Ethereum's Recursive Length Prefix in ACL2

Alessandro Coglio
Kestrel Institute
www.kestrel.edu

August 2019

Abstract

Ethereum's Recursive Length Prefix (RLP) is used to encode a wide variety of data, including transactions. The work described in this paper provides a formal specification of RLP encoding and a verified implementation of RLP decoding, all developed in the ACL2 theorem prover. This work has led to improvements to the Ethereum documentation and additions to the Ethereum test suite.

1 Problem and Contribution

Errors in cryptocurrency code may lead to particularly direct financial losses. This applies not only to smart contracts, but also to the underlying execution engines, to wallets, and to other critical components.

In Ethereum [17], Recursive Length Prefix (RLP) [18, Page 'RLP'] [20, Appendix B] is used to encode a wide variety of data, including transactions. It is thus important for this fundamental building block to be specified precisely and implemented correctly.

The work described in this paper contributes to this goal by providing a formal specification of RLP encoding and a verified implementation of RLP decoding. This work has been developed in the ACL2 theorem prover [7]. The development is available [14, Path `books/kestrel/ethereum/rlp`] and is thoroughly documented [13, Topic `rlp`]. Some of the excerpts of the development shown in this paper are slightly simplified for brevity.

This work has led to improvements to the Yellow Paper [20] and to the Ethereum Wiki [18], which are major components of the Ethereum documentation. It has also led to additions to the Ethereum test suite [16], which contains tests for all Ethereum implementations. See Section 5 for details.

This work is part of an ongoing effort to develop, in ACL2, a formal specification and a verified implementation of a complete Ethereum client [9]. This formal specification will also be useful for formally verifying existing client implementations, smart contracts at the level of the EVM (Ethereum Virtual Machine), and the compilation of higher-level programming languages to EVM code.

2 Background

2.1 RLP

RLP is specified informally in the Ethereum wiki ('WK' for short) [18, Page 'RLP'] and more formally in the Ethereum Yellow Paper ('YP' for short) [20, Appendix B].[1]

[1] WK and YP are evolving artifacts. This paper refers to their versions on the date indicated in the references [18] and [20].

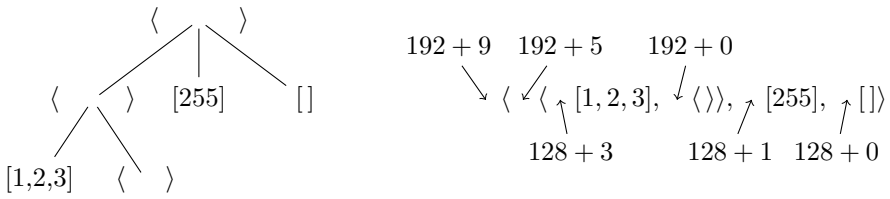

Figure 1: An example RLP tree and its encoding.

RLP encodes arbitrarily nested sequences of bytes into flat sequences of bytes that can be decoded back into the original nested sequences. These nested sequences are finitely branching ordered trees with flat sequences of bytes at their leaf nodes and no additional information at the branching nodes. An example is $\langle\langle[1,2,3],\langle\rangle\rangle,[255],[]\rangle$, where $[\ldots]$ are leaf nodes and $\langle\ldots\rangle$ are branching nodes; this tree is depicted in Figure 1 (left). The leaf node $[]$ with the empty sequence of bytes differs from the branching node $\langle\rangle$ with no subtrees.

WK calls the trees 'items', the branching nodes 'lists', and the leaf nodes 'strings' or 'byte arrays'; it uses Python notations `"..."` for strings (with the implicit assumption that characters consist of 8 bits) and `[...,...]` for lists; it also uses lists `[...,...]` of numbers and characters for flat encodings. YP uses a more explicit tree terminology; it uses a mathematical notation $(\ldots,\ldots)$ for both leaf nodes and branching nodes, as well as for flat encodings.

RLP prescribes to encode non-negative integers by first representing them as base-256 big-endian byte arrays without leading zeros, and then encoding those as any other leaf nodes. Other than that, RLP does not prescribe encodings for any data types, delegating that to the "users" of RLP.

Leaf and branching nodes are encoded into byte arrays by recursively adding a few extra bytes before most nodes to indicate the kind of node (leaf or branching) and the length of the subsequent bytes. These extra bytes tell the decoder how to reconstruct the nodes of the tree.

- Singleton byte arrays whose only byte is below 128 are encoded as themselves, without extra bytes; these are recognized by their starting and only byte being in the range 0–127. Other byte arrays whose length is in the range 0–55 have an extra starting byte in the range 128–183, which is 128 plus the length. Longer byte arrays have 1–8 extra bytes that contain the base-256 big-endian no-leading-zeros length, preceded by an extra byte in the range 184–191 that indicates the number of the base-256 big-endian bytes—184 for 1, 185 for 2, ..., and 191 for 8.
- Branching nodes $\langle\ldots\rangle$ are encoded by first recursively encoding the subtrees and concatenating all the encodings, and then adding one or more extra bytes to indicate the length of the concatenated encodings: if the length is in the range 0–55, there is a single extra byte in the range 192–247, which is 192 plus the length; otherwise, there are 1–8 extra bytes that contain the base-256 big-endian no-leading-zero length, preceded by an extra byte in the range 248–255 that indicates the number of the base-256 big-endian bytes—248 for 1, 249 for 2, ..., and 255 for 8.

For instance, $\langle\langle[1,2,3],\langle\rangle\rangle,[255],[]\rangle$, is encoded as $[201,197,131,1,2,3,192,129,255,128]$. This is illustrated in Figure 1 (right).

There is a symmetry between leaf and branch encodings with respect to small and large lengths: the first 56 values of the 64 values in the range 128–191 or 192–255 are used for small lengths from 0 to 55, while the remaining 8 values are used for large lengths up to $2^{64}-1$, i.e. the maximum value representable in 8 digits in base 256, since $256^8 = 2^{64}$. However, leaf nodes have an additional shorter encoding, when they consist of single bytes below 128.

```
(defun fact (n) ; a function
  (if (zp n)
      1
    (* n (fact (- n 1)))))

(defthm above ; a theorem
  (implies (natp n)
           (>= (fact n) n)))

(defchoose below (b) (n) ; an indefinite description
  (and (natp b)
       (< b (fact n))))

(defun-sk between (n) ; a quantification
  (exists (m)
    (and (natp m)
         (< (below n) m)
         (< m (fact n)))))

(defun gfact (n) ; a guarded function
  (declare (xargs :guard (natp n)))
  (if (zp n)
      1
    (* n (gfact (- n 1)))))
```

Figure 2: Some simple examples of ACL2 functions and theorems.

2.2 ACL2

ACL2 is a general-purpose interactive theorem prover based on an untyped first-order logic of total functions that is an extension of a purely functional subset of Common Lisp [8]. Predicates are functions and formulas are terms; they are false when their value is `nil`, and true when their value is `t` or anything else non-`nil`.

The ACL2 syntax is consistent with Lisp. A function application is a parenthesized list consisting of the function's name followed by the arguments, e.g. $x + 2 \times f(y)$ is written `(+ x (* 2 (f y)))`. Comments extend from semicolons to line endings.

The user interacts with ACL2 by submitting a sequence of theorems, function definitions, etc. ACL2 attempts to prove theorems automatically, via algorithms similar to NQTHM [2], most notably simplification and induction. The user guides these proof attempts mainly by (i) proving lemmas for use by specific proof algorithms (e.g. rewrite rules for the simplifier) and (ii) supplying theorem-specific 'hints' (e.g. to case-split on certain conditions). This paper says that a theorem is proved 'automatically' to mean that the proof involves no hints (at all, or none besides the ones explicitly mentioned) as well as no lemmas that are specific to the theorem; however, it may, and generally does, involve more generic library theorems, i.e. the theorem is typically not proved "from scratch".

The factorial function can be defined like `fact` in Figure 2, where `zp` tests if `n` is 0 or not a natural number. Thus `fact` treats arguments that are not natural numbers as 0. ACL2 functions often handle arguments of the wrong type by explicitly or implicitly coercing them to the right type—since the logic is untyped, in ACL2 'type' often denotes just any subset of the universe of values. This coercion may be implicit, as in `fact`, or explicit by calling a fixing function ('fixer' for short) for the type, i.e. a function that "fixes" values outside the type to be values inside the type, and leaves values already inside the type unchanged.

To preserve logical consistency, recursive function definitions must be proved to terminate via a measure of the arguments that decreases in each recursive call according to a well-founded relation. For `fact`, ACL2 automatically finds a measure and proves that it decreases according to a standard well-founded relation,

but sometimes the user has to supply a measure.

A theorem saying that `fact` is above its argument can be introduced like `above` in Figure 2, where `natp` tests if `n` is a natural number—in Lisp terminology, `natp` is a 'recognizer' of natural numbers. ACL2 proves this theorem automatically (if a standard arithmetic library [14, Path `books/arithmetic`] is loaded), finding and using an appropriate induction rule—the one derived from the recursive definition of `fact`, in this case.

