Reasoning about ACL2 File Input

Jared Davis
Department of Computer Sciences
The University of Texas at Austin
jared@cs.utexas.edu

ABSTRACT

We introduce the logical story behind file input in ACL2 and discuss the types of theorems that can be proven about file-reading operations. We develop a low level library for reasoning about the primitive input routines. We then develop a representation for Unicode text, and implement efficient functions to translate our representation to and from the UTF-8 encoding scheme. We introduce an efficient function to read UTF-8-encoded files, and prove that when files are well formed, the function produces valid Unicode text which corresponds to the contents of the file.

We find exhaustive testing to be a useful technique for proving many theorems in this work. We show how ACL2 can be directed to prove a theorem by exhaustive testing.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs, formal methods*; D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Verification, Performance

Keywords

ACL2, file input, exhaustive testing, Unicode, UTF-8

1. INTRODUCTION

We thought it might be interesting to develop some "typical" software with ACL2. While the lack of networking, threading, and graphics is limiting, ACL2 does include functions to read from files and we could perhaps write some console based programs such as lexers, parsers, compilers, and other utilities. Afterwards, we might be able to use the theorem prover to gain confidence in these programs.

What can we say about a file-reading program? It would be nice to somehow show the basic input routines return exactly the contents of a file as they occur on the disk, but such a correspondence is beyond our scope: we cannot reason about the physical apparatus used to read from some storage device, nor can we reason about the system code responsible for controlling this apparatus. In this paper, we will assume these operations are implemented correctly.

What, then, do we mean when we speak of reasoning about input routines? To describe the file input operations, ACL2 presents us with a logical story — a fiction whereby we can "see" the contents of files on the disk before they are read. Using this fiction, we can reason about functions which perform file input.

We begin by explaining this story (Section 2), and describing a library of basic theorems about the primitive input routines (Section 3). These theorems are particularly mundane, e.g., *reading a byte yields some natural number less than 256*, *reading a byte from an open channel leaves the channel open*, etc., and form the basis for our remaining work.

We then develop a representation for Unicode text, and write a function for reading Unicode text from UTF-8 encoded files (Section 4). We prove a number of well-formedness properties, inspired by the Unicode standard. Ultimately, we prove *reading a valid UTF-8 encoded file produces Unicode text corresponding to the contents of the file.*

During the development of these theorems, we will often find exhaustive testing to be a useful proof method. We can apply exhaustive testing to any problems which can be reduced, in a straightforward way, to checking a finite number of cases. This finite number might be large: in one proof we test 2^{32} cases. The basic idea is to write an efficient testing function which we prove will test every case, then instruct the prover to run this function and observe that it finds no problems.

We have gone to some lengths to make these functions efficient, and we discuss these optimizations and give some rudimentary performance results (Section 5). We conclude with some remarks about writing extended file operations and using the Unicode library (Section 6).

2. THE STORY OF FILE INPUT

Like any other program, ACL2 must interact with the operating system in order to read the contents of a file. This interaction is not visible to the ACL2 user, whom instead sees a fiction of how file input occurs in the ACL2 logic.

At the core of this fiction is *state*, ACL2's logical representation of its host system. Logically speaking, a valid state is an ACL2 object which is recognized by the function `state-`

ACL2 '06 Seattle, Washington USA

p, which returns true only on certain fifteen-tuples which satisfy a litany of requirements. Essentially, state objects are "records" implemented using position-based lists [1].

The ACL2 state includes a representation of the file system. In this paper, we are only concerned with the acts of opening files, reading data from open files, and closing files, so only three of state's fields are relevant: *file-clock*, *readable-files*, and *open-input-channels*.

File-clock. The state's file-clock is an integer which is incremented every time a file is opened or closed, and every time that certain other tasks (e.g., system calls) are performed. This allows us to capture the idea that the state of the file system may change over the course of our program's execution, e.g., our program could create, change, or delete a file.

Readable-files. The state's readable-files field is an association list where:

- The keys are triples of the form `(string type time)`, where string is a file name, time is an integer, and type is one of the three file types: `:character`, `:byte`, or `:object`, and
- The value associated with each key is a list of data. This list represents the contents which *would eventually be read* from the file by our program if we were to open this file at the specified point in time.

