```
(define (with-rubout-session* rubout-port thunk)
  (let ((pd (extensible-input-port-local-data rubout-port))
        (suspended-checkpoint #f))

    (with-raw-mode-rubout-port pd  ; Turn off tty buffering & echoing.
      (dynamic-wind
        ;; Dynamic-wind pre:
        ;;   If we are throwing back into the parse, restore the
        ;;   checkpoint we saved away when we threw out.
        (λ () (set-rubout-port-data:checkpoint pd suspended-checkpoint))

        ;; Dynamic-wind body:
        (λ () (with-syntax-error-handler
                ;; Here's the error handler. If the parser raises a
                ;; syntax error, ring the bell, clear the peek-char cache,
                ;; then trigger a 1-step rewind to rubout the last char,
                ;; which triggered the error.
                (λ (c) (visible-bell (rubout-port-data:ttyout pd))
                       (set-rubout-port-data:peek-char pd #f)
                       ((rubout-port-data:checkpoint pd) #t))

                ;; Set the initial checkpoint, which comes back here and
                ;; restarts the whole parse.
                (call/cc (λ (go-start-parse)
                           (let restart ()
                             (call/cc (λ (icp)
                                (set-rubout-port-data:checkpoint pd icp)
                                (go-start-parse)))
                             ;; Come here when initial checkpoint ICP
                             ;; is invoked:
                             (restart))))

                (thunk))) ; Do the parse.

        ;; Dynamic-wind post:
        ;;   When we're done, clear out the port's checkpoint, so the port
        ;;   won't keep the checkpoint from being gc'd. But... we might not
        ;;   be done. We might be throwing out and later throwing back in.
        ;;   So save the current checkpoint in case we later throw back in.
        (λ () (set! suspended-checkpoint (rubout-port-data:checkpoint pd))
              (set-rubout-port-data:checkpoint pd #f))))))

(define (visible-bell oport)    ; A simple "visible bell:"
  (write-char #\! oport)        ;   print out a !,
  (sleep 1)                     ;   pause one second, then
  (print-rubout-sequence oport));   erase it.
```

Figure 2. Delimiting a rubout-handler session.

Discussion

The big idea The first thing to note about this rubout-handler system is that the technique is not at all limited to interactive text entry. The general idea here is that we have a library of effectful (but reversible) operations. Some client performs a series of effects by issuing a sequence of these operations; as the client computes, the application may decide to rewind the driving computation to some previous operation and do something different—perhaps (as in our rubout-handler scenario) in response to error conditions.

The design pattern we propose here is to instrument the effectful operation library to construct a rollback log for the operations it performs. More specifically, *by constructing this rollback log from continuations, we capture not only the library's effects, but also the client's control state at each operation-invocation point.* This is what permits completely transparent rollback of the client computation: it's all due to the power of `call/cc` to package up a general control state.

Delimiting our checkpoints Let's motivate our next point by considering an extension to our rubout-handler system. Many text-input systems have some kind of "history" mechanism: they save previous lines of input, which the user can recall by entering some special control key that cycles through previous entries. Suppose we wished to provide this kind of functionality—and, of course, keep our rubout-handling capability. Unfortunately, the saved continuations that make up our checkpoints capture *too much control state:* they capture not only the parse computation, but also the computational state of the client that called the parser. If we were to reset to one of the checkpoints from a previous read in, *e.g.*, a Scheme interpreter's read-eval-print loop, we'd reset the entire interpreter back to that earlier state!

This is a problem that is exactly handled by delimited-control operators such as Felleisen's `prompt` (Felleisen, 1988; Sitaram and Felleisen, 1990) or Danvy and Filinski's `shift` (Danvy and Filinski, 1990). We need only delimit the parse computation carried out by the thunk passed to `with-rubout-session*` and we're set. We leave this modification to our code as a (fun) exercise for the interested reader.

The dark side of Church encodings Scheme provides continuations encoded as procedures: we perform a non-local control transfer to a captured continuation by applying it to some argument. This has been a long-standing source of subtle problems using Scheme's continuations. The central issue is that, if we want to call procedure `p`, with argument `a` and continuation `k`, we cannot do the following: `(k (p a))`. This goes

wrong because `k`'s underlying continuation does not actually become the *current* continuation until `p` returns; the whole time `p` is executing, the continuation is an extension of the one extant when we began evaluating the entire `(k (p a))` expression. So if `p` raises an exception, we will resolve it using the exception handlers of that continuation, not `k`'s handlers. If we were hoping to reclaim the original continuation's stack by installing `k` as the current continuation, this won't happen while `p` is executing—if it happens to be some long-running thread (such as a web server), then the original continuation's stack will *never* become free, leading to a subtle space leak. These kinds of space leaks aren't just theoretical oddities; they actually occur when programmers build thread schedulers based on continuations (Biagioni et al., 1998).

The real problem here is that when we Church-encode a kind of data, we can only do one thing with the data: apply it to an argument—that is the only operation permitted on procedures. In the case of continuations, we need to do two things: perform a function call with the given continuation for the call, and compose an $\alpha \to \beta$ function onto the "end" of a β-accepting continuation, producing an α-accepting continuation. These two procedures provide the necessary interface:

`(with-continuation` *cont thunk*`)`
`(compose-continuation` β-*cont* α`->`β`)`

Scheme doesn't have this kind of continuation mechanism, so we have to code in awkward ways to work around the limitations of `call/cc`.

This problem rears its head in our rubout-handler system. Consider the subtle, double-`call/cc` code that creates the new checkpoint when we read a character in `%read-char`. Couldn't we eliminate the inner `call/cc` and just use a simple procedure, instead of an exotic continuation, with the following?

```
(call/cc (λ (ret)
           (set-rubout-port-data:checkpoint port-data
             (λ (just-1?) (if just-1? (ret (lp)) (rubout #f))))
           c))
```

Unfortunately, no. If some future delete command applies this checkpoint to true, we will perform the `(lp)` retry *in that future context*; we won't throw back to the previous control state until the `lp` call returns to the Scheme-style Church-encoded continuation `ret`. All the time that `(lp)` is running, it is running with the wrong exception-handler context, the wrong dynamic-wind undo/redo handlers, and so forth. Furthermore, the run-time system can't reclaim the triggering read's stack and other control context until `lp` returns; we want to reclaim it when `lp` *starts*.

