A Boolean Expression Language

Warren A. Hunt, Jr.

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

hunt@cs.utexas.edu
+ 1 512 471 9748
http://www.cs.utexas.edu/users/hunt

Fall, 2008, Revised 2012, 2013[1]

Abstract. We introduce the syntax and semantics for a language of Boolean expressions. A decision procedure for this language is presented, and the reader is led through exercises that establish its correctness. Later, we formally define an `IF`-expression language and show it that can be correctly translated into our language of Boolean expressions. We assume a familiarity with defining ACL2 functions and proving theorems. Answers are available as an ACL2 script with embedded comments that can be checked by the ACL2 theorem-proving system, but we hope the reader independently solves the problems presented. The foundation for this work was jointly developed by Robert S. Boyer and the author.

1 Introduction

Boolean logic is a basis for many logical systems. Problems expressed in ACL2 are often transformed into a Boolean logic formula, which in turn is checked to determine if it is invalid (never true), valid (always true), or satisfiable (sometimes true); we will make these notions precise.

Through experimentation, we have found that our Boolean expression manipulation functions generally execute at one fourth to one third the speed of the best available BDD packages. Thus, a reader interested in considering problems that can be translated into questions of quantified Boolean logic may find practical use in our presentation.

We start by defining the syntax and semantics of our Boolean-expression language. Later, we define our `IF`-expression language and we prove that it can be translated into our Boolean-expression language. Associated with this text is an ACL2 events file that both provides the answers to the problems posed and once processed by the ACL2 system, ACL2 can be used to decide questions about terms in our `IF`-expression language.

2 Boolean Expressions

We investigate properties of our Boolean expressions by expressing them with the ACL2 logic; thus, we will give both the syntax and the semantics for a Boolean expression using ACL2 functions. A Boolean expression is a rooted binary tree with constant symbols `T` or `NIL` or is a `CONS` *node* of two Boolean expressions with no node equal to `'(T . T)` or to `'(NIL . NIL)`. We first define the syntax and then the semantics of Boolean expressions.

2.1 Syntax of Boolean Expressions

We specify an acceptable syntax for Boolean expressions by defining an ACL2 predicate, `NORMP`, that recognizes syntactically well-formed Boolean expressions. We call such expressions uBDDs for *unlabeled Binary Decision Diagrams*, in the spirit of Bryant's binary-decision diagrams (BDDs). As far as we know, our uBDD representation of Boolean expressions is unique with respect to the BDD implementation literature as our internal uBDDs nodes do not contain references to variable names. uBDDs along with logical operators are a model of a Boolean

[1] Improvements and corrections from my CS389r students and Matt Kaufmann.

logic. Note that we often use DEFN instead of DEFUN; it is just a shorthand version of DEFUN that declares the guard to be T.[2] Below is our NORMP definition.

```
(defn normp (x)
  (if (atom x)
      (booleanp x)
    (and (normp (car x))
         (normp (cdr x))
         (if (atom (car x))
             (not (equal (car x) (cdr x)))
           t))))
```

We often find it convenient to consider vectors of Boolean expressions; for instance, we may wish to express a Boolean function for adding two vectors of Boolean values. We define a recognizer for a list of uBDDs as:

Problem 1.

Define NORMP-LIST with signature

```
(DEFN NORMP-LIST (x) <the-function-body>)
```

by replacing <the-function-body> with a suitable function body.[3]

Before one may use a function memoization feature that has been added to ACL2, it is necessary for a function's guard to be proven; thus we almost always attempt to verify the guards for our function definitions. In fact, we developed ACL2's memoization mechanism using our formalization of uBDDs.

We can explore syntactic properties of uBDD expressions by stating conjectures and attempting to prove them. For example, is a pair of two syntactically well-formed Boolean expressions a syntactically well-formed Boolean expression?

Problem 2.
Prove or disprove:

```
(implies (and (normp x)
              (normp y))
         (normp (cons x y)))
```

Other syntactic properties can be defined as well. For instance, we define the depth of a node as the number of successive CAR and CDR operations required to reach a node. The root node has depth zero; the depth of a node is one more than the depth of its parent.

Problem 3.
Define (MAX-DEPTH X) to calculate the maximum depth of a Boolean expression.

2.2 Semantics of Boolean Expressions

We define the semantics of a Boolean expression by defining the left-right branch choice at each depth, d, by using the d'th Boolean value in a proper list of Boolean values. Thus, our Boolean expressions do not actually have variables names, but variables are instead identified by their depth in a Boolean expression.

