Ethereum's Recursive Length Prefix in ACL2

Alessandro Coglio
Kestrel Institute
`http://www.kestrel.edu`

Recursive Length Prefix (RLP) is used to encode a wide variety of data in Ethereum, including transactions. The work described in this paper provides a formal specification of RLP encoding and a verified implementation of RLP decoding, 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 [20], Recursive Length Prefix (RLP) [21, Page 'RLP'] [23, 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*, developed in the ACL2 theorem prover [11]. The development is available [18, Path `books/kestrel/ethereum/rlp`] and thoroughly documented [17, Topic `rlp`]. Some excerpts of the development shown in this paper are slightly simplified for brevity.

This work has led to improvements to the Ethereum Yellow Paper [23] and to the Ethereum Wiki [21], which are major components of the Ethereum documentation. It has also led to additions to the Ethereum test suite [19], 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 [12]. 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

Ethereum uses RLP to encode transactions, which may include smart contract code to run on the EVM. It also uses RLP to encode blocks, whose hashing plays a critical role in the blockchain paradigm. Thus, errors in RLP encoders and decoders may lead, among other problems, to unexpected smart contract code, or to incorrect block verification or mining.

RLP is specified informally in the Ethereum wiki ('WK' for short) [21, Page 'RLP'] and more formally in the Ethereum Yellow Paper ('YP' for short) [23, Appendix B].

RLP encodes 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 $[\ldots]$ denote leaf nodes and $\langle\ldots\rangle$ denote branching nodes; this tree is

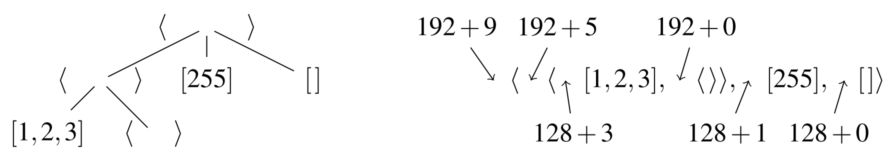

Figure 1: An example RLP tree and its encoding.

depicted in Figure 1 (left). The leaf tree [] with the empty sequence of bytes differs from the branching tree ⟨⟩ with no subtrees.[1]

WK calls the trees 'items', the branching trees 'lists', and the leaf trees '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 sequences of bytes. YP uses a more explicit tree terminology; it uses a mathematical notation $(\ldots,\ldots)$ for both leaf and branching nodes, as well as for flat sequences of bytes. This paper uses the notations $[\ldots]$ and $\langle\ldots\rangle$ shown earlier and in Figure 1.

RLP prescribes to encode non-negative integers by first representing them as base-256 big-endian byte arrays without leading zeros (not even a 0 byte for the integer 0, which is represented as the empty byte array), and then encoding those as any other leaf trees. Other than that, RLP does not prescribe encodings for any data types, delegating that to the "users" of RLP.

Leaf and branching trees 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.

Leaf trees $[\ldots]$ are encoded by encoding their byte arrays as follows. Singleton byte arrays whose only byte is below 128 are encoded as themselves, without extra bytes. 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 from 1 to 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 trees $\langle\ldots\rangle$ 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 from 1 to 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 trees have an additional shorter encoding, when they consist of single bytes below 128.

