UBDD Library Enhancements

(and other random stuff)

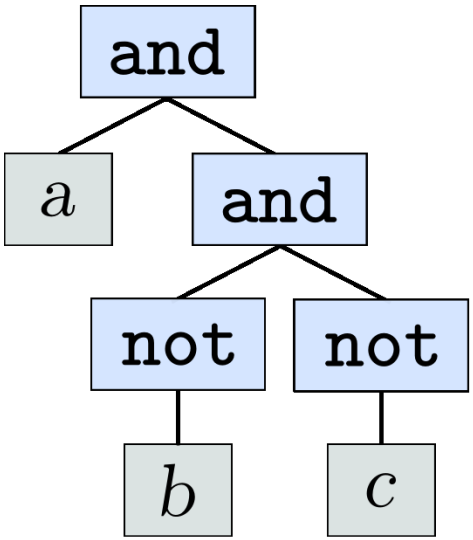
Jared Davis and Sol Swords

Centaur Technology

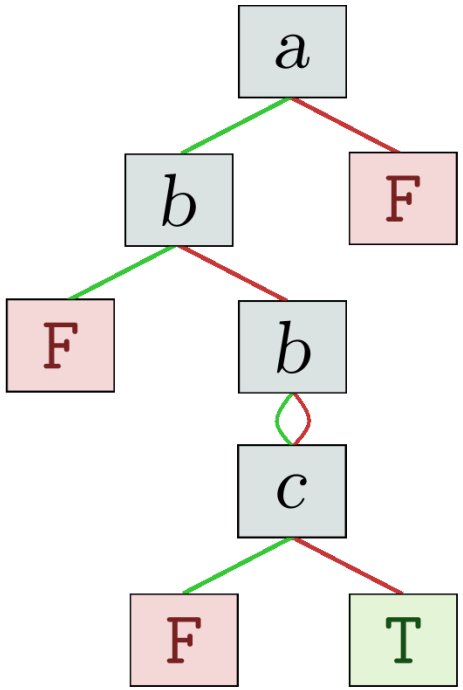
November 5, 2008

Outline

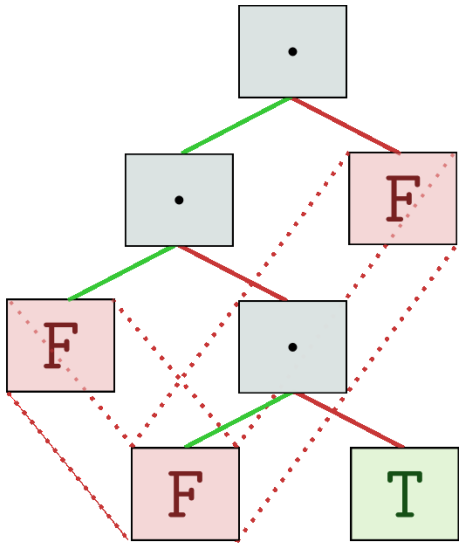
Representations of Boolean functions



S-Expression



Unreduced BDD



UBDD

Interpreting representations of Boolean functions

In most representations, meanings are given by environments mapping variables to values

- `(eval T env) = T`
- `(eval NIL env) = NIL`
- `(eval var env) = (lookup var env)`
- `(eval ‘(and ,a ,b) env) = (and (eval a env) (eval b env))`
- `(eval ‘(or ,a ,b) env) = (or (eval a env) (eval b env))`
- `(eval ‘(not ,a) env) = (not (eval a env))`

Interpreting UBDDs

For UBDDs, meanings are given by list of values telling us to go left or right as we descend

- `(eval-ubdd T vals) = T`
- `(eval-ubdd NIL vals) = NIL`
- `(eval-ubdd ‘(,a . ,b) T::vals) = (eval-ubdd a vals)`
- `(eval-ubdd ‘(,a . ,b) NIL::vals) = (eval-ubdd b vals)`

