A Mechanical Proof of the Cook-Levin Theorem

Ruben Gamboa and John Cowles

University of Wyoming
Department of Computer Science
Laramie, WY 82071

Abstract. As is the case with many theorems in complexity theory, typical proofs of the celebrated Cook-Levin theorem showing the NP-completeness of satisfiability are based on a clever construction. The Cook-Levin theorem is proved by carefully translating a possible computation of a Turing machine into a boolean expression. As the boolean expression is built, it is "obvious" that it can be satisfied if and only if the computation corresponds to a valid and accepting computation of the Turing machine. The details of the argument that the translation works as advertised are usually glossed over; it is the translation itself that is discussed. In this paper, we present a formal proof of the correctness of the translation. The proof is verified with the theorem prover ACL2.

1 Introduction

This paper presents a mechanical proof of the Cook-Levin theorem. A number of reasons led us to this investigation. The Cook-Levin theorem is the central theorem in NP-completeness theory, as it was the first to demonstrate the existence of an NP-complete problem, namely satisfiability [3, 7]. Moreover, having taught several undergraduate and introductory graduate courses on the theory of computer science, one of the authors has always been uncomfortable with the format of most proofs in the field. Many such proofs hinge on an algorithm that translates an instance of a problem from one domain to another. The transformation can be quite intricate, but seldom is its correctness actually proved. More often the correctness of the transformation is left as being obvious. Since the correctness proof is almost certainly tedious, we see it as an opportunity for formal approaches to proof. Other efforts have used the Boyer-Moore theorem prover to prove similar results, such as [1, 2, 8, 9].

We chose to use the theorem prover ACL2 for our formalization. ACL2 is a theorem prover over a first-order logic of total functions, with minimal support for quantifiers. The logic of ACL2 is based on the applicative subset of Common Lisp. Its basic structure and inference mechanisms are taken from its predecessor, the Boyer-Moore theorem prover. In fact, ACL2 arose out of a desire to enhance the Boyer-Moore prover to make it more suitable for industrial use, and in that respect it has succeeded marvelously. For example, it has been used to verify aspects of the floating-point units of microprocessors at AMD and IBM, and it is also in use in the simulation and verification of microprocessors at Rockwell-Collins. ACL2 has also been used in many other verification projects, ranging from the algebra of polynomials to properties of the Java virtual machine [5, 6].

This paper does not assume familiarity with ACL2. However, we will use regular ACL2 syntax to introduce ACL2 definitions and theorems. We assume, therefore, that the reader is comfortable with Lisp notation.

The remainder of the paper is organized as follows. In Sect. 2 we present an informal proof of the Cook-Levin theorem, such as the one found in many introductory texts. We formalize this proof in Sect. 3. The formalization in ACL2 will follow the constructive parts of the informal proof quite closely. In Sect. 4 we present some final thoughts and some directions for further research.

2 An Informal Proof

We assume the reader is familiar with Turing machines and the NP-completeness of satisfiability. In this section we present an informal proof of this fact, merely to fix the terminology and lay the foundation for the formal proof to come later. Our exposition follows [4] quite closely. There are other proofs of the Cook-Levin theorem, some more recent and easier to follow. We chose this particular exposition because we considered it to be the most amenable to mechanization.

Informally, a Turing machine consists of a single tape that is divided into an infinite number of cells. The tape has a leftmost cell but no rightmost cell. The machine has a read/write head that can process a single cell at a time. The head can also move to the left or the right one step at a time. The behavior of the machine is governed by a finite control, with transitions based on its current state and the tape symbol being scanned. More formally a Turing machine $M = (Q, \Sigma, \delta, q_0, q_f)$ where Q is a finite set of states including q_0 and q_f, Σ is the finite alphabet of the tape not including the special blank symbol B, and δ is a relation mapping a state and a symbol into a possible move. The states q_0 and q_f are called the initial and accepting states of M, respectively.

The Cook-Levin theorem shows the relationship between Turing machines and satisfiability:

Theorem 1 (Cook, Levin). *Let M be a Turing Machine that is guaranteed to halt on an arbitrary input x after $p(n)$ steps, where p is a (fixed) polynomial and n is the length of x. $L(M)$, the set of strings x accepted by M, is polynomially reducible to satisfiability.*

Consider the behavior of machine M on input x. Initially, the tape contains the input x followed by blanks, the head is scanning the first symbol of x, and the machine M is in its initial state q_0. The machine goes through a sequence of steps, each of which is characterized by the contents of the tape, the position of the head, and the internal state of the machine. After at most $p(n)$ steps, the machine halts. If it halts while in state q_f the machine accepts input x, and otherwise it rejects x.

So a computation of the machine can be formalized as the sequence of steps S_0, S_1, …, $S_{p(n)}$, where S_0 corresponds to the initial configuration of the machine with input x, and each S_{i+1} follows from S_i according to the rules of the machine M. As a matter of convenience, if the machine halts before $p(n)$ steps, we let the last step repeat so that we always end up with $p(n) + 1$ steps.