This table allows us to capture the relevant state of the file system at any given point in our program's execution. That is, this table putatively contains everything that we could ever read from the file system during our program's execution. Of course, we can never access the contents of this table from within an ACL2 program, because it does not "really exist".

Notice that since the contents each file are represented as a list of elements, we are able to assume that all files are of a finite size! So-called "files" which are actually sockets or devices like `/dev/random` are not accurately modeled by this story, and should not be opened.

Open-input-channels. The state's open-input-channels field is another association list, where:

- The keys are symbols in the `ACL2-INPUT-CHANNEL` package, and
- The values are of the form `((:header type file-name open-time) . elements)`, where file-name is a string, open-time is an integer, type is one of `:character`, `:byte`, or `:object` and elements is a list of elements of the appropriate type.

This table allows us to represent the remaining contents which have not yet been read from each open file. Again, we cannot look at this table at run-time, because it is never actually constructed. We can now appreciate how the basic file operations are understood in the logic:

- `open-input-channel` takes three inputs: a file name to open, the "type" of reading to use, (i.e., `:character`, `:byte`, or `:object`), and the state. It looks up this file in the readable-files table to ensure that it is openable at the current file-clock's time, and if so creates an entry in the open-input-channels table for the file and its contents (which are copied from the readable-files table). Finally, it increments the file-clock.
- `read-byte$` takes as input a channel to read from and the state. It searches for the channel's entry in the open-input-channels table, and if found, modifies the entry by removing the first element, which is returned to the caller. The other reading operations are similar.
- `close-input-channel` takes two inputs: a channel to close and the state. It removes the channel's entry from the open-input-channels table and increments the file-clock.

Keep in mind that this logical story is a fiction which never really exists, and is only interesting from the perspective of theorem proving. Under the hood, reading from files is implemented through system calls as in any other programming language.

3. A BASIC FILE INPUT LIBRARY

ACL2 allows files to be opened in one of three modes, `:character`, `:byte`, or `:object`, which allow us to read an ASCII character, a byte, or a Lisp object from the file at a time. Since our Unicode files are not Lisp or ASCII data, we will largely ignore `:object` and `:character` modes, and instead focus only on `:byte` input.

Unfortunately, ACL2 comes with little support for reasoning about file input operations "out of the box." Starting with the logical definitions of the file operations, we can work our way through proving the mundane but useful theorems we would expect. We bundle these theorems into a low-level input library.

The function `(open-input-channel filename type state)` returns `(mv channel state)`. We prove:

- Whenever `filename` is a string, `type` is one of the valid input modes, and the input `state` is valid, then output `state` is also valid, and
- If the output `channel` is non-`nil` — i.e., if no error has occurred — then `channel` is an open input channel of the appropriate type in the output `state`.

The function `(read-byte$ channel state)` returns as output `(mv byte state)`. We prove:

- Whenever `channel` is an open input channel of the appropriate type in a valid input `state`, then the output `state` is also valid, and
- Furthermore, if the output `elem` is non-`nil` — i.e., if we have not encountered EOF — then `elem` is a character or an integer in the range $[0, 255]$, as appropriate.
- Finally, if the output `elem` is `nil`, then the results of subsequent reads are also `nil`.

Finally, `(close-input-channel channel state)` simply returns `state`. We prove:

- Whenever the input `state` is valid and `channel` is an open input channel of any of the valid types, then the output `state` is also valid.

All of the above will almost certainly be needed for the guard verification of any ACL2 program using these functions. The proofs are completely uninteresting, and are mainly details about state and its components.

Another important fact is that reading a file makes progress, i.e., the number of bytes left to read decreases each time we read a byte. Unfortunately, the file's contents are buried deep inside the state's open-input-channels table. To address this, we add a new function, `(file-measure channel state)`, which retrieves the length of the file's remaining contents. This allows us to prove the appropriate theorems, e.g., reading a byte or character always decreases this measure, unless we are at the end of the file.

3.1 Extended Input Routines

Given our guard verification theorems and the ability to argue that `read-byte$` decreases the `file-measure` of a channel, we can write a function, `(read-file-bytes file-name state)`, which opens a file for input, reads its entire contents into a list of bytes, and closes the file after it is done.

`Read-file-bytes` is perhaps the simplest way to read a file. We use MBE to provide a tail-recursive executable counterpart for efficiency and to avoid stack size limitations, but of course we are still limited to reading files which can fit into memory as a list of bytes. The function is also used as our notion of a file's contents during theorem proving.