[1] In this paper, a 'leaf tree' is one that consists of just one leaf node (which is also necessarily the root), while a 'branching tree' is one that has (at least) a branching node as root.

```
(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 2: Formalization of RLP trees in ACL2.

3 RLP Encoding

3.1 Tree Structures

As explained in Section 2, RLP encodes trees (i.e. nested byte sequences) into flat byte sequences (i.e. byte arrays). These trees are formalized in Figure 2, using the FTY library [17, Topic `fty`].

The `fty::deftagsum` defines a tagged sum type (disjoint union) called `rlp-tree`. Leaf trees are tagged by `:leaf`; branching trees are tagged by `:branch`. A leaf tree has a single component called `bytes` of type `byte-list`, which consists of lists of natural numbers below 256 (its definition is not shown). A branching tree has a single component called `subtrees` of type `rlp-tree-list`, defined by the `fty::deflist` as consisting of lists of trees; `:elt-type` specifies the element type of the lists. The surrounding `fty::deftypes` introduces `rlp-tree` and `rlp-tree-list` as mutually recursive types, with `rlp-trees` as the name of the ensemble.

In more detail, the `fty::deftagsum` introduces a recognizer `rlp-treep`, a fixer `rlp-tree-fix`, constructors `rlp-tree-leaf` and `rlp-tree-branch` for leaf and branching trees, accessors `rlp-tree-leaf->bytes` and `rlp-tree-branch->subtrees` for leaf and branching trees, and several theorems about these functions, which are all guard-verified. The `fty::deflist` introduces a recognizer `rlp-tree-listp`, a fixer `rlp-tree-list-fix`, and theorems about these functions and existing list functions; no specific constructors or accessors are introduced by `fty::deflist`, since the generic ones for lists can be used.

In essence, Figure 2 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 in Figure 3.

The function `rlp-encode-bytes` formalizes the encoding of the byte arrays at the leaves of RLP trees, using the `define` enhancement of `defun` [17, Topic `define`], which supports type annotations for arguments (abbreviating guards) and results (abbreviating theorems). The function `rlp-encode-bytes` takes as input a list of bytes and returns as output a pair [17, Topic `mv`] consisting of a boolean error flag 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 `nil`, but irrelevant. Otherwise, the error flag is `nil` and the second component of the output is the encoding.

First, `rlp-encode-bytes` fixes the argument to be a list of bytes via the fixer `byte-list-fix` (a no-op under the guard), using the `b*` enhancement of `let*` [17, Topic `b*`]. 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) (mv nil (cons (+ 128 (len bytes)) bytes)))
          ((< (len bytes) (expt 2 64))
           (b* ((be (nat=>bebytes* (len bytes))))
             (mv nil (cons (+ 183 (len be)) (append be bytes)))))
          (t (mv t nil)))))

(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)
                      (mv nil (cons (+ 192 (len encoding)) encoding)))
                     ((< (len encoding) (expt 2 64))
                      (b* ((be (nat=>bebytes* (len encoding))))
                        (mv nil (cons (+ 247 (len be)) (append be 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 3: Formalization of RLP encoding in ACL2.

1. If the list consists of one byte and that byte is below 128, 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.
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 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. The library function `nat=>bebytes*` turns a natural number into a list of bytes as big-endian digits in base 256, without leading zeros.
4. Otherwise, the operation fails: 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 by 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 more convenient to return two results, so that each result always has the same type. The function `nat=>bebytes*`, mentioned above, formalizes the function `BE` in YP.

The function `rlp-encode-tree` formalizes the encoding of (both branching and leaf) trees. It takes as input a tree 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—the same output types as `rlp-encode-bytes`.

First, `rlp-encode-tree` performs a case analysis on the argument via the macro `rlp-tree-case`, generated by the `fty::deftagsum` in Figure 2. If `tree` is a leaf tree, `rlp-encode-bytes` is called on the list of bytes in the leaf; the variable `tree.bytes` is bound to `(rlp-tree-leaf->bytes tree)` by `rlp-tree-case`. If instead `tree` is a branching tree, `rlp-encode-tree-list` is called to encode each subtree and concatenate their encodings (more details below); the variable `tree.subtrees` is bound to `(rlp-tree-branch->subtrees tree)` by `rlp-tree-case`. The two results of `rlp-encode-tree-list` are simultaneously bound to `error?` and `encoding` via `b*`'s support for binding patterns such as `(mv ...)`. If `rlp-encode-tree-list` returns an error, `rlp-encode-tree` returns an error too, via `b*`'s early-exit construct `((when ...) ...)`: 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.
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` formalizes the encoding of a list 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, the result is the empty list of bytes `nil`. Otherwise, the first tree is encoded, the remaining list of trees is encoded, 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` [17, Topic `defines`] groups mutually recursive functions that are introduced via `define` (and also provides some enhancements over `mutual-recursion`), 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` 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` formalizes the function s in YP.

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

```
(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))))))
  :skolem-name rlp-tree-encoding-witness)
```

Figure 4: Formalization of valid RLP encodings in ACL2.

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

3.3 Valid Encodings

Valid encodings are formalized in Figure 4, using the `define-sk` enhancement [17, Topic `define-sk`] of `defun-sk` [17, Topic `defun-sk`], which provides conveniences similar to `define`.

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`. This predicate has a declarative, non-executable definition.

3.4 Decodability Properties

RLP encodings are decodable, i.e. trees can be recovered from their encodings—a basic requirement for any encoding method. This is expressed by the theorems in Figure 5: (i) the RLP encoding function is injective, i.e. no two distinct trees have the same encoding; and (ii) the RLP encoding function is prefix-unambiguous, i.e. no valid tree 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.

Typically, the injectivity of a function f is stated as $[x \neq y \Longrightarrow f(x) \neq f(y)]$, with x and y universally quantified, or equivalently as $[f(x) = f(y) \Longrightarrow x = y]$. Because $[x = y \Longrightarrow f(x) = f(y)]$ is always trivially true, injectivity can be equivalently stated as $[f(x) = f(y) \Longleftrightarrow x = y]$, which is usable as a rewrite rule. If f operates on values of a type (predicate) τ, its injectivity restricted to values of that type can be stated as $[\tau(x) \wedge \tau(y) \Longrightarrow (f(x) = f(y) \Longleftrightarrow x = y)]$; if f implicitly or explicitly fixes values outside τ via a fixer ϕ for τ, injectivity can be equivalently stated as $[f(x) = f(y) \Longleftrightarrow \phi(x) = \phi(y)]$,[2] which is generally preferable as a rewrite rule because it has no hypotheses. This is the formulation in Figure 5, applied to the `encoding` results and with the necessary hypotheses that the `error?` results are `nil`; the results are obtained via `mv-nth` [17, Topic `mv-nth`].

First, a theorem `rlp-encode-bytes-injective`, not shown here but analogous to `rlp-encode-tree-injective` (with `rlp-encode-tree` and `rlp-tree-fix` replaced with `rlp-encode-bytes` and

[2]This is obtained by replacing x and y with $\phi(x)$ and $\phi(y)$ in $[\tau(x) \wedge \tau(y) \Longrightarrow (f(x) = f(y) \Longleftrightarrow x = y)]$, and using the fact that $\forall z.\tau(\phi(z))$ (i.e. ϕ is a fixer for τ) and $\forall z.f(\phi(z)) = f(z)$ (i.e. f fixes its argument). Conversely, assuming $\tau(x)$ and $\tau(y)$, $[f(x) = f(y) \Longleftrightarrow \phi(x) = \phi(y)]$ reduces to $[f(x) = f(y) \Longleftrightarrow x = y]$ because $\forall z.\tau(z) \Longrightarrow \phi(z) = z$ (i.e. ϕ is identity over τ).