[2] We will discuss guards later.
[3] In later problems, we will only write "Define (NORMP-LIST X) with the following properties".

```
(defn eval-bdd (x values)
  (if (atom x)
      x
    (if (atom values)
        (eval-bdd (cdr x) nil)
      (if (car values)
          (eval-bdd (car x) (cdr values))
        (eval-bdd (cdr x) (cdr values))))))
```

`(EVAL-bdd x values)` is the 'value' of `X` with respect to the assignment of `VALUES` to variables. Variable `X` is normally a `NORMP` tree and `VALUES` is normally recognized by `BOOLEAN-LISTP`, i.e., a proper-list of `T` and `NIL` values. Of course, since `EVAL-BDD` has a guard of `T`, it can be given any two ACL2 objects as arguments.

If `X` is an atom, then `X` is its own 'value'; otherwise, we use the `CAR` and `CDR` of `VALUES`, say `A` and `D`, to guide us further through `X`. If `VALUES` is an atom, we use `NIL` for both `A` and `D`. If `A` is `NIL` the answer is the value of `(CDR X)` with respect to `D`; otherwise, the answer is the value of `(CAR X)` with respect to `D`. One can think of an empty (atomic) `VALUES` argument to `EVAL-BDD` as representing an infinite list of `NIL` values.

Problem 4.
Prove or disprove:

```
(implies (normp x)
         (booleanp (eval-bdd x vals)))
```

Sometimes, we wish to evaluate a list of uBDDs with respect to a single assignment of values to the levels (variables).

```
(defn eval-bdd-list (bdds values)
  (if (atom bdds)
      nil
    (cons (eval-bdd (car bdds) values)
          (eval-bdd-list (cdr bdds) values))))
```

`EVAL-BDD-LIST` is useful for evaluating lists of uBDDs that might represent the Boolean outputs of some multi-output function, such as a hardware adder.

Evaluation of a uBDD `X` by `(EVAL-BDD X VALUES)` is nothing more than following the left-right instructions given for each depth in `VALUES`. Whenever `X` is an atom, we return the atom; otherwise, we consult `VALUES` for a left-right decision. We will use uBDDs to represent Boolean functions, such as shown below.

```
          0    <-- A        ;  This simple tree could
         / \                ;  represent the disjunction
        /   \               ;  of variables A and B.
       T     0   <-- B
            / \             ;  This uBDD is written
           /   \            ;  (CONS T (CONS T NIL))
          T    NIL          ;  which we abbreviate as '(T T).
```

Remember, our uBDD expressions do not actually have variable names. The introduction of `A` and `B` is only for our benefit; here variable `A` represents the variable at depth `0` and `B` represents the variable at depth `1`. Why do we say that the tree could represent the disjunction of `A` and `B`? Well, we consider the left-right instructions given by the values argument, and we can prove that the following is a theorem.

```
(and
 (let* ((A nil)
        (B nil)
        (values (list A B)))
   (equal (eval-bdd '(t t) values) nil))
 (let* ((A t)
        (B nil)
        (values (list A B)))
   (equal (eval-bdd '(t t) values) t))
 (let* ((A nil)
        (B t)
        (values (list A B)))
   (equal (eval-bdd '(t t) values) t))
 (let* ((A t)
        (B t)
        (values (list A B)))
   (equal (eval-bdd '(t t) values) t)))
```

The variable names `A` and `B` are just place holders in the `LET` expression. The theorem above could have been more compactly written as:

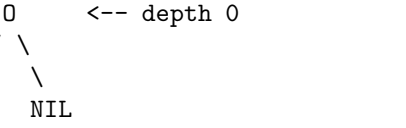
```
(and
 (equal (eval-bdd '(t t) '(nil nil)) nil)
 (equal (eval-bdd '(t t) '(t nil)) t)
 (equal (eval-bdd '(t t) '(nil t)) t)
 (equal (eval-bdd '(t t) '(t t)) t))
```

Problem 5.
Evaluate each of the expressions below.

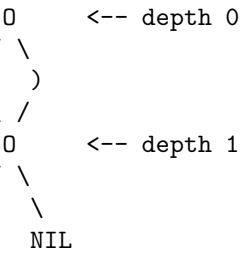
1. `(eval-bdd t '(t nil))`
2. `(eval-bdd '(t) '(t nil))`
3. `(eval-bdd '(nil) '(t nil))`
4. `(eval-bdd '(t nil) '(t nil))`
5. `(eval-bdd '(t t) '(nil nil))`
6. `(eval-bdd '(t t t) '(nil nil nil))`
7. `(eval-bdd '(((t))) '(t t t))`

2.3 uBDD Variables

Writing large uBDD expressions without the aid of variables can be tedious. Each variable is identified by its depth in a uBDD. Consider the uBDD shown below.