Besides the discouraged ability to introduce arbitrary axioms, ACL2 provides logical-consistency-preserving mechanisms to axiomatize new functions, such as indefinite description functions. A function constrained to be strictly below `fact` can be introduced like `below` in Figure 2, where `b` is the variable bound by the indefinite description. This introduces the logically conservative axiom that, for every `n`, `(below n)` is a natural number less than `(fact n)`, if any exists—otherwise, `(below n)` is unconstrained.

ACL2's Lisp-like macro mechanism provides the ability to extend the language with new constructs defined in terms of existing constructs. For instance, despite the lack of built-in quantification in the logic, functions with top-level quantifiers can be introduced. The existence of a value strictly between `fact` and `below` can be expressed by a predicate like `between` in Figure 2, where `defun-sk` is a macro defined in terms of `defchoose` and `defun`, following a well-known construction [1]: $\exists x.\, P(x) \stackrel{\text{def}}{=} P(\varepsilon x.\, P(x))$, where the indefinite description $\varepsilon x.\, P(x)$, which can be read as 'choose an x such that $P(x)$', denotes some unspecified x that satisfies P if at least one exists; if none exists, $\varepsilon x.\, P(x)$ is completely unspecified, but is known not to satisfy P. In ACL2, a `defun-sk` introduces the indefinite description as a 'witness' function via `defchoose` (called `between-witness` for `between` in Figure 2, but not appearing in the figure) and defines the function (`between` in Figure 2) via `defun` with a body obtained by instantiating, in the matrix of the quantification (`(and (natp m) ...)` for `between` in Figure 2), the quantified variable (`m` for `between` in Figure 2) with the witness function applied to the arguments (`(between-witness n)` in Figure 2).

ACL2 functions are total, i.e. well-defined for all possible argument values. Nonetheless, ACL2 supports an optional guard mechanism to constrain function domains and ensure that functions are always called in their constrained domains. The function `gfact` in Figure 2 is a guarded version of `fact`, which requires the argument to be a natural number. The functions called by `gfact` also have guards: the argument of `zp` must be a natural number; the arguments of `-` and `*` must be numbers; and, recursively, the argument of `gfact` must be a natural number. To guard-verify `gfact`, ACL2 generates proof obligations for all the arguments of all the functions called by `gfact`, taking into account the contexts induced by `if`; these proof obligations are proved automatically in this case. To guard-verify functions that call `gfact`, ACL2 generates proof obligations requiring that `gfact` is always called with a natural number as argument.

The executable subset of the ACL2 language is essentially a subset of Lisp, which can therefore run efficiently on the underlying Lisp platform, especially when the functions are guard-verified. When guards are satisfied at run time, the underlying Common Lisp functions cannot cause run-time errors; when guards are verified at compile time, the checks for their satisfaction are omitted at run time.

3 RLP Encoding

3.1 Tree Structures

As explained in Section 2.1, RLP encodes trees (i.e. nested byte sequences) into flat byte sequences (i.e. byte arrays). In ACL2, these trees are formalized as shown in Figure 3, using the FTY macro library for structured recursive types [12].

The macro `fty::deftagsum` introduces a tagged sum type (disjoint union). The name of the type is `rlp-tree`. Leaf nodes are tagged by `:leaf`; branching nodes are tagged by `:branch`. A leaf node has a single component, called `bytes`, whose type is `byte-list` (whose definition if not shown), which consists of finite sequences (lists, in ACL2) of bytes, which are represented as natural numbers below 256 in ACL2. A branching node has a single component, called `subtrees`, whose type is `rlp-tree-list`. The macro `fty::deflist` introduces a type of lists with elements of the type specified just after `:elt-type`. Thus, the

```
(fty::deftypes rlp-trees
  (fty::deftagsum rlp-tree
    (:leaf ((bytes byte-list)))
    (:branch ((subtrees rlp-tree-list))))
  (fty::deflist rlp-tree-list
    :elt-type rlp-tree))
```

Figure 3: Formalization of RLP trees in ACL2.

type `rlp-tree-list` consists of lists of trees. The surrounding macro `fty::deftypes` introduces `rlp-tree` and `rlp-tree-list` as mutually recursive types, with `rlp-trees` as the names of the ensemble.

The `fty::deftagsum` that defines `rlp-tree` in Figure 3 introduces: a boolean recognizer `rlp-treep` that returns `t` exactly on the values of the type, i.e. on trees; constructors `rlp-tree-leaf` and `rlp-tree-branch` to build leaf trees from lists of bytes, and branching trees from subtrees; destructors `rlp-tree-leaf->bytes` and `rlp-tree-branch->subtrees` to retrieve lists of bytes from leaf trees, and subtrees from branching trees; and several theorems about these functions, which are also guard-verified. The `fty::deflist` that defines `rlp-tree-list` in Figure 3 introduces a boolean recognizer `rlp-tree-listp` that returns `t` exactly on the values of the type, i.e. on lists of trees, as well as theorems about this function and existing list functions; no specific constructors or destructors are introduced by `fty::deflist`, since the generic ones for lists can be used.

In essence, Figure 3 defines the set $\mathbb{T}$ of trees as a least fixpoint of the recursive set equation $\mathbb{T} = \{0, \ldots, 255\}^\star \uplus \mathbb{T}^\star$, where $\{0, \ldots, 255\}$ is the set of bytes, $X^\star$ is the set of all the finite sequences of elements in X (Kleene star), and $X \uplus Y$ is the disjoint union of X and Y. This is consistent with the set-theoretic notation in YP, which uses $\mathbb{O}$ for $\{0, \ldots, 255\}$, $\mathbb{B}$ for $\mathbb{O}^\star$, and $\mathbb{L}$ for $\mathbb{T}^\star$.

3.2 Encoding Functions

The encoding of RLP trees into byte arrays is formalized as shown in Figures 4 and 5.

The function `rlp-encode-bytes` in Figure 4 formalizes the encoding of byte arrays (at the leaves of RLP trees). The macro `define` is an enhancement of the built-in `defun` macro (see Section 2.2), that provides conveniences such as the ability to add type annotations to arguments and results: argument type annotations are turned into guards; result type annotations are turned into theorem declarations. The function `rlp-encode-bytes` takes as input a list of bytes `bytes` of type `byte-list` (mentioned in Section 3.1), whose recognizer is `byte-listp`, and returns as output a pair (formed via `mv`, for 'multiple value') consisting of a boolean error flag (`booleanp` is the ACL2 recognizer for the type of booleans) and a list of bytes that is the RLP encoding of the argument. The error flag is `t` exactly when the input consists of 2^{64} or more bytes: in this case, the input cannot be RLP-encoded, and the second component of the output is just the empty list of bytes `nil`, but its specific value is irrelevant when the error flag is `t`. Otherwise, the error flag is `nil` and the second component of the output is the encoding.

The macro `b*` is a binding construct. In its simple form `(b* ((`x_1 e_1`) ... (`x_n e_n`)) `e`)`, as used in Figure 4, it is a sequential `let` construct: the variable x_1 is bound to the expression e_1, then the variable x_2 (which may be the same as x_1) is bound to the expression e_2 (which may reference x_1), and so on; the final result is the value of the expression e.

The macro `cond` is a conditional construct. The value of `(cond (`c_1 e_1`) ... (`c_n e_n`))` is the value of the first e_i such that c_i returns a non-`nil` value and every preceding c_j, with $j < i$, returns `nil`; if $c_1, \ldots, c_n$ all return `nil`, the whole expression returns `nil`. This is equivalent to a suitable sequence of nested `if`s, with `nil` in the very last branch.

First, `rlp-encode-bytes` explicitly coerces the argument to be a list of bytes, via the fixer `byte-list-fix`. Then there are four cases:

```
(define rlp-encode-bytes ((bytes byte-listp))
  :returns (mv (error? booleanp)
               (encoding byte-listp))
  (b* ((bytes (byte-list-fix bytes)))
    (cond ((and (= (len bytes) 1)
                (< (car bytes) 128)) (mv nil bytes))
          ((< (len bytes) 56) (b* ((encoding (cons (+ 128 (len bytes))
                                                   bytes)))
                                (mv nil encoding)))
          ((< (len bytes)
              (expt 2 64)) (b* ((be (nat=>bebytes* (len bytes)))
                                (encoding (cons (+ 183 (len be))
                                                (append be bytes))))
                           (mv nil encoding)))
          (t (mv t nil)))))
```

Figure 4: Formalization of RLP byte array encoding in ACL2.

1. If the list consists of one byte (`len` returns the length of a list) and that byte is below 128 (`car` returns the first element of a list, in this case the only element), the operation is successful (i.e. the `error?` result is `nil`) and the encoding is the singleton list of the byte itself.
2. Otherwise, if the list consists of $l < 56$ bytes, the operation is successful and the encoding is obtained by prepending the byte $128 + l$ to the list of bytes (`cons` prepends an element to a list).
3. Otherwise, if the list consists of $l < 2^{64}$ bytes, the operation is successful and the encoding is obtained by prepending (i) the byte $183 + ll$ to the concatenation (obtained via `append`) of (ii) the base-256 big-endian no-leading-zeros representation of l of length ll ('length of length') and (iii) the initial list of bytes. Here `nat=>bebytes*` is a library function that turns a natural number into a list of bytes as big-endian digits in base 256, without leading zeros.
4. Otherwise, the operation fails (i.e. the `error?` result is `t`), because the list of bytes is too long to be encoded. To encode a list of 2^{64} or more bytes, 9 or more base-256 digits would be needed, i.e. it would be $ll \geq 9$, and therefore the first byte would be 192 or more, overlapping with the encoding of branching trees and preventing a decoder from discriminating leaf and branching trees from the first byte of an encoding.