```
(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-tree-unamb-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))))))
```

Figure 5: ACL2 theorems asserting the decodability of RLP encodings.

`byte-list-fix`), is proved by first proving a variant lemma (not shown) with `byte-listp` hypotheses on `x` and `y` and without `byte-list-fix` (i.e. the form $[\tau(x) \wedge \tau(y) \Longrightarrow (f(x) = f(y) \Longleftrightarrow x = y)]$ above). 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 [17, 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.

Then, 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 (i.e. the form $[\tau(x) \wedge \tau(y) \Longrightarrow (f(x) = f(y) \Longleftrightarrow x = y)]$ above). Since `rlp-encode-tree` is mutually recursive with `rlp-encode-tree-list`, 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, 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 others make use of some fairly specific lemmas (not shown); see the documentation [17, Topic `rlp`] for details. The base case, in which `x` or `y` is a leaf tree, makes use of `rlp-encode-bytes-injective`, the injectivity theorem for `rlp-encode-bytes`.

The injectivity of the RLP encoding functions could be alternatively proved by defining RLP decoding functions, proving that the latter are left inverses of the former, and using the inverse property to prove injectivity.[3] 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 my opinion.

The prefix-unambiguity theorem in Figure 5 says that if a valid encoding is a prefix of another valid encoding, then the two encodings are equal; similarly to the injectivity theorem, this theorem also states the obvious converse implication, making the theorem a more useful rewrite rule. The library function `prefixp` tests whether the first argument is a (not necessarily strict) prefix of the second argument. Thus,

[3]If f has a left inverse g, then given $f(x) = f(y)$, and 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 f is injective.

```
(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))))) ; 2nd result irrelevant
```

Figure 6: Declarative definition of RLP decoding in ACL2.

the theorem prohibits a valid encoding from being a strict prefix of another valid encoding.

The theorem `rlp-encode-tree-unamb-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 equal, 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.

4 RLP Decoding

4.1 Declarative Specification

The RLP decoding of trees from their encodings is formalized in Figure 6.

The function `rlp-decode-tree` 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 4 (more details on this below). Otherwise, the error flag is `t` and the second component of the output is irrelevant.

The existentially quantified function `rlp-tree-encoding-p` is defined in terms of the witness function `rlp-tree-encoding-witness` (under the hood [17, Topic `defun-sk`]; see `:skolem-name` in Figure 4), which is axiomatized, via the matrix of the quantification, to be a right inverse of the encoding function `rlp-encode-tree`: if `encoding` is a valid tree encoding, then `(rlp-tree-encoding-witness encoding)` returns `tree` of type `rlp-treep` such that `(rlp-encode-tree tree)` returns `(mv nil encoding)`, i.e. no error and the original 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 7: if `encoding` is a valid tree encoding (modulo `byte-list-fix`), then `rlp-decode-tree` succeeds and returns `tree`, and `rlp-encode-tree` succeeds on `tree` and returns the original encoding. This 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, `rlp-decode-tree` is also a left inverse of `rlp-encode-tree`,[4] as stated by the theorem `rlp-decode-tree-of-rlp-encode-tree` in Figure 7: if `tree` is (modulo `rlp-tree-fix`) a tree, and `rlp-encode-tree` succeeds and returns `encoding`, then `rlp-decode-tree` succeeds on `encoding` and returns the original tree. This is proved via a few

[4] 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.

```
(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 7: ACL2 theorems asserting that RLP encoding and decoding are mutual inverses.

hints (not shown) to instantiate (replacing `encoding` with `(mv-nth 1 (rlp-encode-tree tree))`) and use the right inverse theorem `rlp-encode-tree-of-rlp-decode-tree`, while the injectivity theorem in Figure 5 is automatically used as a rewrite rule.

Figure 6 defines the RLP decoding function as inverse of the RLP encoding function. This is a declarative, non-executable definition. YP does not explicitly define any RLP decoding function, but, clearly, it implicitly defines it as inverse of the encoding function: this implicit definition is formalized in Figure 6. WK provides Python reference code for RLP decoding, but the definition in Figure 6 is more abstract and manifestly correct.

4.2 Executable Implementation

The function `rlp-decode-tree` in Figure 6 can be regarded as a high-level specification of RLP decoding, which can be implemented by an equivalent executable function.

The executable decoding function is defined in terms of the executable RLP parser shown in Figure 8. This function is mutually recursive with the one in Figure 9. Similarly to Figure 3, these two functions are surrounded by `defines` (not shown).

The function `rlp-parse-tree` 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 one if parsing fails), and the remaining bytes after the parsed encoding (`nil` if parsing fails[5]). The parser stops as soon as a tree is successfully decoded (because of theorem `rlp-encode-tree-unamb-prefix` in Figure 5, 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` takes as input a 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 exact errors are not shown in Figure 8, replaced with ellipses.