The value of this uBDD is decided by the *variable* at depth `0`. Thus, if the variable at depth `0` is `T`, the evaluation of this uBDD is `T`; likewise, if the variable at depth `0` is `NIL`, the evaluation of this uBDD is `NIL`. Thus, we often informally call such a uBDD, "variable `0`". Given this approach, we might depict variable `1` as shown below.



We might write variable `1` as:

```
(let ((var0 (cons t nil))
      (var1 (cons var0 var0)))
     var1)
```

The 'value' of the expression above is independent from every other variable; whether variable `0` is `T` or `NIL` does not affect the result, as either choice leaves `EVAL-BDD` in the same place. Any higher-numbered variable also does not affect the result because `EVAL-BDD` would have finished its evaluation immediately upon finding an atom. This may seem to be a strange way to write variables, but we will later see the utility of this representation.

Problem 6.
Define `(QVAR-N n)` that creates uBDD variable `n`.

We wish to observe several properties about our `QVAR-N` definition. We will later depend on these properties, so, if necessary, the reader may need to alter their definition of `QVAR-N` until it satisfies the three properties identified in the next three problems.

Problem 7.
Prove:

```
(implies (natp n)
         (consp (qvar-n n)))
```

Problem 8.
Prove:

```
(normp (qvar-n n))
```

Problem 9.
Prove:

```
(booleanp (eval-bdd (qvar-n n) values))
```

We have formally defined the syntax and semantics of a Boolean expression language; it is composed only of `CONS` pairs and the terminal symbols `T` and `NIL`. So far, we have not provided ourselves with functions to manipulate our uBDD expressions, and correspondingly, they have little use. In the next section, we will define operations to manipulate our uBDDs.

3 uBDD Operations

Boolean logic can be used to decide many questions of interest. To create and compose uBDD Boolean expressions, we define a number of functions that perform logical operations on uBDDs. Later, we will investigate using uBDDs to represents sets and define uBDD set operations. Important to our development of uBDDs is proving that our uBDD operations correctly manipulate uBDDs; the reader will be asked to consider this issue. We start by considering some properties of our hash-`CONS` implementation.

3.1 Unique Object Representation

We have developed a canonical representation for ACL2 data objects and a function memoization mechanism to facilitate reuse of previously computed results. In the implementation of the ACL2 logic, ACL2 data objects are represented by Common Lisp objects of the same type, and the ACL2 pairing (`CONS`) operation is internally implemented by the Common Lisp `CONS` procedure. We have defined a new ACL2 function `HONS` that is logically identical to the ACL2 `CONS` function, but whose implementation usually reuses an existing pair; this operation has been called Hash `CONS`ing. We also define a new equality function whose implementation takes advantage of our canonical data representation.

Remark: The uBDD work presented here is part of an extension developed by Robert S. Boyer and the author to provide ACL2 users with unique object representation, function memoization, and hash-based association lists. This extension does not change the ACL2 logic; it provides enhancements in the underlying implementation of the ACL2 system. The creation of globally unique `CONS` pairs, enabled partially by the implementation of *Hash*-`CONS`, represents ACL2 data objects in a canonical way; thus, the comparison of any two such objects can be determined in constant time. Guard-checked ACL2 user-defined functions may be memoized; the underlying implementation may conditionally retain the return values of such function calls so that if a repeated function application is requested, a previously computed value may instead be returned. Using hash tables invisible to the user, we have defined fast association list access and update functions. Not part of the ACL2 logic but useful for file I/O, we provide a file reader that identifies and eliminates duplicate representations of repeated objects and a file printer that produces output with no duplicate subexpressions.

We define several built-in functions that are logically defined in the usual manner, but these functions have system-level implementations. Logically, the `HONS` function is just defined to be `CONS`.

```
(defn hons (x y)
  (cons x y))
```

However, this built-in function has a different system-level implementation than that of `CONS`, which just creates a new pair. In fact, in Common Lisp, when `CONS` is called, it is required to produce a new pair distinct from every other existing pair, even when asked to create a pair with the same `CAR` and `CDR` as an existing pair. ACL2 restricts its formalization and usage of Common Lisp to a functional subset. The ACL2 user is not permitted any destructive operations, so it is possible to reuse existing ACL2 data structures. The implementation of `HONS` first checks to see if there already exists a `CONS` pair previously created by a `HONS` operation that has the same two arguments. If so, the previously created `HONS` is returned; otherwise, a new pair is created and an internal reference to this new pair is saved. This mechanism assures that we never twice create the same `HONS` pair, but instead refer to previously created pairs when available — this, in turn, assures that the equality of any two ACL2 data objects can be determined in constant time.