Canonicity

For UBDDs, the following statements are equivalent

- x = y
- $\forall$ vals : `(eval-ubdd x vals)` = `(eval-ubdd y vals)`

Many Boolean-function representations do not have this property

- `(not a)` vs. `(not (not (not a)))`

Efficiency characteristics

- Expensive to construct
- Cheap to compare (pointer equality)

```
(defun 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))))

(defun q-not (x)
  (if (atom x)
      (if x nil t)
    (hons (q-not (car x))
          (q-not (cdr x)))))
```

```
(defun q-ite (x y z)
  (cond ((null x) z)
        ((atom x) y)
        (t
         (let ((y (if (hons-equal x y) t y))
               (z (if (hons-equal x z) nil z)))
           (cond ((hons-equal y z)
                  y)
                 ((and (eq y t) (eq z nil))
                  x)
                 ((and (eq y nil) (eq z t))
                  (q-not x))
                 (t
                  (qcons
                   (q-ite (car x) (qcar y) (qcar z))
                   (q-ite (cdr x) (qcdr y) (qcdr z)))))))))
```

```
(defun q-and (x y)
  (cond ((atom x)
         (if x
             (if (atom y)
                 (if y t nil)
               y)
           nil))
        ((atom y)
         (if y x nil))
        ((hons-equal x y)
         x)
        (t
         (qcons (q-and (car x) (car y))
                (q-and (cdr x) (cdr y))))))
```

Outline

Opportunistic laziness

Sometimes the result of a function call may be apparent even without evaluating all of its arguments

- `(* (fib x) 0)`
- `(difference nil (mergesort x))`
- `(q-and nil (q-not x))`

Matt has improved MBE to facilitate this

- Defthm has improved awareness of MBE
- Restrictions on nested MBEs have been loosened
- Induction schemes may still have some issues

A simple example: q-ite

Avoid evaluating `y` or `z` when `x` evaluates to a constant

```
(defmacro q-ite (x y z)
  `(mbe :logic (q-ite-fn ,x ,y ,z)
        :exec (let ((_x ,x))
                (cond ((null _x) ,z)
                      ((atom _x) ,y)
                      (t
                       (q-ite-fn _x ,y ,z)))))))

(add-macro-alias q-ite q-ite-fn)
(add-untranslate-pattern (q-ite-fn ?x ?y ?z)
                         (q-ite ?x ?y ?z))
```

Identifying additional opportunities

In `(q-and` x_1 x_2 ... x_n`)`, when any x_i = `NIL` then the answer is `NIL`

Which order should we use?

- `(q-and nil (q-not y))`
- `(q-and (q-not x) y)`
- `(q-and (q-not x) (q-not y))`

Surely cheap

- quoted constants, ("don't need to be evaluated")
- variables, ("already evaluated")

So we evaluate the surely-cheap arguments first

Rulesets

Rulesets are extensible deftheories

- `(include-book "tools/rulesets" :dir :system)`

Defining and extending rulesets

- `(def-ruleset foo '(car-cons cdr-cons))`
- `(add-to-ruleset foo '(default-car default-cdr))`

Enabling and disabling rulesets

- `(in-theory (enable* (:ruleset foo)))`
- `(in-theory (disable* append (:ruleset foo) reverse))`
- `(in-theory (e/d* (reverse member) ((:ruleset foo))))`

Ruleset fanciness

Rulesets can contain pointers to other rulesets

- `(def-ruleset foo '(car-cons))`
- `(def-ruleset bar '(cdr-cons (:ruleset foo)))`

These really are like pointers

- `(add-to-ruleset foo '(append))`
- `(in-theory (disable* (:ruleset bar)))` ;; append is disabled

If you use your own package, it's easy to make `FOO::enable` be an alias to `enable*`, etc.

Make-flag

Make-flag generates a flag function for a mutual-recursion

- Non-executable; multiple-values and stobjs are fine
- Measure inferred from existing definitions
- Efficient proof of equivalence theorem
- Adds a macro for proving new theorems about these functions

Make-flag example

```
(include-book "tools/flag" :dir :system)

(FLAG::make-flag flag-pseudo-termp
                 pseudo-termp
                 :flag-var flag
                 :flag-mapping ((pseudo-termp . term)
                                (pseudo-term-listp . list))
                 :hints(( {for the measure theorem} ))
                 :defthm-macro-name defthm-pseudo-termp)

(defthm-pseudo-termp type-of-pseudo-termp
  (term (booleanp (pseudo-termp x)))
  (list (booleanp (pseudo-term-listp lst)))
  :hints(("Goal" :induct (flag-pseudo-termp flag x lst))))
```