[5]Returning `nil` as third result when parsing fail may seem an odd choice, compared to returning the remaining bytes whose parsing caused the error, which could convey information about the error. However, this (and more) information is already included in the error result, whose recognizer `rlp-error-p` is mentioned a few sentences later (but not explained in detail).

```
(define rlp-parse-tree ((encoding byte-listp))
  :returns (mv (error? maybe-rlp-error-p) (tree rlp-treep) (rest byte-listp))
  (b* ((encoding (byte-list-fix encoding))
       ((when (endp encoding)) ...) ; error
       ((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)) ...) ; error
             (bytes (take len encoding))
             ((when (and (= len 1) (< (car bytes) 128))) ...)) ; error
          (mv nil (rlp-tree-leaf bytes) (nthcdr len encoding))))
       ((when (< first 192))
        (b* ((lenlen (- first 183))
             ((when (< (len encoding) lenlen)) ...) ; error
             (len-bytes (take lenlen encoding))
             ((unless (equal (trim-bendian* len-bytes) len-bytes)) ...) ; error
             (encoding (nthcdr lenlen encoding))
             (len (bebytes=>nat len-bytes))
             ((when (<= len 55)) ...) ; error
             ((when (< (len encoding) len)) ...) ; error
             (bytes (take len encoding))
             (encoding (nthcdr len encoding)))
          (mv nil (rlp-tree-leaf bytes) encoding)))
       ((when (<= first 247))
        (b* ((len (- first 192))
             ((when (< (len encoding) len)) ...) ; error
             (subencoding (take len encoding))
             (encoding (nthcdr len encoding))
             ((mv error? subtrees) (rlp-parse-tree-list subencoding))
             ((when error?) ...)) ; error
          (mv nil (rlp-tree-branch subtrees) encoding)))
       (lenlen (- first 247))
       ((when (< (len encoding) lenlen)) ...) ; error
       (len-bytes (take lenlen encoding))
       ((unless (equal (trim-bendian* len-bytes) len-bytes)) ...) ; error
       (encoding (nthcdr lenlen encoding))
       (len (bebytes=>nat len-bytes))
       ((when (<= len 55)) ...) ; error
       ((when (< (len encoding) len)) ...) ; error
       (subencoding (take len encoding))
       (encoding (nthcdr len encoding))
       ((mv error? subtrees) (rlp-parse-tree-list subencoding))
       ((when error?) ...)) ; error
    (mv nil (rlp-tree-branch subtrees) encoding)))
```

Figure 8: Executable parser of RLP encodings in ACL2 (Part 1).

```
(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?) ...) ; error
       ((unless (mbt (< (len encoding1) (len encoding)))) ...) ; error
       ((mv error? trees) (rlp-parse-tree-list encoding1))
       ((when error?) ...)) ; error
    (mv nil (cons tree trees))))
```

Figure 9: Executable parser of RLP encodings in ACL2 (Part 2).

The parser operates as follows (cf. the encoding functions in Figure 3):

1. It fixes the `encoding` input to be a list of bytes (a no-op under the guard).
2. If the input list of bytes is empty, an error is returned: RLP encodings are never empty.
3. Otherwise, the `encoding` input is matched to the `(cons first encoding)` pattern, binding the variable `first` to the first byte of the encoding and the (new) 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, and so the byte is returned as a leaf tree.
5. If the first byte is in the range 128–183, the encoding must be of a byte array with length below 56. 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. An error is also returned if there is just one byte below 128; see Section 4.2.1.
6. If the first byte of the encoding is in the range 184–191, 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. 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 3, 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. If the first byte of the encoding is in the range 192–247, the encoding must be of a branching tree whose subtrees have a total encoded length below 56. 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 returned by `rlp-parse-tree`.
8. If the first byte of the encoding is in the range 248–255, 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. The processing of `lenlen` and `len` is analogous to the case in which the first byte is in the range 184–191 (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 in the range 192–247, `rlp-parse-tree-list` is called with the exact number of bytes to parse, any error from that call is propagated, and ultimately (if no errors occur) the decoded branching tree is returned.

```
(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?) ...) ; error
       ((when (consp rest)) ...)) ; error
    (mv nil tree)))
```

Figure 10: Executable definition of RLP decoding in ACL2.

The function `rlp-parse-tree-list` in Figure 9 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`. When `rlp-parse-tree` calls `rlp-parse-tree-list`, the first component of the measure decreases. When `rlp-parse-tree-list` calls `rlp-parse-tree`, the first component is unchanged but the second decreases. When `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))`, which gets the function definition accepted by ACL2. The `mbt` around the test avoids executing the test at run time [17, Topic `mbt`].

The guard verification proofs, which involve the truth of the aforementioned `mbt` test, 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 theorem (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 function for trees is defined in Figure 10; the '`x`' in its name stands for 'executable', as opposed to the declaratively defined decoding function in Figure 6. This decoding function takes as input a list of bytes, purported to be a complete encoding with no extra bytes,[6] and returns as output a pair consisting of an error indication (`nil` if decoding is successful) and a tree—whose value is irrelevant if the first result is not `nil`. A tree is decoded by calling the parser and ensuring that there are no remaining bytes.

4.2.1 Rejection of Invalid Quasi-Encodings

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

[6]This executable decoding function is meaningful as implementation of the inverse of the encoding function declaratively defined in Figure 6. It 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 in Figure 8 can be used instead.

```
(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-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))))))
```

Figure 11: ACL2 theorems asserting that RLP encoding and parsing are mutual inverses.