We provide additional routines to read from a file in units larger than a byte. In particular, the following functions can accommodate various signedness and byte orders for 1, 2, and 4 byte units. Each have been optimized somewhat via MBE to use fixnum arithmetic, for efficiency.

Function	Bytes	Result Range	Byte Order
`read-byte$`	1	$[0, 2^8 - 1]$	N/A
`read-8s`	1	$[-2^7, 2^7 - 1]$	N/A
`read-16ube`	2	$[0, 2^{16} - 1]$	Big Endian
`read-16ule`	2	$[0, 2^{16} - 1]$	Little Endian
`read-16sbe`	2	$[-2^{15}, 2^{15} - 1]$	Big Endian
`read-16sle`	2	$[-2^{15}, 2^{15} - 1]$	Little Endian
`read-32ube`	4	$[0, 2^{32} - 1]$	Big Endian
`read-32ule`	4	$[0, 2^{32} - 1]$	Little Endian
`read-32sbe`	4	$[-2^{31}, 2^{31} - 1]$	Big Endian
`read-32sle`	4	$[-2^{31}, 2^{31} - 1]$	Little Endian

We prove the obvious properties about these functions, e.g., they leave their input channel open and return an integer in the appropriate range on success. We also provide "block reading" operations based on each of the above functions, e.g., `read-16ube-n` will read up to n unsigned 16-bit quantities from the file using big-endian byte order. These sorts of functions are easy to write, and can be easily chained together.

4. UNICODE REPRESENTATION

Unicode [2] is a popular method for representing text. Where ASCII uses only the natural numbers $0 \leq n \leq$ `127` as characters, Unicode uses two much larger ranges, $0 \leq n \leq$ `#xD7FF` and `#xE000` $\leq n \leq$ `#x10FFFF`, allowing every human language to be simultaneously represented.

Each of these numbers is called a *Unicode scalar value*, and we write a simple predicate, `uchar?`, to recognize them. Note that Unicode scalar values are sometimes also called *Unicode codepoints*.[1] We also provide some additional functions:

- `(ustring? x)` recognizes lists of `uchar?` objects, and is our representation for Unicode text. The Unicode standard would call such lists *Unicode 32-bit Strings* (§3.9-D29d).

- `(ustring x)` creates the corresponding Unicode string from any regular ACL2 `stringp` object. For example, `(ustring "Apples")` produces the list `(65 112 112 108 101 115)`.

- `(ascii x)` creates a `stringp` based on a Unicode string. Note that there are many characters in Unicode which do not exist in ASCII, so this function is not perfect and will insert `?` characters when the Unicode string uses non-representable characters. For example, `(ascii '(1000 65 112 112 108 101 115 1000))` will produce `"?Apples?"`.

There are seven different *Unicode encoding schemes* (§3.9-D39) which can be used for storing Unicode data in files and for sending Unicode data across networks. Of these, *UTF-8* is perhaps the most popular. This is a variable width encoding scheme where each Unicode scalar value is laid out across one to four bytes.

Unicode is a superset of ASCII. In UTF-8 every ASCII character is represented as a single byte, so that every ASCII file is by definition a UTF-8 encoded file. This provides backwards compatibility with existing text documents and good compression for English text.

Other characters are encoded by splitting their bits across a sequence of bytes as governed by Tables 3-5 and 3-6 of the Unicode standard (See figures 1 and 2, below). We provide functions `(utf8-table35-ok? cp x)` and `(utf8-table36-ok? cp x)` which return true only if the codepoint `cp` and byte sequence `x` match with some row in these tables.

4.1 Encoding to UTF-8

Using Table 3-5 as a basis, we write `(uchar=>utf8 x)`, which can be used to convert from `uchar?` objects into UTF-8 byte sequences. One of the basic properties we would like to demonstrate about this function is that given any valid `uchar?` as input, the resulting byte sequence is acceptable under Tables 3-5 and 3-6. More specifically, we would like to prove:

```
(implies (uchar? x)
         (and (utf8-table35-ok? x (uchar=>utf8 x))
              (utf8-table36-ok? x (uchar=>utf8 x))))
```

How can we prove this? The functions `utf8-table35-ok?` and `utf8-table36-ok?` are non-recursive, but involve a number of seemingly arbitrary cases. Furthermore, the function `uchar=>utf8` involves low level bit-manipulation operations such as `ash` and `logior` which are not particularly easy to reason about.