The step S_i can be represented by its tape, the location of the head, and the internal state of the machine M. It is also helpful to store explicitly the move taken by M from state S_i to S_{i+1}. The tape can hold at most $p(n)$ characters, because it takes at least one step to write a character. We may assume that all tapes have exactly $p(n)$ characters, simply by padding the tapes with blanks on the right. Thus the tapes in the computation can be represented by the two-dimensional array $T(i, j)$ where $i \in [0, p(n)]$ is the step of the computation and $j \in [1, p(n)]$ is the position of the character in the tape. We will use the notation $T(i, *)$ to refer to all the cells in a single step of the computation.

To complete the representation of a computation, we need only represent the position of the head and the machine state at each step of the computation, as well as the moves taken between steps of the computation. A convenient way to do this is to encode this information in the array T. The value of $T(i, j)$ is normally a symbol in the tape. But if the head is at position j at step S_i, then $T(i, j)$ is the composite symbol $\langle c, q, m\rangle$, where c is the character in position j of the tape, q is the state of the machine at step S_i, and m is the move taken by the Turing machine from step S_i to step S_{i+1}.

The transformation to satisfiability is carried out using this data structure. It is clear that the value of $T(i, j)$ is in $\mathit{\Gamma} = \mathit{\Sigma} \cup \{B\} \cup (\mathit{\Sigma} \cup \{B\}) \times Q \times img(\delta)$. For each $i \in [0, p(n)]$, $j \in [1, p(n)]$, and $X \in \mathit{\Gamma}$ we define the proposition $C_{i,j,X}$ with informal meaning $T(i, j) = X$. The expression E_x over these variables is the conjunction of the following four subexpressions:

- The truth assignment really does represent a unique array $T(i, j)$. That is, for each i and j precisely one of the $C_{i,j,X}$ is true.
- The values in $T(0, *)$ correspond to the initial configuration of the machine with x in the input tape.
- The machine is in its final accepting state q_f in $T(p(n), *)$.
- For each $i \in [1, p(n)]$, the configuration represented by $T(i, *)$ follows from the configuration at $T(i - 1, *)$.

Taken together, these expressions are satisfiable if and only if there is some valid computation of M that starts with the input x and ends in an accepting state.

3 A Formal Proof

3.1 The Turing Machine Models

As there are many variants of Turing machines, it is important to specify precisely which variant we are using. Our Turing machines have a semi-infinite tape that is allowed to grow without bounds but only to the right. The input is placed at the beginning of this tape. The machine has a single initial and a final state. Once the machine enters the final state, it is constrained to stay there.

We encode a specific Turing machine in a data structure that contains the machine's alphabet, its set of states, the initial and final states, and the transitions specifying how the machine changes state. The transitions are encoded as a list mapping state/symbol pairs into a list of possible moves. We use the functions `ndtm-alphabet`, `ndtm-states`, `ndtm-initial`, `ndtm-final`, and `ndtm-transition` to select individual components from a Turing machine.

A configuration stores the information about a single step in the computation: The current contents of the tape, the position of the read/write head, and the current internal state of the machine. To make traversals of the tape easier, we split the tape into two halves. The right half of the tape begins with the symbol currently being scanned by the head; its remaining elements contain all the symbols to the right of the head in increasing order. The left half of the tape contains all the symbols to the left of the head in *reverse* order. The functions `config-lhs`, `config-rhs`, and `config-state` will be used to access the members of this structure.

The basic mechanics of the Turing machine are modeled by the function `ndtm-step`, which takes in a Turing machine and a configuration and returns all the possible configurations that may follow it:

```
(defun ndtm-step (machine config)
  (let ((moves (ndtm-moves (config-state config)
                           (first (config-rhs config))
                           (ndtm-transition machine))))
    (ndtm-step-with-move-list config moves)))
```

The function `ndtm-moves` returns all the valid transitions that the Turing machine can make when it is at the given state and looking at the given symbol on the tape. The function `ndtm-step-with-move-list` applies the selected moves to the configuration, returning a list containing all the resulting configurations.

We can not allow a machine to move the read/write head to the left when it is in the first cell position. This is enforced in the function `ndtm-step-with-move` (called by `ndtm-step-with-move-list`). When the head attempts to move past the beginning of the tape, we leave the head scanning the first cell of the tape.

We use a breadth-first strategy to model the non-determinism of the Turing machine. Using this search strategy allows us to decouple the search from the acceptance check. The function `ndtm-step-n` returns all the possible configurations that can occur after stepping through an initial configuration n times. We test acceptance with the function `ndtm-accept` which takes a list of configurations and checks to see if any of them are in the accepting state. The function `ndtm-accepts-p` takes a machine, input, and number of steps, and returns true if the machine accepts the given input in that number of steps.

We place some restrictions on the Turing machines: We insist that once a machine enters its final state it should stay there; we require that a machine have *some* transition for every possible combination of internal state and tape symbol read; and we require some syntactic conditions, such as the initial and final states being listed in the possible states, and that each transition write a valid character in the tape and move to a valid state. These properties are encapsulated in the predicate `valid-machine`. We chose to write this as restrictions on the possible Turing machines, rather than to enforce them in the function `ndtm-step`, to simplify the Turing machine model.