- $[129,x]$ 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.
- $[184,l,x,\ldots]$ with $0 < l < 56$: 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.
- $[248,l,x,\ldots]$ with $0 < l < 56$: 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 predicate in Figure 4. While they “could” be encodings in the sense mentioned above, they are “non-optimal”: shorter valid encodings exist.

When implementing RLP decoding, the rejection of these quasi-encodings may be easily overlooked. My initial writing of the parser failed to reject these quasi-encodings; I discovered the problem when failing to prove the right inverse property in Section 4.3. 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 not in the image of the RLP encoding function. Also see `https://gitter.im/ethereum/research/archives/2016/07/30`; search for ‘consensus rule’. A possible problematic scenario is a database that uses RLP encodings as keys: if different encodings for the same item were accepted, the item could be stored multiple times in the database.

4.3 Verification of Correctness

The first step toward proving that the executable definition in Figure 10 is equivalent to the declarative definition in Figure 6 is to prove that the tree parsing and encoding functions are mutual inverses, which is expressed by the theorems in Figure 11.

The first theorem in Figure 11 says that `rlp-parse-tree` is a left inverse of `rlp-encode-tree` over the encodable trees, i.e. the parser recognizes and reconstructs *all* valid encodings of trees. More in detail, it says 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 another theorem (not shown) that 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 [17, 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 [17, Topic `rlp`] for details. The proof also makes use of a previously proved more general rewrite rule (not shown) saying 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 second theorem in Figure 11 says that `rlp-parse-tree` is a right inverse of `rlp-encode-tree` over the valid tree encodings, i.e. 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 says that if `rlp-parse-tree` succeeds, then `rlp-encode-tree` succeeds 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 another theorem (not shown) about `rlp-parse-tree-list` and `rlp-encode-tree-list`, via `make-flag`; the proof, by induction on `rlp-parse-tree` and `rlp-parse-tree-list`[7], is automatic 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 10 is both a left and a right inverse of `rlp-encode-tree` easily follows from the theorems in Figure 11. This fact is asserted by two theorems (not shown) that are written as in Figure 7 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 11 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 second theorem in Figure 11, since the `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 12. 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 12 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:

[7]In ACL2, as in NQHTM [4, Chapt. 15], induction is applicable 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 12: ACL2 theorem asserting the correctness of the executable RLP decoder.

- The first conjunct in Figure 12 is proved via a few hints, using the aforementioned lemma and the definition of `rlp-decode`.
- The second conjunct in Figure 12 is proved via a few hints, from `rlp-encode-tree-injective` and the right inverse properties of `rlp-decode-tree` and `rlp-decodex-tree`.[8]

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

5 Related Work and Impact

RLP is formally defined in YP.[9] Based on the ACL2 development, I contributed some improvements to that definition (see Pull Requests 700, 736, 739, 742, 745, and 746 at `https://github.com/ethereum/yellowpaper`) and helped close some outstanding items (see Pull Request 648 and Issue 116 at `https://github.com/ethereum/yellowpaper`).

RLP is informally defined in WK. Previously, the Python reference code for RLP decoding in WK accepted the quasi-encodings described in Section 4.2.1. Based on the ACL2 development, I contributed a fix to reject those quasi-encodings (see Issue 688 at `https://github.com/ethereum/wiki`).

The KEVM [7] [22] includes executable specifications of RLP encoding and decoding.[10] 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 are no theorems stating that the specified encoding and decoding are mutual inverses. The decoding specification appears to accept some of the quasi-encodings described in Section 4.2.1 (see Issue 413 at `https://github.com/kframework/evm-semantics`).

The Lem EVM [9] [8] includes a partial (apparently in progress) Isabelle/HOL specification of RLP encoding and decoding.[11] The type of RLP trees is much like Figure 2. There is a complete specification of RLP encoding, but only a partial specification of RLP decoding. There are no theorems stating that the specified encoding and decoding are mutual inverses.

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 Issue 1639 at `https://github.com/ethereum/aleth` and Issue 49 at `https://github.com/paritytech/parity-common`).

The Ethereum test suite [19] 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, I contributed additional tests to cover all the five kinds of quasi-encodings (see Pull Request

[8]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 $g(x) = h(x)$.

[9]Using "pencil-and-paper" mathematical notation, not in a theorem prover or similar tool.

[10]The KEVM is an evolving artifact. The following assertions are current at the time of this writing.

[11]The Lem EVM is an evolving artifact. The following assertions are current at the time of this writing.

612 at `https://github.com/ethereum/tests`).

There is extensive work on formal verification and analysis of (Ethereum and other) smart contracts, for which just some references are provided here [1] [2] [3] [5] [6] [7] [9] [10] [13] [14] [15] [16]. That work is complementary to the ACL2 RLP development, whose focus is (a component of) the platform that runs smart contracts. However, as mentioned in Section 1, this platform-focused work can support smart contract verification, by providing the underlying formal semantics of smart contracts.

6 Future Work

Alternative, perhaps more efficient, implementations of the RLP parser and decoder 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 specification in Section 4.1, using the APT (Automated Program Transformations) toolkit [17, Topic `APT`]. This may require the development of additional, but more generally applicable, APT transformation tools.

The RLP parser and decoder 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 extend 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 long encodings, e.g. close to the 2^{64} limits.

The RLP parser and decoder 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 decoder will need to accept invalid encodings that are however 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 [18, Path `books/kestrel/ethereum`] [17, Topic `ethereum`].