The double-`call/cc` pattern establishes the proper context, then captures it with the inner `call/cc`, so things work out properly. It would have been easier and simpler to write this code if we'd had the alternate continuation functionality described above. Our checkpoint-setting code would then look like this:

```
(call/cc (λ (ret)
           (let ((ckpt (compose-continuation ret
                         (λ (just-1?)
                           (if just-1? (lp) (rubout #f))))))
             (set-rubout-port-data:checkpoint port-data ckpt))
           c))


(call/cc
  (λ (ret)
    (set-rubout-port-data:checkpoint port-data
      (compose-continuation ret           ; compose a char cont
                            (λ (just-1?)  ; & a bool->char fun
                              (if just-1? (lp) (rubout #f)))))
    c))
```

An observant reader might similarly have wondered why we didn't use a simple procedure as the checkpoint for the `echo` procedure. This is why. As a matter of style, checkpoints in our rubout-handler system—that is, things stored in the `checkpoint` field of a rubout port's data record—are always and only continuations—that is, things created by `call/cc`.

This issue matters less in the case of `echo`'s checkpoints. If we'd written the checkpoint code for `echo` with the simpler

```
(set-rubout-port-data:checkpoint pd
  (λ (just-1?)
    (print-rubout-sequence oport)
    (reset just-1?)))
```

then the only code that would run in the wrong context would be the `(print-rubout-sequence oport)` call, which is short and presumably terminates with no other control effects such as raising an exception. Still, we preferred to work with the discipline of *only* using continuations for checkpoints.

Backwards and forwards An observant reader might also be wondering: old Lisp systems didn't have `call/cc` or general continuations. So how did these systems provide prompt reading with rubout handling?

The answer is rather ingenious (Pitman, 1995). These systems couldn't back up to prior checkpoints by invoking saved continuations. Instead, the port machinery would log all the characters read during a session. If the user requested a single-character delete, the rubout handler would raise an exception, which would be caught by an exception handler established at the beginning of the rubout-handling session. The exception handler would delete the last character from the log, and then *completely restart* the entire parse. During the new parse attempt, the port would be in a special "replay" mode: whenever the parser requested a character from the port, the port would get the next item from the log, instead of going to the terminal to read a new character. When the log was exhausted, the port would revert to doing actual input from the terminal. So, to delete a character, the rubout handler would simply reread and reparse everything *but* the deleted character. Instead of going one step back, the system would do a complete restart and then go $n - 1$ steps forward.

Why `call/cc`*?* Our final point is the following. Many programmers think of `call/cc` and delimited control operators as being the province of effete language theorists. To which we really must reply[1]: "*Au contraire!*" Continuations are rather tools for hearty, robust systems programmers: we've used them to write a line-discipline driver to replace the one provided by the Unix kernel, and we only needed about 200 lines of code... and our version provides more functionality.

2. The Burke-Fisher functor in SML

Enough s-expressions. Let's look at some *real* syntax:

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle exp \rangle \texttt{ + } \langle exp \rangle \\
& | & \langle num \rangle \\
& | & \langle id \rangle \\
& | & \cdots \\
\langle decl \rangle & ::= & \texttt{val } \langle id \rangle \texttt{ = } \langle exp \rangle \texttt{ ;} \\
& | & \texttt{fun } \langle id \rangle \texttt{ (} \langle id \rangle \texttt{) = } \langle exp \rangle \texttt{ ;}
\end{array}
$$

This is just enough grammar to get us into some interesting trouble. Imagine for a moment being a novice programmer sitting at the REPL. You type

```
> val f(x) = x + 1;
```

[1] In a hearty, robust manner, that is.

expecting to define your first function. But you are instead greeted by

```
SYNTAX ERROR: at (1:6), got '(', expected '='
```

Being a novice, you probably haven't seen the grammar above, which anyway would be much larger in practice. From the error, you only know that the first five characters drove the parser into a state from which `'='` is the only way out. If you also believe that `val` declarations are the sole way of creating bindings—or have forgotten whether it's `fun`, `fn`, `proc` or `function`—you're in for a long and frustrating REPL session.

A more helpful parser might instead respond with

```
> val f(x) = x + 1;
SYNTAX ERROR: at (1:1), did you mean 'fun'?
```

Startlingly, the *location* of the syntax error has changed in this interaction. The parser is recommending that `val` be replaced by `fun`, even though `val` is permitted by the grammar. This recommendation seems more helpful than replacing `'('` with `'='`, but why?

There is a general principle at work, one elucidated in a classic paper by Burke and Fisher (Burke and Fisher, 1987). The principle, loosely stated, is that

> *Syntax errors are usefully explained by finding minimal, nearby edits that allow the parser to substantially progress.*

Looking at our novice REPL session, we see:

- If we replace `'('` with `'='`, the parser will encounter another syntax error almost immediately: the now-unbalanced closing parenthesis. To recover from *that* error, we would probably delete several tokens—the initial replacement leads us straight into a syntactic quagmire.
- If instead we replace `'val'` by `'fun'`, which requires *backing up* from the location where the error was detected, that single-token change results in a grammatically-correct input.

In short, the `'fun'` replacement is better because it explains more of the programmer's original input.

Burke and Fisher outline two requirements for "practicality":

1. Error handling should not substantially increase the space or time needed to parse *correct* input.
2. Error analysis and recovery should take constant time.

The technique outlined in their paper relies on the details of explicit-stack LL and LR parsing, and works by maintaining *two* parser states. The first parser state represents the "real" parser, and is used to detect syntax errors. The second parser state lags some fixed k tokens behind. When the real parser encounters an error, it can be repeatedly reverted to the state of the lagging parser as different modifications to the input are tried. The winning modification is, roughly, the smallest one yielding the greatest amount of progress beyond the original error.[2]

While ingenious, Burke-Fisher error repair is also a vivid example of too-tight coupling between error handling and core logic: the error-handling code relies on complete knowledge of the representation and algorithm of the underlying parser. At the very least, it requires there to *be* an explicit representation of the parser's state, which would seem to rule out the technique for recursive-descent parsers. But as any functional programmer knows, the stack is a representation of a computation's state, and is only a `call/cc` away from being an explicit one.