We have defined the function `HONS-EQUAL` for equality testing.

```
(defn hons-equal (x y)
  (equal x y))
```

We have defined a macro so we may instead write `HQUAL`, defined in the ACL2 community book `misc/hons-help2.lisp`. The implementation of `HONS-EQUAL` can check its arguments to determine if they was created by `HONS`; if so, then the equality check is simply object pointer equality. Thus, the equality of very large `CONS` trees can be determined in constant time. We also take advantage of this capability when memoizing functions.

3.2 Are uBDDs Canonical?

We have mentioned that our underlying implementation for ACL2 data objects is canonical, but we have no means to formally establish such. However, we can consider the canonicity of our uBDDs with respect to `EQUAL`. However, before we do so, we introduce several abbreviations for constructing and traversing (destructing) uBDDs.

Both `QCAR` and `QCDR` check for an atom before 'descending' into a Boolean expression; atoms are left alone. Before we create a new Boolean expression from two existing Boolean expressions, `QCONS` checks to see if its arguments are identical Boolean constants; if so, they are 'reduced'.

```
(defn qcar (x) (if (consp x) (car x) x))


(defn qcdr (x) (if (consp x) (cdr x) x))


(defn qcons (x y)
  (if (or (and (eq x t)   (eq y t))
          (and (eq x nil) (eq y nil)))
      x
    (hons x y)))
```

We certainly want, in all cases, the evaluations of `A` and `B`, with respect to the same variable ordering, to be the same when `A` and `B` are equal; otherwise, we would have no confidence in our uBDD formalization. Thus, the following theorem is obvious.

```
(implies (and (normp a)
              (normp b)
              (equal a b))
         (equal (eval-bdd a v)
                (eval-bdd b v)))
```

That is, if uBDDs `A` and `B` are equal, then so are their evaluations. However, is it the case that if uBDDs `A` and `B` are not equal then their evaluations different? Obviously, for some evaluations they may not be different, but is there always some assignment of values to the variables that will exhibit a difference if uBDDs `A` and `B` are different?

Problem 10.
Find a witness function, `FIND-DIFF`, that creates a different assignment of variable values so the following conjecture can be proven:

```
(implies (and (normp a)
              (normp b)
              (not (equal a b)))
         (not (equal (eval-bdd a (find-diff a b))
                     (eval-bdd b (find-diff a b)))))
```

This statement says that if uBDDs `A` and `B` are different, then there is some assignment of values to variables, as discovered by `FIND-DIFF`, that shows that the evaluations of `A` and `B` are different. Thus, we conclude that uBDDs are a canonical representation of Boolean functions.

The canonicity of uBDDs is a great asset. If uBDD operations are composed and the final result is `T`, then the composition of these operations produced a tautology. If uBDD operations produce `NIL`, then the conjecture represented by the uBDD operations is unsatisfiable. Finally, if a uBDD tree is produced, then the operation produced a Boolean expression in terms of the variable levels. So, due to the canonicity of uBDDs, we have a decision procedure for Boolean logic. A user only needs to compose uBDD operations that are in accordance with some problem, and just the act of composing these operations allows a user to discover if a conjecture (represented through a collection of uBDD operations) is a tautology.

3.3 uBDD Negation

Our first uBDD operation is negation; that is, we define the `Q-NOT` function to replace each terminal node `T` with `NIL` and, respectively, replace each terminal node `NIL` with `T`.

Problem 11.
Define `(Q-NOT x)` that uses `HONS` for pair creation and that 'negates' each Boolean tip value.

Now, we wish to establish the correctness of Q-NOT with respect to our semantics. So, for each uBDD operation we want to establish its syntactic and semantic properties: (a) given NORMP arguments, it returns a NORMP result, and (b) the result is correct with respect to its corresponding specification function.

Problem 12.
Prove:

```
(implies (normp x)
         (normp (q-not x)))
```

Problem 13.
Prove:

```
(implies (normp x)
         (equal (eval-bdd (q-not x) vals)
                (not (eval-bdd x vals))))
```

3.4 The uBDD If-Then-Else Operation

We now are ready to define our uBDD if-then-else operation, Q-ITE. Given, for the moment, that we think of each level in a uBDD as corresponding to a variable, we may think of uBDDs as generally having three characteristics: each path from the root to a tip encounters each (variable) level at most once, every path encounters levels (variables) in a pre-specified order, and each node is *reduced* only in the limited sense that it edges are not both T and its edges are not both NIL. Thus, our uBDD definition does not reduce internal nodes unless both outgoing edges of a uBDD node point to the same constant (T or NIL). So unlike Bryant's BDDs, our uBDD structures do not store a variable name in a node because; instead, we maintain the levels (variables), in order, until we reach a Boolean tip. Recall the definition of QVAR-N: it isn't reduced, since for each variable level except the last, the CAR and CDR are equal; but it is recognized by NORMP.