The function `rlp-encode-bytes` formalizes the function R_{b} in YP, which returns one result: either the encoding (a byte sequence), or $\varnothing$ if the input byte sequence cannot be encoded. In the ACL2 formulation, it is slightly more convenient to return two results, so that each result has always the same type. The ACL2 function `nat=>bebytes*`, mentioned above, formalizes the function `BE` in YP.

The function `rlp-encode-tree` in Figure 5 formalizes the encoding of (both branching and leaf) trees. It takes as input a tree (see Figure 3) and returns as output a pair consisting of a boolean error flag and a list of bytes that is the RLP encoding of the argument; these are the same output types as `rlp-encode-bytes` in Figure 4.

Figure 5 uses more advanced features of the macro `b*` than the single-variable bindings used in Figure 4. In the general form `(b* ((`b_1 e_1`) ... (`b_n e_n`)) `e`)`, each b_i may be a pattern that binds multiple variables simultaneously: for instance, `((mv error? encoding) (rlp-encode-tree-list ...))` in Figure 5 binds `error?` and `encoding` to the first and second results of `rlp-encode-tree-list`—this function is explained below. Each b_i may also be an early-exit condition of the form `(when `c_i`)` or `(unless `c_i`)`, for immediately returning e_i if c_i is not `nil` (for `when`) or `nil` (for `unless`): for instance, `((when error?) (mv t nil))` in Figure 5 immediately returns `(mv t nil)` if `error?` is not `nil`.

The macro `rlp-tree-case` is generated by the `fty::deftagsum` in Figure 3. It performs a case analysis on the argument `tree`. If `tree` is a leaf tree (i.e. `:leaf` tag), `rlp-encode-bytes` is called on the list of

```
(defines rlp-encode-trees
  (define rlp-encode-tree ((tree rlp-treep))
    :returns (mv (error? booleanp)
                 (encoding byte-listp))
    (rlp-tree-case
     tree
     :leaf (rlp-encode-bytes tree.bytes)
     :branch (b* (((mv error? encoding) (rlp-encode-tree-list tree.subtrees))
                  ((when error?) (mv t nil)))
               (cond ((< (len encoding) 56)
                      (b* ((encoding (cons (+ 192 (len encoding))
                                           encoding)))
                        (mv nil encoding)))
                     ((< (len encoding)
                         (expt 2 64))
                      (b* ((be (nat=>bebytes* (len encoding)))
                           (encoding (cons (+ 247 (len be))
                                           (append be encoding))))
                        (mv nil encoding)))
                     (t (mv t nil))))))
  (define rlp-encode-tree-list ((trees rlp-tree-listp))
    :returns (mv (error? booleanp)
                 (encoding byte-listp))
    (b* (((when (endp trees)) (mv nil nil))
         ((mv error? encoding1) (rlp-encode-tree (car trees)))
         ((when error?) (mv t nil))
         ((mv error? encoding2) (rlp-encode-tree-list (cdr trees)))
         ((when error?) (mv t nil)))
      (mv nil (append encoding1 encoding2)))))
```

Figure 5: Formalization of RLP tree encoding in ACL2.

```
(define rlp-encode-scalar ((scalar natp))
  :returns (mv (error? booleanp)
               (encoding byte-listp))
  (rlp-encode-bytes (nat=>bebytes* (nfix scalar))))
```

Figure 6: Formalization of RLP scalar encoding in ACL2.

bytes in the leaf; the variable `tree.bytes` is bound to that list of bytes by `rlp-tree-case`. If instead `tree` is a branching tree (i.e. `:branch` tag), first `rlp-encode-tree-list` is called to encode each subtree and concatenate their encodings (more details below); the variable `tree.subtrees` is bound to those subtrees by `rlp-tree-case`. If `rlp-encode-tree-list` returns an error, `rlp-encode-tree` returns an error too: if any subtree cannot be encoded, the tree cannot be encoded either. Otherwise, there are three cases:

1. If the concatenated subtree encodings consist of $l < 56$ bytes, the overall operation is successful and the overall encoding is obtained by prepending the byte $192 + l$ to the concatenated subtree encodings.
2. Otherwise, if the concatenated subtree encodings consist of $l < 2^{64}$ bytes, the overall operation is successful and the overall encoding is obtained by prepending (i) the byte $247 + ll$ to the concatenation of (ii) the base-256 big-endian no-leading-zeros representation of l (of length ll) and (iii) the concatenated subtree encodings. (The function `nat=>bebytes*` is explained above, for `rlp-encode-bytes`.)
3. Otherwise, the overall operation fails, because the tree is too large to be encoded. To encode a tree whose concatenated subtree encodings consist of 2^{64} or more bytes, 9 or more base-256 digits would be needed, i.e. it would be $ll \geq 9$, and therefore the first byte would be 256 or more, which would not actually be a byte.

The function `rlp-encode-tree-list` in Figure 5 formalizes the encoding of a sequence of (sub)trees and the concatenation of the resulting encodings. It takes as input a list of trees and returns as output a pair consisting of a boolean error flag and a list of bytes that are the concatenated encodings of the argument trees; these are the same output types as `rlp-encode-tree`. When the list of trees is empty (i.e. `endp` holds), the result is the empty list of bytes `nil`. Otherwise, the first tree is encoded, the remaining list of trees is encoded (`cdr` returns the list resulting from removing the first element from a list), and the two resulting lists of bytes are concatenated. If any tree in the list cannot be encoded, an error is returned.

The functions `rlp-encode-tree` and `rlp-encode-tree-list` are mutually recursive. The macro `defines` groups mutually recursive functions that are introduced via `define`, with `rlp-encode-trees` as the name of the ensemble. Termination is proved automatically, based on the decreasing size of the argument trees, which is explicitly supplied as measure (not shown).

The function `rlp-encode-tree` in Figure 5 formalizes the function `RLP` in YP; the `:branch` case of the definition of `rlp-encode-tree` formalizes the function R_{l} in YP. The function `rlp-encode-tree-list` in Figure 5 formalizes the function s in YP.

As mentioned in Section 2.1, RLP prescribes that non-negative integers (scalars, in YP's terminology) be encoded by first representing them as base-256 big-endian byte arrays without leading zeros, and then encoding those as any other leaf nodes. This is formalized as shown in Figure 6, where `nfix` is a fixer for `natp`. Since `nat=>bebytes*` produces a list of bytes without leading zeros, the scalar 0 is encoded as the empty byte array.

The functions `rlp-encode-...` are guard-verified; their guard verification proofs are essentially automatic, with just a hint (not shown) to locally enable a rewrite rule that may be somewhat expensive to be always enabled. The proofs of the result type theorems of the `rlp-encode-...` functions are also proved essentially automatically, with just a hint (not shown) to locally enable a definition that is normally kept disabled and to locally enable a linear arithmetic rule that may be somewhat expensive to be always enabled. All these proofs make use of existing library rules about the functions called by the `rlp-encode-...` functions.

The `rlp-encode-...` functions provide a high-level specification of RLP encoding that also happens to be executable. The definitions of these functions correspond very closely to the definitions in YP. These

```
(define-sk rlp-tree-encoding-p ((encoding byte-listp))
  :returns (yes/no booleanp)
  (exists (tree)
          (and (rlp-treep tree)
               (b* (((mv error? encoding1) (rlp-encode-tree tree)))
                 (and (not error?)
                      (equal encoding1 (byte-list-fix encoding)))))))

(define-sk rlp-bytes-encoding-p ((encoding byte-listp))
  :returns (yes/no booleanp)
  (exists (bytes)
          (and (byte-listp bytes)
               (b* (((mv error? encoding1) (rlp-encode-bytes bytes)))
                 (and (not error?)
                      (equal encoding1 (byte-list-fix encoding)))))))

(define-sk rlp-scalar-encoding-p ((encoding byte-listp))
  :returns (yes/no booleanp)
  (exists (scalar)
          (and (natp scalar)
               (b* (((mv error? encoding1) (rlp-encode-scalar scalar)))
                 (and (not error?)
                      (equal encoding1 (byte-list-fix encoding)))))))
```

Figure 7: Formalization of valid RLP tree, byte array, and scalar encodings in ACL2.

definitions are also similar to the Python reference code in WK.

3.3 Valid Encodings

The valid encodings are formalized as shown in Figure 7. The macro `define-sk` is an enhancement of the built-in `defun-sk` macro (see Section 2.2) that provides similar conveniences as `define` compared to `defun` (see Section 3.2).

The predicate `rlp-tree-encoding-p` returns `t` exactly on the byte arrays that encode some trees. Roughly speaking, this predicate characterizes the image of `rlp-encode-tree`, restricted to the second result of that function, and subject to the constraint that the first result is `nil`. The fixer `byte-list-fix` just fixes the argument to be a list of bytes. The predicates `rlp-bytes-encoding-p` and `rlp-scalar-encoding-p` are analogous to `rlp-tree-encoding-p`, but they are restricted to encodings of byte arrays and scalars.

