Abbreviated Output for Input in ACL2: An Implementation Case Study

Matt Kaufmann
Dept. of Computer Sciences,
University of Texas at Austin
kaufmann@cs.utexas.edu

ABSTRACT

ACL2 has long provided a way to print expressions in an abbreviated manner, where information about hidden subexpressions is lost. We present a new ACL2 feature that allows the missing subexpressions to be recovered. One purpose of this paper is to motivate and explain the new feature. But the main focus is on the design and implementation of this feature, as a case study to give a sense of the process of improving ACL2, especially to enhance its support for user interaction.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.2.2 [**Design Tools and Techniques**]: User interfaces; D.2.3 [**Coding Tools and Techniques**]: Pretty printers; D.2.5 [**Testing and Debugging**]: Debugging aids

General Terms

Algorithms, Design, Documentation, Human Factors, Verification

Keywords

ACL2, iprinting, pretty-printing, debugging

1. INTRODUCTION

Bob Boyer's pretty-printing algorithm [1], which is used by ACL2, was used by the earliest theorem prover written by Boyer and J Moore [3] and by their other provers [4, 5], up through ACL2 [6]. This algorithm was extended very early in ACL2's development (when Boyer and Moore were solely responsible for ACL2) by allowing for *evisceration*: the replacement of sub-objects by small tokens before printing, in a manner analogous to what is supported by Common Lisp [9]. For example, `(A ((B C) D) E F G)` is printed as `(A (# D) E ...)` if the *print-level* is 2 and the *print-length* is 3. Clearly the latter cannot be read back in, so if for example it is printed as the result of an evaluation, then the elided subexpressions cannot be recovered from the output.

Some recent applications of ACL2 have trafficked in very large structures, in particular with the `HONS`/memoization extension of ACL2 [2]. Users have referred to some such objects as "galactic", and have had to interrupt ACL2 because the objects were too large to print. These applications have made clear the importance of this abbreviation mechanism. It is increasingly desirable to set up an ACL2 environment where evisceration is done routinely and yet eviscerated output may be inspected without loss of information. That ability to inspect eviscerated output is an example of the evolution of ACL2's support for user interaction in increasingly demanding industrial applications.

We present a new ACL2 feature, fleshing out an idea floated by J Moore, that permits eviscerated output to be read back in. The above example might thus be printed as

```
(A (#@1# D) E . #@2#)
```

and ACL2 can read this form back in by looking up indices 1 and 2 in a global structure that provides the values of the elided subexpressions. We call the new feature *iprinting* to suggest "interactive printing" or, thinking of "`#@i#`", to suggest "`i` printing" or "index printing", `i` being an *iprint index* into a global structure containing elided values.[1] We call each such '`#@i#`' an *iprint token*.

The larger goal of this paper is to provide a sense of what goes into adding new interactive features to ACL2. We describe significant design and implementation challenges (and their solutions) in adding iprinting to ACL2. These challenges include choosing and managing relevant data structures, including ACL2 arrays, and maintaining suitable invariants; reclaiming storage; defining when an iprint token is legal input and providing an appropriate error message when it is not; reacting suitably to interrupts and "wormhole" (brr) printing; and avoiding unsoundness caused by the contextual nature of iprinting. We also discuss the design and implementation of utilities for controlling the use of iprinting. Our hope is that as a by-product of this view into the ACL2 development process, the ACL2 community will find some practical ACL2 programming ideas.

Throughout this paper we will quote from the ACL2 documentation, which we may refer to by: "`:DOC` *`name-of-`*

[1]We hope that this explanation, together with the fact that the 'p' in "iprint" is not capitalized, will earn us forgiveness for using the trendy "i" prefix.

topic." In each case, the documentation is taken (with minor reformatting) from a development version of ACL2 preceding the release of Version 3.5. We may also refer to ACL2 documentation topics simply by underlining text. The reader is invited to look at ACL2 Version 3.5 or later [8] for the most recent documentation [7].

The next section describes iprinting at the *user level*. The reader interested only in *using* iprinting can stop there (or indeed, can just go to the ACL2 documentation topic set-iprint and skip this paper). We then discuss the *design* of iprinting, followed by a discussion of some *implementation challenges*. Of course, there is significant overlap between what might be identified as *design* or *implementation*, but the design section focuses on high-level organizational considerations while the implementation section focuses on lower-level issues that needed to be managed. We conclude with possible future work and some reflections.

2. IPRINTING AT THE USER LEVEL

ACL2 printing routines use a so-called evisc-tuple to specify which subexpressions are to be elided. The following explanation of evisc-tuples from `:DOC evisc-tuple` suffices for our purposes. (See `:DOC fmt` for a discussion of ACL2 formatted printing in general, including `fms`.)