Thus, using similar techniques to the eager REPL, we can liberate Burke-Fisher error repair from its assumptions about parsing, achieving clean separation between processing valid input and handling erroneous input.

Plot summary

The basic trick is to slide in error correction between a parser and its source of input, a lexer—about which we assume essentially nothing. As with the eager REPL, we do this by feeding the parser an instrumented, checkpointing lexer. To keep things interesting, we switch to SML, and use the module system to be very clear about the separation of concerns. In the end, we write a single functor, `BurkeFisher`, that can add error repair to *any* module of signature `PARSER`.

We use two extensions to the language: delimited control and higher-order functors, both of which can be had in SML/NJ (Kuan, 2011; Herman, 2007). Why? Read on.

Setting the scene: the signature of a parser

We start with the `LEXER` signature shown in Figure 3, which provides a type `tok` of tokens and `tok_stream` of token streams. We assume nothing about the internal structure of token streams, but can observe the next token and remaining stream, if any, using `lex`.

[2] To work in constant time, a test of a repair should only venture some fixed number of tokens beyond the original error.

```
signature LEXER =
sig
  type tok
  type tok_stream
  val lex : tok_stream -> (tok * tok_stream) option
end

signature PARSER =
sig
  type tok
  type result
  functor ForLexer(L : LEXER where type tok = tok):
  sig
    exception ParseError of L.tok_stream list
    val parse : L.tok_stream -> result
  end
end
```

Figure 3. The LEXER and PARSER signatures

A PARSER is parameterized by its LEXER. Here is our first use of SML/NJ's higher-order module system: a module implementing PARSER must include a functor, ForLexer, which can be applied to lexers with a compatible notion of tokens. ForLexer in turn provides a parse function specialized to the token stream of the given lexer. Of course, all this could be replaced by an appropriate use of polymorphism, but then, so can SML's module system (Rossberg et al., 2010). Explicit structuring via signatures, modules and functors allows us to clearly articulate and name the separate concepts with which we wish to work.

Because result is defined outside the ForLexer functor, the result type of a parser cannot depend on its lexer. On the other hand, the ParseError it raises on detecting a syntax error is parameterized by the token stream—in fact, it takes a *list* of token streams. This is the one concession a parser must make to error repair: it must signal the detection of an error by throwing an exception holding the stream just after the error was detected. If the parser performs backtracking choice, but every choice fails, it should throw the exception with a list containing *each* failure point.

A simple parser

The module DeclRecognizer in Figure 4 implements a recognizer for the ⟨*decl*⟩ grammar, providing an example realization of the PARSER signa-

```
structure DeclRecognizer =
struct
  datatype tok = VAL | FUN | LPAREN | RPAREN | ID | EQ | NUM
               | PLUS | SEMI
  type result = unit

  functor ForLexer (L : LEXER where type tok = tok) =
  struct
    exception ParseError of L.tok_stream list

    fun want t = fn s =>
      case L.lex s
       of NONE => raise ParseError [s]
        | SOME (t', s') =>
            if t=t' then s'
            else raise ParseError [s']
                   (* stream /after/ the error *)

    infix >>
    fun p >> q = fn s => q (p s)

    infix <|>
    fun p <|> q = fn s =>
      p s handle ParseError ss =>
      q s handle ParseError ss' =>
      raise ParseError (ss @ ss')

    val wantExp  = (* ... *)
    val wantDecl =
         (want FUN >> want ID >> want LPAREN >> want ID >>
          want RPAREN >> want EQ >> wantExp >> want SEMI)
     <|> (want VAL >> want ID >> want EQ >> wantExp >>
          want SEMI)
    fun parse s = wantDecl s; ()
  end
end
```

Figure 4. A combinator-style recognizer for ⟨*decl*⟩

ture. Being a recognizer, there are only two possible outcomes of `parse`: a unit value if successful, and an exception if not. The implementation of `DeclParser` uses a few parser combinators (Hutton and Meijer, 1998), which provide interaction with both the lexer and, ultimately, error repair.

The combinator `want` simply checks that the given token is the next one on the stream, returning an advanced stream if so and raising an exception if not. Usually we compose recognizers sequentially with `>>`, letting errors propagate. To handle nonterminals like ⟨*decl*⟩, which have multiple productions, we instead use `<|>`, which provides backtracking choice. With choice we see concretely the concession the parser must make to error repair: it must bundle together the possible error locations.

Checkpointing a lexer

Parsers are conveniently parameterized over lexers, leaving the perfect loophole through which to checkpoint parser state. Unlike the eager REPL, we will employ *delimited* continuations (Felleisen, 1988; Danvy and Filinski, 1990) for checkpointing. A delimited continuation captures the evaluation context only up to the most recent delimiter. Delimitation will allow us to tentatively run the rest of a parse with various repairs, searching for the best one, without going on to execute the rest of the program.

We use an implementation of delimited control, due to David Herman in an ICFP pearl (Herman, 2007), that is parameterized by a single *answer* type:

```
signature CONTROL =
sig
  type ans
  val shift : (('a -> ans) -> ans) -> 'a
  val reset : (unit -> ans) -> ans
end
```

Delimiters are inserted using `reset`, which expects an answer-producing thunk. On the other hand, `shift` aborts to the nearest delimiter, but reifies the evaluation context up to the delimiter as a reusable function. For example,

```
> (reset (fn () => shift (fn k => 1) + 1)) * 2;
2
> (reset (fn () => shift (fn k => k (k 1)) + 1)) * 2;
6
```

In the first case, the captured context [ ] + 1 is simply discarded; the reset is replaced by the answer 1, which is then doubled. In the second case, the captured context [ ] + 1 is applied, as a function, twice; the reset is replaced by the answer 3, which is then doubled.