A valid encoding of a scalar is also a valid encoding of a byte array, and a valid encoding of a byte array is also a valid encoding of a tree, as asserted by some easily proven theorems (not shown).

3.4 Decodability Properties

RLP encodings are decodable, i.e. trees (or byte arrays, or scalars) can be recovered from their encodings—a basic requirement for any encoding method. This is expressed by the theorems in Figures 8 and 9: (i) the RLP encoding functions are injective, i.e. no two distinct trees (or bytes arrays, or scalars) have the same encoding; and (ii) the RLP encoding functions are prefix-unambiguous, i.e. no valid tree (or byte array, or scalar) encoding is a strict prefix of another one. The second property ensures the ability to decode from byte streams without "end-of-encoding" markers: if a valid encoding could be a strict prefix of another valid encoding, then after reading the former, a decoder could either stop there or proceed to decode a longer encoding, giving rise to an ambiguity.

```
(defthm rlp-encode-bytes-injective
  (implies (and (not (mv-nth 0 (rlp-encode-bytes x)))
                (not (mv-nth 0 (rlp-encode-bytes y))))
           (equal (equal (mv-nth 1 (rlp-encode-bytes x))
                         (mv-nth 1 (rlp-encode-bytes y)))
                  (equal (byte-list-fix x)
                         (byte-list-fix y)))))

(defthm rlp-encode-tree-injective
  (implies (and (not (mv-nth 0 (rlp-encode-tree x)))
                (not (mv-nth 0 (rlp-encode-tree y))))
           (equal (equal (mv-nth 1 (rlp-encode-tree x))
                         (mv-nth 1 (rlp-encode-tree y)))
                  (equal (rlp-tree-fix x)
                         (rlp-tree-fix y)))))

(defthm rlp-encode-scalar-injective
  (implies (and (not (mv-nth 0 (rlp-encode-scalar x)))
                (not (mv-nth 0 (rlp-encode-scalar y))))
           (equal (equal (mv-nth 1 (rlp-encode-scalar x))
                         (mv-nth 1 (rlp-encode-scalar y)))
                  (equal (nfix x)
                         (nfix y)))))
```

Figure 8: ACL2 theorems asserting the injectivity of RLP encoding.

Typically, the injectivity of a function f is stated as $[x \neq y \implies f(x) \neq f(y)]$, with x and y universally quantified, or equivalently as $[f(x) = f(y) \implies x = y]$. Because $[x = y \implies f(x) = f(y)]$ is always trivially true, injectivity can be also stated as $[f(x) = f(y) \iff x = y]$. If f is an ACL2 function that operates on values of a type (predicate) τ, its injectivity restricted to values of that type can be stated as $[\tau(x) \wedge \tau(y) \implies (f(x) = f(y) \iff x = y)]$; if f implicitly or explicitly fixes values outside τ via a fixer ϕ for τ, injectivity can be stated as $[f(x) = f(y) \iff \phi(x) = \phi(y)]$,[2] which is generally preferable as a rewrite rule in ACL2 because it has no hypotheses. This is the formulation in Figure 8, with the necessary hypotheses that the encoding functions return no error on `x` and `y`. The fixer `rlp-tree-fix` for RLP trees is automatically generated by the `fty::deftagsum` in Figure 3. The ACL2 function `mv-nth` projects the components (0-based) of an `mv` value, such as the one returned by the `rlp-encode-...` functions: thus `(mv-nth 0 ...)` is the `error?` result and `(mv-nth 1 ...)` is the `encoding` result (cf. Figures 4, 5, and 6).

The theorem `rlp-encode-bytes-injective` is proved by first proving a variant lemma (not shown) with `byte-listp` hypotheses on `x` and `y` and without `byte-list-fix`. The lemma is proved automatically, once the definition of `rlp-encode-bytes` is enabled via a hint, which leads ACL2 to consider nine cases—three for `x` and three for `y`, corresponding to the three branches in the definition of `rlp-encode-bytes`; see the documentation [13, Topic `rlp`] for details. Then the theorem is easily proved from the lemma via a hint (not shown) to use the instance of the lemma where `x` and `y` are replaced with `(byte-list-fix x)` and `(byte-list-fix y)`. Attempting to prove the theorem directly, with just the hint to enable the definition of `rlp-encode-bytes`, fails.

The theorem `rlp-encode-tree-injective` is also proved by first proving a variant lemma analogous to the one for `rlp-encode-bytes-injective` described above. Since `rlp-encode-tree` is mutually recursive with `rlp-encode-tree-list` (see Figure 5), this lemma is proved by induction along with a similar lemma about the injectivity of `rlp-encode-tree-list` (not shown). The induction schemas that ACL2 automatically generates for trees and for the encoding functions, which operate on single variables, do not work for these lemmas: a new induction schema (not shown) is provided that operates on two variables simultaneously,

[2] This is obtained by replacing x and y with $\phi(x)$ and $\phi(y)$ in $[\tau(x) \wedge \tau(y) \implies (f(x) = f(y) \iff x = y)]$, and using the facts that (i) $\forall z.\tau(\phi(z))$ (i.e. ϕ is a fixer for τ) and (ii) $\forall z.f(\phi(z)) = f(z)$ (i.e. f fixes its argument).

```
(defthm rlp-encode-bytes-unamb-prefix
  (implies (and (not (mv-nth 0 (rlp-encode-bytes x)))
                (not (mv-nth 0 (rlp-encode-bytes y))))
           (equal (prefixp (mv-nth 1 (rlp-encode-bytes x))
                           (mv-nth 1 (rlp-encode-bytes y)))
                  (equal (mv-nth 1 (rlp-encode-bytes x))
                         (mv-nth 1 (rlp-encode-bytes y))))))

(defthm rlp-encode-tree-umamb-prefix
  (implies (and (not (mv-nth 0 (rlp-encode-tree x)))
                (not (mv-nth 0 (rlp-encode-tree y))))
           (equal (prefixp (mv-nth 1 (rlp-encode-tree x))
                           (mv-nth 1 (rlp-encode-tree y)))
                  (equal (mv-nth 1 (rlp-encode-tree x))
                         (mv-nth 1 (rlp-encode-tree y))))))

(defthm rlp-encode-scalar-unamb-prefix
  (implies (and (not (mv-nth 0 (rlp-encode-scalar x)))
                (not (mv-nth 0 (rlp-encode-scalar y))))
           (equal (prefixp (mv-nth 1 (rlp-encode-scalar x))
                           (mv-nth 1 (rlp-encode-scalar y)))
                  (equal (mv-nth 1 (rlp-encode-scalar x))
                         (mv-nth 1 (rlp-encode-scalar y))))))
```

Figure 9: ACL2 theorems asserting the prefix umambiguity of RLP encoding.

such as `x` and `y` in the lemmas; this new induction schema is more general than its use in this proof. Some of the base and step cases of the induction are proved automatically, while other make use of some fairly specific lemmas (not shown); see the documentation [13, Topic `rlp`] for details. The base case, in which `x` or `y` is a leaf tree, makes use of the injectivity theorem for `rlp-encode-bytes`.

The theorem `rlp-encode-scalar-injective`, after enabling the definition of `rlp-encode-scalar`, is proved automatically. The proof makes use of the injectivity theorem for `rlp-encode-bytes`, as well as for a library injectivity theorem for `nat=>bebytes*`.

If a function f has a left inverse g, then f is injective: given $f(x) = f(y)$, applying g to both sides to obtain $g(f(x)) = g(f(y))$, the left inverse property yields $g(f(x)) = x = y = g(f(y))$. Thus, the injectivity of the RLP encoding functions could be alternative proved by defining RLP decoding functions, proving that the latter are left inverses of the former, and then deriving injectivity as sketched for f and g. In contrast, the injectivity proofs explained above, which are not particularly difficult, are solely in terms of the encoding functions, which is more elegant and abstract in the author's opinion.

The prefix-unambiguity theorems in Figure 9 say that if a valid encoding encoding is a prefix of another valid encoding, then the two encodings are equal; similarly to the injectivity theorems, these theorem also state the easily proved converse implication, so that the theorems are more useful rewrite rules. The library function `prefixp` says whether the first argument is a (not necessarily strict) prefix of the second argument. Thus, the theorems prohibit a valid encoding from being a strict prefix of another valid encoding.