> The following example illustrates the use of an evisc-tuple that limits the print-level to 3 — only three descents into list structures are permitted before replacing a subexpression by '`#`' — and limits the print-length to 4 — only the first four elements of any list structure will be printed before replacing its tail by '`...`'.

```
ACL2 !>
 (fms "~x0~%"
      (list (cons #\0 '((a b ((c d))
                              e f g)
                           u v w x y)))
      *standard-co*
      state
      (evisc-tuple 3 4 nil nil))

((A B (#) E ...) U V W ...)
<state>
ACL2 !>
```

> Notice that it is impossible to read the printed value back into ACL2, since there is no way for the ACL2 reader to interpret '`#`' or '`...`'. To solve this problem, see `:DOC set-iprint`.

The new command `set-iprint` is used to enable iprinting, so that eviscerated forms can be read back in. When `set-iprint` is called with value `t` or `nil`, the only state change is that subsequent evisceration uses or does not use iprinting. There is no change in the interpretation of existing iprint tokens.

The following log shows how the tokens '`#`' and '`...`' are replaced by iprint tokens '`#@i#`'. It also shows how those tokens can then be read back in: the ACL2 reader replaces '`#@i#`' by the subexpression that was hidden when '`#@i#`' was printed.

```
ACL2 !>(set-iprint t)

ACL2 Observation in SET-IPRINT:  Iprinting has
been enabled.
ACL2 !>
 (fms "~x0~%"
      (list (cons #\0 '((a b ((c d))
                              e f g)
                           u v w x y 1 2 3 4 5)))
      *standard-co*
      state
      (evisc-tuple 3 4 nil nil))

((A B (#@1#) E . #@2#) U V W . #@3#)
<state>
ACL2 !>'#@1#
(C D)
ACL2 !>'((A B (#@1#) E . #@2#) U V W . #@3#)
((A B ((C D)) E F G)
 U V W X Y 1 2 3 4 5)
ACL2 !>
```

Notice that when ACL2 prints the value of the last (quoted) form above, it does so in full because no global evisceration has been specified; only the above call of `fms` has specified an evisc-tuple. The utility `set-evisc-tuple` has been introduced concurrently with the introduction of iprinting, to provide a single interface for controlling ACL2 printing, both for evaluation results and in other settings such as proof output and errors. The keyword `:ld` in the following input form illustrates setting the ld-evisc-tuple, sometimes called the `:LD` evisc-tuple, which is used for printing results as shown.

```
ACL2 !>(set-evisc-tuple
        (evisc-tuple 3   ; print-level
                     4   ; print-length
                     nil ; alist
                     nil ; hiding-cars
                     )
        :iprint t ; enable iprinting
        :sites :ld)

ACL2 Observation in SET-IPRINT:  Iprinting has
been enabled.
 (:LD)
ACL2 !>'((a b ((c d))
            e f g)
         u v w x y 1 2 3 4 5)
((A B (#@1#) E . #@2#) U V W . #@3#)
ACL2 !>
```

It is tempting to quote this result in order to see the missing subexpressions. But the result is printed with evisceration again![2]

[2]See Section 5 for a discussion about possible reuse of iprint tokens.

```
ACL2 !>'((A B (#@1#) E . #@2#) U V W . #@3#)
((A B (#@4#) E . #@5#) U V W . #@6#)
ACL2 !>
```

Of course, we can explore by quoting the iprint tokens, as follows.

```
ACL2 !>'#@1#
(C D)
ACL2 !>'#@2#
(F G)
ACL2 !>'#@3#
(X Y 1 2 . #@4#)
ACL2 !>
```

Notice however that the last value printed above still contains an iprint token. We could continue to explore, but for large expressions this iterative process could be exhausting.

To circumvent this problem, a utility `without-evisc` has been introduced concurrently with iprinting. It turns off evisceration during both evaluation of the form and printing of the resulting value.

```
ACL2 !>(without-evisc
        '((A B (#@1#) E . #@2#) U V W . #@3#))
((A B ((C D)) E F G)
 U V W X Y 1 2 3 4 5)
ACL2 !>
```

Instead of using `without-evisc` one can invoke `(set-evisc--tuple nil)` to disable evisceration globally, not just for the next form. In that case, a better choice might be to invoke `(set-evisc-tuple :default)`, so that evisceration has its initial behavior, occurring only in limited cases including error messages.

Next we discuss the `:sites` argument of `set-evisc-tuple`, shown as `:ld` in the example above. It can be any of the four keyword values listed below. These correspond to four printing contexts, as explained in `:DOC set-evisc-tuple` (we omit some details here), where by default, the `:ABBREV` context restricts the print-level (to 5) and the print-length (to 7).