Halfhearted attempts to simply unleash arithmetic books such as `arithmetic-3` and `ihs` were unsuccessful, and led to many failed subgoals which appeared to be nontrivial arithmetic problems. Perhaps, with a more genuine effort, we could develop an arithmetic library which could handle this problem for us, but we have found a more convenient solution which we will use throughout this paper.

[1]The codepoints are a superset of the scalar values, which additionally include the numbers `#xD800` $\leq n \leq$ `#xDFFF`, which are not valid scalar values. (§3.4-D4b, §3.9-D28)

Figure 1: Table 3-5: UTF-8 Bit Distribution (§3.9, D36)

Scalar Value	1st Byte	2nd Byte	3rd Byte	4th Byte
00000000 0xxxxxxx	0xxxxxxx			
00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Figure 2: Table 3-6: Well Formed UTF-8 Byte Sequences (§3.9, D36)

Code Points	1st Byte	2nd Byte	3rd Byte	4th Byte
0000..007F	00..7F			
0080..07FF	C2..DF	80..BF		
0800..0FFF	E0	A0..BF	80..BF	
1000..CFFF	E1..EC	80..BF	80..BF	
D000..D7FF	ED	80..9F	80..BF	
E000..FFFF	EE..EF	80..BF	80..BF	
10000..3FFFF	F0	90..BF	80..BF	80..BF
40000..FFFFF	F1..F3	80..BF	80..BF	80..BF
100000..10FFFF	F4	80..8F	80..BF	80..BF

4.2 Proofs by Exhaustive Testing.

We know that all of the valid `uchar?` objects naturals in $[0, \texttt{\#x10FFFF}]$, so why not just exhaustively test each of these numbers? It is quite easy to write a function to do this, e.g.,

```
(defun test-uchar=>utf8 (i)
  (and (implies
        (uchar? i)
        (and (utf8-table36-ok? i (uchar=>utf8 i))
             (utf8-table35-ok? i (uchar=>utf8 i))))
       (or (zp i)
           (test-uchar=>utf8 (1- i)))))
```

We can convince ourselves that the goal theorem is true by running our test function on any input which covers all `uchar?` objects, e.g., `#x10FFFF`, and observing that `t` is returned. A similar argument can be used to convince ACL2 that our goal theorem is true.

We first prove the following lemma by induction, which shows that if the testing function succeeds on input i, then all naturals $j \leq i$ satisfy the desired property. Importantly, this proof does not need to use the definitions of `uchar=>utf8`, `utf8-table35-ok?`, and `utf8-table36-ok?`.

```
(defthmd lemma
  (implies
   (and (test-uchar=>utf8 i)
        (natp i)
        (natp j)
        (<= j i)
        (uchar? j))
   (and (utf8-table35-ok? j (uchar=>utf8 j))
        (utf8-table36-ok? j (uchar=>utf8 j)))))
```

Now, our goal theorem is submitted with the following hint:

```
(defthm goal
  (implies
   (uchar? x)
   (and (utf8-table35-ok? x (uchar=>utf8 x))
        (utf8-table36-ok? x (uchar=>utf8 x))))
  :hints(("Goal" :use (:instance lemma
                                 (i #x10FFFF)
                                 (j x)))))
```

ACL2's proof proceeds as follows. First, the goal is augmented as indicated by the `:use` hint, producing the new goal:

```
(implies
 (implies
  (and (test-uchar=>utf8 1114111)
       (natp 1114111)
       (natp x)
       (<= x 1114111)
       (uchar? x))
  (and (utf8-table35-ok? x (uchar=>utf8 x))
       (utf8-table36-ok? x (uchar=>utf8 x))))
 (implies (uchar? x)
  (and (utf8-table35-ok? x (uchar=>utf8 x))
       (utf8-table36-ok? x (uchar=>utf8 x)))))
```

Since `(test-uchar=>utf8 1114111)` is a ground term, ACL2 evaluates it to `t` by running our testing function. Evaluation can similarly simplify `(natp 1114111)` to `t`. Furthermore, since we have `(uchar? x)`, ACL2 is able to relieve `(uchar? x)`, `(natp x)` and `(<= x 1114111)`, leaving a trivial tautology. Altogether, the proof takes about 2 seconds, most of which is spent running the testing function.

