An Assistant for Reading Nqthm Proof Output

Matt Kaufmann

Technical Report 85 November, 1992

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

ABSTRACT: *Inspection of the output of failed proof attempts is crucial to the successful use of the Boyer-Moore theorem prover. We introduce a utility that assists with the navigation of the prover's output in an Emacs environment. This utility should be a major help to beginning users of the Boyer-Moore prover, and should also be a timesaver for more advanced users.*

Experienced users of the Boyer-Moore prover ‘Nqthm’ [1] have a knack for getting information from the output of a failed proof attempt, by *focusing* on the most useful parts of that (often voluminous) output. Such information can be crucial in formulating appropriate lemmas that can help the proof to succeed. This report documents a utility that should help all Nqthm users get such information. Although the primary beneficiaries of this technology may well be novice users, the utility can also be a timesaver for experienced users.

We make the following assumptions of the user of this facility (and reader of this report).

- The user is at least slightly familiar with Nqthm.
- The user is at least slightly familiar with Emacs [2].
- The user is prepared to run Nqthm inside Emacs.

The utility is based on the notion of a ‘‘checkpoint,’’ roughly as defined in Chapter 9 of [1]. The idea is that when the prover tries certain of its tricks (heuristics), we should consider saying to ourselves, ‘‘Gee, I wonder if it could avoid that trick if only I first prove a suitable rewrite rule.’’ This is an important question to ask, because its tricks often turn out not to be helpful. The way this utility works is to modify slightly the way proofs are printed out so as to mark some checkpoints, and to help the user to peruse them, starting with the ‘‘best’’ one first and then working through the others if that’s desired. The implementation actually includes Common Lisp code for modifying the output by placing certain text above some ‘‘checkpoints,’’ as well as Emacs code that ‘‘knows’’ how to search for that text.

The first section is a very brief summary that probably is sufficient preparation for using the facility. The second section contains detailed documentation. The third section contains a demonstration of how the facility works on a particular example. We conclude with a brief discussion of how to modify some parameters of the system, followed by appendices with the Lisp and Emacs code.

Finally, let us recognize that there is likely to be room for lots of improvement in this facility. While we do expect it to be useful in its present form, it may also be appropriate to view this as a first step toward providing more useful interfaces to Nqthm and its interactive enhancement Pc-Nqthm.

1. Brief summary.

Here is a very brief summary of what is needed in order to start using the utility. If you intend to read the entire document, then you can skip this section.

Of course, the filenames below need to be adjusted if they are not in your current directory.

First start up Nqthm or Pc-Nqthm [3], and then compile as follows.

```
(compile-file "checkpoints.lisp")
```

Then to use the system, either right away or at any time in the future, submit the following form to (Pc-)Nqthm.

```
(load "checkpoints")
```

Next, load the Emacs file "checkpoints.el" (again, with an appropriate pathname if this file is not in the current directory):

```
meta-x load-file checkpoints.el
```

You may also want to load an optional Emacs file that defines the **`control-t`** key bindings, described later. This is done with the command:

```
meta-x load-file checkpoints-keys.el
```

Now try a proof, and follow that with any of the following commands. (If you don't load the optional file shown above, you'll have to give Emacs commands the long way, for example **`meta-x first-checkpoint`** instead of **`control-t 1`**.)

`control-t 1` *[the number 1]***`:  first-checkpoint`**
Go to the first (''best'') checkpoint in the proof

`control-t n :  next-checkpoint`
Go to the ''next-best'' checkpoint in the proof

`control-t p :  previous-checkpoint`
Go to the ''next-worst'' checkpoint in the proof

`control-t g :  goto-checkpoint-level`
Go to the ''best'' checkpoint at the indicated level

`control-t c :  checkpoint-options`
Used interactively to set checkpoint options

At this point you can just experiment with the system. Or you can read on for details.

2. Documentation

2.1 Definitions.

Intuitively a *checkpoint* is a point in an Nqthm proof at which some ''daring'' transition happens, for example one that can turn a provable goal into one that is not provable (such as generalization). That is, a checkpoint is a point in the proof that may well bear some careful inspection by the user. Actually, what we call a ''checkpoint'' below will be the goal printed *just before* such a transition takes place, for example just before a non-simplification step (such as elimination, generalization, or induction) is tried for the first time.

In order to define the notion of a checkpoint, we first need a notion of a *goal segment*. And for that purpose, let us say that a *critical goal* is one that has been pushed for proof by induction. Then a *goal segment* is a sequence of printed goals concluding with a critical goal, or with a failed or aborted proof, and extending backwards toward the beginning of a proof up to (but not including) the preceding critical goal (if any). For example, in the example in the next section, Cases 2 and 2.2 together constitute a goal segment, and Case 2.1 all by itself constitutes a goal segment.

The *level* of a goal is simply the number of critical goals that precede it.

A *checkpoint* is any member of a goal segment satisfying at least one of the conditions below. In each case, a *priority* is assigned, where 0 is the highest priority. Checkpoints are ordered is first by level (as defined above) and then by priority, as follows. (Note that we abuse terminology slightly in that the ''priority 4 checkpoint'' shown below is not really a member of a goal segment.)

- Priority 0: The goal is about to have destructor elimination applied to it, and the only previous steps in this goal segment (if any) have been simplification.
- Priority 1: The goal is about to be generalized.

- Priority 2: The goal is a critical goal (one being pushed for proof by induction).
- Priority 3: The goal is about to have cross-fertilization or elimination of irrelevance applied to it, and the only previous steps in this goal segment (if any) have been either simplification or elimination.
- Priority 4: The prover is about to start proving a goal by induction. (When this happens, a message is printed that describes the induction scheme. For our purposes, we think of that message as being a checkpoint with priority 4.)

In the final section we'll see how to change this assignment of priorities.

Finally, the *checkpointed goal sequence* for a proof is the sequence of all its checkpoints, ordered first by level and then by priority. That is, one checkpoint precedes another in this sequence if and only if either the first has a smaller level than the second, or else they have the same level and the first has a higher priority (i.e. smaller priority number) than the second.