For our purposes, a single delimiter surrounding an entire parse will suffice. The checkpoints captured by the lexer will correspond to the remaining execution of the parser, starting from its request for a token at some point in the stream, and continuing to the point where it returns the result of the parse. The type ans for a parser P will therefore be P.result.

The checkpointing instrumentation is performed by a functor that takes LEXERs to (instrumented) LEXERs. The instrumented lexer provides the same type of tokens as the original one, so it will be compatible with the same parsers. The type of *token streams*, however, differs. In particular, an instrumented lexer can operate in one of two modes, checkpointing or passthrough, and the current mode is part of the stream representation. Checkpointing is used during the initial parse, until a syntax error is found. The last k checkpoints are maintained in the stream state, using a window, so that instrumentation imposes only a constant-bounded space overhead:

```
signature WINDOW =
sig
  type 'a window
  val empty : 'a window
  (* keeps only last k pushes *)
  val push  : 'a window -> 'a -> 'a window
  val list  : 'a window -> 'a list
end
```

Figure 5 shows the instrumentation. In passthrough mode, instrumentation has no effect. In checkpointing mode, the lexer uses shift to abort while capturing the continuation up to the end of parsing. The resulting delimited continuation k is immediately invoked—effectively undoing the abort—yielding the same token t that the underlying lexer would have. However, a checkpoint is pushed that, when invoked, reruns the parser from the point of the shift, providing an alternative token t' and switching to passthrough mode.

In addition, an instrumented lexer provides wrap to inject an underlying lexer, unwrap to project the underlying lexer, and checkpoints to extract the last k checkpoints.

```
functor WrappedLexer (C : CONTROL) (L : LEXER) =
struct
  type tok = L.tok
  type checkpoint = L.tok -> C.ans
  datatype wrapper
    = CHECKPOINT of checkpoint Window.window
    | PASSTHRU
  type tok_stream = L.tok_stream * wrapper

  fun lex (s, PASSTHRU) =
      (case L.lex s
        of NONE => NONE
         | SOME (t, s') => SOME (t, (s', PASSTHRU)))

    | lex (s, CHECKPOINT w) =
      (case L.lex s
        of NONE => NONE
         | SOME (t, s') => SOME (C.shift (fn k =>
           (* first time through, yield t *)
             k (t,  (s', CHECKPOINT (Window.push w (fn t' =>
           (* on checkpoint invocation, yield t' *)
             k (t', (s', PASSTHRU)))))))))

  fun wrap s = (s, CHECKPOINT Window.empty)
  fun unwrap (s, _) = s
  fun checkpoints (_, CHECKPOINT w) = Window.list w
    | checkpoints (_, PASSTHRU)     = []

end
```

Figure 5. The checkpointing lexer

Putting it together: the Burke-Fisher functor

Nearly all the ingredients are now in place. However, to search for repairs, we'll need to know something about the available tokens:

```
signature TOK =
sig
  type tok
  val toks : tok list
  val toString : tok -> string
end
```

The list `toks` provides an instance of each token that should be used for replacement or insertion during error repair.

With that, we can specify the shape of functional Burke-Fisher error repair using SML/NJ's `funsig` form:

```
funsig BURKE_FISHER
         (T : TOK)
         (P : PARSER where type tok = T.tok) =
  PARSER where type tok = T.tok
```

Figure 6 gives an implementation—the `BurkeFisher` functor. To keep the presentation short, the functor is not quite true to the original algorithm:

- It only performs a single repair.
- It requires the repair to make the entire input valid, rather than choosing a repair that makes maximal (but bounded) progress. This means, in particular, that repair is not constant-time.
- It only attempts repairs of Hamming distance 1, that is, single-token replacements.

These limitations can be easily addressed by modifying the functor, without any change to the underlying parser.

When applied to a parser `P`, `BurkeFisher` produces a parser with result type `P.result * string option`. The string component is a description of the single repair (if any) performed.

Internally, the functor uses Herman's `GreatEscape` functor (Herman, 2007), which provides delimited control by using SML/NJ's call/cc facility. Crucially, this implementation of delimited control interacts properly with exceptions: delimited continuations captured by `shift` also capture exception handlers up to the delimiter—no more, no less. Thus, as Herman writes,

```
reset
  (fn _ =>
      (shift (fn k => (k 0)
                      handle Fail _ => 1))
    + (raise Fail "uncaught"))
handle Fail _ => 2
```

returns 1 rather than 2. Since we use exceptions to communicate parse failures, and thus exception handlers to institute repair, we rely on this behavior.

Once `BurkeFisher` is applied to a particular parser and lexer, it instantiates the parser with an instrumented version of the lexer, yielding

a module `UP` (for "underlying parser"). To parse an underlying stream `us`, it wraps the stream and feeds it to the underlying parser—within a `reset`, which sets the delimiter for the checkpointing lexer. The result of the parse is paired with `NONE`, meaning no repair was necessary, which is returned if all goes well. But *outside* this pair (in particular, outside the `reset`), there is an exception handler that kicks off the repair process. Placing the handler outside the `reset` guarantees it won't be captured in the checkpoints.

At last—when the underlying parser throws a syntax error, the `BurkeFisher` functor catches it and its list of wrapped streams `wss`. Each wrapped stream represents a possible syntax error within a choice. Concatenating the checkpoints from each stream, the repairing `parse` uses `tryCPs` to attempt a repair at each checkpoint in turn. If successful, `tryCPs` returns a pair of the `result` from the underlying parser and the replacement token it used to get that successful parse. Otherwise `tryCPs` returns `NONE`, and `parse` re-raises the parse error using the underlying lexer streams.

Supposing we've written modules `Tok` and `SimpleLexer` (which represents streams as lists of tokens), we can sit down at the REPL and see

```
> structure RP  = BurkeFisher(Tok)(DeclRecognizer);
> structure RPP = RP.ForLexer(SimpleLexer);
> RPP.parse [VAL,ID,LPAREN,ID,RPAREN,EQ, NUM,PLUS,NUM,SEMI];
((), SOME "Did you mean 'fun'?")
```

A caveat: choice points are commit points

There's a subtlety regarding choice: if we factor out " = ⟨*exp*⟩ ; ", which is common to both forms of ⟨*decl*⟩, the repair no longer works.