Generalization clause processor

Simple-generalize-cp lets you specify how a clause should be generalized

```
(include-book "clause-processors/generalize" :dir :system)

(defstub foo (x) x)
(defstub bar (x) x)

(thm (equal (foo x) (bar y))
  :hints(("Goal"
          :clause-processor
          (simple-generalize-cp clause '(((bar y) . z))))))

We now apply the verified :CLAUSE-PROCESSOR function SIMPLE-GENERALIZE-
CP to produce one new subgoal.

Goal'
(EQUAL (FOO X) Z).
```

Supporting hint-directed generalization

Tools for generating fresh variables

- `(make-n-vars n root m avoid)`
- `(term-vars x)` and `(term-vars-list x)`

Examples:

```
ACL2 !>(make-n-vars 3 'foo 0 '(x y z foo0 foo1 foo2))
(FOO3 FOO4 FOO5)

ACL2 !>(term-vars '(if x y z))
(X Y Z)
```

Outline

Reasoning about UBDDs

Why do we care?

No ACL2 reasoning is needed for equivalence checking

- Build a UBDD for the circuit (execution)
- Build a UBDD for the specification (execution)
- Check if they are equal (execution)

But there are other, critical uses of UBDDs

- Parameterization — partitions an input space into UBDDs
- AIG conversion — builds a UBDD from an AIG
- G System — represents symbolic objects as lists of UBDDs

The direct approach

The "recursion and induction" approach does not work very well

Some problems

- Finding workable induction schemes
- Case-splits in UBDD construction (`q-car`, `q-cdr`, `q-cons`)

It also "feels wrong"

- Structural, low-level view of Boolean functions
- Not applicable to other representations (AIGs, ...)

Similar to the problem of reasoning about ordered sets

```
(defthm q-and-equiv
  (implies (and (normp x)
                (normp y))
           (equal (q-and x y)
                  (q-ite x y nil))))
```

ACL2 can do the proof directly (0.7s)

- Merges induction schemes of `normp` and `q-ite`
- *1/22 inductive subgoals
- Many subsequent case splits

```
(defun q-xor (x y)
  (cond ((atom x)
         (if x (q-not y) y))
        ((atom y)
         (if y (q-not x) x))
        ((hons-equal x y)
         nil)
        (t
         (qcons (q-xor (car x) (car y))
                (q-xor (cdr x) (cdr y))))))

(defthm q-xor-equiv
  (implies (and (normp x)
                (normp y))
           (equal (q-xor x y)
                  (q-ite x (q-not y) y))))
```

```
Subgoal *1/7.97.164.8'
(IMPLIES (AND (CONSP X)
              Y (CONSP Y)
              (NOT (EQUAL X (Q-NOT Y)))
              (NOT (EQUAL (Q-NOT Y) Y))
              (EQUAL (Q-ITE (CAR X) (CAR (Q-NOT Y)) NIL)
                     T)
              (NOT (EQUAL (Q-ITE (CDR X) (CDR (Q-NOT Y)) (CDR Y))
                          T))
              (NOT (CAR Y))
              (CDR Y)
              (CONSP (CDR Y))
              (EQUAL (Q-XOR (CDR X) (CDR Y))
                     (Q-ITE (CDR X) (Q-NOT (CDR Y)) (CDR Y)))
              (NORMP (CAR X))
              (NORMP (CDR X))
              (CONSP (CAR X))
              (NORMP (CDR Y))
              (NOT (EQUAL (Q-NOT Y) T)))
         (NOT (Q-NOT Y)))
```

Pick-a-point proofs

Prove: $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$

Proof: Let x be an arbitrary element. We will show x is in $(A \cup B) \cap C$ exactly when it is in $(A \cap C) \cup (B \cap C)$.

$$\begin{aligned} x \in (A \cup B) \cap C &\leftrightarrow (x \in A \cup B) \wedge x \in C \\ &\leftrightarrow (x \in A \vee x \in B) \wedge x \in C \end{aligned}$$

$$\begin{aligned} x \in (A \cap C) \cup (B \cap C) &\leftrightarrow x \in A \cap C \vee x \in B \cap C \\ &\leftrightarrow (x \in A \wedge x \in C) \vee (x \in B \wedge x \in C) \\ &\leftrightarrow (x \in A \vee x \in B) \wedge x \in C \end{aligned}$$

Q.E.D.