Each of the theorems `rlp-encode-bytes-unamb-prefix` and `rlp-encode-tree-umamb-prefix` is proved via a case split, specified as a hint (not shown), on whether the lengths of the encodings are equal or not. If the lengths are equal, the encodings must be equal since one is a prefix of the other; this is proved automatically via library theorems about `prefixp`. If the lengths are not equal, a more general theorem (not shown) is used to show that the lengths must be actually actual, proving this case by contradiction. That more general theorem says that the length of an encoding is determined by the first few bytes of the encoding, because encodings start with length information. Additional hints (not shown) guide ACL2 to recognizing that those first few bytes must be the same for the two encodings if one is a prefix of the other.

```
(define rlp-decode-tree ((encoding byte-listp))
  :returns (mv (error? booleanp)
               (tree rlp-treep))
  (b* ((encoding (byte-list-fix encoding)))
    (if (rlp-tree-encoding-p encoding)
        (mv nil (rlp-tree-encoding-witness encoding))
      (mv t (rlp-tree-leaf nil)))))

(define rlp-decode-bytes ((encoding byte-listp))
  :returns (mv (error? booleanp)
               (bytes byte-listp))
  (b* ((encoding (byte-list-fix encoding)))
    (if (rlp-bytes-encoding-p encoding)
        (mv nil (rlp-bytes-encoding-witness encoding))
      (mv t nil))))

(define rlp-decode-scalar ((encoding byte-listp))
  :returns (mv (error? booleanp)
               (scalar natp))
  (b* ((encoding (byte-list-fix encoding)))
    (if (rlp-scalar-encoding-p encoding)
        (mv nil (rlp-scalar-encoding-witness encoding))
      (mv t 0))))
```

Figure 10: Declarative definitions of RLP decoding in ACL2.

The theorem `rlp-encode-scalar-unamb-prefix` is proved automatically after enabling the definition of `rlp-encode-scalar`. This replaces `(rlp-encode-scalar ...)` with `(rlp-encode-bytes (nat=>bebytes* ...))`, then `rlp-encode-bytes-unamb-prefix` applies, turning `prefixp` into `equal`, and finally the library injectivity theorem about `nat=>bebytes*` applies, yielding the equality of the encodings. In contrast, no hints to enable the definitions of `rlp-encode-bytes` and `rlp-encode-tree` are needed in their prefix-unamibiguity theorems; those definitions participate only indirectly, via the rules about those two functions used to prove the two theorems.

4 RLP Decoding

4.1 Declarative Specification

The RLP decoding of trees, byte arrays, and scalars from their flat encodings is formalized as shown in Figure 10.

The function `rlp-decode-tree` in Figure 10 takes as input a list of bytes that is the purported encoding and returns as output a pair consisting of a boolean error flag and a tree. The error flag is `nil` exactly when the input is a valid encoding of a tree: in this case, that tree is returned, via the witness function `rlp-tree-encoding-witness` associated to `rlp-tree-encoding-p` in Figure 7 (more on this below). Otherwise, the error flag is `t` and the second component of the output is just a leaf tree with the empty list of bytes, but this second component is irrelevant in this case.

As explained in Section 2.2, an existentially quantified function like `rlp-tree-encoding-p` is defined in terms of a witness function, `rlp-tree-encoding-witness` in this case, which is axiomatized, via the matrix of the quantification, to be a right inverse of the encoding function `rlp-encode-tree` over the valid encodings: if `encoding` is a valid tree encoding, then `(rlp-tree-encoding-witness encoding)` returns a tree `tree` of type `rlp-treep` such that `(rlp-encode-tree tree)` returns `(mv nil encoding)`, i.e. no

```
(defthm rlp-encode-tree-of-rlp-decode-tree
  (implies (rlp-tree-encoding-p encoding)
           (b* (((mv d-error? tree) (rlp-decode-tree encoding))
                ((mv e-error? encoding1) (rlp-encode-tree tree)))
             (and (not d-error?)
                  (not e-error?)
                  (equal encoding1
                         (byte-list-fix encoding))))))

(defthm rlp-decode-tree-of-rlp-encode-tree
  (b* (((mv e-error? encoding) (rlp-encode-tree tree))
       ((mv d-error? tree1) (rlp-decode-tree encoding)))
    (implies (not e-error?)
             (and (not d-error?)
                  (equal tree1
                         (rlp-tree-fix tree))))))
```

Figure 11: ACL2 theorems asserting that tree encoding and decoding functions are mutual inverses.

error and the original encoding `encoding`. This readily implies that `rlp-decode-tree` is a right inverse of `rlp-encode-tree`, as stated by the theorem `rlp-encode-tree-of-rlp-decode-tree` in Figure 11: if `encoding` is (modulo `byte-list-fix`) a valid tree encoding, then `rlp-decode-tree` succeeds and returns a tree `tree`, and `rlp-encode-tree` succeeds on `tree` and returns the original encoding. This theorem is proved automatically, using easily proved theorems (not shown) about `rlp-tree-encoding-witness`.

Since, as discussed in Section 3.4, `rlp-encode-tree` is injective, it follows that `rlp-decode-tree` is also a left inverse of `rlp-encode-tree`, as stated by the theorem `rlp-decode-tree-of-rlp-encode-tree` in Figure 11: if `tree` is (modulo `rlp-tree-fix`) an RLP tree, then `rlp-encode-tree` succeeds and returns an encoding `encoding`, and `rlp-decode-tree` succeeds on `encoding` and returns the original tree. In general, if a function f is injective and has a right inverse g, then g is also a left inverse of f: given the right inverse property $f(g(x)) = x$, replacing x with $f(y)$ yields $f(g(f(y))) = f(y)$, which the injectivity of f reduces to $g(f(y)) = y$, i.e. the left inverse property. The left inverse theorem `rlp-decode-tree-of-rlp-encode-tree` is proved via a few hints (not shown) to instantiate (i.e. replace x with $f(y)$ as above) and use the right inverse theorem `rlp-encode-tree-of-rlp-decode-tree`, while the injectivity theorem in Figure 8 is automatically used as a rewrite rule.

The functions `rlp-decode-bytes` and `rlp-decode-scalar` in Figure 10 are analogous to `rlp-decode-tree`. Left and right inverse theorems (not shown) that relate these two functions to `rlp-encode-bytes` and `rlp-encode-scalar` are analogous to the ones in Figure 11, and are proved analogously.

Figure 10 defines the RLP decoding functions as inverses of the RLP encoding functions. These are declarative, non-executable definitions. YP does not explicitly define any RLP decoding functions, but, clearly, it implicitly defines them as inverses of the encoding functions: this implicit definition is formalized in Figure 10. WK provides Python reference code for RLP decoding, but the definitions in Figure 10 are more abstract and manifestly correct.

4.2 Executable Implementation

The functions in Figure 10 can be regarded as high-level specification of RLP decoding, which can be implemented by equivalent executable functions.

The executable decoding functions are defined in terms of the executable RLP parser shown in Figures 12, 13, and 14. This function is mutually recursive with the one in Figure 15. Similarly to Figure 5, these two functions are surrounded by a `defines` (not shown).

```
(define rlp-parse-tree ((encoding byte-listp))
  :returns (mv (error? maybe-rlp-error-p)
               (tree rlp-treep)
               (rest byte-listp))
  (b* ((encoding (mbe :logic (byte-list-fix encoding) :exec encoding))
       (irrelevant (rlp-tree-leaf nil))
       ((when (endp encoding)) (mv (rlp-error-no-bytes) irrelevant nil))
       ((cons first encoding) encoding)
       ((when (< first 128)) (mv nil (rlp-tree-leaf (list first)) encoding))
       ((when (<= first 183))
        (b* ((len (- first 128))
             ((when (< (len encoding) len))
              (mv (rlp-error-fewer-bytes-than-short-length (list first)
                                                           len
                                                           (len encoding))
                  irrelevant
                  nil))
             (bytes (take len encoding))
             ((when (and (= len 1)
                         (< (car bytes) 128)))
              (mv (rlp-error-non-optimal-short-length (list first
                                                            (car bytes)))
                  irrelevant
                  nil))
             (encoding (nthcdr len encoding)))
          (mv nil (rlp-tree-leaf bytes) encoding)))
       ... ; see next figure
```

Figure 12: Executable parser of RLP tree encodings in ACL2 (Part 1).

```
... ; see previous figure
((when (< first 192))
 (b* ((lenlen (- first 183))
      ((when (< (len encoding) lenlen))
       (mv (rlp-error-fewer-bytes-than-length-of-length (list first)
                                                        lenlen
                                                        (len encoding))
           irrelevant
           nil))
      (len-bytes (take lenlen encoding))
      ((unless (equal (trim-bendian* len-bytes)
                      len-bytes))
       (mv (rlp-error-leading-zeros-in-long-length (cons first
                                                         len-bytes))
           irrelevant
           nil))
      (encoding (nthcdr lenlen encoding))
      (len (bebytes=>nat len-bytes))
      ((when (<= len 55))
       (mv (rlp-error-non-optimal-long-length (cons first len-bytes))
           irrelevant
           nil))
      ((when (< (len encoding) len))
       (mv (rlp-error-fewer-bytes-than-long-length (cons first
                                                         len-bytes)
                                                   len
                                                   (len encoding))
           irrelevant
           nil))
      (bytes (take len encoding))
      (encoding (nthcdr len encoding)))
   (mv nil (rlp-tree-leaf bytes) encoding)))
... ; see next figure
```