2.2 The proof region and other display issues

The various commands are *not* sensitive to the current cursor position, as long as the cursor is ''inside'' the current proof. That is, the cursor be in the *current proof region*; and for that, it suffices that the cursor be *strictly after* the line with the prompt ">" or "->:" on which the proof was begun, and before or on the next line starting with either of these prompts. See the top of Appendix A for esoteric details. Note that the akcl break prompt, '>>', does *not* play any role in defining the proof region. In fact, in order for '>' to define the final line of a proof region, this character must be preceded by a carriage return and followed by a character other than another '>'. Note however that the end of the buffer always defines the last line of the current proof region.

In particular, if you have attempted a proof and wish to explore the output using this facility, but you have already submitted other Lisp commands since the completion of the proof, then you'll need to move the cursor inside the current proof region before proceeding.

If you are running the theorem prover in a Common Lisp other than kcl or akcl, then you'll probably need to tell Emacs about the Lisp prompt. Simply submit the expression

```
(setq lisp-prompt <string>)
```

to Emacs in that case, where `<string>` is the prompt enclosed in double quotes. In fact <string> can be a regular expression, so for example, to get back the akcl prompt, you could execute the following in emacs.

```
(setq lisp-prompt ">[^>]")
```

After all this emphasis on the Lisp prompt, we should also mention that the Pc-Nqthm prompt '->:' also serves to begin and end proof regions in the same way that the Lisp prompt does. Hence, this facility can be used to inspect prover output arising from the PROVE and REDUCE commands in Pc-Nqthm.

For each of the `control-t` commands displayed above, if the requested checkpointed goal exists then that goal will be the one displayed just below the cursor. If not, the system will beep at you and print an appropriate message in the Emacs mode line at the bottom of the Emacs screen.

2.3 Using the system

We give relative pathnames below, which need to be adjusted if the files are not in the current directory.

In order to use the system, you need to start up Nqthm-1992 or Pc-Nqthm-1992 (i.e. pc-nqthm) and issue the following command:

```
(load "checkpoints.lisp")
```

This will load the compiled file for a Sun 3 or a Sparc, depending on your machine's architecture.

Also, you'll need to load an Emacs file, as follows.[1]

```
meta-x load-file checkpoints.el
```

There is also an optional Emacs file you may want to load. This file defines an Emacs *keymap*[2], hung on the key **`control-t`**, and defines several keys for the keymap. If you already use this keymap[3], this should be harmless except that the keys shown below will be redefined. If not, and you've been using **`control-t`** to transpose characters, you'll want to hit two **`control-t`** characters for that purpose henceforth.

```
meta-x load-file checkpoint-keys.el
```

This file defines certain keyboard stroke sequences as follows. If you don't load it, then you can still give these commands, e.g. **`meta-x first-checkpoint`** instead of **`control-t 1`**.

`control-t 1` *[the number 1]*: **`first-checkpoint`**
Go to the first (''best'') checkpoint in the proof

`control-t n :  next-checkpoint`
Go to the ''next-best'' checkpoint in the proof

`control-t p :  previous-checkpoint`
Go to the ''next-worst'' checkpoint in the proof

`control-t g :  goto-checkpoint-level`
Go to the ''best'' checkpoint at the indicated level

`control-t c :  checkpoint-options`
Used interactively to set checkpoint options

In order to use this utility, type **`control-t 1`** if you've loaded the optional file checkpoints-keys as discussed above; or, type **`meta-x first-checkpoint`**.[4] The cursor will be positioned just above the first (''best'') checkpoint, i.e. the first checkpoint in the checkpointed goal sequence, as defined at the end of the preceding subsection. To advance to the next checkpointed goal that is marked, type **`control-t n`**; to move to the previous checkpointed goal, type **`control-t p`**.

---

[1]Some Emacs users may wish to byte-compile this file, though it's not clear to us that this makes much difference.

[2]A detailed knowledge of Emacs is not necessary here; in particular, it is not necessary to know anything about keymaps. The interested reader is welcome, however, to refer to the Emacs manual, [2]

[3]thanks to Bob Boyer for defining this keymap

[4]That's the number '**`1`**', *not* the letter 'l'.

Currently, our system does not print a note above every checkpoint. Instead, it only prints a note above selected checkpoints. To keep it simple, the checkpoints that are noted *in a given goal segment* are as follows.

- goal pushed for induction;
- the first non-simplification;
- the first step other than simplification or elimination;
- the first generalization.

Remark. Certainly the hope here is that this ‘‘checkpoints’’ assistant will provide a pleasant, helpful interface. However, perhaps the word ‘‘assistant’’ should be emphasized here. Users will probably desire to explore the prover's output from time to time in various ways other than what this facility offers. In order to support such exploration, the various commands displayed above always push the new point as a mark. That is, if you type **`control-t 1`**, say, and then scroll around in the Emacs buffer, you can return to that first checkpoint by typing **`control-u control-@`** (or on many terminals, **`control-u control-<space>`**).

