Propositional Logic in ACL2

Let's take a bit of break now, and illustrate how we can do the kind of propostional logic reasoning we've done for the last two days in ACL2—well, really, this can be done in any programming language that has Boolean values and conditionals, and lets you define functions, but ACL2 has a nice feature that comes in handy, as we will soon see.

So, how do you write down propositional formulas in ACL2? First off, we already know what the truth values look like: `T` is true, and `NIL` is false.

It is completely straightforward how to define the logical connectives, *once you realize that we have already expressed the logical connectives in terms of an* **if** $-$ **then** $-$ **else** $-$ *connective.*

```
(defun not (x) (if x T NIL))

(defun and (x y) (if x y x))

(defun or (x y) (if x x y))

(defun implies (x y) (if x y T))

(defun iff (x y) (if x (if y T NIL) (if y NIL T)))
```

(As a side note, there was a discussion in class about the right definition of `and`—an alternative was

```
(defun and (x y) (if x y NIL))
```

I'm sorry I was confusing on this point; these two definitions are in fact completely equivalent, and it has nothing to do with short-circuiting evaluation or anything like that. In fact, when the time comes to prove things about ACL2 programs, remind me of this, and we will prove that the two definitions are indeed equivalent. It'll be a good and easy exercise.)

All the above definitions are already provided in ACL2, so you don't need to enter them. And the provided definitions are indeed the ones above—well, okay, that's a slight lie. The definitions of `and` and `or`, while essentially right, are in fact more general. In ACL2, `and` and `or` can take an arbitrary number of arguments:

- `(and x y ... z)` is true exactly when all of `x`, `y`, ..., `z` are true; we also have `(and)` returning `T`, and `(and x)` returning `x`;
- `(or x y ... z)` is true exactly when at least one of `x`, `y`, ..., `z` is true; we also have `(or)` returning `NIL`, and `(or x)` returning `x`.

(We haven't seen a way to define functions with arbitrary arguments, and we won't, it's way outside the scope of what we need for the course. If you're curious, check out *macros* in the reference manual.)

Okay, first bit of trivia now. The above are ACL2 functions, meaning that they must be total, that is, they must return a meaningful value when given arguments outside their intended domain. Clearly, the intended domain of those functions is the Booleans. So what happens when we given an integer or a string to one of those operations? Following what we saw for other functions, we expect the value to be treated as a Boolean. But which one?

The answer is that when a Boolean is expected, primitive operations such as `if` and `cond` will consider a non-Boolean value to be `T`. Thus, the interpretation of `if c th el)` is to evaluate `th` if `c` evaluates to `T` or any non-Boolean value, and to evaluate `el` if `c` evaluates to `NIL`. Similarly for `cond`: if any condition evaluates to something which is not `NIL`, the condition succeeds and the corresponding branch is evaluated.

So, we can check that `(and 0 0)` returns `0`, since `0` is treated as true, and the definition of `and` says that we should return the second argument in that case. Similarly, `or "hello" 10)` returns `"hello"`, since `"hello"` is treated as true, and the definition of `or` says to return that value. Note that `and`, `or`, and `implies` are not guaranteed to return Booleans if given non-Boolean values, while `not` and `iff` always return Boolean values.

So what can we do with these operations? Well, first, we can evaluate their truth value at various truth assignments. For instance, consider the formula $a \wedge (a \Rightarrow b) \equiv a \wedge b$, which you can verify is valid directly using a truth table or indirectly using an equational proof. The formula in ACL2 is written

```
(iff (and a (implies a b)) (and a b))
```

By itself, this is meaningless, because `a` and `b` are unbound variables. But if we bind them to truth values, we can evaluate the formula. We will bind truth values using `let`, which is a way to bind values locally. (`let` is like `local`, except it cannot be used to define local functions.)

```
(let ((a T)
      (b NIL))
    (iff (and a (implies a b)) (and a b)))
```

which evaluates to `T`, as expected. In fact, not matter what truth assignment you use, the formula should evaluate to `T`. (Why?)

So how can you check if a formula is valid? Easy: we simply check that every truth assignment makes the formula return `T`. It's a bit of a pain to this by hand, though.[1] But this is where ACL2 differs from other programming languages. You can *ask* it whether a formula of propositional logic is valid, and it will tell you yes or no. It will do so by essentially doing a mix of truth tables and equational proof, it turns out, although you won't get to see the details. To check if a formula is valid, you query it using `thm`. Let's look at an example, with some sample output from ACL2. First, what happens when we check a formula that is indeed valid. (Note that the details of what gets output differ slightly from version to version, but the essence of it is the same.)

```
ACL2 !>(thm (iff (and a (implies a b)) (and a b)))

But we reduce the conjecture to T, by case analysis.

Q.E.D.

Summary
Form:  ( THM ...)
Rules: NIL
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)

Proof succeeded.
```

All nice and positive, a pleasant green in Eclipse. This tells you that the system determined that the formula was valid. What about a formula that is not valid?

```
ACL2 !>(thm (implies a b))

This simplifies, using trivial observations, to

Goal'
NIL.


Summary
Form:  ( THM ...)
Rules: NIL
Warnings:  None
Time:  0.04 seconds (prove: 0.03, print: 0.01, other: 0.00)
```

[1] If you are bored one day, write a function that loops through all the possible assignments, and checks that evaluating the formula on each truth assignment returns `T`.

```
---
The key checkpoint goal, below, may help you to debug this failure.
See :DOC failure and see :DOC set-checkpoint-summary-limit.
---

*** Key checkpoint at the top level: ***

Goal'
NIL

******** FAILED ********  See :DOC failure  ******** FAILED ********
```

Negativity galore—failed, failure. The system was not able to prove validity, meaning that the formula is not valid. (When checking for validity of propositional logic formulas, if the system fails to show validity, then the formula is not valid.)

Note that when a formula cannot be proved valid, ACL2 *does not* give you a counterexample. So it's a bit like our equational proof—you can use them to check validity, but you don't get a counterexample out. Only truth tables, of the techniques we've seen, can be used to get a counterexample with a formula that is not valid.

I suggest you use ACL2 as an oracle to verify your results when you try to determine if a formula is valid or not.