The problem is this: once a choice has been successfully parsed, only the checkpoints of the taken branch are retained. The effective result is that repair checkpoints do not always have access to all possible branches.

We used exceptions carrying a list of streams to join together the checkpoints of an unsuccessful choice. To deal with successful choices, we need to maintain all checkpoints even when backtracking—we need the backtracking control structure to play nicely with the checkpointing control structure. An easy and fairly pleasing way to do this is to parameterize the underlying parser by an implementation of back-

```
functor BurkeFisher (T : TOK)
                    (P : PARSER where type tok = T.tok) =
struct
  type tok = T.tok
  type repair = string
  type result = P.result * string option

  (* Herman's delimited control from call/cc *)
  structure C = GreatEscape(type ans = P.result)

  functor ForLexer (L : LEXER where type tok = tok) =
  struct
    exception ParseError of L.tok_stream list
    structure WL = WrappedLexer (C) (L)
    structure UP = P.ForLexer (WL)

    fun tryReps k []      = NONE
      | tryReps k (t::ts) = SOME (k t, t)
          handle UP.ParseError _ => tryReps k ts

    fun tryCPs []      = NONE
      | tryCPs (k::ks) = case tryReps k T.toks
                          of NONE    => tryCPs ks
                           | SOME rt => SOME rt

    fun parse us =
      (C.reset (fn () => UP.parse (WL.wrap us)), NONE)
      handle UP.ParseError wss =>
        case tryCPs (List.concat (map WL.checkpoints wss))
         of NONE =>
              raise ParseError (map WL.unwrap wss)
          | SOME (r, t) =>
              (r, SOME (concat ["Did you mean ",
                                T.toString t, "?\n"]))
  end
end
```

Figure 6. The Burke-Fisher functor

tracking choice—which `BurkeFisher` can then provide—which also allows `ParseError` exceptions to carry just a single stream:

```
fun choose p q = fn (us, w) =>
  p s handle ParseError (us', w') =>
  q (us, w')
```

Discussion

The big idea As with the eager REPL, the key benefit of control operators is the ability to hook into—even manipulate—the computation of a parser, without any understanding of how that computation is structured.

There are also two significant differences from the eager REPL. First, we assume that parsing and lexing are free from observable side effects, so we do not use effect logging (and, in fact, avoid state—see below). Thus, we are *only* logging control state. Second, by having a modicum of knowledge about the structure of input to the parser (via the `Tok` signature), we are able to intelligently explore the space of input repairs, while keeping the computational structure of the parser abstract.

Look Ma, no refs! Contra the eager Scheme REPL, we haven't used mutable state in implementing Burke-Fisher repair. We were able to avoid it because the `PARSER` signature requires parsers to use their lexers *functionally*—in particular, to thread the token stream through the parsing processes. The downside to this approach is that checkpointing is sensitive to the way the parser threads the token stream, which as we noted can cause problems when the parser itself backtracks.

But, of course, an imperative interface is also workable, and it has the benefit that *every* use of the lexer is checkpointed, regardless of how the token stream is threaded. What's more, the token streams can be dropped from the `ParseError` exception, further shrinking the interface between the parser and the repair functor.

To type-checking, and beyond Having generalized Burke-Fisher repair to arbitrary parsers, it's natural to wonder if we can go even farther. By encapsulating the notion of input streams and local repairs in a separate module taken as an extra parameter, the `BurkeFisher` functor could be pared down to handling just the checkpointing and error-catching process. We suspect, for example, that Lerner *et al.*'s Seminal tool (Lerner et al., 2007), which attempts to explain type errors by searching for repairs, could be restructured to use Burke-Fisher-style local search—becoming just one more instantiation of the functor.

The real deal The `BurkeFisher` functor came out of work done building new lexing and parsing infrastructure for SML/NJ; it is in use in production code, and available with recent versions of the compiler.[3] The implementation includes the full suite of Burke-Fisher repairs, allows for multiple repairs, and judges goodness of repairs using a sophisticated metric.

Historical note

As an undergraduate at the University of Chicago, Turon had a gig under John Reppy building infrastructure for SML/NJ—in particular, new lexing and parsing tools that were meant to be robust, but somewhat out of the mainstream. The parser, ml-antlr, was built in the style of the famous Antlr parser (Parr and Quong, 1995) for Java, and generated readable, recursive-descent code. Once it was up and running, Reppy tasked Turon with improving its terrible error messages. Fatefully, he handed Turon a copy of Burke and Fisher's paper, which led to a question: but what if you don't have an explicit parser stack to keep a duplicate of? Turon had by that time heard of `call/cc`—he was Reppy's student, after all—and the rest was inevitable.

3. Once more, with free monads

The *free monad*, `(:*) f`, of a functor, `f`, presented in Haskell thus

```
data f :* x = Ret x | Do (f (f :* x))

instance Functor f => Monad ((:*) f) where
  return = Ret
  Ret x   >>= g   = g x
  Do ffx  >>= g   = Do (fmap (>>= g) ffx)
```

is a standard construction characterizing what it is to *be* a computation whose notion of 'effect' is given by `f`, without saying what it is to *run* such a computation. We gain modularity just by separating the business of constructing a program to an interface, given by `f`, from the business of interpreting the `f`-data in that program as some sort of command.

The underlying data in `f :* x` are simply trees whose leaves represent 'returning an `x`' and whose internal nodes represent 'doing one `f`-operation then carrying on'. That is, each node is an `f`-shaped structure with subtrees for children, explaining how to proceed in each circumstance possible arising from the action.

[3] See `http://smlnj.org/`

The 'return' of the monad just delivers a value in a leaf, with no preceding `f`-operations. The 'bind' copies the `f`-actions of an existing tree, but grafts a further tree in place of each leaf, computed from the value found at that leaf, thus delivering a continuation of the computation with more `f`-actions.