The order of the uBDD variables is implicit — there are no names, just the depth from the root. Thus, a uBDD with only one variable can be either the reduced values T or NIL, or it can be, (HONS T NIL), or (HONS NIL T); (HONS T T) and (HONS NIL NIL) are not permitted, but reduced to T and NIL, respectively.

The definition of Q-ITE is a bit more subtle than defining uBDD negation as we perform several simplifications that ensure that the result is a canonical representation of some Boolean expression; by canonical we mean that each Boolean function is represented with exactly one unique uBDD. Our definition is designed to assure this canonical form by performing several simplifications. Below is our definition, which we follow with some additional explanation.

```
(defn q-ite (x y z)
  (cond
   ((null x) z)                                    ; (if NIL y z) => z
   ((atom x) y)                                    ; (if  T  y z) => y
   (t (let ((y (if (hqual x y) t y))               ; Simplify left branch
            (z (if (hqual x z) nil z)))            ; Simplify right branch
        (cond ((hqual y z) y)                      ; (if x y y)   => y
              ((and (eq y t) (eq z nil)) x)        ; (if x T NIL) => x
              ((and (eq y nil) (eq z t)) (q-not x)) ; (if x NIL T) => (Q-NOT X)
              (t (qcons (q-ite (car x) (qcar y) (qcar z))
                        (q-ite (cdr x) (qcdr y) (qcdr z)))))))))
```

The function call (Q-ITE X Y Z) can be read as 'if X, then Y else Z'. The (NULL X) check is simple, and, if true, we return Z. Since we expect Q-ITE to be called with NORMP arguments, the (ATOM X) test is essentially a check to see if X is T; if so, we return Y.

If we proceed to the the LET expression, then X is a pair and we simplify Y and Z with respect to X. Thus, if X and Y are equal (HQUAL is an abbreviation for HONS-EQUAL), then whenever X is true, Y will be also true so we

replace `Y` by `T`. Similarly, if `X` and `Z` are equal, then whenever `X` is false, `Z` will be false so we replace `Z` by `NIL`. In a sense, `X` 'governs' `Y` and `Z`, and we make use of that information.

The final conditional expression performs two simplifications that are critical to keep the return value canonical. If `Y` and `Z` are identical, the value of `X` is irrelevant. The next two tests ensure that there is just one representation for any expression `X`; the second of these two tests is not strictly necessary but we call `Q-NOT` for execution efficiency; in our function-memoization implementation, single-argument functions are more efficiently memoized than multiple-argument functions. Finally, we just return the result of recursively dividing the problem.

Problem 14.
Prove:

```
(implies (and (normp x)
              (normp y)
              (normp z))
         (normp (q-ite x y z)))
```

Problem 15.
Prove:

```
(implies (and (normp x)
              (normp y)
              (normp z))
         (equal (eval-bdd (q-ite x y z) vals)
                (if (eval-bdd x vals)
                    (eval-bdd y vals)
                  (eval-bdd z vals))))
```

3.5 Other uBDD Operations

A variety of uBDD operations can be created; these operations take uBDDs as arguments and produce a uBDD. We have seen two examples of such functions, and many other such operations can be defined. Function `Q-ITE` can be used to implement other logical uBDD operations. For instance, `Q-NOT-ITE` is defined as follows.

```
(defn q-not-ite (x)
  (q-ite x nil t))
```

Of course, we already have defined `Q-NOT`, which `Q-ITE` will appeal to in this case. We can also use `Q-ITE` to define `Q-AND-ITE`.

```
(defn q-and-ite (x y)
  (q-ite x y nil))
```

The execution performance of our uBDD definitions will be affected by their corresponding memoization tables. Although we could use `Q-ITE` for everything, it is usually more efficient to define a particular uBDD operation for a specific function. For instance, `Q-AND` only has two arguments, thus its memoization need only concern itself with two arguments.

Problem 16.
Define a recursive function `(Q-AND X Y)` that makes no use of `Q-ITE`.

Problem 17.
Prove:

```
(implies (and (normp x)
              (normp y))
         (normp (q-and x y)))
```

Problem 18.
Prove:

```
(implies (and (normp x)
              (normp y))
         (equal (eval-bdd (q-and x y) vals)
                (and (eval-bdd x vals)
                     (eval-bdd y vals))))
```

We leave to the reader the definition and verification of other uBDD logical operations.