Pick-a-point proofs of UBDDs

Sets

$$x = y \leftrightarrow \forall a : has(x, a) = has(y, a)$$

UBDDs

$$x = y \leftrightarrow \forall a : eval\text{-}bdd(x, a) = eval\text{-}bdd(y, a)$$

Some familiar set-theory operations

- `NIL`, the empty set
- `T`, the universal set
- `Q-NOT`, set complement
- `Q-AND`, set intersection
- `Q-OR`, set union

Osets-style automation

Suppose `(bdd-lhs)`, `(bdd-rhs)`, and `(bdd-hyp)` satisfy

```
(implies (and (bdd-hyp)
              (normp (bdd-lhs))
              (normp (bdd-rhs)))
         (equal (eval-bdd (bdd-lhs) vals)
                (eval-bdd (bdd-rhs) vals)))
```

Then, we can prove

```
(implies (and (bdd-hyp)
              (normp (bdd-lhs))
              (normp (bdd-rhs)))
         (equal (bdd-lhs) (bdd-rhs)))
```

A default hint functionally instantiates this theorem when our goal is to show two `normp`'s are equal (and other approaches have failed)

Preparing for pick-a-point proofs

For ordered sets

- `(setp (union x y))`
- `(in a (union x y))` = `(in a x)` ∨ `(in a y)`

For UBDDs

- `(normp x)`, `(normp y)` → `(normp (q-or x y))`
- `(eval-bdd (q-or x y) a)` = `(eval-bdd x a)` ∨ `(eval-bdd y a)`

These proofs are done in the "recursion and induction" style
They tend to be easy

```
(add-bdd-fn q-and)

(defthm q-and-equiv
  (implies (and (normp x)
                (normp y))
           (equal (q-and x y)
                  (q-ite x y nil))))
```

```
We now appeal to EQUAL-BY-EVAL-BDDS in an attempt to show that (Q-AND X Y)
and (Q-ITE X Y NIL) are equal because all of their evaluations under
EVAL-BDD are the same.  (You can disable EQUAL-BY-EVAL-BDDS to avoid
this.  See :doc EQUAL-BY-EVAL-BDDS for more details.)

We augment the goal with the hypothesis provided by the :USE hint.
The hypothesis can be derived from EQUAL-BY-EVAL-BDDS via functional
instantiation, provided we can establish the constraint generated;
the constraint can be simplified using case analysis.  We are left
with the following two subgoals.
```

```
Subgoal 2
(IMPLIES (AND (IMPLIES (AND (AND (NORMP X) (NORMP Y))
                            (NORMP (Q-AND X Y))
                            (NORMP (Q-ITE X Y NIL)))
                       (EQUAL (EQUAL (Q-AND X Y) (Q-ITE X Y NIL))
                              T))
              (NORMP X)
              (NORMP Y))
         (EQUAL (Q-AND X Y) (Q-ITE X Y NIL))).
```

But simplification reduces this to T, using the :executable-counterparts of EQUAL and NORMP, primitive type reasoning, the :rewrite rules NORMP-OF-Q-AND and NORMP-OF-Q-ITE and the :type-prescription rule NORMP.

```
Subgoal 1
(IMPLIES (AND (NORMP X)
              (NORMP Y)
              (EQUAL (LEN ARBITRARY-VALUES)
                     (MAX (MAX-DEPTH (Q-AND X Y))
                          (MAX-DEPTH (Q-ITE X Y NIL))))
              (BOOLEAN-LISTP ARBITRARY-VALUES)
              (NORMP (Q-AND X Y))
              (NORMP (Q-ITE X Y NIL)))
         (EQUAL (EVAL-BDD (Q-AND X Y) ARBITRARY-VALUES)
                (EVAL-BDD (Q-ITE X Y NIL) ARBITRARY-VALUES))).
```

But simplification reduces this to T, using the :definition MAX, the :executable-counterpart of NORMP, primitive type reasoning, the :rewrite rules EVAL-BDD-OF-NON-CONSP-CHEAP, EVAL-BDD-OF-Q-AND, EVAL-BDD-OF-Q-ITE, NORMP-OF-Q-AND and NORMP-OF-Q-ITE and the :type-prescription rule NORMP.

Q.E.D.