The functions `ndtm-step-n` and `ndtm-accept` faithfully model traditional Turing machines. But the proof of the Cook-Levin theorem makes use of computations, i.e., paths through the tree explored by `ndtm-step-n`. In contrast these functions only store the frontier of the tree, since Turing machines do not keep a "memory" of their previous states. To bridge this gap, we introduced another model of Turing machines, one based on computations instead of configurations.

A computation consists of a sequence of configurations and the Turing machine transitions or moves that link them together. Consider the sequence S_1, S_2, $\ldots$, S_n of configurations, and further let m_i be the move that transforms S_{i-1} into S_i. Then we represent this with the list `( (`S_n `.` m_n`) (`S_{n-1} `.` m_{n-1}`) ...(` S_1`, nil ) )`. Notice that the list contains the last (or current) configuration in the front, making it easier to extend recursively.

The functions `ndtm-comp-step-n` and `ndtm-comp-accept` are direct analogs of `ndtm-step-n` and `ndtm-accept`. In particular, we use the exact same search strategy in the `ndtm-comp-*` functions as we do in the `ndtm-*` functions. This simplifies the proof of the equivalence between the two Turing machine models.

3.2 The Model of Satisfiability

Boolean expressions are considerably simpler than Turing machines. We must make clear that by "boolean expression" we mean any expression made up of propositional variables and the connectives "and," "or," and "not." In particular, we do not restrict ourselves to clausal representation.

What we need to model boolean expressions is an interpreter that can input arbitrary expression trees over `and`, `or`, and `not` as well as a list associating variables with values, and return the value of the expression. We defined the interpreter `booleval` that fits this description. For example, the expression

```
(booleval '(and (or p q) (not r)) '((r . nil) (p . t) (q . t)))
```

returns true, i.e., `t`. In addition, we proved a number of simple theorems about `booleval`, such as the following:

```
(defthm booleval-and
  (implies (equal (first x) 'and)
           (equal (booleval x a)
                  (and (booleval (second x) a)
                       (booleval (third x) a)))))
```

For the remainder of the proof, the actual definition of `booleval` was irrelevant and in fact disabled. Only properties such as the above were used in the proof.

3.3 The Translation

In this section, we describe the formal translation from a Turing machine instance into satisfiability. Rather than presenting the complete translation, we will focus only on the functions that will be needed in the formal proofs to follow.

Recall that the boolean expression E_x consists of the conjunction of four parts, with rough semantics equal to "the assignment consistently represents an array," "the first configuration is the initial configuration for the input," "the final configuration is an accepting configuration," and "successive configurations follow each other legally, according to the rules of the machine." Formally we build this expression as follows:

```
(defun ndtm2sat (machine input nsteps ncells)
  (let ((alphabet (ndtm2sat-alphabet machine)))
    (fold-and (is-a-2d-array 1 nsteps ncells alphabet)
              (first-string-is-input ncells input machine)
              (last-string-accepts nsteps ncells machine)
              (valid-computation nsteps ncells machine))))
```

The function `ndtm2sat-alphabet` builds the alphabet of the array $T(i, j)$. This includes not just the alphabet of the Turing machine, but also all the composite symbols $\langle x, q, \delta \rangle$ encoding a tape symbol, a state, and a legal move. Note: The function `fold-and` returns an expression corresponding to the conjunction of its arguments.

We will now consider each subexpression, starting with `is-a-2d-array`. This function loops over all steps making sure each one is a valid string:

```
(defun is-a-2d-array (step nsteps ncells alphabet)
  (declare (xargs :measure (nfix (1+ (- nsteps step)))))
  (if (or (not (integerp nsteps)) (not (integerp step))
          (> step nsteps))
      t
    (list 'and
          (is-a-string step 1 ncells alphabet)
          (is-a-2d-array (1+ step) nsteps ncells alphabet))))
```

The `:measure` is used to justify the termination of this function. All ACL2 functions are total, so ACL2 tries to prove a function terminates before accepting it. When the termination argument is non-obvious, it is necessary to provide an explicit `:measure` that ACL2 can use in the termination proof.

The definition of `is-a-string` is just like that of `is-a-2d-array`, except that it iterates over the function `is-a-character`, which returns a boolean expression that is sasisfiable precisely when there is exactly one character at position $T(i, j)$:

```
(defun is-a-character (step cell alphabet)
  (list 'and
        (is-one-of-the-characters step cell alphabet)
        (is-not-two-characters step cell alphabet)))
```

Checking that the symbol is one of the characters is straightforward. We need only iterate over the `alphabet` and take the disjunction of all terms `(prop step cell X)` where `X` is one of the members of `alphabet`. Similarly, to make sure there are not two different characters at this position, we consider each member of `alphabet` against each of the *remaining* elements of `alphabet`:

```
(defun is-not-2nd-character (step cell char alphabet)
  (if (endp alphabet)
      t
    (list 'and
          (list 'not
                (list 'and
                      (prop step cell char)
                      (prop step cell (first alphabet))))
          (is-not-2nd-character step cell char (rest alphabet)))))

(defun is-not-two-characters (step cell alphabet)
  (if (or (endp alphabet) (endp (rest alphabet)))
      t
    (list 'and
          (is-not-2nd-character step cell (first alphabet)
                                (rest alphabet))
          (is-not-two-characters step cell (rest alphabet)))))
```

A similar story explains `first-string-is-input`. We already know what the input should be, so we need only check that the appropriate propositions are true. The function `string-holds-values` performs such a check. Given a step, a beginning and end tape position, and a list, it creates a conjunction specifying that the tape holds the characters in the given list. The only complication is that the first character is actually a composite symbol, so we do not know exactly which proposition will be true. This forces us to iterate over all legal moves when the machine is in its initial state and scanning the first character of the input.