3. An example

Suppose we start up the theorem prover and load the appropriate file. Here is an example of what we might see. (I'll truncate the proof in order to avoid clutter.) Notice that some ‘‘checkpoints’’ have been printed out by the system. I've indicated in italics what happens if you type **`control-t 1`** and then type **`control-t n`** several times. Notice that the first checkpoint below (put on the screen when one types **`control-t 1`**) is *not* the first checkpoint printed by the prover, because of the priorities.

```
thunder:kaufmann[119]% nqthm-1992
AKCL (Austin Kyoto Common Lisp)  Version(1.615) Thu Oct 29 15:17:16 CST 1992
Contains Enhancements by W. Schelter

Nqthm-1992.
Initialized with (BOOT-STRAP NQTHM) on November 9, 1992  08:32:18.
>(load "checkpoints.lisp")
Loading checkpoints.lisp
Finished loading checkpoints.lisp
T

>(prove-lemma times-comm (rewrite)
   (equal (times x y) (times y x)))


     Give the conjecture the name *1.

!!CHECKPOINT LEVEL 1; PRIORITY 4; ID 3  <<< Checkpoint #2, obtained from C-t n>>>


     We will appeal to induction.  Two inductions are suggested by terms in
the conjecture, both of which are flawed.  We limit our consideration to the
two suggested by the largest number of nonprimitive recursive functions in the
conjecture.  Since both of these are equally likely, we will choose
arbitrarily.  We will induct according to the following scheme:
      (AND (IMPLIES (ZEROP X) (p X Y))
           (IMPLIES (AND (NOT (ZEROP X)) (p (SUB1 X) Y))
                    (p X Y))).
Linear arithmetic, the lemma COUNT-NUMBERP, and the definition of ZEROP inform
us that the measure (COUNT X) decreases according to the well-founded relation
LESSP in each induction step of the scheme.  The above induction scheme
produces the following two new conjectures:

Case 2. (IMPLIES (ZEROP X)
                 (EQUAL (TIMES X Y) (TIMES Y X))).

  This simplifies, expanding the functions ZEROP, EQUAL, and TIMES, to the
  following two new conjectures:

  Case 2.2.
          (IMPLIES (EQUAL X 0)
                   (EQUAL 0 (TIMES Y 0))).

    This again simplifies, obviously, to:

!!CHECKPOINT LEVEL 1; PRIORITY 2; ID 7 <<< Checkpoint #1, obtained from C-t 1>>>

          (EQUAL 0 (TIMES Y 0)),

    which we will name *1.1.

!!CHECKPOINT LEVEL 2; PRIORITY 2; ID 9 <<< Checkpoint #3>>>

  Case 2.1.
          (IMPLIES (NOT (NUMBERP X))
                   (EQUAL 0 (TIMES Y X))).

    Name the above subgoal *1.2.

Case 1. (IMPLIES (AND (NOT (ZEROP X))
                      (EQUAL (TIMES (SUB1 X) Y)
                             (TIMES Y (SUB1 X))))
                 (EQUAL (TIMES X Y) (TIMES Y X))).

  This simplifies, opening up ZEROP and TIMES, to the new conjecture:

!!CHECKPOINT LEVEL 3; PRIORITY 0; ID 12 <<< Checkpoint #4>>>

        (IMPLIES (AND (NOT (EQUAL X 0))
                      (NUMBERP X)
                      (EQUAL (TIMES (SUB1 X) Y)
                             (TIMES Y (SUB1 X))))
                 (EQUAL (PLUS Y (TIMES Y (SUB1 X)))
                        (TIMES Y X))).

  Applying the lemma SUB1-ELIM, replace X by (ADD1 Z) to eliminate (SUB1 X).
  We employ the type restriction lemma noted when SUB1 was introduced to
  restrict the new variable.  This produces the new conjecture:

        (IMPLIES (AND (NUMBERP Z)
                      (NOT (EQUAL (ADD1 Z) 0))
                      (EQUAL (TIMES Z Y) (TIMES Y Z)))
                 (EQUAL (PLUS Y (TIMES Y Z))
                        (TIMES Y (ADD1 Z)))),

  which further simplifies, obviously, to:

!!CHECKPOINT LEVEL 3; PRIORITY 3; ID 14 <<< Checkpoint #6>>>

        (IMPLIES (AND (NUMBERP Z)
                      (EQUAL (TIMES Z Y) (TIMES Y Z)))
                 (EQUAL (PLUS Y (TIMES Y Z))
                        (TIMES Y (ADD1 Z)))).

  We now use the above equality hypothesis by substituting (TIMES Z Y) for
  (TIMES Y Z) and throwing away the equality.  This generates:

!!CHECKPOINT LEVEL 3; PRIORITY 2; ID 15 <<< Checkpoint #5>>>

        (IMPLIES (NUMBERP Z)
                 (EQUAL (PLUS Y (TIMES Z Y))
                        (TIMES Y (ADD1 Z)))).
  Name the above subgoal *1.3.

!!CHECKPOINT LEVEL 4; PRIORITY 4; ID 16 <<< Last checkpoint (#8)>>>


     We will appeal to induction.  There are three plausible inductions.  They
merge into two likely candidate inductions.  However, only one is unflawed.
We will induct according to the following scheme:
      (AND (IMPLIES (ZEROP Z) (p Y Z))
           (IMPLIES (AND (NOT (ZEROP Z)) (p Y (SUB1 Z)))
                    (p Y Z))).
Linear arithmetic, the lemma COUNT-NUMBERP, and the definition of ZEROP
establish that the measure (COUNT Z) decreases according to the well-founded
relation LESSP in each induction step of the scheme.  The above induction
scheme leads to the following two new formulas:

Case 2. (IMPLIES (AND (ZEROP Z) (NUMBERP Z))
                 (EQUAL (PLUS Y (TIMES Z Y))
                        (TIMES Y (ADD1 Z)))).

  This simplifies, expanding the functions ZEROP, NUMBERP, EQUAL, TIMES, and
  ADD1, to:

        (IMPLIES (EQUAL Z 0)
                 (EQUAL (PLUS Y 0) (TIMES Y 1))),

  which again simplifies, trivially, to:

!!CHECKPOINT LEVEL 4; PRIORITY 2; ID 20 <<< Checkpoint #7>>>

        (EQUAL (PLUS Y 0) (TIMES Y 1)),

  which we will name *1.3.1.

<<< etc. >>>
```

Sometimes the prover starts over, and attempts to prove the original goal by induction. When that is the case, typing **control-t 1** will cause the system to beep and print a message on the bottom of the screen (the Emacs mode line). The message simply informs the user that this is what has happened, and suggests that **control-t g** can now be used to move to the ‘‘best’’ checkpoint that occurs *after* that ‘‘starting over’’ takes place. This is often a good strategy, since often it is useful to think of the proof attempt that precedes the induction attempt as being a ‘‘mistake.’’ However, this is not always appropriate; if you think that the prover shouldn’t have started over in some sense, then you’ll want to look at the checkpoints that it prints out before it starts over.

4. Modifying the system’s behavior.

All you really have to remember if you want to modify the behavior of the system is to type **control-t c** (assuming that you have loaded the file **"checkpoints-keys.el"** in Emacs -- otherwise, type **meta-x checkpoint-options**). This will cause Emacs to prompt you for some answers, as explained below.

```
Toggle ignoring of proved checkpoints (currently on)?
```

The system is smart enough not to take you to a checkpointed goal that has ultimately been proved. In fact, even if the goal has been pushed to be proved by induction, as long as that proof succeeds then the goal may be considered proved. Really, we have a recursive definition here: a goal is considered *proved* if either it is **T** or else all of its descendents have been proved. (We omit defining the reasonably intuitive notion of *descendents*.) At any rate, the option printed above allows you to change this situation, by having the system ignore the issue of whether a goal has been proved or not. Usually you'll want to answer **`n`** to this question so that the status quo is preserved; an answer of **`y`** turns off the ''ignore'' feature the first time, and then alternates between ''ignoring'' and ''not ignoring'' henceforth.