Figure 13: Executable parser of RLP tree encodings in ACL2 (Part 2).

```
   ... ; see previous figure
   ((when (<= first 247))
    (b* ((len (- first 192))
         ((when (< (len encoding) len))
          (mv (rlp-error-fewer-bytes-than-short-length (list first)
                                                       len
                                                       (len encoding))
              irrelevant
              nil))
         (subencoding (take len encoding))
         (encoding (nthcdr len encoding))
         ((mv error? subtrees) (rlp-parse-tree-list subencoding))
         ((when error?) (mv (rlp-error-subtree error?) irrelevant nil)))
      (mv nil (rlp-tree-branch subtrees) encoding)))
   (lenlen (- first 247))
   ((when (< (len encoding) lenlen))
    (mv (rlp-error-fewer-bytes-than-length-of-length (list first)
                                                     lenlen
                                                     (len encoding))
        irrelevant
        nil))
   (len-bytes (take lenlen encoding))
   ((unless (equal (trim-bendian* len-bytes)
                   len-bytes))
    (mv (rlp-error-leading-zeros-in-long-length (cons first len-bytes))
        irrelevant
        nil))
   (encoding (nthcdr lenlen encoding))
   (len (bebytes=>nat len-bytes))
   ((when (<= len 55))
    (mv (rlp-error-non-optimal-long-length (cons first len-bytes))
        irrelevant
        nil))
   ((when (< (len encoding) len))
    (mv (rlp-error-fewer-bytes-than-long-length (cons first len-bytes)
                                                len
                                                (len encoding))
        irrelevant
        nil))
   (subencoding (take len encoding))
   (encoding (nthcdr len encoding))
   ((mv error? subtrees) (rlp-parse-tree-list subencoding))
   ((when error?)
    (mv (rlp-error-subtree error?) irrelevant nil)))
(mv nil (rlp-tree-branch subtrees) encoding)))
```

Figure 14: Executable parser of RLP tree encodings in ACL2 (Part 3).

```
(define rlp-parse-tree-list ((encoding byte-listp))
  :returns (mv (error? maybe-rlp-error-p)
               (trees rlp-tree-listp))
  (b* (((when (endp encoding)) (mv nil nil))
       ((mv error? tree encoding1) (rlp-parse-tree encoding))
       ((when error?) (mv error? nil))
       ((unless (mbt (< (len encoding1) (len encoding))))
        (mv (rlp-error-no-bytes) nil))
       ((mv error? trees) (rlp-parse-tree-list encoding1))
       ((when error?) (mv error? nil)))
    (mv nil (cons tree trees))))
```

Figure 15: Executable parser of RLP tree list encodings in ACL2.

The function `rlp-parse-tree` (see Figure 12) takes as input a list of bytes and returns as output a triple consisting of an error indication (`nil` if parsing is successful), a decoded tree (an irrelevant tree value if parsing fails), and the remaining bytes after the parsed encoding (`nil` if parsing fails). The parser stops as soon as a tree is successfully decoded (because of theorem `rlp-encode-tree-unamb-prefix` in Figure 9, there cannot be a longer encoding to parse), returning the remaining bytes for further parsing: thus, as lists of trees are recursively parsed, the input bytes are threaded through, and consumed chunk-wise. The function `rlp-parse-tree-list` (see Figure 15) takes as input as list of bytes and returns as output a pair consisting of an error indication (similarly to `rlp-parse-tree`) and a list of decoded trees; it returns no remaining bytes because it is always called (by `rlp-parse-tree`) on a sublist of the input of known length that must exactly encode zero or more trees. The predicate `maybe-rlp-error-p` recognizes `nil` (for no error) and error values (recognized by `rlp-error-p`, not shown) that convey information about different possible parsing errors. The functions `rlp-error-...` that appear in the parser construct error values, but are not discussed in detail here.

The parser operates as follows:

1. Starting in Figure 12, the `(mbe :logic ... :exec ...)` expression, where `mbe` stands for 'must be equal', logically fixes the `encoding` input to be a list of bytes, without actually doing that during execution since the guard guarantees that `encoding` is a list of bytes; see [3] for details.
2. If the input list of bytes is empty, an error is returned: RLP encodings are never empty.
3. Otherwise, the pattern `(cons first encoding)` is matched to `encoding`, binding the variable `first` to the first byte of the encoding and the variable `encoding` to the rest of the encoding. This way, the first byte can be examined to determine what to do next.
4. If the first byte is below 128, it encodes the singleton list of itself (see `rlp-encode-bytes` in Figure 4), and so the byte is returned as a leaf tree.
5. If the first byte is between 128 and 183, the encoding must be of a byte array with length below 56 (see `rlp-encode-bytes` in Figure 4). The length is calculated: if not enough bytes occur after the first, an error is returned; otherwise, the bytes are returned as a leaf tree. The ACL2 function `take` returns the prefix of its second argument (a list) of length specified by its first argument (a natural number). The ACL2 function `nthcdr` returns the suffix of its second argument (a list) after removing the prefix whose length is specified by its first argument (a natural number). An error is also returned if there is just one byte below 128; see Section 4.2.1.
6. Continuing in Figure 13, if the first byte of the encoding is between 129 and 191, the encoding must be of a byte array whose length `len` is encoded by the next `lenlen` bytes after the first one, as a base-256 big-endian no-leading-zeros list (see `rlp-encode-bytes` in Figure 4). After obtaining `lenlen` from the first byte, an error is returned if there are not enough bytes to encode the length. An error is also returned if the encoded length has leading zeros (the library function `trim-bendian*` removes all the leading zeros from a list of big-endian digits); see Section 4.2.1. Otherwise, the library function

`bebytes=>nat`, inverse of `nat=>bebytes*` in Figure 4, is used to calculate the length `len` of the encoded byte array. An error is returned if this length is below 56; see Section 4.2.1. If there are enough bytes after the first and the next `lenlen` bytes, they are returned as a leaf tree.

7. Continuing in Figure 14, if the first byte of the encoding is between 192 and 247, the encoding must be of a branching tree whose subtrees have a total encoded length below 56 (see `rlp-encode-tree` in Figure 5). That exact number of bytes is passed to `rlp-parse-tree-list` (which is described in more detail below), which returns an error indication (`nil` if no error) and the list of decoded trees, which are wrapped into a branching tree and returned. As above, an error is returned if there are not enough bytes in the input. Any error from `rlp-parse-tree-list` is propagated.

8. If the first byte of the encoding is between 248 and 255, the encoding must be of a branching tree whose length `len` is encoded by the next `lenlen` bytes after the first one, as a base-256 big-endian no-leading-zeros list (see `rlp-encode-tree` in Figure 5). The processing of `lenlen` and `len` is analogous to Figure 13 (explained above), including returning errors if there are leading zeros in the length or if `len` is below 56 (see Section 4.2.1). Similarly to the case in which the first byte is between 192 and 247, `rlp-parse-tree-list` is called with the exact number of bytes to parse, errors from there are propagated, and ultimately (if no errors occur) the decoded branching tree is returned.

The function `rlp-parse-tree-list` in Figure 15 takes as input a list of bytes purported to encode zero or more trees and returns as output a pair consisting of an error indication (`nil` if no error) and a list of decoded RLP trees. Unlike `rlp-parse-tree`, it does not return any remaining input bytes, as explained earlier.

The body of `rlp-parse-tree-list` decodes trees while there are input bytes available, stopping if an error occurs and propagating that error. After each tree is decoded, the remaining bytes are recursively parsed. The `mbt` expression serves to prove termination, as discussed below.

The termination of the mutually recursive functions `rlp-parse-tree` and `rlp-parse-tree-list` is proved automatically once a measure (not shown) is provided. The measure is lexicographic: it consists of the length of the input list of bytes, followed by a linear ordering of the two functions defined by `rlp-parse-tree` being smaller than `rlp-parse-tree-list`. Where `rlp-parse-tree` calls `rlp-parse-tree-list`, the first component of the measure decreases. Where `rlp-parse-tree-list` calls `rlp-parse-tree`, the first component is unchanged but the second decreases. Where `rlp-parse-tree-list` calls itself, the first component also decreases, because RLP encodings are never empty, and therefore the preceding call of `rlp-parse-tree` must have consumed some input bytes. The latter is a property of `rlp-parse-tree`, but in order to prove this property, the function must be first accepted by ACL2, requiring its termination to be proved. This circularity is broken via the run-time test `(< (len encoding1) (len encoding))` in Figure 15, which gets the function definition accepted by ACL2. The `mbt` that surrounds the test stands for 'must be true': it is a mechanism, related to `mbe` (mentioned above), provided by ACL2 to include logical tests without executing them because they are always true under the guard; see [3] for details.

The guard verification proofs, which involve the aforementioned obligations for `mbe` and `mbt`, are automatic after proving a theorem saying that the third result of `rlp-parse-tree`, i.e. the remaining input bytes, is strictly shorter than the input—this is for proving the satisfaction of the `mbt` test described above. The theorem about input and output lengths (not shown) is proved automatically after supplying a hint to expand the definition of `rlp-parse-tree`, which otherwise ACL2's heuristics apparently prevent from expanding.

The executable RLP decoding functions for trees, byte arrays, and scalars are defined as shown in Figure 16; the '`x`' in their names stands for 'executable', as opposed to the declaratively defined decoding functions in Figure 10. These decoding functions take as input a list of bytes, purported to be a complete encoding with no extra bytes,[3] and return as output a pair consisting of an error indication (`nil` if decoding is successful) and a tree, byte array, or scalar—whose values are irrelevant if the first result is not `nil`.