Acknowledgments

- The Ethereum Foundation for supporting this work.
- Eric McCarthy for useful discussions on RLP and for valuable comments on the initial draft of this paper, including suggesting the RLP database scenario mentioned at the end of Section 4.2.1.
- The anonymous reviewers for useful suggestions.

References

[1] Sidney Amani, Myriam Bégel, Maksym Bortin & Mark Staples (2018): *Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL.* In: *Proc. 7th International Conference on Certified Programs and Proofs (CPP)*, doi:10.1145/3167084.

[2] Danil Annenkov & Bas Spitters (2019): *Towards a Smart Contract Verification Framework in Coq.* In: *Proc. 1st Workshop on Formal Methods for Blockchains (FMBC).*

[3] Karthukeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy & Santiago Zanella-Béguelin (2016): *Formal Verification of Smart Contracts*. In: *Proc. 16th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*, pp. 91–96, doi:10.1145/2993600.2993611.

[4] Robert S. Boyer & J Strother Moore (1979): *A Computational Logic*. Academic Press, doi:10.1016/C2013-0-10411-4.

[5] Ting Chen, Xiaoqi Li, Xiapu Luo & Xiaosong Zhang (2017): *Under-Optimized Smart Contracts Devour Your Money*. In: *Proc. 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 442–446, doi:10.1109/SANER.2017.7884650.

[6] Sylvain Conchon, Alexandrina Korneva & Fatiha Zaïdi (2019): *Verifying Smart Contracts with Cubicle*. In: *Proc. 1st Workshop on Formal Methods for Blockchains (FMBC)*.

[7] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth & Grigore Roşu (2017): *KEVM: A Complete Semantics of the Ethereum Virtual Machine*. Technical Report, University of Illinois Urbana-Champaign. `http://hdl.handle.net/2142/97207`.

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

[9] Yoichi Hirai (2017): *Defining the Ethereum Virtual Machine for Interactive Theorem Provers*. In: *Proc. 1st Workshop on Trusted Smart Contracts (WTSC)*, LNCS, doi:10.1007/978-3-319-70278-0_33.

[10] Sukrit Kalra, Seep Goel, Mohan Dhawan & Subodh Sharma (2018): *ZEUS: Analyzing Safety of Smart Contracts*. In: *Proc. 25th Annual Network and Distributed System Security Symposium (NDSS)*, doi:10.14722/ndss.2018.23082.

[11] Matt Kaufmann & J Strother Moore: *The ACL2 Theorem Prover: Web Site*. `http://www.cs.utexas.edu/users/moore/acl2`.

[12] Kestrel Institute: *ACL2 Ethereum Project*. `https://www.kestrel.edu/home/projects/ethereum/`.

[13] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena & Aquinas Hobor (2016): *Making Smart Contracts Smarter*. In: *Proc. 23rd Conference on Computer and Communication Security (CCS)*, pp. 254–269, doi:10.1145/2976749.2978309.

[14] Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi & Natasha Sharygina (2018): *Computing Exact Worst-Case Gas Consumption for Smart Contracts*. In: *Proc. 8th International Symposium on Leveraging Applications of Formal Methods (ISoLA), LNCS* 11247, pp. 450–465, doi:10.1007/978-3-030-03427-6_33.

[15] Zeinab Nehaï & François Bobot (2019): *Deductive Proof of Industrial Smart Contracts Using Why3*. In: *Proc. 1st Workshop on Formal Methods for Blockchains (FMBC)*.

[16] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena & Aquinas Hobor (2018): *Finding the Greedy, Prodigal, and Suicidal Contracts at Scale*. In: *Proc. 34th Annual Computer Security Applications Conference (ACSAC)*, pp. 653–663.

[17] The ACL2 Community: *The ACL2 Theorem Prover and Community Books: Documentation*. `http://www.cs.utexas.edu/~moore/acl2/manuals/current/manual`.

[18] The ACL2 Community: *The ACL2 Theorem Prover and Community Books: Source Code*. `http://github.com/acl2/acl2`.

[19] The Ethereum Community: *The Ethereum Test Suite*. `https://github.com/ethereum/tests`.

[20] The Ethereum Community: *The Ethereum Web Site*. `https://ethereum.org`.

[21] The Ethereum Community: *The Ethereum Wiki*. `https://github.com/ethereum/wiki`.

[22] The KEVM Team: *JelloPaper: Human Readable Semantics of EVM in K*. `https://jellopaper.org`.

[23] Gavin Wood: *Ethereum: A Secure Decentralized Generalised Transaction Ledger*. `https://github.com/ethereum/yellowpaper`.

Appendix: Background on ACL2

This Appendix is not part of the published paper on EPTCS, but is included in the web version of the paper to make it more self-contained. Without this Appendix, the two documents are the same.

Logic

ACL2 is a general-purpose interactive theorem prover.

It is based on an untyped first-order logic of total functions that is an extension of a purely functional subset of Common Lisp.[12]

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`.

Syntax

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.

Names of variables, functions, theorems, and other entities are Lisp symbols, which are partitioned into packages. The canonical way to denote a symbol is `pkg::sym`, where `pkg` is the package name and `sym` is the symbol name. However, if `pkg` is the current package (as declared at the top of the file where the symbol reference occurs), `sym` suffices to denote the symbol; examples are `x`, `y`, `+`, `*`, and `f` above.