The function `last-string-accepts` is also quite simple. It iterates over all the cells in a given step, checking to see if that cell is one of the elements of the final alphabet.

Not surprisingly, `valid-computation` is the hardest part of the translation. The function iterates over successive steps checking that the second follows from the first. We do this by looping over each of the cells in the second tape, making sure that it is correct. The difficulty lies with validating a single cell.

We use the function `valid-cell` to perform this check. A minor difficulty has to do with boundary conditions. The cell $T(i,j)$ depends on the values of $T(i-1,j-1)$, $T(i-1,j)$, and $T(i+1,j)$, which we call the neighbors of $T(i,j)$. But when the cell j is at the beginning or end of the tape, we must drop the neighbors that lie outside the edges. This also prevents the read/write head from scanning past the left edge of the tape. So `valid-cell` performs a case split to check the position of the cell. We will avoid this complication in this presentation, since it only splits the proof into four very similar cases.

For a cell in the "middle" of the tape, there are four ways in which $T(i, j)$ can follow from $T(i-1, *)$. First, it is possible that $T(i-1, j-1)$ is a composite symbol corresponding to a move of the read/write head towards the right, in which case $T(i, j)$ will become a composite symbol. A similar story holds if $T(i-1, j+1)$ represents a move to the left. When $T(i-1, j)$ is composite, then $T(i, j)$ will change as the machine will write a (possibly new) symbol on cell j. In all other cases, $T(i, j)$ will retain the old value of $T(i-1, j)$.

The functions `valid-moves-left`, `-right`, `-middle`, and `-rest` check for these cases. The first three functions scan the valid composite symbols to find the ones that may affect the symbol $T(i, j)$. These functions return two values. The first value is the boolean expression that will be true if and only if $T(i-1, j')$ is a composite symbol resulting in $T(i, j)$. The second value is a list of the composite symbols examined. This list is needed by `valid-moves-rest`, so it can perform the "else" case. To make this clear, consider the definition of `valid-moves-left`:

```
(defun valid-moves-left (prevstep curstep curcell machine)
  (let* ((alphabet (cons nil (ndtm-alphabet machine)))
         (composites (strip-right
                      (composite-symbols
                       alphabet (ndtm-states machine)
                       (ndtm-transition machine)))))
    (cons (make-valid-moves prevstep (1- curcell)
                            curstep curcell
                            composites alphabet machine)
          (prop-list prevstep (1- curcell) composites))))
```

This function handles the case where $T(i-1, j-1)$ is a composite affecting $T(i, j)$. The alphabet consists of the alphabet of the Turing machine and the designated blank symbol, which is represented by `nil`. The auxiliary function `strip-right` returns all the relevant moves, i.e., those that move to the right. The function `prop-list` stores the propositions representing the fact that $T(i-1, j-1)$ is a relevant composite symbol. The function `make-valid-moves` loops over the relevant composite symbols in $T(i-1, j-1)$ and possible tape symbols in $T(i, j)$ and calls `make-valid-move` to generate the given constraint. The definition of `make-valid-move` is given below:

```
(defun make-valid-move (prevstep prevcell curstep curcell
                        composite symbol machine)
  (let* ((newstate (move-nextstate (symb-move composite)))
         (moves (ndtm-moves newstate symbol
                            (ndtm-transition machine))))
    (list 'and
          (prop prevstep prevcell composite)
          (prop prevstep curcell symbol)
          (make-valid-move-list curstep curcell
                                newstate symbol moves))))
```

This function depends on `make-valid-move-list` which loops over the given moves and generates the appropriate composite symbol:

```
(defun make-valid-move-list (curstep curcell state symbol moves)
  (if (endp moves)
      nil
    (list 'or
          (prop curstep curcell (symb symbol state (first moves)))
          (make-valid-move-list curstep curcell state
                                symbol (rest moves)))))
```

The function `valid-moves-right` is completely symmetrical; in fact, it uses many of the same auxiliary functions. The function `valid-moves-middle` is also very similar, but it is slightly more complicated because it takes into account the new symbol written by the machine. That leaves `valid-moves-rest` which handles the else case. That is, if a given cell is not affected by a neighboring composite symbol, then it retains its previous value:

```
(defun valid-moves-rest (prevstep curstep curcell machine cases)
  (list 'and
        (list 'not (fold-or cases))
        (remains-unchanged prevstep curstep curcell
                           (cons nil (ndtm-alphabet machine)))))
```

The list `cases` contains all of the propositions encoding neighboring composite symbols. This is compiled from the second value of the other `valid-moves-*` functions. The function `remains-unchanged` iterates over the given alphabet making sure that $T(i-1, j) = T(i, j)$ and is a member of the alphabet.