- `:TERM` — used for printing terms
- `:ABBREV` — used for printing informational messages for errors, warnings, and queries
- `:LD` — used by the ACL2 read-eval-print loop
- `:TRACE` — used for printing trace output

The `:sites` argument may evaluate to any of these four keywords; to `:ALL`, denoting a list of all four of these keywords; or to a sublist of that list. This argument need not be supplied when `set-evisc-tuple` is called, but then the user will be queried for its value. The argument `:iprint` is sometimes required; see Section 3.3.

Note that `set-evisc-tuple` can be called without supplying keyword arguments, though users may be queried to supply are missing. We explain more about required arguments in Section 3.3.

We conclude our user-level discussion of iprinting by noting that eventually, iprinting will "roll over" so that iprint tokens begin again at '`#@1#`', then '`#@2#`', and so on. This feature is important in order to provide some bounding of memory usage to store the values of iprint tokens. One might expect such rollover to occur at each top-level command; but a call of `ld` or `certify-book`, for example, can generate very large amounts of output. Here we summarize what a user may find useful to know about when rollover occurs, deferring to later sections lower-level design and implementation considerations.

Let us refer to the *last iprint index* as that value of `i` for which '`#@i#`' is the most recently printed iprint token. Then rollover can occur in either of two ways. The more common way is likely to be at the top level of the ACL2 read-eval-print loop, immediately after input is read (and before its evaluation): if the last iprint index exceeds the iprinting *soft bound*, then the next iprint token will be '`#@1#`'. Rollover can occur in a second way: After printing an object (for example, a formula during a proof), if the last iprint index exceeds the iprinting *hard bound*, then the next iprint token will be '`#@1#`'. A precise description of which iprint tokens can be read back in after rollover may be found in Section 3.2. In a nutshell: immediately after rollover occurs, every iprint index up to and including the last iprint index is available and remains so until the next rollover, though values are overwritten as iprint tokens '`#@1#`', '`#@2#`', and so on, are written.

Note that rollover never occurs during a call of `fmt` or any other formatted printing function. In particular, the result printed for evaluation of a top-level form can always be read back in.

We may change the defaults for the iprinting soft bound or hard bound, which are 1,000 and 10,000 (respectively), according to user feedback.

We conclude this section by discussing the introduction of the utility `set-evisc-tuple`. The set of global evisc-tuples has evolved over time, without particularly well-documented and clear interfaces for setting each of them. This work on iprinting motivated us to clean that up, providing `set--evisc-tuple` as a single point of interaction to set the global evisc-tuples. A bonus is that this interface can bring iprinting to the user's attention, by requiring either the `:iprint` argument or response to an associated query.

3. IPRINTING DESIGN CONSIDERATIONS

Our hope is that the ACL2 user can make effective use of iprinting by understanding the preceding section and, perhaps, reading the documentation for topics `evisc-tuple`, `set-evisc-tuple`, `set-iprint`, and `without-evisc`. In this section we discuss some design considerations that support natural, clearly-documented iprinting behavior.

3.1 Rollover

The default values for the iprinting soft and hard bounds are deliberately set high, in order to minimize the chance of reading stale values for iprint tokens. These bounds are important for managing space, because the values associated with iprint indices are not garbage collected. The `set-iprint` utility allows specification of new iprinting soft and hard bounds, for example for those want lower bounds for space-intensive applications,

Why are there both a soft bound and a hard bound? If there were only a soft bound, then a single command could use an unbounded amount of storage; imagine a call of `certify-book` that generates a large amount of proof output. If there were only a hard bound, then one might be tempted to set it where we now set the (smaller) soft bound, so that under normal circumstances only modest storage would be needed to support iprinting. But that could be unfortunate, since by the time the user tries to read in an eviscerated object printed several commands earlier, several rollovers may have occurred, in which case the iprint indices would be stale. Note that this explanation shows why we only check the soft bound at the top level of the ACL2 read-eval-print loop, rather than at forms read by subsidiary calls of `ld`.

3.2 Valid Iprint Indices

When is a positive integer a valid iprint index? And what happens when there is an attempt to read the iprint token '`#@i#`' when `i` is not a valid iprint index?

We focus on the second question first. In a nutshell, the ACL2 reader causes an error when reading an iprint token with an invalid index, or indeed any invalid iprint token starting with '`#@`'. Every character encountered must be a base-10 digit. We insist on base 10 because iprinting is done in base 10, as confusion might otherwise arise if the print-base is changed between the time output is printed and the time at which it is read back in. If a character other than a base-10 digit is encountered after '`#@`' before the terminating '`#`' character, then the remaining input is flushed in order to avoid additional but spurious read errors.

So suppose that in the iprint token '`#@i#`', '`i`' is a sequence of base-10 digits, representing a natural number. If this number is not a legal iprint index, then an error is caused, and as in the case above, remaining input is flushed.

It remains to say when a natural number is a legal iprint index. Clearly 0 is illegal as an iprint index, which must always be at least 1. If there has not yet been a rollover then the largest legal iprint index is the last iprint index (as defined above). Otherwise, the largest legal iprint index is the last iprint index before the most recent rollover.