If you have turned off this ''ignoring'' feature, then until you toggle this feature again, the message displayed above will of course say **`(currently off)`** instead of **`(currently on)`**.

```
Set cursor line number for checkpoints?
```

Normally, when you go to a checkpoint using one of the commands provided, the cursor goes wherever Emacs usually puts cursors at the end of searches. Typically, that's in the middle of the screen, unless no scrolling is necessary. However, you may find it convenient to have the cursor placed on a given line every time. So for example, suppose you answer **`y`** to the above question, and then answer **`7`** to:

```
Enter line number:
```

Then you'll find that henceforth, the cursor is always on line 7 (where line 0 is at the top of the screen).

Although these are typical queries, there are variations. For example, if you previously set the line position and then hit **`control-t c`**, then you'll be given the option of cancelling that setting. Also, when the line position is not set, the system will give you the option of having the cursor line always be the center line in the window; except if that option is already set, then instead you'll have the option of cancelling it. All of this is straightforward and you'll see what is going on when you use **`control-t c`**.

When the queries end, the system will suggest that you submit the form **`(CHECKPOINT-OPTIONS)`** to Nqthm. This form will give you the ability to turn off (or, turn back on) the checkpointing facility. It will also given you the option of having the system print the names of the Nqthm ''process'' that produces each checkpoint, just below the checkpoint marker. Finally, it prints (upon request) information on how to change the priorities assigned to these ''processes.''

Acknowledgements. A number of my colleagues at Computational Logic, Inc. have given me feedback on this facility, and I thank them. A special thanks goes to Rich Cohen, who made useful comments on an earlier version of this report.