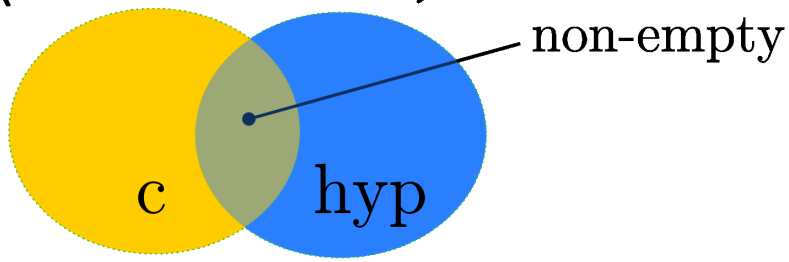
A subset-oriented approach

Our simple pick-a-point approach sometimes led to goals whose hypotheses were difficult to use effectively

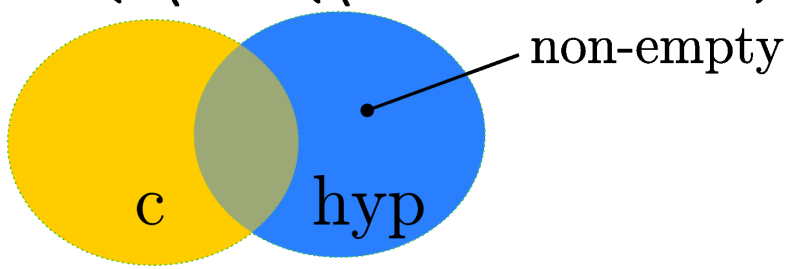
```
(IMPLIES (AND (NORMP C)
              (NORMP HYP)
              (Q-ITE C HYP NIL)
              (NOT (EQUAL (Q-ITE C HYP NIL) HYP))
              HYP
              (NOT (EQUAL C T))
              (NOT (Q-ITE C NIL HYP))
              (NOT (EVAL-BDD C ARBITRARY-VALUES)))
         (NOT (EVAL-BDD HYP ARBITRARY-VALUES))))
```

A graphical view

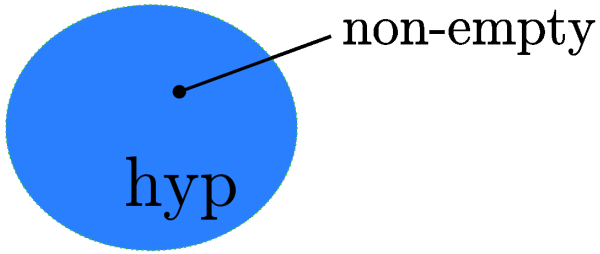
`(Q-ITE C HYP NIL)`



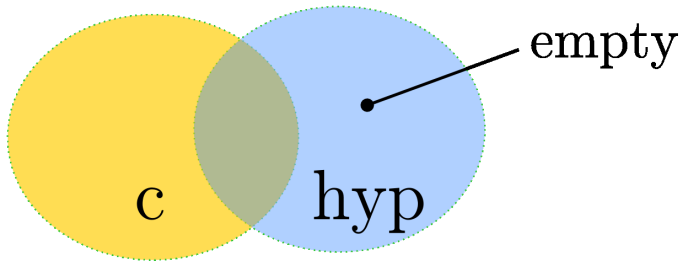
`(NOT (EQUAL (Q-ITE C HYP NIL) HYP))`



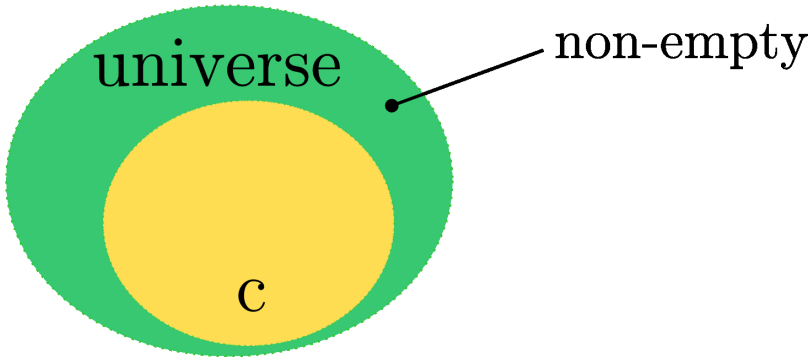
`HYP`



`(NOT (Q-ITE C NIL HYP))`



`(NOT (EQUAL C T))`



Subset mode

`(qs-subset x y)`: $\forall$ `vals` : `(eval-bdd x vals)` $\longrightarrow$ `(eval-bdd y vals)`