At one time we considered reading '`#@i#`' as `nil` for `i` denoting a positive integer that is an illegal index. But it seems much more helpful to the user to see an error in that case, rather than perhaps proceeding under the mistaken assumption that nothing was wrong.

No matter how much documentation is available, it may still surprise the user to see an error caused by an out-of-bounds index. This case is addressed directly in `:DOC set-iprint`, and as with many ACL2 error messages, the user is directed to that topic, for example as follows.

```
**********************************************
************ ABORTING from raw Lisp ***********
Error:  Out-of-bounds index in #@5#.
See :DOC set-iprint.
**********************************************
```

Remark. The reader may wonder why the error is caused by raw Lisp. This was convenient given the implementation of the `#@` reader macro; see Section 4.1.

We conclude our discussion of valid iprint indices by considering the following case: a form is read that contains iprint tokens '`#@j#`' and '`#@k#`' such that `k` exceeds the last iprint index but `j` does not. In such a case, `k` comes from before the most recent rollover and `j` comes from after the rollover, so the two iprint tokens couldn't have been stored while printing the same expression. It was thus tempting to cause an error in this case. But we decided against that because the user might want to read a list of forms, some of which were printed before the last rollover while others were printed after the last rollover.

We turn now to the general issue of how to acquaint the user with iprinting in a gentle way.

3.3 Transitioning the User to Iprinting

A basic goal is to encourage the user to take advantage of iprinting whenever evisceration is used, while not surprising the uninitiated user by printing mystifying '`#@i#`' iprint tokens.

ACL2 thus starts up with iprinting disabled, but it is desirable for the system to make the user aware of the possibility of iprinting. It does so when the user attempts to use `set-evisc-tuple`: unless iprinting has previously been enabled, either the keyword argument `:iprint` must be supplied or the user will be queried on whether to turn on iprinting.

It is tempting therefore to initialize all four global evisc-tuples to `nil`, so that the system won't generate any iprint tokens until the user calls set-evisc-tuple, which as described above should help make the user aware of iprinting. However, it is important that the `:ABBREV` global evisc-tuple have modest print-level and print-length, so that large structures do not overwhelm users during informational messages. Therefore, when the user sees `:ABBREV` evisceration with iprinting disabled, a suggestion appears to see `:DOC set-iprint`, as in the following example.

```
ACL2 !>(defun foo '(a b c d e f g h i))


ACL2 Error in ( DEFUN FOO ...):  A definition
must be given three or more arguments, but
```

```
(FOO '(A B C D E F G ...)) has length only 2.
(See :DOC set-iprint to be able to see elided
values in this message.)
```

We have employed user interaction as one way to bring awareness of iprinting, as discussed above in the cases of `set-evisc-tuple` and informational messages (such as error messages). A second way was through providing abundant documentation. To that end we have written new documentation topics, mentioned the new capability in the release notes (note-3-5), and added well-placed hyperlinks that point to `:DOC set-iprint` in the documentation for evisc-tuple, cw-gstack, set-trace-evisc-tuple, set-evisc-tuple, note-3-5, without-evisc, and proof-checker. Although the task of writing ACL2 documentation is a time-consuming activity, it is a critical part of implementation work — in this case especially so, in order to make users aware of a new feature.

3.4 A Soundness Consideration

A trip through the ACL2 documentation [7] reveals that many ACL2 features do not seem connected to automated reasoning in the classical sense. Iprinting is one such feature. However, as with many pieces of ACL2, even iprinting has a logical aspect: it can render ACL2 unsound if not implemented carefully, as we now show.

Consider two ACL2 sessions in which the `:LD` evisc-tuple has been set to specify print-level 2 and print-length 3. In the first session we have

```
ACL2 !>'(a b c d)
(A B C . #@1#)
ACL2 !>
```

while in the second, fresh session we have the following.

```
ACL2 !>'(a b c e)
(A B C . #@1#)
ACL2 !>
```

Thus, iprint index 1 is bound to `(D)` in the first session and to `(E)` in the second session. Now imagine certifying a book in the first session containing the following event.

```
(defthm d-this-time
  (equal '#@1# '(d))
  :rule-classes nil)
```

Finally, imagine including that book in the second session. Then because iprint index 1 is bound to `'(E)` in the second session, we have included the theorem `(equal '(D) '(E))`, which is unsound!

Of course, a reasonable user will probably not deliberately place an iprint token into a book. But it is easy to make cut-and-paste errors, and besides, soundness is not conditioned on the reasonableness of users.

ACL2 therefore disallows the use of iprint tokens during `certify-book`. Moreover, ACL2 writes out the certificate file using source function `print-object$`, which is not sensitive to iprinting or evisc-tuples. Otherwise, unsoundness could arise as above by using iprinting when writing out portcullis commands from the certification world.