Appendix A
The Emacs code

Here is the code for the Emacs file **"checkpoints.el"**.

```
;; Documentation of proof regions:

;; A start is a carriage return followed by the Lisp prompt (see
;; below) or a Pc-Nqthm prompt.  The position associated with a
;; start is the beginning of the line immediately following the
;; start, except that if there is no such line then it's the end of
;; the line of the start.

;; (By default, the Lisp prompt is '>' followed by any character
;; besides '>', so that interrupts are ignored.  However, this can be
;; reset by running the emacs command checkpoint-options.)

;; The previous start is the last start that ends before or on the
;; line of the current cursor, or else the buffer's start if there is no
;; such start.  The next start is the first start that begins after
;; the current cursor (and thus on a later line), or else the buffer's
;; end if there is no such start.  Notice that these notions only
;; depend on the line of the current cursor, not on where the cursor
;; appears on that line.

;; The current proof region extends from the position associated with
;; the previous start (inclusively) to the position associated with the
;; next start (exclusively).  Notice that these regions are
;; non-overlapping.  Also notice that the line containing a prompt
;; belongs to the preceding proof region, not the next one.  This seems
;; to be the simplest way to deal with the regions issue if we want to
;; allow the search to start from the last line in the buffer when there
;; is a prompt on that line.  We could check for that, of course, but
;; then if we type a single carriage return this would change things,
;; which is probably bad.

(defvar proved-checkpoints-enabled t)
(defvar next-start-for-maximum-checkpoint-level)
(defvar next-start-for-proved-checkpoints)

(defvar proved-checkpoints)

(defvar checkpoint-level 0)

(defvar maximum-checkpoint-level)

(defvar checkpoint-priority 0)

;; *** Fix the following if we add more priorities.
(defvar max-priority 4)

(defvar checkpoint-line-position t)

(defvar start-over-level)

;; I used to use
;; \\*\\*\\*\\*\\* Now entering the theorem prover \\*\\*\\*\\*\\*:
;; but maybe that's not a good marker really when searching backwards.
;; Maybe it's OK, but it seems that using prompts and control-l as
;; the only 'separators' is simpler.

(defvar lisp-prompt ">[^>]")

(defvar find-marker-string-after-prompt
  "\\|
->:\\|
\C-l")

(defvar find-marker-string
  (format "\n%s%s" lisp-prompt find-marker-string-after-prompt))
```

```
(defun set-find-marker-string (prompt-string)
  (progn (setq lisp-prompt prompt-string)
         (setq find-marker-string
               (format "\n%s%s" lisp-prompt find-marker-string-after-prompt))))

(defun find-previous-marker ()
  "Puts us just before the start of line with the previous marker."
  (re-search-backward find-marker-string nil t))

(defun find-next-marker ()
  "Puts on just after the next marker, hence on a strictly later line."
  (re-search-forward find-marker-string nil t))

(defun previous-start ()
  "Finds the previous start from the end of the current line."
  (let* ((saved-point (point))
         (success (progn (end-of-line)
                         (find-previous-marker)))
         (ans (progn (if success
                         (progn (forward-line 2)
                                (beginning-of-line)))
                     (point))))
    (goto-char saved-point)
    (if success
        ans
      (point-min))))

(defun next-start ()
  "Finds the next start lying on a strictly later line."
  (let* ((saved-point (point))
         (success (find-next-marker))
         (ans (progn (backward-char)    ;in case we’ve already jumped to the next line
                     (end-of-line)
                     (forward-line 1)
                     (beginning-of-line)
                     (point))))
    (goto-char saved-point)
    (if success
        ans
      (point-max))))

(defun go-to-beginning-of-proof ()
  (goto-char (previous-start)))

(defun cp-member (x y)
  ;; from doctor-member
  "Like memq, but uses  equal  for comparison"
  (while (and y (not (equal x (car y))))
    (setq y (cdr y)))
  y)

(defun proved-checkpoints ()
  "Returns the list of all IDs of proved checkpoints."
  (and proved-checkpoints-enabled
       (let ((next-start (next-start)))
         (if (equal next-start-for-proved-checkpoints next-start)
             ;; then presumably there’s no change in proof output since
             ;; the last call of this function, so we return that same answer
             proved-checkpoints
           (let ((saved-point (point)))
             (setq proved-checkpoints nil)
             (setq next-start-for-proved-checkpoints next-start)
             (go-to-beginning-of-proof)
             (while (search-forward "!!NOTE PROVED CLAUSES " next-start t)
               (setq proved-checkpoints
                     (append (read (current-buffer))
                             proved-checkpoints)))
             (goto-char saved-point)
             proved-checkpoints)))))

(defun next-checkpoint ()
  "Part of the Nqthm checkpoint facility, this puts the cursor just above
the checkpoint of next-lower significance relative to the checkpoint
most recently visited."
  (interactive)
  (if (not (and (boundp ’maximum-checkpoint-level)
                maximum-checkpoint-level))
      (error "You must go to the first checkpoint before asking for the next.")
    (let ((current-checkpoint-level checkpoint-level)
          (current-checkpoint-priority checkpoint-priority))
      (catch 'next-checkpoint
        (while (<= checkpoint-level (maximum-checkpoint-level))
          (increment-checkpoint-level-and-priority)
          (let ((ans (next-checkpoint-at-current-level)))
            (if ans
                (throw 'next-checkpoint ans))))
        (progn (setq checkpoint-level current-checkpoint-level)
               (setq checkpoint-priority current-checkpoint-priority)
               (beep)
               (message "No more checkpoints.")
               nil)))))

(defun next-checkpoint-at-current-level ()
  (catch 'next-checkpoint-at-current-level-exit
    (progn (while (not (find-one-checkpoint))
             (if (= checkpoint-priority max-priority)
                 (throw 'next-checkpoint-at-current-level-exit nil)
               (setq checkpoint-priority (+ 1 checkpoint-priority))))
           t)))

(defun increment-checkpoint-level-and-priority ()
  (if (= checkpoint-priority max-priority)
      (progn (setq checkpoint-priority 0)
             (setq checkpoint-level (+ 1 checkpoint-level)))
    (setq checkpoint-priority (+ 1 checkpoint-priority))))

(defun previous-checkpoint ()
  "Part of the Nqthm checkpoint facility, this puts the cursor just above
the checkpoint of next-higher significance relative to the checkpoint
most recently visited."
  (interactive)
  (if (not (and (boundp 'maximum-checkpoint-level)
                maximum-checkpoint-level))