Note in particular that `remains-unchanged` is called only for characters that are part of the real alphabet of the tape, i.e., the machine alphabet and the special blank character. No composite symbols are ever passed through this function, since the composite symbols always change according to the rules of the `valid-moves-*` functions.

All of these constraints come together in `valid-cell`, which ties these functions while taking care of the special cases. The following excerpt will suffice to show how this function operates:

```
(defun valid-cell (prevstep curstep curcell ncells machine)
  (if (> curcell 1)
      (if (< curcell ncells)
          (let ((left (valid-moves-left prevstep curstep
                                        curcell machine))
                (middle (valid-moves-middle prevstep curstep
                                            curcell machine))
                (right (valid-moves-right prevstep curstep
                                          curcell machine)))
            (fold-or (first left) (first middle) (first right)
                     (valid-moves-rest
                      prevstep curstep curcell machine
                      (append (rest left) (rest middle)
                              (rest right)))))))
          ...)
```

3.4 Case I: The Turing Machine Accepts

In this section we will show that the boolean expression generated in Sect. 3.3 is satisfied when the Turing machine accepts the input. The expression consists of the conjunction of four main subexpressions which we can consider in turn.

Before delving into the details, it is worth a moment to look at the basic structure of the proofs. Suppose we have a valid computation of the machine accepting x. We want to show that a term `(booleval expr alist)` is true, where `expr` is E_x and `alist` is a truth assignment generated from the accepting computation. The `expr` is constructed by piecing together a large number of *local* terms. For example, the subexpression for `valid-cell` will only examine propositions that correspond to neighboring cells. The `alist` is also constructed in this manner. We will process the computation and translate pieces of it into truth assignments which are then joined together. So the essence of the proof will be to dive into both `expr` and `alist`, such that we can show a particular subexpression `expr1` is true under the truth assignment `alist1`. Then we will "lift" this result to the complete truth assignment, so that `expr1` is satisfied by `alist`. Finally, we put together all the subexpressions to complete the proof.

We begin our study of the proof with the extraction of a truth assignment from a computation. A computation is a list of configurations and the moves that link them together, and a configuration consists of a tape and a state. The most basic extraction function, therefore, converts a tape into a truth assignment:

```
(defun convert-tape-to-assignment (tape step cell ncells)
  (declare (xargs :measure (nfix (1+ (- ncells cell)))))
  (if (or (not (integerp ncells)) (not (integerp cell))
          (> cell ncells))
      nil
    (cons (cons (prop step cell (first tape)) t)
          (convert-tape-to-assignment (rest tape) step (1+ cell)
                                      ncells))))
```

This is the only place where we will assign a value to a proposition; notice in particular that the only propositions assigned are given a true value. The following routine is used to extract an assignment from a configuration:

```
(defun convert-config-move-to-assignment (config move step ncells)
  (convert-tape-to-assignment
   (append (reverse (config-lhs config))
           (cons (symb (first (config-rhs config))
                       (config-state config)
                       move)
```

```
          (rest (config-rhs config))))
   step 1 ncells))
```

To finish the conversion of a computation to a truth assignment, it is only necessary to step over all the configurations in the computation and append the resulting assignments. However, there is a slight complication. The computations associate a configuration with the move that results in that configuration, while the composite symbols associate a state and symbol with the move that is possible *from* that configuration. So we must stagger the moves as we process them. In addition, we must explicitly find a possible (e.g., the first) move extending the last configuration, since this is needed to form the composite symbol but it is not present in the computation.

Now that the truth assignment is constructed, let us consider the proof that it represents a 2D array. We break the boolean expression down into its smallest terms and find the corresponding local section of the truth assignment. Recall how `is-a-2d-array` is defined in terms of `is-a-string`, all the way down to `is-one-of-the-characters`. So we begin by considering the latter function:

```
(defthm tape-to-assignment-is-one-of-the-characters-aux
  (implies (member symbol alphabet)
           (booleval (is-one-of-the-characters step cell alphabet)
                     (cons (cons (prop step cell symbol) t)
                           alist))))
```

As the theorem shows, the simplest truth assignment that makes this expression true is one that begins with a boolean proposition corresponding to this particular cell. As it turns out, the function `convert-tape-to-assignment` has just this property, so it is easy to show the following:

```
(defthm tape-to-assignment-is-one-of-the-characters
  (implies (and (member (first tape) alphabet)
                (integerp ncells) (integerp cell)
                (<= cell ncells))
           (booleval (is-one-of-the-characters step cell alphabet)
                     (convert-tape-to-assignment tape step cell
                                                 ncells))))
```

The satisfiability of `is-not-two-characters` is easy to establish in the same way. So now we are ready to lift the result higher in the truth assignment.

But this is not as simple as it would appear at first. The problem is that the instance of `convert-tape-to-assignment` used in the theorem above hardcodes the value of `cell`. We need to generalize this theorem to allow other cell values, such as the ones in the call from `convert-config-move-to-assignment`. This assignment has some values in front of, not just behind, the one we need.