This proof method is convenient in that it allows us to barbarically force our way through a proof, skipping the time and energy which would have been needed to develop good arithmetic rules for these bit operations. Another feature: if our property is violated, we can easily augment our testing function to print the counterexamples which cause the failure.

4.3 Decoding from UTF-8.

Now that we can convert any `uchar?` into a UTF-8 byte sequence, we will consider the reverse problem. That is, suppose that x is the UTF-8 encoded byte sequence for some Unicode scalar value, and our goal is to recover the value which maps to x.

If we know the length of x, we can easily recover the scalar value associated with it by using the bit manipulation suggested by Table 3-5. We begin by writing functions to perform these combinations. For example, `(utf8-combine2 x1 x2)` produces the Unicode scalar value associated with the two-byte sequence `(x1 x2)`. Similarly, `(utf8-combine3 x1 x2 x3)` handles the three-byte case, and `(utf8-combine4 x1 x2 x3 x4)` handles the four-byte case. (We do not bother with a function for the one-byte case, since the transformation is just the identity.) For each of these functions, we also produce a guard function, e.g., `utf8-combine2-guard`, which checks to ensure that the byte sequence is valid according to Tables 3-5 and 3-6.

We use exhaustive testing to prove that when each guard is met, the combined value matches its input under each table, and that `utf8=>uchar` applied to the result will produce the original input list. In the four-byte case of these proofs, the exhaustive tester must consider 2^{32} possibilities. After fixnum optimizations are added to our combination functions, slightly over 3 minutes of run time are needed to finish this proof on a 2.8 GHz Pentium 4. We are probably near the upper limit of feasibility for this proof method.

We can now create a function to decodes a single UTF-8 character by first examining the length of the character's byte sequence, and then calling the appropriate combination function:

```
(defund utf8-char=>uchar (x)
  (declare (xargs
            :guard (and (unsigned-byte-listp 8 x)
                        (<= 1 (len x))
                        (<= (len x) 4))))
  (and (mbt (true-listp x))
       (case (len x)
         (1 (if (utf8-table35-byte-1/1? (first x))
                (first x)
              nil))
         (2 (let ((x1 (first x))
                  (x2 (second x)))
              (if (utf8-combine2-guard x1 x2)
                  (utf8-combine2 x1 x2)
                nil)))
         (3 (let ((x1 (first x))
                  (x2 (second x))
                  (x3 (third x)))
              (if (utf8-combine3-guard x1 x2 x3)
                  (utf8-combine3 x1 x2 x3)
                nil)))
         (4 (let ((x1 (first x))
                  (x2 (second x))
                  (x3 (third x))
                  (x4 (fourth x)))
              (if (utf8-combine4-guard x1 x2 x3 x4)
                  (utf8-combine4 x1 x2 x3 x4)
                nil)))
         (otherwise nil))))
```

We use exhaustive testing to prove that this function is the inverse of `uchar=>utf8` for all valid `uchar?`s. Furthermore, using the theorems proven about our combining functions, we can conclude that whenever `utf8-char=>uchar` returns a non-nil value, the result is a `uchar?` and satisfies Tables 3-5 and 3-6. Finally, we show that `uchar=>utf8` is also the inverse of `utf8=>uchar` for all acceptable inputs.

4.4 Decoding Byte Sequences.

Since we can encode an arbitrary `uchar?` into UTF-8, it is easy to encode any `ustring?` by successively encoding each character and `append`ing the results. But decoding UTF-8 strings is not so easy: our character decoding operation, `utf8-char=>uchar`, works by first considering the length of the UTF-8 character's byte sequence, but this is not known if we wish to process a string of bytes which, as in files, are laid out one after the next with no structure to tell us where characters begin and end.

We say that a UTF-8 character is a list of one to four bytes which can be successfully converted into a Unicode scalar value. In code:

```
(defun utf8-char? (x)
  (and (unsigned-byte-listp 8 x)
       (<= 1 (len x))
       (<= (len x) 4)
       (utf8-char=>uchar x)))
```

Furthermore, we say that a UTF-8 string is a list of such UTF-8 characters. It is easy to decode a UTF-8 string by successively applying our character decoding function. Finally, we say a list of bytes is valid UTF-8 data if it can be *partitioned* into a UTF-8 string.