[3]These executable decoding functions are meaningful as implementations of the inverses of the encoding functions declaratively defined in Figure 10. They can be used when the lengths of the purported encodings are known. When the lengths are not known, e.g. when decoding from byte streams, the executable parser can be used instead.

```
(define rlp-decodex-tree ((encoding byte-listp))
  :returns (mv (error? maybe-rlp-error-p)
               (tree rlp-treep))
  (b* (((mv error? tree rest) (rlp-parse-tree encoding))
       ((when error?) (mv error? (rlp-tree-leaf nil)))
       ((when (consp rest))
        (mv (rlp-error-extra-bytes rest) (rlp-tree-leaf nil))))
    (mv nil tree)))

(define rlp-decodex-bytes ((encoding byte-listp))
  :returns (mv (error? maybe-rlp-error-p)
               (bytes byte-listp))
  (b* (((mv error? tree) (rlp-decodex-tree encoding))
       ((when error?) (mv error? nil))
       ((unless (rlp-tree-case tree :leaf))
        (mv (rlp-error-non-leaf-tree (list (car encoding))) nil))
       (bytes (rlp-tree-leaf->bytes tree)))
    (mv nil bytes)))

(define rlp-decodex-scalar ((encoding byte-listp))
  :returns (mv (error? maybe-rlp-error-p)
               (scalar natp))
  (b* (((mv error? bytes) (rlp-decodex-bytes encoding))
       ((when error?) (mv error? 0))
       ((unless (equal (trim-bendian* bytes) bytes))
        (mv (rlp-error-leading-zeros-in-scalar bytes) 0))
       (scalar (bebytes=>nat bytes)))
    (mv nil scalar)))
```

Figure 16: Executable definitions of RLP decoding in ACL2.

In `rlp-decodex-tree`, a tree is decoded by calling the parser and ensuring that there are no remaining bytes—the ACL2 function `consp` tests if a list is not empty. In `rlp-decodex-bytes`, a byte array is decoded by decoding a tree and ensuring that it is a leaf tree, returning the underlying byte array. In `rlp-decodex-scalar`, a scalar is decoded by decoding a byte array ensuring that it has no leading zeros (see Section 2.1), and treating it as a base-256 big-endian representation.

4.2.1 Rejection of Invalid Quasi-Encodings

As mentioned above, the parser rejects "quasi-encodings" of the following forms:

- $[129, x]$ with $x < 128$: this "could" encode (a leaf tree consisting of) a singleton byte array $[x]$, but RLP prescribes the encoding $[x]$ in this case.
- $[183 + ll, l_1, \ldots, l_{ll}, x, \ldots]$ with $1 \leq ll \leq 8$ and $l_1 = 0$: this "could" encode (a leaf tree consisting of) a byte array $[x, \ldots]$ of length $l = \sum_{1 \leq i \leq ll} l_i \times 256^{ll-i}$, but RLP prescribes the absence of leading zeros in the base-256 big-endian representation of l.
- $[183 + ll, l_1, \ldots, l_{ll}, x, \ldots]$ with $1 \leq ll \leq 8$ and $l = \sum_{1 \leq i \leq ll} l_i \times 256^{ll-i} < 56$: this "could" encode (a leaf tree consisting of) a byte array $[x, \ldots]$ of length l, but RLP prescribes the encoding $[128 + l, x, \ldots]$ in this case.
- $[247 + ll, l_1, \ldots, l_{ll}, x, \ldots]$ with $1 \leq ll \leq 8$ and $l_1 = 0$: this "could" encode a branching tree whose concatenated encoded subtrees have length $l = \sum_{1 \leq i \leq ll} l_i \times 256^{ll-i}$, but RLP prescribes the absence of leading zeros in the base-256 big-endian representation of l.

```
(defthm rlp-parse-tree-of-rlp-encode-tree
  (b* (((mv e-error? encoding) (rlp-encode-tree tree))
       ((mv d-error? tree1 rest) (rlp-parse-tree encoding)))
    (implies (not e-error?)
             (and (not d-error?)
                  (not (consp rest))
                  (equal tree1 (rlp-tree-fix tree))))))

(defthm rlp-parse-tree-list-of-rlp-encode-tree-list
  (b* (((mv e-error? encoding) (rlp-encode-tree-list trees))
       ((mv d-error? trees1) (rlp-parse-tree-list encoding)))
    (implies (not e-error?)
             (and (not d-error?)
                  (equal trees1 (rlp-tree-list-fix trees))))))

(defthm rlp-encode-tree-of-rlp-parse-tree
  (b* (((mv d-error? tree rest) (rlp-parse-tree encoding))
       ((mv e-error? encoding1) (rlp-encode-tree tree)))
    (implies (not d-error?)
             (and (not e-error?)
                  (equal (append encoding1 rest)
                         (byte-list-fix encoding))))))

(defthm rlp-encode-tree-list-of-rlp-parse-tree-list
  (b* (((mv d-error? trees) (rlp-parse-tree-list encoding))
       ((mv e-error? encoding1) (rlp-encode-tree-list trees)))
    (implies (not d-error?)
             (and (not e-error?)
                  (equal encoding1 (byte-list-fix encoding))))))
```

Figure 17: ACL2 theorems asserting that tree encoding and parsing functions are mutual inverses.

- $[247 + ll, l_1, \ldots, l_{ll}, x, \ldots]$ with $1 \leq ll \leq 8$ and $l = \sum_{1 \leq i \leq ll} l_i \times 256^{ll-i} < 56$: this "could" encode a branching tree whose concatenated encoded subtrees have length l, but RLP prescribes the encoding $[192 + l, x, \ldots]$ in this case.

These are not valid encodings because they do not satisfy the predicates in Figure 7. While they "could" be encodings in the sense mentioned above, they are "non-optimal", in the sense that shorter valid encodings exist.

When implementing RLP decoding, the rejection of these quasi-encodings may be easily overlooked. For instance, the Python reference code in WK used to accept these quasi-encodings, and some existing RLP implementations used to accept or still accept them as well (see Section 5). While it may seem benign to accept these quasi-encodings, they are just not in the image of the RLP encoding functions.

4.3 Verification of Correctness

The first step toward proving that the executable definitions in Figure 16 are equivalent to the declarative definitions in Figure 10 is to prove that the tree parsing and encoding functions are mutual inverses, which is expressed by the theorems in Figure 17.

The first theorem in Figure 17 asserts that `rlp-parse-tree` is a left inverse of `rlp-encode-tree` over the encodable trees, i.e. that the parser recognizes and reconstructs *all* valid encodings of trees. More in detail, it asserts that if `rlp-encode-tree` succeeds, then `rlp-parse-tree` succeeds on the resulting encoding,

returning the original tree (modulo fixing) and no remaining bytes (i.e. it consumes the whole encoding). Since `rlp-parse-tree` is mutually recursive with `rlp-parse-tree-list` and `rlp-encode-tree` is mutually recursive with `rlp-encode-tree-list`, that theorem is proved at the same time as the second theorem in Figure 17, which says that `rlp-parse-tree-list` is a left inverse of `rlp-encode-tree-list`. These two theorems are proved by induction on `rlp-encode-tree` and `rlp-encode-tree-list`, via the `make-flag` macro [13, Topic `make-flag`]. The proof is automatic once the definitions of these and other functions are enabled, and some hints (not shown) are provided to force the expansion of some calls of `rlp-parse-tree`; see the documentation [13, Topic `rlp`] for details. The proof also makes use of a previously proved more general rewrite rule (not shown) asserting that if `rlp-parse-tree` succeeds on `encoding` returning a tree and some remaining bytes `rest`, it also succeeds on an extended input `(append encoding more-bytes)`, returning the same tree and `(append rest more-bytes)` as remaining bytes.

The third theorem in Figure 17 asserts that `rlp-parse-tree` is a right inverse of `rlp-encode-tree` over the valid tree encodings, i.e. that the parser recognizes and reconstructs *only* valid encodings of trees: if it accepted an invalid encoding and returned a tree, `rlp-encode-tree` would have to map that tree back to the encoding, which would be therefore valid, contradicting the hypothesis that it is invalid. More in detail, the theorem asserts that if `rlp-parse-tree` succeeds, then `rlp-encode-tree` succeed on the resulting tree, returning the prefix of the original encoding (modulo fixing) that omits the remaining bytes returned by `rlp-parse-tree`. Analogously to the left inverse theorem described above, this theorem is proved at the same time as the fourth theorem in Figure 17, by induction on `rlp-parse-tree` and `rlp-parse-tree-list`[4] via `make-flag`, automatically once the definitions of these and other functions are enabled.

If the parser accepted the quasi-encodings discussed in Section 4.2.1, the left inverse theorem would still hold, but the right inverse theorem would not.

The fact that `rlp-decodex-tree` in Figure 16 is both a left and a right inverse of `rlp-encode-tree` easily follows from the first and third theorems in Figure 17. This fact is asserted by two theorems (not shown) that are written as in Figure 11 but where `rlp-decode-tree` is replaced with `rlp-decodex-tree`. The left inverse theorem is proved automatically once the definition of `rlp-decodex-tree` is enabled; the first theorem in Figure 17 applies as a rewrite rule. The right inverse theorem, besides enabling the definition of `rlp-decodex-tree`, requires a couple of hints (not shown) to use the third theorem in Figure 17, since the presence of `append` in it does not make it readily applicable as a rewrite rule in this case.