This is not straightforward. It is possible that one of the assignments in front gives a different value to a proposition. Even if the assignments are disjoint; i.e., if they assign values to different propositions, it is possible for the combination to provide unexpected results. The reason for this is that `booleval` implicitly assigns a value of false to any proposition that is not explicitly assigned, which is a valuable property of `booleval` because it allows truth assignments to be built incrementally. Compatibility of truth assignments depends not only on the assignments themselves, but on the variables used in the term being evaluated.

To continue the proof, therefore, we have to consider the propositions that are assigned values by a truth assignment, as well as the propositions used in an expression. We defined the functions `assigned-vars` and `vars-in-term` for this purpose. Typical theorems about these include the following:

```
(defthm vars-in-term-one-of-the-characters
  (implies (member prop (vars-in-term
                         (is-one-of-the-characters step cell
                                                   alphabet)))
           (and (equal (prop-step prop) step)
                (equal (prop-cell prop) cell))))

(defthm assigned-vars-convert-tape-to-assignment
  (implies (member prop (assigned-vars
                         (convert-tape-to-assignment
                          tape step cell ncells)))
           (and (equal (prop-step prop) step)
                (<= cell (prop-cell prop))
                (<= (prop-cell prop) ncells))))
```

Now it is possible to lift the theorem to bigger truth assignments. We only need lemmas specifying how `booleval` composes the truth assignment. The following lemma is the one we need for this specific case:

```
(defthm booleval-append-alist-left
  (implies (and (not (intersectp-equal (vars-in-term x)
                                       (assigned-vars a)))
                (alistp a))
           (equal (booleval x (append a b))
                  (booleval x b))))
```

This suffices to lift the theorem so that we know the truth assignment generated by `convert-config-move-to-assignment` satisfies `is-a-string`:

```
(defthm move-to-assignment-is-a-string
  (implies (and (no-duplicates alphabet)
                (subsetp (config-lhs config) alphabet)
                (subsetp (config-rhs config) alphabet)
                (member (symb (first (config-rhs config))
                              (config-state config)
                              move)
                        alphabet)
                (member nil alphabet)
                (not (zp ncells)))
           (booleval (is-a-string step 1 ncells alphabet)
                     (convert-config-move-to-assignment
                      config move step ncells))))
```

Notice the requirements that the move appear in the alphabet, and that the tape is a subset of the alphabet. This is needed because `is-a-string` tests not only that the truth assignment is a consistent representation of a string, but also that the string is over a particular alphabet.

To complete the satisfiability of `is-a-string`, we need to apply the theorem `move-to-assignment-is-a-string` to all the configurations in the computation. In particular, we need to show that the hypothesis of this lemma will be satisfied by all the configurations in the computation. But this follows when the initial tape uses symbols only from the alphabet, since subsequent tapes will also satisfy the requirement as long as the transitions in the machine are valid. So we have completed the proof of the satisfiability of `is-a-2d-array`.

The proof of the other three major subexpressions follows the same pattern. The proofs of `first-string-is-input` and `last-string-accepts` do not bring anything new to the table, so we will omit them. It is only worth noting that the proof of `last-string-accepts` depends on the fact that the last configuration has a composite symbol. In particular, the left tape is not allowed to grow by more than the number of steps, which is straightforward to show.

That leaves the proof that the assignment satisfies `valid-computation`. Our plan is to split the tape into three parts. In the middle are the cells around the read/write head, which could possibly be affected by a move. The remaining cells are considered to be either to the left or to the right.

So our first task is to see what happens to a character that is (far enough) to the left of the head. Consider what happens to the actual tape. Suppose `config1` and `config2` are valid configurations. We explore every possible way in which `config2` can follow `config1`. The following theorem is representative:

```
(defthm cdr-lhs-tape-does-not-change-left-move-possible
  (implies (and (equal (move-direction move) 'left)
                (consp (config-lhs config1))
                (valid-step machine config1 move config2))
           (equal (rest (config-lhs config1))
                  (config-lhs config2))))
```

Using this lemma, it is possible to show that if a cell has a given value and the cell is (far enough) to the left of the read/write head, the cell has the same value at the next iteration. Since propositions explicitly in the truth assignments are assigned true, truth is equivalent to membership in the assignment. This results in the following theorem, which is representative of the various cases to consider:

```
(defthm early-cell-in-convert-config-left-move-possible
  (implies (and (consp (config-lhs config1))
                (not (zp ncells))
                (<= (len (config-lhs config1)) ncells)
                (member prop (assigned-vars
                              (convert-config-move-to-assignment
                               config1 move step ncells)))
                (< (prop-cell prop) (len (config-lhs config1)))
                (equal (move-direction move) 'left)
                (valid-step machine config1 move config2))
           (member (prop (1+ step) (prop-cell prop)
                         (prop-char prop))
                   (assigned-vars
                    (convert-config-move-to-assignment
                     config2 move2 (1+ step) ncells)))))
```

Naturally, the next step is to lift this result to the larger truth assignments.