      (error "You must go to the first checkpoint before asking for the previous.")
    (let ((current-checkpoint-level checkpoint-level)
          (current-checkpoint-priority checkpoint-priority))
      (catch 'previous-checkpoint
        (while (>= checkpoint-level 0)
          (decrement-checkpoint-level-and-priority)
          (let ((ans (previous-checkpoint-at-current-level)))
            (if ans
                (throw 'previous-checkpoint ans))))
        (progn (setq checkpoint-level current-checkpoint-level)
               (setq checkpoint-priority current-checkpoint-priority)
               (beep)
               (message "No preceding checkpoints.")
               nil)))))

(defun previous-checkpoint-at-current-level ()
  (catch 'previous-checkpoint-at-current-level-exit
    (progn (while (not (find-one-checkpoint))
             (if (= checkpoint-priority 0)
                 (throw 'previous-checkpoint-at-current-level-exit nil)
               (setq checkpoint-priority (- checkpoint-priority 1))))
           t)))

(defun decrement-checkpoint-level-and-priority ()
  (if (= checkpoint-priority 0)
      (progn (setq checkpoint-priority max-priority)
             (setq checkpoint-level (- checkpoint-level 1)))
    (setq checkpoint-priority (- checkpoint-priority 1))))

(defun find-one-checkpoint ()
  (interactive)
  (let ((saved-point (point))
        (done nil))
    (go-to-beginning-of-proof)
    (let ((next-start (next-start)))
      (if (progn
            (while (and (not done)
                        (search-forward
                         (format "!!CHECKPOINT LEVEL %s; PRIORITY %s; ID "
                                 checkpoint-level checkpoint-priority)
                         next-start t))
              (setq done (not (memq (read (current-buffer)) (proved-checkpoints)))))
            done)
          (progn (push-mark saved-point)
                 (backward-char)
                 (beginning-of-line)
                 (if (or (eq checkpoint-line-position nil)
                         (numberp checkpoint-line-position))
                     (recenter checkpoint-line-position))
                 t)
        (progn (goto-char saved-point)
               nil)))))

(defun maximum-checkpoint-level ()
  ;; returns nil if there are no checkpoints
  (let ((saved-point1 (point)))
    (forward-line -1)
    (let ((next-start (next-start)))
      (if (and (equal next-start-for-maximum-checkpoint-level next-start)
               maximum-checkpoint-level)
          ;; ... then we have a good guess; let's go with it.
          (progn (goto-char saved-point1)
                 maximum-checkpoint-level)
        (let ((ans nil))
          (setq next-start-for-maximum-checkpoint-level next-start)
          (goto-char next-start)
          ;; we may be a line too late, so let's be sure not simply
          ;; to find the 'start' we're already looking at
          (forward-line -1)
          (beginning-of-line)
          (let ((new-point (point)))
            (let ((marker (progn (find-previous-marker)
                                 (point))))
              (goto-char new-point)
              (if (and marker (search-backward "!!CHECKPOINT LEVEL" marker t))
                  (progn (forward-char 19)
                         (setq ans (read (current-buffer)))))
              (goto-char saved-point1)
              (setq maximum-checkpoint-level ans)))))))

(defun look-for-start-over ()
  (let ((saved-point (point))
        (next-start (next-start)))
    (go-to-beginning-of-proof)
    (setq start-over-level nil)
    (if (search-forward "disregard" next-start t)
        (let ((next-level (if (search-forward "!!CHECKPOINT LEVEL " next-start t)
                              (read (current-buffer))
                            nil)))
          (beep)
          (setq start-over-level next-level)
          (if next-level
              (message
               (format "Note: the prover is inducting on the input, starting at level %s. Try C-t g."
                        next-level))
            (message (format "Note: the prover is inducting on the input."
                             next-level)))))
    (goto-char saved-point)))

(defun first-checkpoint ()
  "Part of the Nqthm checkpoint facility, this starts the search for checkpoints
in a failed (or failing) Nqthm proof.  It puts the cursor just above the most
significant checkpoint."
  (interactive)
  (setq checkpoint-level -1)
  (setq checkpoint-priority max-priority)
  (setq next-start-for-proved-checkpoints nil)
  (setq next-start-for-maximum-checkpoint-level nil)
  (setq maximum-checkpoint-level
        (maximum-checkpoint-level))
  (if maximum-checkpoint-level
      (progn (next-checkpoint)
             (look-for-start-over))
    (error "No checkpoints appear in this proof")))
```

```
(defun checkpoint-options ()
  "Used interactively to set options for the Nqthm checkpointing feature."
  (interactive)
  (if (y-or-n-p
       (format "Toggle ignoring of proved checkpoints (currently %s)? "
               (if proved-checkpoints-enabled 'on 'off)))
      (setq proved-checkpoints-enabled
            (not proved-checkpoints-enabled)))
  (if (y-or-n-p "Change what is considered to be the Lisp prompt? ")
      (set-find-marker-string (read-from-minibuffer "Enter prompt: ")))
  (if (or (eq checkpoint-line-position t)
          (eq checkpoint-line-position nil))
      (if (y-or-n-p "Set cursor line number for checkpoints? ")
          (setq checkpoint-line-position (read-minibuffer "Enter line number: "))
        (if (eq checkpoint-line-position t)
            (if (y-or-n-p "Begin recentering cursor line for checkpoints? ")
                (setq checkpoint-line-position nil))
          (if (y-or-n-p "Stop recentering cursor line for checkpoints? ")
              (setq checkpoint-line-position t))))
    (if (y-or-n-p "Stop setting cursor line number for checkpoints? ")
        (if (y-or-n-p "Begin recentering cursor line for checkpoints? ")
            (setq checkpoint-line-position nil)
          (setq checkpoint-line-position t))))
  (message "Done. For a related utility, submit (CHECKPOINT-OPTIONS) to Nqthm."))

(defun goto-checkpoint-level ()
  "Go to the ''best'' checkpoint at the indicated level."
  (interactive)
  (let ((n (read-from-minibuffer "Go to checkpoint at level: "
                                 (if start-over-level
                                     (format "%s" start-over-level)
                                   nil)
                                 nil t)))
    (let ((saved-priority checkpoint-priority)
          (saved-level checkpoint-level))
      (setq checkpoint-priority 0)
      (setq checkpoint-level n)
      (if (next-checkpoint-at-current-level)
          (progn (setq start-over-level nil)
                 t)
        (progn (beep)
               (message (format "There is no checkpoint at level %s." n))
               (setq checkpoint-priority saved-priority)
               (setq checkpoint-level saved-level))))))
```