The *partitionings* of `x` are those lists which can be flattened into `x` by `append`ing their members. For example, we consider `((a b) (c d e) (f) () (g h))` to be a partitioning of $x =$ `(a b c d e f g h)`. We can describe partitionings as lists of lengths, e.g., the partitioning above can be described as `(2 3 1 0 2)`.

An examination of the *1st Byte* column in Table 3-5 reveals that any byte matches at most one of these bit patterns. Hence, if our goal is to partition a list of bytes, x, into a UTF-8 string, we can use the first byte of x to determine how large the first partition will need to be. Based on this observation, we introduce the `(utf8-partition x)`, which returns `(mv successp sizes)`, where `successp` indicates if x contains valid UTF-8 data, and `sizes` is a list of lengths which will partition x into a UTF-8 string. We prove:

- *Soundness.* If `utf8-partition` is successful, the list of sizes returned will partition x into a valid UTF-8 string.
- *Completeness.* If any partitioning of x would result in a valid UTF-8 string, then `utf8-partition` is successful.
- *Uniqueness.* If any partitioning of x would result in a valid UTF-8 string, then `utf8-partition` returns exactly that partitioning.

Given this partitioning algorithm, we can now (inefficiently) convert a raw list of UTF-8 bytes into Unicode by first partitioning it, then applying our UTF-8 string conversion function. We call this function `utf8=>ustring`, and we can prove that it and `ustring=>utf8` are inverses.

We finally provide a much more efficient algorithm for decoding a list of bytes without first partitioning it and so forth. We make this function tail recursive, optimize it to make use of fixnums, and use MBE so that `utf8=>ustring` uses this efficient function for its execution. We will not show the body of this function, because it is long and quite similar to the function discussed in the next section.

4.5 An Efficient UTF-8 Reader

We can now read and decode a UTF-8 files by first running `read-file-bytes` to read the file into a list of bytes, then using `utf8=>ustring` to efficiently decode this list into Unicode. However, this is somewhat inefficient, because we must `cons` together this intermediate list of bytes. It would be more efficient to decode the file as we read it.

Towards this end, we write a new function that decodes the file as it is read. The core of this routine is heavily optimized and tail recursive, and is presented in Figure 3, which makes up the last 3 pages of this paper. Care must be taken to ensure that both the same data and stream are returned by the `:logic` and `:exec` portions of the MBE, i.e., we must be careful to ensure that both read exactly the same number of bytes.

5. PERFORMANCE

The test system we used is a 2.8 GHz Pentium 4, running Ubuntu Linux 2.6.13.2. We built our books using ACL2 2.9.4 on GCL 2.6.7 configured with 2^{19} maxpages. The hard disk information reported by `dmesg` is: WDC WD400-BB-75JHC0, ATA DISK drive, max request size: 128KiB, 78125000 sectors (40000 MB) w/2048KiB Cache, CHS=-65535/16/63, UDMA(100), cache flushes supported.

5.1 Basic File Operations

To get a baseline reading on the system's performance, we wrote programs in C and C++ which repeatedly read a character from the file and discard it, using the following loops, respectively:

```
for(i = 0;i < size;++i)  // int i, size
    c = fgetc(in);       // char c, FILE* in

for(i = 0;i < size;++i)  // int i, size
    c = in.get();        // char c, ifstream in
```

We then wrote Lisp equivalents of these programs, according to the following template:

```
(defun ,test (n channel state)
  (declare (xargs :guard (and (natp n) ...)))
  (if (mbe :logic (zp n)
           :exec (= (the-fixnum n) 0))
      state
    (mv-let (data state)
            (,function channel state)
            (declare (ignore data))
            (,test (the-fixnum (1- (the-fixnum n)))
                   channel state)))))
```

We instructed each program to read 300 MB from a test file. We ran each test four times, then threw out the worst time and averaged the remaining results in order to minimize noise from caching or other processes. We summarize our performance results below:

Test	Average Time	Throughput
C fgetc	5.73s	52.37 MB/s
C++ ifstream	10.78	27.83
`read-byte`	9.46	31.71
`read-8s`	11.95	25.10
`read-16ube`	14.89	20.15
`read-16ule`	14.84	20.21
`read-16sbe`	16.00	18.75
`read-16sle`	14.90	20.13
`read-32ube`	50.52	5.94
`read-32ule`	50.93	5.89
`read-32sbe`	15.74	19.06
`read-32sle`	15.99	18.76

The abysmal performance of `read-32ube` and `read-32ule` is due to our inability to fully fixnum-optimize these operations. That is, the `long`s which GCL uses to represent fixnums are signed 32-bit numbers on our test machine, and cannot represent the whole range of values which `read-32ube` and `read-32ule` might return.