A package `pkg2` may import a symbol `pkg::sym`, in which case `pkg2::sym` and `pkg::sym` denote the same symbol. In particular, `sym` denotes that symbol when the current package is `pkg` or `pkg2`.

Symbols in the built-in `keyword` package are written as `:sym`. These are often used as tags that look the same in any current package.

Macros

ACL2's Lisp-like macro mechanism provides the ability to extend the language with new constructs defined in terms of existing constructs.

Examples are `cond`, `b*`, and `defun-sk`, described below. Other examples used in this paper are `define`, `define-sk`, `fty::deftagsum`, and `fty::deflist`.

Conditionals

The basic conditional `(if a b c)` is non-strict: in execution, first `a` is evaluated, then either `b` or `c` is evaluated (never both), based on whether `a` evaluates to non-`nil` or to `nil`.

The generalized conditional `(cond ((`c_1 e_1`) ... (`c_n e_n`)))` evaluates each c_i in sequence until one returns non-`nil`, in which case the value of the corresponding e_i is returned as result of the `cond`; if every c_i evaluates to `nil`, the result of the `cond` is `nil`. It is common for c_n to be `t`, for 'otherwise'. The conditional `cond` is a macro defined as a nest of `if`s, from which it inherits non-strictness.

[12]Matt Kaufmann and J Strother Moore, "A Precise Description of the ACL2 Logic", Technical Report, Department of Computer Sciences, University of Texas at Austin, April 1998, `https://www.cs.utexas.edu/users/moore/publications/km97a.pdf`.

Binders

The basic binder `(let ((`x_1 e_1`) ... (`x_n e_n`))` e`)` simultaneously binds each variable x_i (which must be distinct from the others) to the value of e_i and then returns the value of e.

The macro `b*` is a more general binder, which is defined in terms of `let`s and `if`s. In its simple form `(b* ((`x_1 e_1`) ... (`x_n e_n`))` e`)`, it is a sequential `let` construct: the variable x_1 is bound to the value of e_1, then the variable x_2 (which may be the same as x_1) is bound to e_2 (which may reference x_1), and so on; the final result is the value of e. In its general form `(b* ((`b_1 e_1`) ... (`b_n e_n`))` e`)`, each b_i may be a pattern that binds multiple variables simultaneously, or an early-exit condition of the form `(when` c_i`)` or `(unless` c_i`)` for immediately returning the value of e_i if c_i is non-`nil` (for `when`) or `nil` (for `unless`) and no early-exit b_j with $j < i$ applies.

Types, Recognizers, and Fixers

Since the ACL2 logic is untyped, in ACL2 'type' denotes just any subset of the universe of values.

A type is normally characterized by a 'recognizer' (in Lisp terminology), i.e. a predicate that returns `t` (or non-`nil`) exactly on the values of the type. For example, `natp` is the recognizer of natural numbers. As another example, `bytep` is the recognizer of (8-bit) bytes.[13]

A type may also have a fixing function, or 'fixer' for short, i.e. a function that "fixes" values outside the type to be values inside the type, and leaves values already inside the type unchanged. For example, `nfix` maps every natural number to itself and all other values to the natural number 0. As another example, `byte-fix` maps every byte to itself and all other values to the byte with all 0 bits.

The FTY library [17, Topic `fty`] provides macros to associate recognizers and fixers to type names, and to build structured, possibly recursive, types (records, unions, etc.) out of other types. For example, the recognizer `natp` and the fixer `nfix` are associated to the type name `nat`. As another example, the recognizer `bytep` and the fixer `byte-fix` are associated to the type name `byte`. When the FTY macros are used to build types out of other types, they generate not only recognizers and fixers, but also constructors and accessors, as well as many boilerplate theorems that relate all these functions.

Proofs and Automation

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 [4], 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 (at all, or none besides the ones explicitly mentioned). However, it may, and generally does, involve more generic library theorems, i.e. the theorem is typically not proved "from scratch".

Functions

Function definitions are introduced via `defun`.

For example, the factorial function can be defined as

[13]In ACL2, as in Lisp, it is common to end predicate names with `p`.

```
(defun fact (n)
  (if (zp n) 1 (* n (fact (- n 1)))))
```

where `zp` tests if `n` is 0 or not a natural number, `*` is multiplication, and `-` is subtraction. ACL2 integers are unbounded,[14] as in Lisp.

Via the `zp` test, `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. This coercion may be implicit, as in `fact`, or explicit by calling a fixer for the type.

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.

Theorems

Theorems are introduced via `defthm`.

For example, a theorem saying that `fact` is (non-strictly) above its argument can be stated as

```
(defthm above
  (implies (natp n) (>= (fact n) n)))
```

where `natp` tests if `n` is a natural number (as also explained earlier), `>=` is the greater-than-or-equal-to relation on numbers, and `implies` is logical implication.

If a standard arithmetic library [18, Path `books/arithmetic`] is loaded, ACL2 proves this theorem automatically, finding and using an appropriate induction rule—the one derived from the recursive definition of `fact`, in this case.

Indefinite Descriptions

Besides the discouraged ability to introduce arbitrary axioms, ACL2 provides some logical-consistency-preserving mechanisms to axiomatize new functions, such as indefinite description functions.

For example, a function constrained to be strictly below `fact` can be axiomatized as

```
(defchoose below (b) (n)
  (and (natp b) (< b (fact n))))
```

where `b` is the variable bound by the indefinite description, `<` is the less-than relation on numbers, and `and` is logical conjunction. 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.