A `Functor` FROM AN `interface`

We can imagine specifying an `interface` of commands for interacting with some sort of external system. For example, a `Teletype` might offer us the ability to get and put characters. We might imagine expressing these abilities as commands, `getC` returning a `Char`, and `putC` parametrized by a `Char`, returning a trivial value of unit type. Such a declaration might conceivably resemble the following:

```
interface Teletype where
  getC       :: Char
  putC Char  :: ()
```

Imagination notwithstanding, we can systematically unpack such a declaration to construct a Haskell `Functor`, `Teletype`, whose free monad admits `getC` and `putC` operations. The `Teletype` functor explains how to obtain a value after *exactly one* `Teletype`-operation: a `Teletype x` value comprises a choice of an operation from the interface and its parameters, together with a function to `x` from its result. The associated `fmap` acts by postcomposition on the latter.

```
getC    ::          Teletype  :*   Char
putC    :: Char ->  Teletype  :*   ()

data Teletype x  =  GetC        (  Char  -> x  )
                 |  PutC  Char  (  ()    -> x  )

instance Functor Teletype where
  fmap g         (  GetC        k  )  = GetC    (g . k)
  fmap g         (  PutC  c     k  )  = PutC c  (g . k)

getC    = do1    (  GetC        )
putC c  = do1    (  PutC  c     )
```

Figure 7. Haskell code for `interface Teletype`

Wouter Swierstra gives a general account of constructing free monads from an interface, including machinery for combining interfaces

and checking interface inclusion using type class machinery (Swierstra, 2008), but our purposes require only the basic idea.

The `getC` and `putC` operations are then given by one-node trees in the `Teletype` free monad, `Doing` one command, then `Returning`, after the following pattern:

```
do1 :: (forall x. (t -> x) -> f x) -> f :* t
do1 h = Do (h Ret)
```

Monadic computations performing `getC` and `putC` operations to obtain a value of type `x` thus correspond to command-response trees in `Teletype :* x`. The free monad captures what it is to be a process which communicates via the `Teletype` interface without giving any particular semantics to its commands. We can supply such a semantics by interpreting those command-response trees in terms of other operations, for example, the actual `getChar` and `putChar` of Haskell's `IO` monad.

```
ioTeletype :: Teletype :* x -> IO x
ioTeletype (Ret x)           = return x
ioTeletype (Do (GetC    k))  = do
  c <- getChar
  ioTeletype (k c)
ioTeletype (Do (PutC c  k))  = do
  putChar c
  ioTeletype (k ())
```

An `interface` for reading input

In much the same way, we can characterize operations that support reading text with a one character lookahead. The `ReadLine interface` allows a process to peek at the next character, to read it and move on once its meaning is clear, or to abort once its meaninglessness is inevitable.

```
interface ReadLine where
  peekC :: Char
  readC :: ()
  abort :: a
```

The polymorphic `abort` operation can be used at any type, requiring an 'existential type' in the construction of the corresponding `ReadLine Functor`, which otherwise works just as before:

```
peekC :: ReadLine :* Char
readC :: ReadLine :* ()
abort :: ReadLine :* a

data ReadLine x  =             PeekC  (  Char  -> x  )
                 |             ReadC  (  ()    -> x  )
                 |  forall a.  Abort  (  a     -> x  )

instance Functor ReadLine where
  fmap g (PeekC  k)  = PeekC  (g . k)
  fmap g (ReadC  k)  = ReadC  (g . k)
  fmap g (Abort  k)  = Abort  (g . k)

peekC  = do1 ( PeekC )
readC  = do1 ( ReadC )
abort  = do1 ( Abort )
```

Just as in our earlier Scheme version, we can implement a parser for *s-expressions*, working to the `ReadLine` interface without the need for a particular implementation of its operations in mind; see Figure 8.

Handling `ReadLine` in batch mode

We are now free to implement `ReadLine` in a variety of ways. To parse s-expressions from a `String`, we can interpret `peekC` and `readC` directly for non-empty input. If we run out of input too early or the parser `aborts`, we must indicate failure, hence we target a `Maybe` type.

```
readLineString ::  ReadLine :* x -> String -> Maybe x
readLineString (Ret x)         _          = Just x
readLineString (Do (PeekC k))  s@(c : _)  = readLineString (k c)   s
readLineString (Do (ReadC k))  (_ : s)    = readLineString (k ())  s
readLineString _               _          = Nothing
```

Handling `ReadLine` on a `Teletype`

However, we should also like to deliver the same sort of rollback for online parsing on a teletype with backspace as we achieved with continuations above. Accordingly, let us interpret the `ReadLine` interface in terms of the `Teletype` interface.

```
teletypeReadLine :: ReadLine :* x -> Teletype :* x
teletypeReadLine p = logRun p (Began p, Nothing)
```

```
data SExp = A String | SExp :.: SExp deriving (Show, Eq)

sexp :: ReadLine :* SExp
sexp = peekC >>= \ c -> case c of
    '.'  ->  abort
    ')'  ->  abort
    '('  ->  readC >> open
    ' '  ->  readC >> sexp
    _    ->  return A <*> atom

open :: ReadLine :* SExp
open = peekC >>= \ c -> case c of
    '.'  ->  abort
    ')'  ->  readC >> return (A "")
    ' '  ->  readC >> open
    _    ->  return (:.:) <*> sexp <*> cdr

cdr :: ReadLine :* SExp
cdr = peekC >>= \ c -> case c of
    ' '  ->  readC >> cdr
    '.'  ->  readC >> return const <*> sexp <*> close
    _    ->  open

close :: ReadLine :* ()
close = peekC >>= \ c -> case c of
    ' '  ->  readC >> close
    ')'  ->  readC >> return ()
    _    ->  abort

atom :: ReadLine :* String
atom = peekC >>= \ c -> if elem c " ()."
    then     return ""
    else     readC >> return (c :) <*> atom
```

Figure 8. An *s-expression* parser, written to the `ReadLine` interface.