Appendix B
The key bindings

Here is the code for the emacs key bindings, i.e. the file `"checkpoint-keys.el"`.

```
(if (not (boundp ’ctl-t-keymap))
    (progn
      (defvar ctl-t-keymap)
      (setq ctl-t-keymap (make-sparse-keymap))
      (message "Redefining control-T; from now on hit it twice to transpose characters.")
      (define-key (current-global-map) "\C-T" ctl-t-keymap)
      (define-key ctl-t-keymap "\C-T" ’transpose-chars)
      (define-key ctl-t-keymap "\C-t" ’transpose-chars)))

(define-key ctl-t-keymap "1" ’first-checkpoint)
(define-key ctl-t-keymap "n" ’next-checkpoint)
(define-key ctl-t-keymap "p" ’previous-checkpoint)
(define-key ctl-t-keymap "c" ’checkpoint-options)
(define-key ctl-t-keymap "g" ’goto-checkpoint-level)
```

Appendix C
The Common Lisp Code

Here is the Common Lisp code, i.e. the file `"checkpoints.lisp"`.

```
;; At this point I need a pretty clear specification of what I'll checkpoint.

;; Note the following bit of code from Nqthm and Nqthm-1992:

;; (DEFUN ADD-PROCESS-HIST (PROCESS PARENT PARENT-HIST DESCENDANTS HIST-ENTRY)
;;   (IO PROCESS PARENT PARENT-HIST DESCENDANTS HIST-ENTRY)
;;   (CONS (CONS PROCESS (CONS PARENT HIST-ENTRY)) PARENT-HIST))

;;
;; To keep it simple, we checkpoint each of the following since the last
;; goal pushed for induction (or the proof's start):
;;
;; goals pushed for induction;
;;
;; the first non-simplification;
;;
;; the first execute-process step other than simplification or elimination;
;;
;; the first generalization.

(defparameter process-print-flag nil)

(defvar total-hist)

;; The following has entries of the form (clause id . dependent-clauses),
;; except that here a ''clause'' can be something like *1.1.
(defvar hist-clause-alist)

;; Here is the list of all printed clause ids.
(defvar all-clause-ids)

(defvar *newly-proved-clause-ids*)

(defvar *saved-random-seed*)

(defun initialize-total-hist ()
  ;;(format (or prove-file t) "~%<<< Beginning checkpointed proof >>>~%")
  (setq hist-clause-alist nil)
  (setq all-clause-ids nil)
  (setq total-hist nil))

(defun remove-proved-clauses-from-hist-clause-alist (proved-clauses a-hist-clause-alist)
  ;; We want to remove all proved clauses, including the given ones
  ;; from a-hist-clause-alist.  This assumes that a-hist-clause-alist respects
  ;; dependencies, in the sense that the dependents of a clause appear "towards
  ;; the front" from that clause.  Note that here a ''clause'' can be something
  ;; like *1.1.

  ;; This has the side effect of setting the global variable *newly-proved-clause-ids*
  ;; to the list of all ids of clauses that have been removed from a-hist-clause-alist.
  (if a-hist-clause-alist
      (let* ((entry (car a-hist-clause-alist))
             (clause (car entry))
             (id (cadr entry))
             (descendants (cddr entry))
             (new-descendants (set-diff-eq descendants proved-clauses)))
        (if new-descendants
            (cons (list* clause id new-descendants)
                  (remove-proved-clauses-from-hist-clause-alist
                   proved-clauses
                   (cdr a-hist-clause-alist)))
          (progn (setq *newly-proved-clause-ids*
                       (cons id *newly-proved-clause-ids*))
                 (remove-proved-clauses-from-hist-clause-alist
                  (cons clause proved-clauses)
                  (cdr a-hist-clause-alist)))))
```

```
    nil))

(DEFUN SETUP (FORM CLAUSES LEMMAS)
  (initialize-total-hist)
  (SETQ ORIGTHM FORM)
  (SETQ EXPAND-LST HINTED-EXPANSIONS)
  (SETQ TERMS-TO-BE-IGNORED-BY-REWRITE NIL)
  (SETQ INDUCTION-HYP-TERMS NIL)
  (SETQ INDUCTION-CONCL-TERMS NIL)
  (SETQ ALL-LEMMAS-USED LEMMAS)
  (SETQ STACK NIL)
  (SETQ FNSTACK NIL)
  (SETQ LAST-PRINT-CLAUSES NIL)
  (SETQ TYPE-ALIST NIL)
  (SETQ LITS-THAT-MAY-BE-ASSUMED-FALSE NIL)
  (SETQ CURRENT-LIT 0)
  (SETQ CURRENT-ATM 0)
  (SETQ ANCESTORS NIL)
  (COND (REWRITE-PATH-STK-PTR
         (SETQ REWRITE-PATH-STK-PTR -1)
         (SETQ REWRITE-PATH-FRAME-CNT 0)
         (SETQ REWRITE-PATH-PERSISTENCE-ALIST NIL)))
  (INIT-LEMMA-STACK)
  (INIT-LINEARIZE-ASSUMPTIONS-STACK)
  (SETQ LAST-PRINEVAL-CHAR (QUOTE |.|))
  (RANDOM-INITIALIZATION ORIGTHM)
  (IO (QUOTE SETUP)
      (LIST ORIGTHM)
      NIL CLAUSES LEMMAS))

(DEFUN DEFN-SETUP (EVENT)
  (initialize-total-hist)
  (SETQ ORIGEVENT EVENT)
  (SETQ LAST-PROCESS (QUOTE SETUP))
  (SETQ EXPAND-LST HINTED-EXPANSIONS)
  (SETQ TERMS-TO-BE-IGNORED-BY-REWRITE NIL)
  (SETQ INDUCTION-HYP-TERMS NIL)
  (SETQ INDUCTION-CONCL-TERMS NIL)
  (SETQ STACK NIL)
  (SETQ FNSTACK NIL)
  (SETQ TYPE-ALIST NIL)
  (SETQ LITS-THAT-MAY-BE-ASSUMED-FALSE NIL)
  (SETQ CURRENT-LIT 0)
  (SETQ CURRENT-ATM 0)
  (SETQ ANCESTORS NIL)
  (COND (REWRITE-PATH-STK-PTR
         (SETQ REWRITE-PATH-STK-PTR -1)
         (SETQ REWRITE-PATH-FRAME-CNT 0)
         (SETQ REWRITE-PATH-PERSISTENCE-ALIST NIL)))
  (INIT-LEMMA-STACK)
  (INIT-LINEARIZE-ASSUMPTIONS-STACK)
  (SETQ LAST-PRINEVAL-CHAR (QUOTE |.|))
  (RANDOM-INITIALIZATION ORIGEVENT)
  EVENT)

(defun new-hist-clause-alist (id)
  ;; assumes that we have a checkpointed goal
  (cond
   ((eq process 'induct)
    (cons (list* (car hist-entry) id descendants)
          hist-clause-alist))
   ((eq process 'store-sent)
    (cons (list parent id (car hist-entry))
          hist-clause-alist))
   (descendants
    (cons (list* parent id descendants)
          hist-clause-alist))
   ((not (eq process 'finished))
    (remove-proved-clauses-from-hist-clause-alist
     (setq *newly-proved-clause-ids* (list parent))
     hist-clause-alist))))

(defun io2 ()
  (let* ((checkpoint-info (make-checkpoint process))
         (id (length total-hist))
         (*newly-proved-clause-ids* nil))
    (setq total-hist
          (cons (list (cons PROCESS checkpoint-info) PARENT PARENT-HIST DESCENDANTS HIST-ENTRY)
                total-hist))
    (setq hist-clause-alist
          (new-hist-clause-alist id))
    (print-checkpoint-info process checkpoint-info)
    (print-removed-clause-ids *newly-proved-clause-ids*)
    (io1)))

(setq io-fn ’io2)

(DEFUN RANDOM-INITIALIZATION (EVENT)
  (setq *saved-random-seed*
        (SETQ *RANDOM-SEED* (CONS-COUNT EVENT))))

;; We make a checkpoint when a goal is pushed for induction,
;; when we do our first non-simplification, and when we do our
;; first non-simplification&non-elimination.  The result is
;; to add a "label" to the process component of a history element.

(defparameter non-simplification-execute-processes
  (QUOTE (;;  SIMPLIFY-CLAUSE SETTLED-DOWN-CLAUSE
          FERTILIZE-CLAUSE
          ELIMINATE-DESTRUCTORS-CLAUSE GENERALIZE-CLAUSE
          ELIMINATE-IRRELEVANCE-CLAUSE STORE-SENT)))

(defparameter non-simplification-elimination-execute-processes
  (QUOTE (;;  SIMPLIFY-CLAUSE SETTLED-DOWN-CLAUSE
          FERTILIZE-CLAUSE
          ;;  ELIMINATE-DESTRUCTORS-CLAUSE
          GENERALIZE-CLAUSE