The equivalence of `rlp-decodex-tree` and `rlp-decode-tree` is stated by the theorem in Figure 18. Since `rlp-decode-tree` returns a boolean error result while `rlp-decodex-tree` returns a richer range of error results, the first results of these two functions are only `iff`-equivalent, i.e. one is `nil` if and only if the other one is `nil`; their second results are always equal instead.

The theorem in Figure 18 is proved by cases on whether `(rlp-tree-encoding-p encoding)` holds or not. A preliminary lemma (not shown) is proved, with a few hints and a couple of simple intermediate lemmas, asserting the equivalence of (i) `rlp-tree-encoding-p` returning `t` and (ii) `rlp-decodex-tree` returning a `nil` error result. If `rlp-tree-encoding-p` holds:

- The first conjunct in Figure 18 is proved via a few hints, using the aforementioned lemma and the definition of `rlp-decode`.
- The second conjunct in Figure 18 is proved via a few hints, from the injectivity of `rlp-encode-tree` and the right inverse properties of `rlp-decode-tree` and `rlp-decodex-tree`. In general, if an injective function f has right inverses g and h, then $g = h$: from the right inverse properties $f(g(x)) = x = f(h(x))$, injectivity gives $g(x) = h(x)$.

If instead `rlp-tree-encoding-p` does not hold, both conjuncts in Figure 18 are proved via a few hints, using the aforementioned preliminary lemma and the definitions of `rlp-decode-tree` and `rlp-decodex-tree`.

The equivalence of `rlp-decodex-bytes` and `rlp-decode-bytes` is stated by a theorem similar to Figure

[4] In ACL2, as in NQHTM [2, Chapt. 15], induction is applicable to recursive functions when the arguments that decrease in the recursion are variables. Thus, while the left inverse theorems are proved by induction on `rlp-encode-tree` and `rlp-encode-tree-list`, the right inverse theorems are proved by induction on `rlp-parse-tree` and `rlp-parse-tree-list`.

```
(defthm rlp-decode-tree-is-rlp-decodex-tree
  (and (iff (mv-nth 0 (rlp-decode-tree encoding))
            (mv-nth 0 (rlp-decodex-tree encoding)))
       (equal (mv-nth 1 (rlp-decode-tree encoding))
              (mv-nth 1 (rlp-decodex-tree encoding)))))
```

Figure 18: ACL2 theorem asserting the correctness of the executable tree decoder.

18. The proof follows easily from the theorem in Figure 18, the definition of `rlp-decodex-bytes` (which is in terms of `rlp-decodex-tree`), and a theorem (not shown), proved via a few hints, that expresses `rlp-decode-bytes` in terms of `rlp-decode-tree` (unlike the definition in Figure 10). The equivalence of `rlp-decodex-scalar` and `rlp-decode-scalar` is stated and proved similarly.

5 Related Work

RLP is formally defined in YP. Based on the ACL2 development described here, the author has contributed some improvements to that definition [20, Pull Request 700] [20, Pull Request 736] [20, Pull Request 739] [20, Pull Request 742] [20, Pull Request 745] [20, Pull Request 746], and has helped close the outstanding [20, Pull Request 648] and [20, Issue 116].

RLP is informally defined in WK. Prior to this work, the Python reference code for RLP decoding in WK accepted the quasi-encodings described in Section 4.2.1. Based on the ACL2 development described here, the author has contributed a fix to reject the quasi-encodings [18, Issue 688].

The KEVM [4] [19] includes executable specifications of RLP encoding and decoding. The decoding specification covers all encodings, while the encoding specification only covers byte arrays and some data types that are encoded like byte arrays. There appear to be no proofs of theorems stating that the specified encoding and decoding are mutual inverses.[5]

The Lem EVM [6] [5] includes a partial (apparently in progress) Isabelle/HOL specification of RLP encoding and decoding. The type of RLP trees is much like Figure 3; there is a complete specification of RLP encoding, but only a partial specification of RLP decoding.

There are several implementations of RLP, in libraries and Ethereum clients, written in mainstream programming languages. Some of these implementations used to accept or still accept the quasi-encodings described in Section 4.2.1, e.g. see [11, Issue 49] and [15, Issue 1639].

The Ethereum test suite [16] contains JSON-formatted tests for all Ethereum implementations, including tests for RLP encoding and decoding. Previously, this test suite included a few tests for rejecting some, but not all, of the five kinds of quasi-encodings described in Section 4.2.1. Based on the ACL2 development described here, the author has contributed additional tests to cover all the five kinds of quasi-encodings [16, Pull Request 612].

6 Future Work

Alternative, perhaps more efficient, implementations of the RLP parser and decoders in Section 4.2 could be written and verified, comparing their proof efforts and techniques with Section 4.3.

It could be investigated how to derive, via stepwise refinement, verified and efficient RLP parser and decoder implementations from the declarative specifications in Section 4.1, using the APT (Automated Program Transformations) toolkit [10]. This may require the development of additional, but more generally applicable, APT transformation tools.

[5] The KEVM is an evolving artifact. This paper refers to its version on the date indicated in the reference [19].

The RLP parser and decoders in Section 4.2 are given the complete purported encodings, or more, as input. However, practical implementations may read the input bytes as needed, e.g. from a socket. It could be investigated how to modify the specification, implementation, and proof to accommodate this approach. In this case, additional customizable length checks should be probably added, to thwart denial-of-service-style attacks consisting in supplying the first few bytes of very encodings, e.g. close to the 2^{64} limits.

The RLP parser and decoders in Section 4.2 process encodings completely, i.e. to the full depth of the RLP trees. However, practical implementations may process encodings up to a specified tree depth initially, then to lower depths as needed. It could be investigated how to extend the specification, implementation, and proof to accommodate this approach. These new parser and decoders will need to accept invalid encodings that they are valid up to the specified depths, requiring a corresponding weakening of their specification.

While RLP is an important component of Ethereum, clearly a lot more work remains to extend the development described in this paper to a complete Ethereum client. This remaining work is in progress [14, Path `books/kestrel/ethereums`] [13, Topic `ethereum`].

Acknowledgments

Thanks to Eric McCarthy for useful discussions on RLP.

References

[1] Jeremy Avigad and Richard Zach. The epsilon calculus. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2016 edition, 2016. `https://plato.stanford.edu/archives/sum2016/entries/epsilon-calculus/`.

[2] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979.

[3] David A. Greve, Matt Kaufmann, Panagiotis Manolios, J Strother Moore, Sandip Ray, José Luis Ruiz-Reina, Rob Sumners, Daron Vroon, and Matthew Wilding. Efficient execution in an automated reasoning environment. *Journal of Functional Programming*, 18(1):15–46, January 2008.

[4] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Roşu. KEVM: A complete semantics of the ethereum virtual machine. Technical report, University of Illinois Urbana-Champaign, August 2017. `http://hdl.handle.net/2142/97207`.

[5] Yoichi Hirai. Formalization of Ethereum Virtual Machine in Lem. `https://github.com/pirapira/eth-isabelle`.

[6] Yoichi Hirai. Defining the Ethereum Virtual Machine for interactive theorem provers. In *1st Workshop on Trusted Smart Contracts (WTCS)*, April 1997.

[7] Matt Kaufmann and J Strother Moore. The ACL2 theorem prover: Web site. `http://www.cs.utexas.edu/users/moore/acl2`.

[8] Matt Kaufmann and J Strother Moore. A precise description of the ACL2 logic. Technical report, Department of Computer Sciences, University of Texas at Austin, 1998.

[9] Kestrel Institute. ACL2 Ethereum project. `https://www.kestrel.edu/home/projects/ethereum/`.

[10] Kestrel Institute. APT (Automated Program Transformations). `http://www.kestrel.edu/home/projects/apt`.

[11] Parity Technologies. Collection of crates used in Parity projects. `https://github.com/paritytech/parity-common`.

[12] Sol Swords and Jared Davis. Fix your types. In *Proc. 13th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2015.

[13] The ACL2 Community. The ACL2 theorem prover and community books: Documentation. `http://www.cs.utexas.edu/~moore/acl2/manuals/current/manual`.

[14] The ACL2 Community. The ACL2 theorem prover and community books: Source code. `http://github.com/acl2/acl2`.

[15] The Ethereum Community. Aleth – Ethereum C++ client, tools, and libraries. `https://github.com/ethereum/aleth`.

[16] The Ethereum Community. The Ethereum test suite. `https://github.com/ethereum/tests`.

[17] The Ethereum Community. The Ethereum web site. `https://ethereum.org`.

[18] The Ethereum Community. The Ethereum wiki. `https://github.com/ethereum/wiki` – Version of July 1st, 2019.

[19] The KEVM Team. JelloPaper: Human readablee semantics of EVM in K. `https://jellopaper.org` – Version of July 1st, 2019.

[20] Gavin Wood. Ethereum: A secure decentralized generalised transaction ledger. `https://github.com/ethereum/yellowpaper` – Version of July 1st, 2019.