In order to facilitate rollback as we interpret process `p`, we maintain a `Log` initialized with a copy of `p`. Command-response trees naturally give rise to a generic notion of log—the *zipper* (Huet, 1997)—recording the current position within the tree in a way which permits not only progress but also retreat to explore alternative paths.

```
logRun ::  ReadLine :* x ->
           (Log x, Maybe Char) -> Teletype :* x
logRun (Ret x)           _               = Ret x
logRun (Do (PeekC    k))  (log, Just c)   = logRun (k c) (log, Just c)
logRun (Do (PeekC    k))  (log, Nothing)  = getC >>= \ c -> case c of
  '\b'  -> unroll log
  _     -> logRun (k c) (PeekC1 log k, Just c)
logRun (Do (ReadC  k))  (log, Just c)   = do
  putC c
  logRun (k ()) (ReadCed log, Nothing)
logRun (Do (Abort   _))  (log, _)        = unroll log
```

Figure 9. Interpreting a `ReadLine` process

For our purposes, we shall need only a fragment of the zipper, logging just the *first* `peekC` and the `readC` of each character. That is, we record only operations whose undoing requires noticeable action.

```
data Log x  =  PeekC1 (Log x) (Char -> ReadLine :* x)
            |  ReadCed (Log x)
            |  Began (ReadLine :* x)
```

To interpret a `ReadLine` process, we maintain this `Log`, along with a buffer to hold a character which has been `peekC`ed at but not yet `readC`ed. The `peekC` command receives the character in the buffer if there is one, with no change to the log. However, if the buffer is empty, we must fetch a fresh character from the teletype. If the character is 'ordinary', we feed it to the `peekC`'s continuation, but we also *log* that continuation in case the user backspaces over it and types something else. It would not make sense to invoke the continuation for a *buffered* `peekC` at a different character: we must ensure that backspace `unroll`s all the way to the `peekC` which brought the last character into the buffer. The `abort` signal, indicating that the last character typed has made a successful read impossible, also triggers `unroll`ing. The `readC` operation empties the buffer, echoing the character it contained. In effect, we log the state of the `ReadLine` process each time the underlying `Teletype` process performs an operation that we might need to undo.

In case of a backspace keystroke or an `abort`, we `unroll` the log just until we find an opportunity to feed in an alternative character. As soon as we find a 'first peek', we restart with the same continuation following a fresh peek. A physical backspace happens whenever we unroll an `readC`, rubbing out the character it had echoed by overwriting with a space. A backspace at the very start will `unroll` the initial log: we respond by restarting the process.

```
unroll :: Log x -> Teletype :* x
unroll (PeekC1 log k)   =  logRun (peekC >>= k) (log, Nothing)
unroll (ReadCed log)  =  do
  putC '\b' >> putC ' ' >> putC '\b'
  unroll log
unroll (Began p)       =  teletypeReadLine p
```

Discussion

Once again, the business of managing rollback is entirely separated from the implementation of the parser. This time, the notion of a 'rubout port' is delivered by a kind of 'device driver' sitting between the `ReadLine` and `Teletype` interfaces, logging the progress of the `ReadLine` process at the points when significant `Teletype` operations occur.

As we can handle `ReadLine` in terms of `Teletype` and `Teletype` in terms of `IO`, it is straightforward to wire our parser up to the 'real world'. The operation

```
(ioTeletype . teletypeReadLine) sexp :: IO SExp
```

reads a self-delimited s-expression from the console, with backspace acting as it should.

Free monads capture the essence of programming to an interface of operations, but the naïve definition is not especially performant, as tree-grafting is notoriously inefficient (much like appending) when left-nested. One remedy is to restore a continuation-based treatment systematically by means of the *codensity* transformation (Voigtländer, 2008). In some sense, the use of a free monad for the `ReadLine interface` makes explicit and gives licence to just exactly the continuation chicanery required by a 'rubout client' implemented in direct style.

Indeed, in a calculus of algebraic effects and handlers (Plotkin and Pretnar, 2009), as used in Bauer and Pretnar's Eff language (Bauer and Pretnar, 2012), one might hope to program in direct style to an explicit interface of operations, with a denotational semantics in the corresponding free monad. The explicit punctuation of monadic style programming in Haskell imposes a notational cost for the privilege of policing effect permissions, perhaps higher than one might desire or require.

Historical note

Shivers presented the account of modular rollback via `call/cc` at the 2011 meeting of IFIP Working Group 2.8 in Texas: McBride, in the audience, was fascinated. A little later, it transpired that the pair were

marooned in Austin airport, waiting for the same delayed flight to Boston. McBride was working on *Kleisli arrows of outrageous fortune*, studying the notion of a 'free monad' for value-indexed types. It seemed natural to wonder if there was also a free monad account of modular rollback, and as the plane finally made it off the runway, McBride and Shivers fired up `ghc` and developed the code in this section, which Turon then presented at the 2011 Continuation Workshop.

4. Epilogue

We have now seen one idea presented in the idiom of three different languages: Scheme, Standard ML, and Haskell. A natural question remains, namely, what is the relationship between these presentations?

As it happens, a deeper version of that question has already been answered. Andrzej Filinski explained how the "direct style" of effectful programming in a language like ML is related to the "monadic style" of effectful programming in Haskell *in general* (Filinski, 1994). The central result is a representation theorem saying that delimited control is a kind of "universal effect": in a language with delimited control, "*any* expressible monadic structure can be added as a purely definitional extension, without requiring a reinterpretation of the whole language." The key to this remarkable result is using delimited control to *dynamically* turn a direct-style program into monadic style. The definition of each effectful operator includes a use of `shift` to capture the rest of the not-yet-monadic computation, which can then be used as the continuation argument `k` to an explicit monadic bind:

```
(* monadic reflection in ML *)
performEffect e = shift (fn k => bind e k)
```

Thus, to perform an effect, you simply insert a bind between you and your continuation. The continuation will be transformed into monadic style by need, if and when the bind makes use of it.

The technique of sneaking a checkpointing lexer into an unsuspecting parser is actually an instance of sneaking a monad into some direct-style code—that is, an application of Filinski's ideas. Our Haskell presentation makes clear what this monad really is, first by identifying its interface (as a free monad) and then by (re)interpreting programs written against that interface using checkpointing. Our Scheme and SML presentations, by contrast, fused the process of transformation to monadic style with the process of interpreting the monad in a particular way, *i.e.*, with checkpoints.