4 A Higher-Level Boolean Language

It is difficult to read uBDDs, especially when they represent large functions. It can also be clumsy to construct large uBDDs using only uBDD operations. In this section, we introduce the more abstract `IF`-expression language for representing Boolean functions, we give its semantics, and we consider the correctness of translating `IF`-expression language statements into uBDDs.

We have defined the `IF`-expression language so that a user may write Boolean expressions using variables. Before presenting the `IF`-expression language, we define a recognizer for `IF` expressions and a function that normalizes `IF` expressions. For brevity, we will often write `IF` expressions as `IF` terms.

4.1 Well-formed IF Expressions and Their Meaning

What is an `IF` expression? It is a term that is either a variable recognized by the `EQLABLEP` predicate or an `IF`-function call term with three arguments that are themselves `IF` terms. Consider the following examples.

```
(if x y z)
x
(if (if a b c) x (if q r t))
```

Note that the last example has an `IF` term as its first argument. As a part of our conversion of an `IF` term to a uBDD, we will convert `IF` terms into canonical uBDD expressions.

We define the semantics of our `IF` expression language with an evaluator. We first define a recognizer for `IF` terms suitable for conversion to uBDDs.

```
(defn eqlablep (x)
  (or (acl2-numberp x)
      (symbolp x)
      (characterp x)))
```

```
(defn qnorm1-guard (x)
  (if (atom x)
      (eqlablep x)
    (let ((fn (car x))
          (args (cdr x)))
      (case fn
        (if (and (consp args)
                 (consp (cdr args))
                 (consp (cddr args))
                 (null (cdddr args))
                 (qnorm1-guard (car args))
                 (qnorm1-guard (cadr args))
                 (qnorm1-guard (caddr args))))
        (quote (and (consp args)
                    (normp (car args))
                    (null (cdr args))))
        (otherwise nil)))))
```

Next, we define a function that is used to look up a variable binding given a variable, a list of variables, and a corresponding list of variable bindings. Of course we could have use an association list, but we choose to keep the variables and their values separated in the spirit of `EVAL-BDD`.

```
(defun sym-val (term vars vals)
  (declare (xargs :guard (and (eqlable-listp vars)
                              (boolean-listp vals))))
  (if (endp vars) nil
    (if (eql term (car vars))
        (car vals)
      (sym-val term (cdr vars) (cdr vals)))))
```

Finally, we define the `IF`-term evaluator `TERM-EVAL`; this function gives the semantics of `IF` expressions.

```
(defun term-eval (term vars vals)
  (declare (xargs :guard (and (qnorm1-guard term)
                              (eqlable-listp vars)
                              (boolean-listp vals))))
  (cond ((eq term t) t)
        ((eq term nil) nil)
        ((eqlablep term)
         (sym-val term vars vals))
        (t (let ((fn (car term))
                 (args (cdr term)))
             (case fn
               (if (if (term-eval (car args) vars vals)
                       (term-eval (cadr args) vars vals)
                     (term-eval (caddr args) vars vals)))
               (quote (eval-bdd (car args) vals))
               (t nil))))))
```

4.2 Converting IF terms to uBDDs

We now present a IF-term to uBDD conversion algorithm. Thus, if we can prove this conversion algorithm correct, then we have shown that expressions in our IF term language can be mapped into uBDDs and manipulated using our uBDD functions.

```
(defn qnorm1 (term vars)
  (declare (xargs :measure (acl2-count term)
                  :guard (and (qnorm1-guard term)
                              (eqlable-listp vars))))
  (cond ((eq term t) t)
        ((eq term nil) nil)
        ((atom term) (var-to-tree term vars))
        ((eq (car term) 'if)
         (let ((test (qnorm1 (cadr term) vars)))
           (cond ((eq test t)
                  (qnorm1 (caddr term) vars))
                 ((eq test nil)
                  (qnorm1 (cadddr term) vars))
                 (t (q-ite
                     test
                     (qnorm1 (caddr term) vars)
                     (qnorm1 (cadddr term) vars))))))
        ((eq (car term) 'quote) (cadr term))
        (t (list "Bad arg to qnorm1   a." term))))
```

The function QNORM1 takes an IF term and a variable order, and converts it into a unique uBDD. Given the following TERM-ALL-P recognizer, it should be possible to prove that QNORM1 produces a term recognized by NORMP.

```
(defun term-all-p (term vars)
  (declare (xargs :guard (and (qnorm1-guard term)
                              (eqlable-listp vars))))
  (if (atom term)
      (or (booleanp term)
          (member term vars))
    (let ((fn (car term))
          (args (cdr term)))
      (if (eq fn 'if)
          (and (term-all-p (car args) vars)
               (term-all-p (cadr args) vars)
               (term-all-p (caddr args) vars))
        t))))
```

Problem 19.
Prove or disprove:

```
  (implies (and (qnorm1-guard term)
                (term-all-p term vars))
           (normp (qnorm1 term vars)))
```

4.3 Verifying the Correctness of QNORM1

We now want to prove that QNORM1 correctly translates IF terms into uBDDs. Given that the variables in VARS contain no duplicates, that every variable in the term to be converted appears in VARS, that we have a well-formed IF term, and a Boolean list of VALS, then we want to verify the translation.