5.2 UTF-8 Performance

The performance of our UTF-8 reading function is based in part upon the data it is given, so we ran it against both the concatenated text of our Library's source code (written in English) and against a history book (written in Chinese).

English Text. Our test file is about 19.7 million characters long. Starting from a fresh ACL2 session, the test system reads and decodes the file in 5.19 seconds, for a throughput of 3.81 million characters per second, i.e., about 3.8 MB/sec.

Chinese Text. Our test file is the same size in MB, but is only about 7.2 million characters long, since Chinese characters can take multiple bytes. Starting from a fresh ACL2 session, the test system reads and decodes the file in 2.02 seconds, for a throughput of about 3.6 million characters per second, or about 9.8 MB/sec.

Note that, in these tests, we are not simply throwing data away as we read it, but instead we are constructing a `ustring?` corresponding to the contents of the file. As a result, it is harder to get reliable timings due to allocation and garbage collection time.

6. CONCLUSIONS

Normally, ACL2 is used to reason about models of other systems, and not to write console programs. But ACL2 programs can have pretty good performance, and if we write these programs in ACL2 itself then we can directly reason about their behavior with the integrated theorem prover. This development model might be more straightforward than developing a model of some other programming system and reasoning about a deeply embedded interpreter.

Of course, this approach is far from perfect. The theorem prover does not consider some aspects of an ACL2 function's execution, for example stack overflow and memory usage. Also, the story of file input may not perfectly match the semantics of a Unix system with other running programs, e.g., being able to claim that every file is finite.

In any event, this library should be a useful building block for anyone wishing to write such a utility and reason about file operations. In total, the library runs slightly over 9,000 lines of ACL2 (including about 1,200 lines of whitespace and 1,200 lines of comments) with about 140 `defuns` and

550 `defthms`. It is freely available under the terms of the GNU General Public License, and is distributed with ACL2 3.0.

7. REFERENCES

[1] Matt Kaufmann and Rob Sumners. Efficient rewriting of operations on finite structures in ACL2. In *Third International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-2002)*, April 2002.

[2] The Unicode Consortium. *The Unicode Standard, Version 4.0.* Addison-Wesley, 2003.