What is the lesson here? Monadic representation is not just about making monadic programming more pleasant. It can also be used to boost modularity, writing a module in a pure-seeming way, and then *externally* introducing effects that systematically reinterpret its code in a particular monad. Here we have used the technique to introduce user interaction and error repair to an already-written parser, with a clean separation of concerns. But surely other opportunities for monadic reinterpretation are lurking.

The ICFP/CW 2011 presentations

Parts of this work were presented at the ICFP 2011 conference and the co-located Continuation Workshop. Here we briefly summarize the angle of those presentations; the former can be viewed at

`http://dl.acm.org/citation.cfm?id=2034783`

The ICFP presentation focused on the `BurkeFisher` functor, which has a compelling, easy to grasp motivation. The theme of the talk was parser infiltration, with Control (*Delimited* Control) acting as our agent 007. Somewhat surprisingly, we were able to walk through essentially the entire body of code by making a few simplifications: we did not handle backtracking, token windows, or give any of the parser's code. Surely this is an indication of the power `call/cc` to radically—but concisely—alter the control structure of a program.

The CW presentation, in contrast, gave the free-monadic perspective (and, transitively, the rubout-handler example). That talk was more tutorial in nature, and it began with an explanation of free monads as well as Filinski's monadic representation. We demonstrated how a judicious use of `shift` and `reset` wrapping an input stream could transform a direct-style parser into one producing a computation in a free monad, which we could then reinterpret with checkpointing. Slides for the CW talk are available at

`http://www.ccs.neu.edu/home/turon/cw2011-slides.pdf`

Acknowledgements

This article reports on work supported by the Defense Advanced Research Projects Agency under Air Force Research Laboratory (AFRL/Rome) Contract No. FA8650-10-C-7090 and Cooperative Agreement No. FA8750-10-2-0233. The views expressed are those of the authors and do not

reflect the official policy or position of the Department of Defense or the U.S. Government.

Turon is currently supported by a Microsoft fellowship, and was supported by NSF award CNS-0454136 while writing the ML-Antlr parser at the University of Chicago.

John Reppy provided much guidance. Matthias Felleisen provided encouragement and insightful comments to Shivers while he was working out the rubout-handler system; Mark Feeley provided Shivers an early opportunity to air the design at a Scheme workshop. The quotation on line 31 of Figure 1 is due to a prince in Denmark. Kent Pitman and Dan Weinreb were fonts of wisdom concerning sophisticated details of classic Lisp technology. Thanks go to Vincent St-Amour, Sam Tobin-Hochstadt, and Jesse Tov for feedback on a draft of the paper.

Shivers would like to dedicate this set of twin pearls to another such set: Olin and Avery.

References

Bauer, A. and M. Pretnar: 2012, 'Programming with algebraic effects and handlers'. *The arXiv (CoRR)* **abs/1203.1539**.

Biagioni, E., K. Cline, P. Lee, C. Okasaki, and C. Stone: 1998, 'Safe-for-space threads in Standard ML'. *Higher Order and Symbolic Computation (HOSC)* **11**, 209–225.

Burke, M. G. and G. A. Fisher: 1987, 'A practical method for LR and LL syntactic error diagnosis and recovery'. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **9**(2), 164–197.

Danvy, O. and A. Filinski: 1990, 'Abstracting control'. In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP'90)*. pp. 151–160.

Felleisen, M.: 1988, 'The theory and practice of first-class prompts'. In: *Proceedings of the Fifteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*. pp. 180–190.

Filinski, A.: 1994, 'Representing monads'. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*. pp. 446–457.

Herman, D.: 2007, 'Functional pearl: The great escape, or how to jump the border without getting caught'. In: *Proceedings of the Twelfth ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*.

Huet, G. P.: 1997, 'The zipper'. *Journal of Functional Programming (JFP)* **7**(5), 549–554.

Hutton, G. and E. Meijer: 1998, 'Monadic parsing in Haskell'. *Journal of Functional Programming* **8**(4), 437–444.

Kelsey, R. and J. Rees, 'The Scheme48 system'. `http://s48.org`.

Kuan, G.: 2011, 'A True Higher-order Module System'. Ph.D. thesis, University of Chicago.

Lerner, B. S., M. Flower, D. Grossman, and C. Chambers: 2007, 'Searching for type-error messages'. In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*.

Parr, T. J. and R. W. Quong: 1995, 'ANTLR: A predicated-LL(k) parser generator'. *Software: Practice and Experience* **25**(7), 789–810.

Pitman, K. M.: 1995, 'Ambitious evaluation: a new reading of an old issue'. *Lisp Pointers* **VIII**(2).

Plotkin, G. D. and M. Pretnar: 2009, 'Handlers of algebraic effects'. In: G. Castagna (ed.): *ESOP*, Vol. 5502 of *Lecture Notes in Computer Science*. pp. 80–94, Springer.

Rossberg, A., C. Russo, and D. Dreyer: 2010, 'F-ing modules'. In: *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'10)*.

Shivers, O., B. Carlstrom, M. Gasbichler, and M. Sperber: 2004, 'The scsh manual, release 0.6.6'. `http://scsh.net`.

Shivers, O. and A. Turon: 2011, 'Modular rollback through control logging: a pair of twin functional pearls'. In: *Proceedings of the Sixteenth ACM SIGPLAN International Conference on Functional Programming (ICFP'11)*.

Sitaram, D. and M. Felleisen: 1990, 'Control delimiters and their hierarchies'. *Lisp and Symbolic Computation* **3**, 67–99.

Swierstra, W.: 2008, 'Data types à la carte'. *Journal of Functional Programming (JFP)* **18**(4), 423–436.

Voigtländer, J.: 2008, 'Asymptotic improvement of computations over free monads'. In: P. Audebaud and C. Paulin-Mohring (eds.): *MPC*, Vol. 5133 of *Lecture Notes in Computer Science*. pp. 388–403, Springer.