Theorems such as the one above show precisely what happens to all the cells in a tape, except for two cells around the read/write head, the one which the head is scanning and the one to which the head will move. We have to handle these cases separately. Although these results are more interesting, in the sense that this is where the machine is performing some action, they are easier to prove than the ones above because we know precisely which cells are involved. That means we can prove an exact theorem, such as the following:

```
(defthm middle-cell-in-convert-config-move-left-move-possible-1
  (implies (and (consp computation)
                (consp (rest computation))
                (equal (first (first computation)) config2)
                (equal (rest (first computation)) move)
                (equal (first (first (rest computation))) config1)
                (consp (config-lhs config1))
                (not (zp ncells))
                (<= (len (config-lhs config1)) ncells)
                (equal step (len computation))
                (equal (move-direction move) 'left)
                (valid-computation machine computation))
           (and (member (prop (1- step)
                              (len (config-lhs config1))
                              (first (config-lhs config1)))
                        (assigned-vars
                         (convert-config-move-to-assignment
                          config1 move (1- step) ncells)))
                (member (prop step
                              (len (config-lhs config1))
                              (symb (first (config-lhs config1))
                                    (config-state config2)
                                    prevmove))
                        (assigned-vars
                         (convert-config-move-to-assignment
                          config2 prevmove step ncells))))))
```

This theorem covers the cell position to which the head moves. A similar theorem takes care of the cell position originally containing the read/write head.

These lemmas are almost ready to be stitched together into the final theorem. The missing piece is the fact that the "else" case in the definition of `valid-cell`, which allows a cell in the tape that is away from the head to retain its value, requires the cell not to have a relevant neighbor. So we must prove that when a cell is (far enough) away from the head, the cells around it are not composite.

Combining all the theorems proved so far shows that `valid-computation` is satisfied by the generated truth assignment. Then combining that with the other parts of the condition, we get the final result:

```
(defthm valid-transformation-computation-best
  (implies (and (integerp n) (< 1 n)
                (valid-machine machine)
                (alphabet-symbol-list-p (ndtm-alphabet machine))
                (no-duplicates (ndtm2sat-alphabet machine))
                (ndtm-accepts-p machine input (1- n))
                (subsetp input (ndtm-alphabet machine)))
           (booleval (ndtm2sat machine input n n)
                     (convert-computation-to-assignment
                      machine
                      (accepting-witness machine input (1- n))
                      n))))
```

Note: The function `accepting-witness` searches for a valid, accepting computation.

3.5 Case II: The Expression is Satisfiable

In this section we explore the other half of the proof. We wish to show that when the expression E_x is satisfiable, the input x is accepted by the machine. To do this, we will extract a computation from a truth assignment that satisfies E_x by looking at all the propositional formulas $C_{i,j,X}$ for each i and j, and selecting the one X that makes it true. So the most fundamental function is `extract-char-alist`, which finds the X that makes $C_{i,j,X}$ true:

```
(defun extract-char-alist (step cell alphabet alist)
  (if (endp alphabet)
      nil
    (if (booleval (prop step cell (first alphabet)) alist)
        (first alphabet)
      (extract-char-alist step cell (rest alphabet) alist))))
```

Notice that we must know the relevant alphabet a priori.

With this function we can define `extract-lhs-tape` and `extract-rhs-tape`, which extract the left and right halves of the tape, respectively. It is only necessary to know where to split the tape. We do this by iterating over all the cells in the tape until we find a composite cell. We wrote different functions for the left and right halves of the tape since the former is stored in reversed order. Once these functions are defined, it is a simple matter to write `extract-config`, which extracts a configuration, and use that to define `extract-computation` which extracts the candidate valid, accepting computation.

At this point, we have a situation similar to the one we faced in trying to prove E_x is satisfiable if there is a valid computation. It would appear that the remaining part of the proof is as difficult as what has gone before, since in both cases we are considering an expression of the form `(booleval expr alist)`. But there is a key difference. Previously, we had a valid computation and we used that to extract an `alist`. The extraction process was localized, so it was necessary to dig into portions of the `alist` to find the part that made a particular expression true. But in this case, the `alist` is known a priori, so we need only split `expr` into its subexpressions, leaving the `alist` unchanged.

The key point is that we break up the structure of `expr`, not of `alist`. Since the function `booleval` is defined precisely in this way, this leads to much simpler lemmas, without worrying about issues such as inconsistent truth assignments. This came as a very pleasant discovery for us. We noticed that the proof in this direction was much easier partly because so much more of the proof was discovered automatically by ACL2. It was in trying to understand why we were so lucky that we discovered the delicious asymmetry of `booleval`. Since the proof is much more mechanical in this direction, we will only present an outline.

Notice that the function `extract-computation` is guaranteed to extract only one computation. However, it is possible that more computations can be extracted from the truth assignment. Of course this is not the case, and at first we thought there was no real need to prove this, but it turns out that this uniqueness property is crucial in the other proofs. Many times it will not be enough to know that $C_{i,j,X}$ is true; we must also know that $C_{i,j,Y}$ is false for all $Y \neq X$.