Problem 20.
Prove or disprove:

```
(implies (and (qnorm1-guard term)
              (no-duplicatesp vars)
              (term-all-p term vars)
              (boolean-listp vals))
         (equal (eval-bdd (qnorm1 term vars) vals)
                (term-eval term vars vals)))
```

5 The TO-IF Language

Our IF-expression language is also quite limited, so we define the function TO-IF to produce a uBDD if it is able to complete successfully; thus, the TO-IF function is both a recognizer for X the TO-IF language as well as a conversion function. When the expression (TO-IF X) is evaluated either an IF term is returned or a CONS pair is returned with the cars set to a string that may help explain in what sense X is not in the TO-IF language. If X is in the TO-IF language, then (TO-IF x) returns an equivalent member of the TO-IF language expressed in the limited vocabulary of IF, T, NIL, and variables. The result returned is not in any particular normal form, but it is in the form expected by the function QNORM1, which we define later.

Informally, in the TO-IF language, T and NIL both are and denote the Boolean constants. All eqlable ACL2 atoms (i.e., symbols, integers, rational, complex numbers, characters, but not strings) are variables in the TO-IF language. The variables denote Boolean values, i.e., T and NIL. The integer 2 is a variable in the TO-IF language, odd as that may seem at first, but the string "2" is not a TO-IF variable.

In the TO-IF language, (IF x y z) means what Y means if X means T and means what Z means if X means NIL. This is similar to the ACL2 IF function when variables are constrained to be Boolean. In a TO-IF expression one may also use the unary operator NOT and the binary operators AND, OR, IFF, IMPLIES, XOR, NAND, NOR, ANDC1, ANDC2, ORC1, and ORC2.

Before we define the TO-IF function, We first define three helper functions. This function recognizes an expression as being an error.

```
(defn to-if-error-p (x)
  "Recognize error as a pair with a string."
  (and (consp x)
       (stringp (car x))))
```

We next define NMAKE-IF which makes an IF term out of three terms. It performs some basic simplifications, but does not attempt to completely normalize its result.

```
(defn nmake-if (test true false)
  "Partially normalize and simplify IF expressions."
  (declare (xargs :guard (and (good-to-if-p test)
                              (good-to-if-p true)
                              (good-to-if-p false))))
  (cond ((eq test t)
         true)
        ((eq test nil)
         false)
        ;; (IF (IF x NIL T) u v) ==> (IF x v u)
```

```
((and (consp test)
      (eq 'if (car test))
      (null (caddr test))
      (eq t (cadddr test)))
 (nmake-if (cadr test) false true))
;; Simplify true and false branches
(t (let* ((true (if (hqual test true) t true))
          (true (if (and (consp true)
                         (hqual test (cadr true)))
                    (caddr true)
                  true))
          (false (if (hqual test false) nil false))
          (false (if (and (consp false)
                          (hqual test (cadr false)))
                     (cadddr false)
                   false)))
     ;; Two simplifications; otherwise return new term
     (cond ((hqual true false) true)
           ((and (eq true t) (eq false nil))
            test)
           (t (hist 'if test true false)))))))
```

Finally, we define function `TO-IF-SUBST` that everywhere substitutes a new term for a particular existing term in an `IF` term.

```
(defn to-if-subst (new old term)
  "Everywhere substitute existing term by a new term in an IF expression."
  (declare (xargs :guard (good-to-if-p term)))
  (cond ((atom term)
 (cond ((eq term t) t)
((eq term nil) nil)
((equal term old) new)
(t term)))
 (t (hist 'if
  (to-if-subst new old (cadr term))
  (to-if-subst new old (caddr term))
  (to-if-subst new old (cadddr term))))))
```