- Good properties: reflexive, transitive, membership-preserving
- Similar pick-a-point approach for proving qs-subset

`(QS-SUBSET-MODE T)` – an alternate normal form

- `(equal x y)` $\Rightarrow$ `(qs-subset x y)` $\wedge$ `(qs-subset y x)`
- `(not x)` $\Rightarrow$ `(qs-subset x nil)`
- `x` $\Rightarrow$ `(not (qs-subset x nil))`

- `(qs-subset (q-and x y) x)`
- `(qs-subset (q-and x y) y)`
- `(qs-subset x (q-or x y))`
- `(qs-subset y (q-or x y))`

Rewrite rules for subset mode (without normp hyps)

```
(equal (qs-subset w (q-ite x y z))
       (and (qs-subset (q-ite w x nil) y)
            (qs-subset (q-ite x nil w) z)))

(implies (and (syntaxp (not (equal y ''nil)))
              (syntaxp (not (equal z ''nil))))
         (equal (qs-subset (q-ite x y z) w)
                (and (qs-subset (q-ite x y nil) w)
                     (qs-subset (q-ite x nil z) w))))

(equal (qs-subset (q-ite x nil y) x)
       (qs-subset y x))

(equal (qs-subset (q-ite x nil y) nil)
       (qs-subset y x))
```

Subset mode in action

`(not (equal (q-ite c hyp nil) hyp))` $\longrightarrow$ `(not (qs-subset hyp c))`

```
(not (equal (q-ite c hyp nil) hyp))
==> (not (and 1. (qs-subset (q-ite c hyp nil) hyp)
                 ==> t
              2. (qs-subset hyp (q-ite c hyp nil))))
                 ==> (and 2a. (qs-subset (q-ite hyp c nil) hyp)
                              ==> t
                          2b. (qs-subset (q-ite c nil hyp) nil)))
                              ==> (qs-subset hyp c)
==> (not (qs-subset hyp c))
```

`(not (q-ite c nil hyp))` $\longrightarrow$ `(qs-subset hyp c)`

```
(not (q-ite c nil hyp))
==> (qs-subset (q-ite c nil hyp) nil)
==> (qs-subset hyp c)
```

A witness-oriented approach

Subset-mode often works in practice, but does not seem ideal

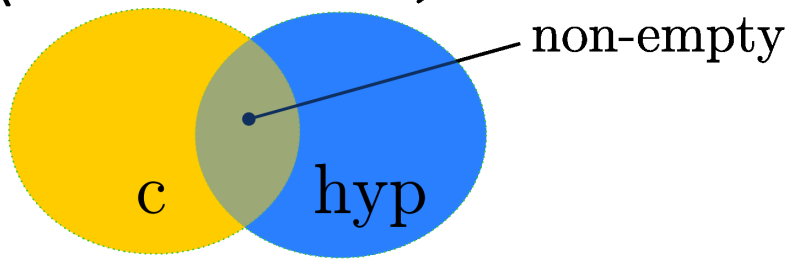
- Strange normal form that affects all booleans
- Strange iff-rewrites needed for all UBDD-making functions
- Free variables in transitivity and the preservation of membership
- Rules about `q-ite` seem somehow fragile

Witness-mode is a more advanced alternative

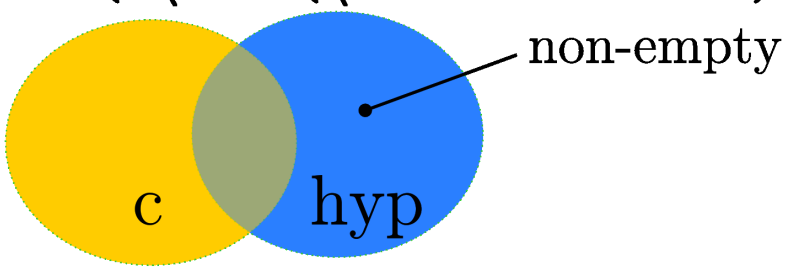
- Intuitively, "Pick all of the probably-relevant points"
- Casts everything in terms of `eval-bdd`
- Works with existing normal forms