As before, the strategy is to isolate what happens around the read/write head. This corresponds to the composite symbol, so it is necessary to know that there is only one composite symbol at any step in the truth assignment. We do this by counting the number of composite symbols in a given step. If we know that this number is equal to 1 and we find a composite symbol at some cell, then we are guaranteed that none of the symbols in other cells are composite.

Next we show that if a cell changes from one step to the next, then one of its neighbors must be a composite symbol. Moreover, the composite symbol is restricted based on its relationship to the cell that changed. For example, if $T(i,j) \neq T(i-1,j)$ and $T(i-1,j-1)$ is a composite, then it must be a composite symbol corresponding to a right move of the tape.

To complete the proof we observe that every step in the computation has exactly one composite symbol. It is easy to show that if one configuration has only one composite symbol, the next one can have at most one such symbol, and if a configuration has no composite symbols neither does the next. Since the initial and final configurations have one composite symbol, so must all the other ones in the computation, and this is the one found when we split the left and right tapes in the transformation.

Essentially the proof is now complete. The cells that are sufficiently to the left of *the* composite symbol in a step are unchanged, as are the ones that are sufficiently to the right. The behavior of the cells immediately around the composite symbol is also known. This is enough to show that two successive configurations extracted from the truth assignment legally follow according to the rules of the Turing machine.

The only remaining complication is that the computation that is extracted is not the computation that `ndtm-comp-step-n` will enumerate. But the only difference is that the extracted computations pads the right tape with blanks to make $p(n)$ cells. It is easy to show that such starting configurations are equivalent, in the sense that if one of them ends in an accepting state so does the other.

3.6 Timing Analysis

Thus far we have ignored the issue of timing. But it is an important aspect of the Cook-Levin theorem that the transformation take only polynomial time, so we would like to address this as well.

Unlike higher-order theorem provers, ACL2 does not provide any introspection mechanisms that can be used for cost measurement. It does provide a mechanism for defining an interpreter over certain functions, but this interpreter is unsuitable for measuring costs, since it uses the functions directly to evaluate results without opening up their definitions.

Curiously, ACL2's prececessor, the Boyer-Moore theorem prover, did have a facility that would be useful in this context. In that theorem prover, every function definition extended a set of built-in interpreters, including `v&c$` which computed the value and the cost of an expression.

Without such an interpreter, however, we are forced to proceed differently. What we did was to define a `cost-*` version of each function used in the translation. This function returns a pair, the first element being the normal value of the function, and the second a measure of the cost used to compute this value. For each such function, we also proved two theorems about it. The first states that the `cost-*` function accurately computes the function it emulates. The second gives an upper bound for the cost.

4 Conclusions

In this paper we described a formal proof in ACL2 of the Cook-Levin theorem. The formal proof fills in the gaps typically left by higher-level proofs. In particular, we showed that the transformation mapping instances of Turing machines to satisfiability really does work.

We attempted to use this proof while teaching a one-hour graduate course introducing students to ACL2. The format of the course requires each student to make a presentation during at least one class period. One of the challenges in teaching a course like this is keeping students interested in each other's presentations. By having each student make a small contribution to a larger research project, we hoped to establish a continuity between the presentations that would involve them throughout the semester. Unfortunately this did not work as planned. The students did gain experience with ACL2, and some of them are becaming proficient in it, but they found the Cook-Levin theorem too difficult to formalize. Not having had a course that covered this theorem in detail, many found even the informal proof too difficult to follow.

This is a shame because the proof does follow many classic patterns of formal proofs: It builds a formal model of the entities involved, namely Turing machines and boolean expressions; it constructs mappings between them; and it shows that the mappings are connected in important ways. Moreover, in doing the proof we discovered that the asymmetry in the definition of `booleval` led to one half of the proof being much easier than the other. This is a beautiful example of the deep connection between recursion and induction. One direction is easier because its natural induction scheme mirrors the recursive structure of the function, making everything work smoothly.

The major weakness in the formalization lies in the analysis of the time complexity of the translation. It is an important aspect of the proof that the translation can be performed in polynomial time. But this is not the sort of reasoning that comes naturally in ACL2. Currently we are investigating a way to extend ACL2 to introduce interpreters that can compute the cost of evaluating an expression as well as its value. There are some very interesting challenges, such as the termination proof for the interpreter.

References

1. R. S. Boyer and J Moore. A mechanical proof of the turing completeness of pure Lisp. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years*. American Mathematical Society, 1984.
2. R. S. Boyer and J Moore. A mechanical proof of the unsolvability of the halting problem. *Journal of the ACM*, 1984.
3. S. Cook. The complexity of theorem proving procedures. In *Proceedings of the 3rd ACM Symposium on Theory of Computing*, 1971.
4. J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.
5. M. Kaufmann, P. Manolios, and J S. Moore. *Computer Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
6. M. Kaufmann and J S. Moore. The ACL2 home page. http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html.
7. L. A. Levin. Universal sorting problems. *Problemi Peredachi Informatsii*, 1973.
8. N. Shankar. *Proof Checking Metamathematics.* PhD thesis, University of Texas, 1984.
9. N. Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1985.