As a convenience, we have defined a number of synonyms for various logical operators.

```
(defconst *and-synonyms*   '(and & *))
(defconst *or-synonyms*    '(or \| +))
(defconst *iff-synonyms*   '(iff eq eql equal eqv xnor =
                             == equiv <-> <=>))
(defconst *if-synonyms*    '(if ite mux))
(defconst *not-synonyms*   '(not ~))
(defconst *xor-synonyms*   '(xor exor))
(defconst *nand-synonyms*  '(nand))
(defconst *nor-synonyms*   '(nor))
(defconst *andc1-synonyms* '(andc1))
(defconst *andc2-synonyms* '(andc2))
(defconst *orc1-synonyms*  '(orc1 implies -> =>))
(defconst *orc2-synonyms*  '(orc2))
```

```
(defn to-if (term)
  (cond ((atom term)
         (cond ((eqlablep term) term)
               (t (hist "Illegal argument to to-if   a." term))))
        ;; Zero Arguments
        ((to-if-error-p term) term)
        ((not (eqlablep (car term)))
         (hist "Illegal argument to to-if   a." term))
        ((atom (cdr term))
         (cond ((not (null (cdr term)))
                (hist "Illegal argument to to-if   a." term))
               ((member (car term) *and-synonyms*) t)
               ((member (car term) *or-synonyms*) nil)
               (t (hist "Illegal argument to to-if   a." term))))
        ;; One Argument
        ((atom (cddr term))
         (cond ((not (null (cddr term)))
                (hist "Illegal argument to to-if   a." term))
               (t (let ((arg1 (to-if (cadr term))))
                    (cond ((to-if-error-p arg1)
                           (hist "Illegal argument to to-if   a."
                                 term))
                          ((member (car term) *not-synonyms*)
                           (nmake-if arg1 nil t))
                          ((or (member (car term) *and-synonyms*)
                               (member (car term) *or-synonyms*))
                           arg1)
                          (t (hist "Illegal arg to to-if   a."
                                   term)))))))
        ;; Two Arguments
        ((atom (cdddr term))
         (cond ((not (null (cdddr term)))
                (hist "Illegal argument to to-if   a." term))
               (t (let ((arg1 (to-if (cadr term)))
                        (arg2 (to-if (caddr term))))
                    (cond ((to-if-error-p arg1) arg1)
                          ((to-if-error-p arg2) arg2)
                          ((member (car term) *and-synonyms*)
                           (nmake-if arg1 arg2 nil))
                          ((member (car term) *or-synonyms*)
                           (nmake-if arg1 t arg2))
                          ((member (car term) *iff-synonyms*)
                           (nmake-if arg1 arg2 (nmake-if arg2 nil t)))
                          ((member (car term) *orc1-synonyms*)
                           (nmake-if arg1 arg2 t))
                          ((member (car term) *orc2-synonyms*)
                           (nmake-if arg1 (nmake-if arg2 nil t) t))
                          ((member (car term) *andc1-synonyms*)
                           (nmake-if arg2 (nmake-if arg1 nil t) nil))
                          ((member (car term) *andc2-synonyms*)
                           (nmake-if arg1 (nmake-if arg2 nil t) nil))
                          ((member (car term) *xor-synonyms*)
```

```
                  (nmake-if arg1 (nmake-if arg2 nil t) arg2))
                 ((member (car term) *nand-synonyms*)
                  (nmake-if arg1 (nmake-if arg2 nil t) t))
                 ((member (car term) *nor-synonyms*)
                  (nmake-if arg1 nil (nmake-if arg2 nil t)))
                 (t (hist "Illegal arg to to-if   a."
                          term)))))))
;; LET Expression
((and (null (cddddr term)) (eq (car term) 'let))
 (let ((var (cadr term))
       (val (caddr term))
       (body (cadddr term)))
   (cond ((or (not (symbolp var))
              (eq var t)
              (eq var nil))
          (hist "Bad bound variable   a." var))
         (t (let ((valt (to-if val))
                  (bodyt (to-if body)))
              (cond ((to-if-error-p valt) valt)
                    ((to-if-error-p bodyt) bodyt)
                    (t (to-if-subst valt var bodyt))))))))
;; IF Expression
((and (null (cddddr term)) (member (car term) *if-synonyms*))
 (let ((arg1 (to-if (cadr term)))
       (arg2 (to-if (caddr term)))
       (arg3 (to-if (cadddr term))))
   (cond ((to-if-error-p arg1) arg1)
         ((to-if-error-p arg2) arg2)
         ((to-if-error-p arg3) arg3)
         (t (nmake-if arg1 arg2 arg3)))))
(t (list "Illegal argument to to-if   a." term))))
```

This concludes the presentation of the `TO-IF` function. Can the correctness of the `TO-IF` function be established in a like manner as our `IF` expression language converter?

6 Conclusion

We have formally defined two languages and we have proved that our `IF` expression language can be converted into uBDDs. Thus, we can manipulate `IF` terms with the performance advantage of canonical uBDDs.