The witness approach, graphically

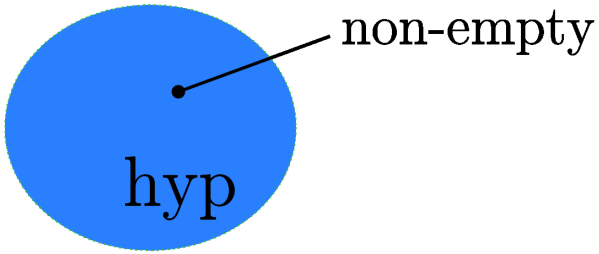
`(Q-ITE C HYP NIL)`



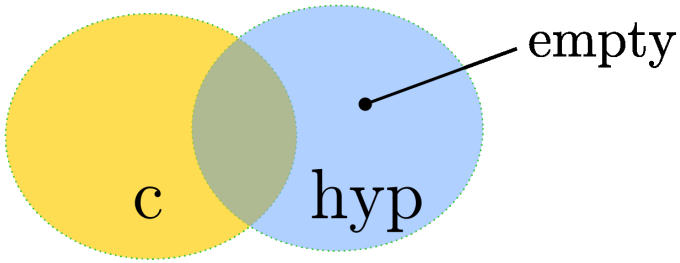
`(NOT (EQUAL (Q-ITE C HYP NIL) HYP))`



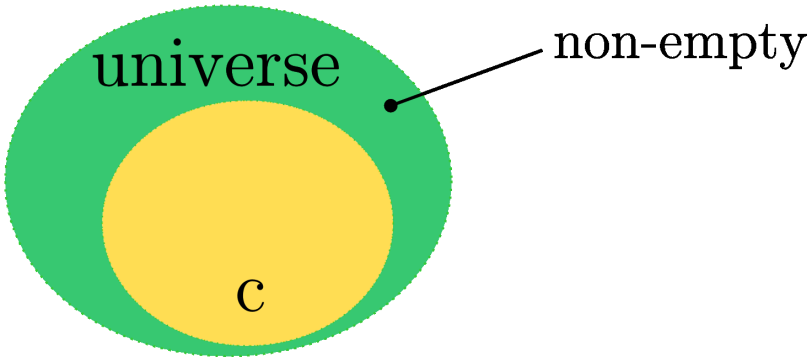
`HYP`



`(NOT (Q-ITE C NIL HYP))`



`(NOT (EQUAL C T))`



The basic transformation

Hypothesis: `x` $\neq$ `y` (or `x`)

- Means $\exists$ `v` : `(eval-bdd x v)` $\neq$ `(eval-bdd y v)`
- Introduce a new variable, `v`
- Replace the hyp with `(eval-bdd x v)` $\neq$ `(eval-bdd y v)`

Hypothesis: `x` $=$ `y` (or `(not x)`)

- Means $\forall$ `v` : `(eval-bdd x v)` $=$ `(eval-bdd y v)`
- Collect all `v` occurring in the clause
- Replace the hyp with `(eval-bdd x v)` $=$ `(eval-bdd y v)`

Transformation example

```
(IMPLIES (AND ;; (NORMP C)
              ;; (NORMP HYP)
              (Q-ITE C HYP NIL)
              (NOT (EQUAL (Q-ITE C HYP NIL) HYP))
              HYP
              (NOT (EQUAL C T))
              (NOT (Q-ITE C NIL HYP))
              (NOT (EVAL-BDD C ARBITRARY-VALUES)))
         (NOT (EVAL-BDD HYP ARBITRARY-VALUES))))
```

```
(IMPLIES (AND (NOT (EQUAL (EVAL-BDD (Q-ITE C HYP NIL) V1)
                          (EVAL-BDD NIL V1)))
              (NOT (EQUAL (EVAL-BDD (Q-ITE C HYP NIL) V2)
                          (EVAL-BDD HYP V2)))
              (NOT (EQUAL (EVAL-BDD HYP V3)
                          (EVAL-BDD NIL V3)))
              (NOT (EQUAL (EVAL-BDD C V4)
                          (EVAL-BDD T V4)))
              (NOT (Q-ITE C NIL HYP))
              (NOT (EVAL-BDD C ARBITRARY-VALUES)))
         (NOT (EVAL-BDD HYP ARBITRARY-VALUES))))
```