In formal logic, an indefinite description is typically written as $\varepsilon x. P(x)$ and read as 'choose an x such that $P(x)$'. It denotes some unspecified x that satisfies P if at least one exists; if none exists, $\varepsilon x. P(x)$ is unspecified, but is known not to satisfy P. In ACL2, `defchoose` introduces a named, possibly parameterized, indefinite description; in more conventional notation, the function `below` could be written as $below(n) = \varepsilon b.\ [b \in \mathbb{N} \wedge b < n!]$.

Quantification

Despite the lack of built-in quantification in the logic, functions with top-level quantifiers can be introduced via macros.

For instance, the existence of a value strictly between `fact` and `below` can be expressed by a predicate defined as

[14] Subject to memory limitations during execution.

```
(defun-sk between (n)
  (exists (m) (and (natp m) (< (below n) m) (< m (fact n)))))
```

where `defun-sk` is a macro defined in terms of `defchoose` and `defun`, following a well-known construction:[15] $\exists x.\,P(x) \overset{\text{def}}{=} P(\varepsilon x.\,P(x))$, where the indefinite description $\varepsilon x.\,P(x)$ is explained above. A `defun-sk` introduces the indefinite description as a 'witness' function via `defchoose` (called `between-witness` for `between`, but not appearing explicitly in the definition of `between` above) and defines the function (i.e. `between`) via `defun` with a body obtained by instantiating, in the matrix of the quantification (i.e. `(and (natp m) ...)` for `between`), the quantified variable (i.e. `m` for `between`) with the witness function applied to the arguments (i.e. `(between-witness n)` for `between`). The default name of the witness function if obtained by concatenating the function name with `-witness`; the name of the witness function can be customized via the `:skolem-name` option (not shown in the definition of `between` above).

Guards

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.

For example, a guarded version of `fact` can be introduced as

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

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 context 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.

Lists

Finite sequences are represented as lists in ACL2. List operations include the following:

- `nil` denotes the empty list. (It also denotes logical falsehood, as explained earlier.)
- `(cons a x)` adds the element `a` in front of the list `x`.
- `(list a1 ... an)` returns the list with the given elements, in that order; the list has length `n`.
- `(car x)` returns the first element of the (non-empty) list `x`.
- `(cdr x)` removes the first element from the (non-empty) list `x`.
- `(len x)` returns the length (i.e. number of elements) of the list `x`.
- `(endp x)` returns `t` if the list `x` is empty, otherwise it returns `nil`.
- `(nth n x)` returns the element at position `n` (0-based) in the list `x` of length greater than `n`.
- `(take n x)` returns the list of the first `n` elements of the list `x` of length at least `n`.
- `(nthcdr n x)` removes the first `n` elements from the list `x` of length at least `n`.

[15]Jeremy Avigad and Richard Zach, "The Epsilon Calculus", *The Stanford Encyclopedia of Philosophy*, Edward N. Zalta (editor), Summer 2016 Edition, Metaphysics Research Lab, Stanford University, `https://plato.stanford.edu/archives/sum2016/entries/epsilon-calculus/`.

Typed Lists

Lists with elements of the same type generally have recognizers called `<type>-listp`, fixers called `<type>-list-fix`, and (possibly) FTY types called `<type>-list`.

For example, lists of natural numbers have recognizer `nat-listp`, fixer `nat-list-fix`, and type `nat-list`. As another example, lists of bytes have recognizer `byte-listp`, fixer `byte-list-fix`, and type `byte-list`.

Multiple Results

Functions that return multiple results may do so by returning a list. Using `mv` (for 'multiple value') to construct the list is logically equivalent to using `list` (see above), but is more efficient in execution. It avoids constructing, and thus allocating memory for, the list; instead, the values are kept and handled separately. ACL2 enforces certain restrictions on the use of `mv` in order to maintain this separation.

For example, a function to return both quotient and remainder of a division of natural numbers can be defined as

```
(defun quorem (a b)
  (declare (xargs :guard (and (natp a) (natp b) (not (equal b 0)))))
  (mv (truncate a b) (rem a b)))
```

When accessing components of multiple results in logical formulas, `mv-nth` is used, which is logically equivalent to `nth` (see above). This is because the expansion of `mv` binders (e.g. in `b*`) produces `mv-nth` calls.

Execution

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 Lisp functions cannot cause run-time errors; when guards are verified at compile time, the checks for their satisfaction are omitted at run time.

Expressions, including defining bodies of functions, may have the form `(mbe :logic a :exec b)`, where `mbe` means 'must be equal'. Logically, the expression is equivalent to `a`, but in execution it is equivalent to `b`; as part of guard verification, it must be proved that `a` and `b` are equal under the guards. Typically, `b` is more efficient, while `a` is logically simpler and handles values outside the guards.

The expression `(mbt a)`, where `mbt` means 'must be true', abbreviates `(mbe :logic a :exec t)`. It logically performs tests that are satisfied under the guards, without executing them at run time.

Some fixers have guards requiring the argument, say `x`, to satisfy the corresponding recognizer, and have a defining body of the form `(mbe :logic ... :exec x)`. In this case, fixing is a no-op during execution, but it logically coerces values outside the type to be treated as values inside the type. An example is the fixer `byte-list-fix`, used in this paper.