In reality, the checksum stored in the certificate would probably save us from the above soundness problem. However, it is not appropriate for soundness to rely on the heuristic guidance of checksums. Moreover, as a practical matter, if an event form in a book has an iprint token, say because of cutting and pasting from ACL2 output, it is best to catch this problem early during certify-book rather than to encounter numerous include-book failures in the future.

As always, it is important to make the error message intelligible. The following example illustrates how this is accomplished.

```
***********************************************
************ ABORTING from raw Lisp ***********
Error:  Illegal reader macro during certify-book, #@1#.
See :DOC set-iprint.
***********************************************
```

We don't restrict the reading of iprint tokens by include-book because it's not necessary. We already disallow such a book from being certified, and for all we know, a user's preferred methodology may be to include uncertified books while doing proof-hacking, cleaning up only after getting more clarity.

3.5 Modifying Existing Source Code

With iprinting, it is no longer necessary to make exceptions to print some expressions in full that would normally be abbreviated, say because they are printed as part of an error message. After all, one can use iprinting so that full expressions can be recovered when necessary, and as mentioned above, this is even suggested by any error message that eviscerates without iprinting.

For example, ACL2 code for clause-processor rules had included `nil` as an evisc-tuple for error messages, so that illegal clause-processor rules could be printed in full. Consider for example the following illegal rule, adapted from file `books/clause-processors/basic-examples.lisp` in the ACL2 distribution), but with variable `aaaa` on the last line of the formula where `a` is expected.

```
(defthm correctness-of-strengthen-cl
  (implies
   (and (pseudo-term-listp cl)
        (alistp a)
        (evl (conjoin-clauses
              (clauses-result
               (strengthen-cl cl term state)))
             a))
   (evl (disjoin cl) aaaa))
  :rule-classes :clause-processor)
```

Then the error message for that illegal rule now prints the above formula as follows (with iprinting disabled). Notice that the subterm `(strengthen-cl cl term state)` has been elided in favor of `#`.

```
(IMPLIES (AND (PSEUDO-TERM-LISTP CL)
              (ALISTP A)
              (EVL (CONJOIN-CLAUSES (CLAUSES-RESULT #))
                   A))
         (EVL (DISJOIN CL) AAAA))
```

Before the introduction of iprinting, the formula would have been printed in full. That's harmless in this example, but imagine if the elided term above had involved a very large quoted constant, perhaps because it was generated by a macro rather than directly by the user. Previous versions of ACL2 printed the formula in full because that was the only way to see the full formula if necessary. The iprinting design allowed us to eliminate cluttering of ACL2 source code with evisc tuples while keeping error messages modest in size, since iprinting provides access to the parts that were not printed.

4. IPRINTING IMPLEMENTATION CHALLENGES

Programming exercises typically require dealing with challenges. Here we discuss some of the implementation challenges we faced with iprinting, to shed light on ACL2 system development in particular and on ACL2 programming in general. For more details on the iprinting implementation, see the long comment "Essay on Iprinting" in ACL2 source file `basis.lisp`.

4.1 Modifying the Lisp Reader

A simple way to eviscerate for readability might seem to be to print something like `(@ xi)`, for example `(@ x17)`, where `@` accesses the value of the indicated state global variable (see `:DOC assign`). But this doesn't work if we quote the printed expression. For example, suppose that the object `(A B)` is stored in state global variable `x7`, and that evaluation of the form `(quote ((A B) C))` thus results in printing `((@ x17) C)`. If then we quote that form and thus submit `(quote ((@ x17) C))` for evaluation, the result will be `((@ x17) C)`, not `((A B) C)` as presumably intended. The point here is that the ACL2 loop's reader has to be able to access the stored values even when in the scope of a `quote`.

ACL2 reads in expressions by employing the reader provided by the underlying Common Lisp implementation. Fortunately, Common Lisp [9] allows programs to modify the default behavior of the reader. ACL2 takes advantage of that capability as follows.

Common Lisp defines the character '`#`' to be a *dispatching macro character*. This causes the reader to call a function based on the next character read, which for an iprint token is '`@`'. ACL2 installs the function `sharp-atsign-read` for this purpose, as follows.

```
(set-dispatch-macro-character
   #\#
   #\@
   #'sharp-atsign-read)
```

The raw Lisp function `sharp-atsign-read` then reads from the current input stream and returns an object. In particular, it collects base-10 digits into an iprint index and return the object that corresponds to this index, namely the object stored when the iprint token with that index was last printed.

The discussion above leaves open just how the iprint index is checked to be in range, as specified in Section 3.2. In fact this is done quite efficiently using an ACL2 array, as we describe below.