Figure 3: Main UTF-8 Reading Routine

```
(defund read-utf8-fast (channel state acc)
  (declare (xargs :guard (and (state-p state)
                              (symbolp channel)
                              (open-input-channel-p channel :byte state)
                              (ustring? acc))
                  :measure (file-measure channel state)))
  (mbe
   :logic
   (if (and (state-p state)
            (symbolp channel)
            (open-input-channel-p channel :byte state))
       (mv-let (x1 state)
               (read-byte$ channel state)
               (if (not x1)
                   (mv (reverse acc) state)
                 (let ((len1 (utf8-table35-expected-length x1)))
                   (if (not len1)
                       (mv 'fail state)
                     (mv-let (x2-x4 state)
                             (take-bytes (1- len1) channel state)
                             (let* ((x1-x4 (cons x1 x2-x4))
                                    (first (utf8-char=>uchar x1-x4)))
                               (if (not first)
                                   (mv 'fail state)
                                 (read-utf8-fast channel state (cons first acc)))))))))
     (mv 'fail state))
   :exec
   (mv-let
    (x1 state)
    (read-byte$ channel state)
    (if (not x1)
        (mv (reverse acc) state)
      (cond

       ((<= (the-fixnum x1) 127)
        ;; Expected length 1.  We don't need to do any further checking; we can
        ;; just recur very quickly.  Note that this will give us very good
        ;; performance for English text, where characters are typically only a
        ;; single byte.
        (read-utf8-fast channel state (cons x1 acc)))

       ((in-range? (the-fixnum x1) 194 223)
        ;; Expected length 2.  (We excluded 192,193 because they are not
        ;; permitted under Table 3-6.)
        (mv-let (x2 state) (read-byte$ channel state)
           (if (and x2 (in-range? (the-fixnum x2) 128 191))
               ;; Manually-inlined utf8-combine2 operation.
               (read-utf8-fast
                channel state
                (cons
                 (the-fixnum
                  (logior
                   (the-fixnum (ash (the-fixnum (logand (the-fixnum x1) 31)) 6))
                   (the-fixnum (logand (the-fixnum x2) 63))))
                 acc))
             (mv 'fail state))))

       ((in-range? (the-fixnum x1) 224 239)
        ;; Expected length 3.  (We cover all options here.)
        (mv-let (x2 state) (read-byte$ channel state)
         (mv-let (x3 state) (read-byte$ channel state)
```

```
    (if (and x2 x3
             (cond ((= (the-fixnum x1) 224)
                    (in-range? (the-fixnum x2) 160 191))
                   ((= (the-fixnum x1) 237)
                    (in-range? (the-fixnum x2) 128 159))
                   (t
                    (in-range? (the-fixnum x2) 128 191)))
             (in-range? (the-fixnum x3) 128 191))
        (read-utf8-fast
         channel state
         (cons
          (the-fixnum
           (logior
            (the-fixnum
             (ash (the-fixnum (logand (the-fixnum x1) 15)) 12))
            (the-fixnum
             (logior
              (the-fixnum
               (ash (the-fixnum (logand (the-fixnum x2) 63)) 6))
              (the-fixnum (logand (the-fixnum x3) 63))))))
          acc))
      (mv 'fail state)))))

((in-range? (the-fixnum x1) 240 244)
 ;; Expected length 4.  (We only accept 240-244 because of Table 3-6;
 ;; i.e., we exclude 245, 246, and 247.)
 (mv-let (x2 state) (read-byte$ channel state)
  (mv-let (x3 state) (read-byte$ channel state)
   (mv-let (x4 state) (read-byte$ channel state)
     (if (and x2 x3 x4
              (cond ((= (the-fixnum x1) 240)
                     (in-range? (the-fixnum x2) 144 191))
                    ((= (the-fixnum x1) 244)
                     (in-range? (the-fixnum x2) 128 143))
                    (t
                     (in-range? (the-fixnum x2) 128 191)))
              (in-range? (the-fixnum x3) 128 191)
              (in-range? (the-fixnum x4) 128 191))
         (read-utf8-fast
          channel state
          (cons
           (the-fixnum
            (logior
             (the-fixnum
              (ash (the-fixnum (logand (the-fixnum x1) 7)) 18))
             (the-fixnum
              (logior
               (the-fixnum
                (ash (the-fixnum (logand (the-fixnum x2) 63)) 12))
               (the-fixnum
                (logior
                 (the-fixnum
                  (ash (the-fixnum (logand (the-fixnum x3) 63)) 6))
                 (the-fixnum
                  (logand (the-fixnum x4) 63)))))))))
          acc))
       (mv 'fail state))))))
```

```
;; This is a little obscure.  As an optimization above, we did not
;; consider cases for first byte = 192, 193, 245, 246, and 247, because
;; these are not allowed under Table 3-6.
;;
;; However, utf8-table35-expected-length predics the lengths of these
;; as 2, 2, 4, 4, and 4, respectively.  So, for our MBE equivalence, we
;; need to make sure to advance the stream just like we do in the
;; :logic mode.
((or (= (the-fixnum x1) 192)
     (= (the-fixnum x1) 193))
 (mv-let (x2 state)
         (read-byte$ channel state)
         (declare (ignore x2))
         (mv 'fail state)))

((or (= (the-fixnum x1) 245)
     (= (the-fixnum x1) 246)
     (= (the-fixnum x1) 247))
 (mv-let (x2 state)
         (read-byte$ channel state)
         (declare (ignore x2))
         (mv-let (x3 state)
                 (read-byte$ channel state)
                 (declare (ignore x3))
                 (mv-let (x4 state)
                         (read-byte$ channel state)
                         (declare (ignore x4))
                         (mv 'fail state)))))

(t
 (mv 'fail state)))))))
```