Values: `V1`, `V2`, `V3`, `V4`, `ARBITRARY-VALUES`

```
(IMPLIES (AND (NOT (EQUAL (EVAL-BDD (Q-ITE C HYP NIL) V1)
                          (EVAL-BDD NIL V1)))
              (NOT (EQUAL (EVAL-BDD (Q-ITE C HYP NIL) V2)
                          (EVAL-BDD HYP V2)))
              (NOT (EQUAL (EVAL-BDD HYP V3)
                          (EVAL-BDD NIL V3)))
              (NOT (EQUAL (EVAL-BDD C V4)
                          (EVAL-BDD T V4)))

              (EQUAL (EVAL-BDD (Q-ITE C NIL HYP) V1)
                     (EVAL-BDD NIL V1))
              (EQUAL (EVAL-BDD (Q-ITE C NIL HYP) V2)
                     (EVAL-BDD NIL V2))
              (EQUAL (EVAL-BDD (Q-ITE C NIL HYP) V3)
                     (EVAL-BDD NIL V3))
              (EQUAL (EVAL-BDD (Q-ITE C NIL HYP) V4)
                     (EVAL-BDD NIL V4))
              (EQUAL (EVAL-BDD (Q-ITE C NIL HYP) ARBITRARY-VALUES)
                     (EVAL-BDD NIL ARBITRARY-VALUES))

              (NOT (EVAL-BDD C ARBITRARY-VALUES)))
         (NOT (EVAL-BDD HYP ARBITRARY-VALUES))))
```

```
(IMPLIES (AND (EVAL-BDD (Q-ITE C HYP NIL) V1)
              (NOT (EQUAL (EVAL-BDD (Q-ITE C HYP NIL) V2)
                          (EVAL-BDD HYP V2)))
              (EVAL-BDD HYP V3)
              (NOT (EVAL-BDD C V4))

              (NOT (EVAL-BDD (Q-ITE C NIL HYP) V1))
              (NOT (EVAL-BDD (Q-ITE C NIL HYP) V2))
              (NOT (EVAL-BDD (Q-ITE C NIL HYP) V3))
              (NOT (EVAL-BDD (Q-ITE C NIL HYP) V4))
              (NOT (EVAL-BDD (Q-ITE C NIL HYP) ARBITRARY-VALUES))

              (NOT (EVAL-BDD C ARBITRARY-VALUES)))
         (NOT (EVAL-BDD HYP ARBITRARY-VALUES))))
```

Follows from cases introduced by `eval-bdd-of-q-ite`

The eval-bdd-cp clause processor (1/2)

`(diff x y)`

- When `x` $\neq$ `y`, `(eval-bdd x (diff x y))` $\neq$ `(eval-bdd y (diff x y))`

1a.. Gather hyps of the form `x` $\neq$ `y`, where `x`, `y` are (likely) UBDDs

- A hyp which is just `x` also counts: `x` $\neq$ `NIL`

1b.. For each `x` $\neq$ `y` found, replace the hyp with

```
(implies (and (normp x) (normp y)))
         (eval-bdd x (diff x y)) ≠ (eval-bdd y (diff x y))
```

This is sound

- In the `normp` case, the clauses are equivalent
- Otherwise, the new clause implies the original

The eval-bdd-cp clause processor (2/2)

2. As a convenience, generalize away all `(diff x y)` terms just introduced with fresh variables. (trivially sound)

3. Gather up all `v` which are used, anywhere, as arguments to `eval-bdd`, i.e., `(eval-bdd x v)`.

4a. Gather hyps of the form `x` = `y` found, where `x`, `y` are (likely) UBDDs

- A hyp which is `(not x)` also counts: `x` = `NIL`

4b. Replace these hyps with `(eval-bdd x v)` = `(eval-bdd y v)`, for all `v` found in step 3. (trivially sound)

Automating eval-bdd-cp

We use a default hint

- The clause must be `stable-under-simplificationp`
- The definition of `eval-bdd-cp-hint` must be enabled
- The transformation must modify the clause

The hint we give

```
(:or (:clause-processor ...)
     (:no-op t))
```

Future directions

Maybe: A non-UBDD convention, UBDD-fixing, and guards

Names and packages

Similar libraries for AIGs, other representations