4.2 Obtaining Efficient Access Using an Array

ACL2 arrays permit constant-time access in an applicative setting. We use a 1-dimensional ACL2 array, `(@ iprint-ar)`, to store the association of iprint indices with values. An association list might well be fine in most cases, but scalability is a fundamental design goal for ACL2, which ideally provides good support for applications that read in large expressions with many iprint tokens. It may have been yet a bit more efficient to use single-threaded objects (stobjs), but that benefit seemed insignificant balanced against the possibility that stobjs might require significantly more programming effort. The main advantage of stobjs would be to avoid a few conses during printing, which seems minor compared to the cost of printing.

A nice benefit of using an array is a constant-time legality check when reading an iprint token, using a combination of the value at index 0 and the `:DEFAULT` field of the header of `iprint-ar`. For the `:DEFAULT` field we store `nil` initially and then, after the first rollover, the last iprint index just before the most recent rollover. At index 0 we store the last iprint index (initially, 0). The following code implements the legality check.

```
(defun iprint-ar-illegal-index (index state)
  (declare (xargs :guard
                  (and (natp index)
                       (state-p state))))
  (or (zp index)
      (let* ((iprint-ar (f-get-global 'iprint-ar
                                      state))
             (bound (default 'iprint-ar iprint-ar)))
        (if (null bound)
            (> index (iprint-last-index* iprint-ar))
          (> index bound)))))
```

Another side benefit of using an array is that it provides constant-time access to whether or not iprinting is enabled, avoiding the introduction of an additional state global variable for that purpose. If the value at index 0 is a number then iprinting is enabled and that value is the last iprint index (initially 0). Otherwise that value is the one-element list containing the last iprint index, and iprinting is disabled.

We conclude with a discussion of the maintenance of fast access for the `iprint-ar`. The basic idea is to compress the array initially (see `:DOC compress1`) and then update the array using `aset1` or `compress1`. But additional attention is necessary.

First, there is some delicate maintenance of an invariant to ensure that the `iprint-ar` never exceeds its `maximum-length`. Details may be found in the ACL2 source code, specifically the "Essay on Iprinting" and in comments in the definition of function `rollover-iprint-ar`.

Second, in order to maintain fast access we need to consider user interrupts (control-C). We have organized the code so that rather than updating the global `iprint-ar` every time a new iprint index is associated with a hidden expression, instead all such pending updates are collected into an alist during printing. At the conclusion of printing it is very fast to do all the updates with `aset1`, or instead at rollover with a call of `compress1`, so that an interrupt is very unlikely to break into the middle of that process. We have considered disabling interrupts during that process, but that seems cumbersome and unnecessary since in the worst case, we have slow array accesses rather than unsoundness. And sanity can be restored in the very unlikely case that fast access fails, because a slow-array-warning will appear and the user can then re-initialize iprinting by calling `set-iprint` with argument `:RESET` or `:RESET-ENABLE`.

Finally we need to make sure that we have fast access after returning from a so-called "wormhole", such as the one that implements the break-rewrite loop. Fortunately ACL2 has a function already responsible for undoing the effects of wormholes, which for example is responsible for compressing the global "enabled-structure" (in support of the current-theory) when returning from a wormhole. It was reasonably straightforward to add a similar call of `compress1` for the global `iprint-ar` in that same function (named `push-wormhole-undo-formi`, in ACL2 source file `axioms.lisp`).

4.3 Without-evisc

The implementation of `without-evisc` (see Section 2) presented a technical challenge. This section discusses that challenge at a necessarily technical level, and may be skipped by those unfamiliar with, and uninterested in, ACL2 implementation methods.

ACL2 implements three of the four evisc-tuples — `:LD`, `:TERM`, and `:ABBREV` — using state global variables. The `:TRACE` evisc-tuple has been handled quite differently, and we chose not to reconsider its design; it is ignored by `without-evisc`. So it may appear that we can easily implement `without-evisc` by using the ACL2 utility `state-global-let*` to bind the above three state globals to `nil`, say:

```
(defmacro without-evisc (form)
  `(state-global-let* ((ld-evisc-tuple nil)
                       (term-evisc-tuple nil)
                       (abbrev-evisc-tuple nil))
                      ,form))
```

A fundamental problem is that `state-global-let*` requires its second argument to evaluate to an *error-triple* of the form `(mv erp val state)`. So for example, if the given form evaluates to an ordinary value, `state-global-let*` cannot be used as above. A rather complicated workaround may be to use the ACL2 evaluator, `trans-eval`, which always returns an error triple specifying both the returned result and that result's output signature (which determines multiple values and stobjs). That leaves the problem of printing that result in the case of stobjs, but ACL2 source function `replace-stobjs` should be useful in solving that problem.

However, there is a second fundamental problem: the result of evaluating a top-level form is printed using the `:LD` evisc-tuple. So the above approach may avoid evisceration caused by printing during evaluation, for example during proof output, but will not avoid evisceration in printing the final result, which can occur if there is a non-trivial `:LD` evisc-tuple.