          ELIMINATE-IRRELEVANCE-CLAUSE STORE-SENT)))

(defparameter io-execute-processes
  (QUOTE (SIMPLIFY-CLAUSE
          ;; SETTLED-DOWN-CLAUSE
          FERTILIZE-CLAUSE
          ELIMINATE-DESTRUCTORS-CLAUSE
          GENERALIZE-CLAUSE
          ELIMINATE-IRRELEVANCE-CLAUSE STORE-SENT)))

(defun processes-not-seen-p (history processes)
  (let (process)
    (if (and history
             (not (eq (setq process (caaar history)) ’store-sent)))
        (and (not (member-eq process processes))
             (processes-not-seen-p (cdr history) processes))
      t)))

(defun some-process-seen-p (history)
  (if history
      (or (member-eq (caaar history) io-execute-processes)
          (some-process-seen-p (cdr history)))
    nil))

(defun current-store-sent-level ()
  ;; could make this more efficient by storing the level, most likely
  (current-store-sent-level-rec total-hist))

(defun current-store-sent-level-rec (hist)
  (if hist
      (if (eq (caaar hist) ’store-sent)
          (1+ (current-store-sent-level-rec (cdr hist)))
        (current-store-sent-level-rec (cdr hist)))
    0))

(defvar checkpoint-priority-alist
  ;; Do not modify this alist, except possibly to shuffle the numbers.
  ;; FERTILIZE-CLAUSE and ELIMINATE-IRRELEVANCE-CLAUSE must both be
  ;; assigned the same value, and other than that all processes must
  ;; be assigned different values, and all between 0 and 4, where 0
  ;; is the most important.  The only place we rely on this specification
  ;; is in the emacs code, where there is a maximum of 4 in the variable
  ;; max-priority, and where only the first priority at a given store-sent-level
  ;; is recognized.
  ’((ELIMINATE-DESTRUCTORS-CLAUSE . 0)
    (FERTILIZE-CLAUSE . 3)
    (ELIMINATE-IRRELEVANCE-CLAUSE . 3)
    (GENERALIZE-CLAUSE . 1)
    (STORE-SENT . 2)
    (INDUCT . 4)))

(defun checkpoint-priority (process)
  (or (cdr (assoc-eq process checkpoint-priority-alist))
      (er hard (process) |Attempted| |to| |assign| |a|
          |priority| |to| |process| (!ppr process (quote |.|)))))

(defun checkpoint ()
  (let ((id (length total-hist)))
    (setq all-clause-ids (cons id all-clause-ids))
    (list (current-store-sent-level)
          (checkpoint-priority process)
          id                            ;unique identifier
          (and process-print-flag process))))

(defun make-checkpoint (process)
  ;; note that settled-down-clause and store-sent
  ;; should not be among current-processes-seen
  (if (and (member-eq process execute-processes)
           (some-process-seen-p total-hist))
      (case process
            (ELIMINATE-DESTRUCTORS-CLAUSE
             ;; make checkpoint if this is the first non-simplification
             (when (processes-not-seen-p total-hist
                                         non-simplification-execute-processes)
                   (checkpoint)))
            ((SIMPLIFY-CLAUSE SETTLED-DOWN-CLAUSE)
             nil)
            ((FERTILIZE-CLAUSE ELIMINATE-IRRELEVANCE-CLAUSE)
             (when (processes-not-seen-p total-hist
                                         non-simplification-elimination-execute-processes)
                   (checkpoint)))
            (GENERALIZE-CLAUSE
             ;; always checkpoint a generalization
             (checkpoint))
            ;; make checkpoint if this is the first non-simplification-elimination
            (otherwise                  ;STORE-SENT
             (checkpoint)))
    (if (eq process 'induct)
        (checkpoint)
      nil)))

(defun process-print-name (process)
  (or (cdr (assoc-eq process
                     '((simplify-clause . simplify)
                       (fertilize-clause . cross-fertilize)
                       (eliminate-destructors-clause . eliminate-destructors)
                       (generalize-clause . generalize)
                       (eliminate-irrelevance-clause . eliminate-irrelevance))))
      process))

(defun print-checkpoint-info (process ci)
  (when ci
        (format (or prove-file t)
                "~%~%!!CHECKPOINT LEVEL ~s; PRIORITY ~s; ID ~s"
                (car ci) (cadr ci)
                (caddr ci)              ;unique identifier
                )
        (when process-print-flag
              (format (or prove-file t)
                      "~&~s"
                      (process-print-name (cadddr ci))))))

(defun print-removed-clause-ids (ids)
  (let ((printed-ids (intersection ids all-clause-ids)))
    (when printed-ids
          (format (or prove-file t) "~&~%!!NOTE PROVED CLAUSES ~s" printed-ids))))

(defmacro pf (form &optional (filename "proof.out"))
  `(with-open-file (prove-file ,filename :direction :output)
                   ,form))
```

```
(defmacro checkpoint-y-or-n-p (msg var yes-ans no-ans &rest args)
  ‘(progn
     ,(list* ’format t msg args)
     (let ((ans (read t)))
       (cond
        ((or (eq ans ’y)
             (eq ans ’yes))
         (setq ,var ,yes-ans))
        ((or (eq ans ’n)
             (eq ans ’no))
         (setq ,var ,no-ans))
        ((eq ans ’a)
         (throw ’checkpoint-options ’abort))
        ((eq ans ’s)
         (format t "~&  No change.~%"))
        (t (format t "~&  Response unknown... no change.~%"))))))

(defun checkpoint-options ()
  (catch ’checkpoint-options
    (format t "~&Please answer each question with either Y (yes), N (no), A (abort queries), or~%~
             S (Skip) or any other character (if you don’t want a change).~&~%~
             For a similar utility in emacs, do control-t c.~%~%")
    (checkpoint-y-or-n-p "Do you want checkpoint-printing on (currently ~a)?  "
                         io-fn ’io2 ’io1
                         (if (eq io-fn ’io2) "off" "on"))
    (checkpoint-y-or-n-p "Do you want process names printed with checkpoints (currently ~a)?  "
                         process-print-flag t nil
                         (if process-print-flag "on" "off"))
    (let (ans)
      (and (checkpoint-y-or-n-p "Would you like information about changing priorities?  "
                                ans t nil)
           (format t "~%To change priorities, (setq checkpoint-priority-alist <new-alist>)~%~
                      where <new-alist> looks like the current value,~%~%   ~s.~%~%~
                      Note however that all priorities should be between 0 and 4 (or you’ll~%~
                      have to modify emacs), and that only the first checkpoint at a given~%~
                      priority will be read by emacs.~%~%"
                   checkpoint-priority-alist)))))
```

References

1. R. S. Boyer and J S. Moore, *A Computational Logic Handbook*, Academic Press, Boston, 1988.

2. Richard M. Stallman, Free Software Foundation, *GNU EMACS Manual*, Sixth ed., 1987.

3. Matt Kaufmann, ‘‘A User’s Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover’’, Technical Report 19, Computational Logic, Inc., May 1988.

Table of Contents