We chose therefore to implement `without-evisc` using a call of `ld`. The enclosing call of `er-progn`, below, avoids printing the `:EOF` returned by the call of `ld` (see `:DOC ld`).

```
(defmacro without-evisc (form)
  `(without-evisc-fn ',form state))

(defun without-evisc-fn (form state)
  (state-global-let*
   ((abbrev-evisc-tuple
     nil
     set-abbrev-evisc-tuple-state)
    (term-evisc-tuple
     nil
     set-term-evisc-tuple-state))
   (er-progn (ld (list form)
                 :ld-verbose nil
                 :ld-prompt nil
                 :ld-evisc-tuple nil)
             (value :invisible))))
```

Here, `set-abbrev-evisc-tuple-state` and `set-term-evisc-tuple-state` are versions of functions `set-abbrev-evisc-tuple` and `set-term-evisc-tuple` that return state, and those two functions simply invoke `set-evisc-tuple` with `:sites :ABBREV` and `:TERM`.

By using `ld` values `nil` for keyword arguments `:ld-verbose` and `:ld-prompt`, and by returning `(value :invisible)`, noise is avoided and all that is seen is the result of evaluating the given `form` — just as the user presumably intended to see.

4.4 Protecting the ACL2 State

ACL2 provides a notion of *untouchable* functions and variables, which are available only to the implementation: untouchable functions may not be called by the user, and untouchable variables may not be set directly by the user. This mechanism has proved useful in protecting the system from corruption by the user.

The four global evisc-tuples can be modified with the utility `set-evisc-tuple`, described above. This utility checks that the proposed evisc-tuple has an acceptable shape. The implementation uses lower-level functions to install the evisc-tuple into the ACL2 state. Those lower-level functions are declared untouchable so that users cannot subvert the acceptability checks. Similarly, corresponding state global variables are declared untouchable so that they cannot be set directly, but rather, only through appropriate interfaces like `set-evisc-tuple`.

Another kind of state protection is in place to support the `make-event` utility. The ACL2 implementation uses a constant, `*protected-system-state-globals*`, to restore built-in state global variables after `make-event` expansion (which is somewhat similar to macroexpansion). However, this constant excludes state global variables that the user might appropriately want to modify permanently during `make-event` expansion. We have added the global evisc-tuples as well as the `iprint-ar` and corresponding variables holding the soft and hard bounds, so that `make-event` expansion can be used to modify these — using the approved interfaces, of course.

4.5 Dealing with Troublesome Source Code

ACL2 has a complex mechanism for reporting guard violations. Part of that mechanism is a function, `ev-fncall--guard-er-msg`, that creates a suitable error message. That function, which uses a hardwired evisc-tuple with print-level 3 and print-length 4, presented a problem for our goal of using the `:ABBREV` evisc-tuple to print error messages, mainly because of its own subtle tricks with evisc-tuples in order to deal properly with stobjs.

Fortunately, ACL2 already has a utility that permits the user to get full information on guard violations: `print-gv`. So it was easy to decide to avoid the labor-intensive work of modifying the definition of `ev-fncall-guard-er-msg`.

Another source function, `print-ldd-full-or-sketch`, also uses a hardwired evisc-tuple, this time with print-level 2 and print-length 3. This function supports history query utilities such as `:pbt` and `:pcb`. ACL2 users have not complained to the implementers (as best we recall) about the evisceration used by such utilities, so we decided to leave their functionality unchanged.[3]

5. FUTURE WORK

We have identified a few places where additional work might improve the implementation of iprinting.

One possible enhancement would be to arrange to use the same iprint token when the same value is encountered. Consider the following example.

```
  ACL2 !>'(a (((b))) c d e)
  (A (#@7#) C . #@8#)
  ACL2 !>'(A (#@7#) C . #@8#)
  (A (#@9#) C . #@10#)
  ACL2 !>
```

Under the enhancement we imagine, the second result would print the same as the first. The question quickly arises: What do we mean by the *same value*? A quick test would be to use `eq`, but calling it on list values such as those above would represent a guard violation, and might not give the notion of "same" that the user expects. Using a full equality test might be slow, though that problem essentially disappears in the "HONS" experimental extension of ACL2 [2]. If such an extension is considered, it will be important to think carefully about "stale" values. For example, suppose we are about to roll over and we encounter a value that is associated with an iprint index that is on the verge of becoming illegal. Do we really want to re-use that iprint index?

[3]One referee has since made such a complaint; we might reconsider that decision.

Another enhancement pertains to an efficiency hack we use, to avoid creating more than one copy of the same iprint token string in most cases. We build an ACL2 array whose length is the default iprinting hard bound, associating index i with the string "#@*i*#". To save time, we admitted the necessary support functions in `:program` mode, even though we put them in source file `axioms.lisp` where most functions are in `:logic` mode. (We had to put them in a source file processed before the file where they are called in creating the ACL2 array constant, so that they would be compiled, thus eliminating tail recursion and avoiding stack overflow.) A nice exercise is to put these functions in `:logic` mode with guards verified.

```
(defun make-sharp-atsign (i)
  (declare (xargs :guard (natp i)
                  :mode :program))
  (concatenate 'string
               "#@"
               (coerce (explode-nonnegative-integer
                        i 10 nil)
                       'string)
               "#"))

(defun sharp-atsign-alist (i acc)
  (declare (xargs :guard (natp i) :mode :program))
  (cond ((zp i) acc)
        (t (sharp-atsign-alist
            (1- i)
            (acons i (make-sharp-atsign i) acc)))))
```

Finally, note that the proof-checker interactive loop uses the `:TERM` evisc-tuple to do nearly all its printing. This is a deliberate decision, since it presents a simple story and avoids a proliferation of global evisc-tuples. Indeed, there was a fifth global evisc-tuple, `brr-term-evisc-tuple`, that has been eliminated in the name of simplicity. However, it might be better for the proof-checker to have its own evisc-tuple, or even more than one: say, one for terms and one for everything else, such as commands. As is often the case, such a change is likely to occur only if driven by user feedback.

6. CONCLUSION

ACL2 now has an *iprinting* capability, which allows the abbreviation of large objects during printing in a manner that allows the user to read those objects back in. This capability exemplifies the evolution of ACL2 to support user interaction in ever more demanding applications.

This paper illustrates the care exercised when adding a new feature to ACL2:

- We addressed interaction of the new feature with existing capabilities, such as soundness implications from interaction with `certify-book` (Section 3.4), and such as interaction with `make-event` (Section 4.4).
- We considered efficiency and scalability, in particular by using arrays rather than alists (Section 4.2) and by implementing rollover (Section 2).
- We took care that user interaction be natural with the new feature. In the case of rollover, for example, both hard and soft bounds were provided so that rollover would occur appropriately according to the amount of intermediate output (like proof output), not merely result output, and where the soft bound is applied only at the top level of `ld` (Section 3.1). We also considered interrupts (Section 4.2), handling of illegal indices (Section 3.2), and other details that together are intended to minimize user frustration.
- We updated documentation, along with error and warning messages, to advertise and clarify the new feature. As usual, writing documentation and messages took much longer than expected!
- We met several implementation challenges (Section 4), while carefully considering how best to prioritize available time (Section 4.5).

ACL2 is much more than a reasoning engine, to an extent well beyond most or all other mechanized theorem provers. It is a highly interactive system, providing a programming environment with a read-eval-print loop that supports large objects and configurable I/O, and with "administrative" functions including `make-event` and the management of book certificates. The addition of new features to ACL2 thus requires attention to issues as described in the bulleted list above: harmony with existing ACL2 features (especially, but not solely, with respect to soundness); efficiency and scalability; user interaction; documentation, including error and warning messages; and prioritization.

Acknowledgements

We thank J Moore for the idea of abbreviating ACL2 output to be read back in by printing natural numbers as placeholders, and for helpful remarks on drafts of this paper. More generally, we thank J for being a terrific collaborator in ACL2 development. We also thank Bob Boyer and Warren A. Hunt, Jr. for their encouragement in developing iprinting. We are grateful to Sandip Ray for helpful discussions about this paper, whose existence is due in part to his encouragement to write it. We also much appreciate much useful feedback from the referees.

This work was supported by ForrestHunt, Inc. and Warren A. Hunt, Jr.; by DARPA and the National Science Foundation under Grant No. CNS-0429591; and by the National Science Foundation under Grant No. EIA-0303609.

7. REFERENCES

[1] R. S. Boyer. Pretty-print, 1973. Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh, Memo No 64; `http://www.cs.utexas.edu/~boyer/pretty-print.pdf`.

[2] R. S. Boyer and W. A. Hunt, Jr. Function memoization and unique object representation for ACL2 functions. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 81–89, New York, NY, USA, 2006. ACM.

[3] R. S. Boyer and J S. Moore. Proving theorems about pure lisp functions. *JACM*, 22(1):129–144, 1975.

[4] R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[5] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition.* Academic Press, New York, 1997.

[6] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, Boston, MA., 2000.

[7] M. Kaufmann and J S. Moore. ACL2 User's Manual, `http://www.cs.utexas.edu/users/moore/acl2/current/acl2-doc.html#User's-Manual`.

[8] M. Kaufmann and J S. Moore. The ACL2 home page. In *http://www.cs.utexas.edu/users/moore/acl2/*. Dept. of Computer Sciences, University of Texas at Austin, 2009.

[9] G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, 30 North Avenue, Burlington, MA. 01803, 1990.