Reasoning About Programs

Panagiotis Manolios
Northeastern University

January 26, 2017
Version: 97

1 Introduction

These lecture notes were developed for Logic and Computation, a freshman-level class taught at the College of Computer and Information Science of Northeastern University. Starting in Spring 2008, this is a class that all students in the college are required to take.

The goals of the Logic and Computation course are to provide an introduction to formal logic and its deep connections to computing. Logic is presented from a computational perspective using the ACL2 Sedan theorem proving system. The goal of the course is to introduce fundamental, foundational methods for modeling, designing, specifying and reasoning about computation. The topics covered include propositional logic, recursion, contracts, testing, induction, equational reasoning, termination analysis, term rewriting, and various proof techniques. We show how to use logic to formalize the syntax and semantics of the core ACL2s language, a simple LISP-based language with contracts. We then use the ACL2s language to formally reason about programs, to model systems at various levels of abstraction, to design and specify interfaces between systems and to reason about such composed systems. We also examine decision procedures for fragments of first-order logic and how such decision procedures can be used to analyze models of systems.

The students taking the Logic and Computation class have already taken a programming class in the previous semester, in Racket. The course starts by reviewing some basic programming concepts. The review is useful because at the freshman level students benefit from seeing multiple presentations of key concepts; this helps them to internalize these concepts. For example, in past semesters I have asked students to write very simple programs (such as a program to append two lists together) during the first week of classes and a surprisingly large number of students produce incorrect code.

During the programming review, we introduce the ACL2s language. This is the language we use throughout the semester and it is similar to Racket. The syntax and semantics of the core ACL2s language are presented in a mathematical way. We provide enough information so that students can determine what sequence of glyphs form a well-formed expression and how to formally evaluate well-formed expressions potentially containing user-defined functions with constants as arguments (this is always in a first-order setting). This is a pretty big jump in rigor for students and is advanced material for freshmen students, but they already have great intuitions about evaluation from their previous programming class. This intuition helps them understand the rigorous presentation of the syntax and semantics, which in turns helps strengthen their programming abilities.

The lecture notes are sparse. It would be great to add more exercises, but I have not done that yet. Over the course of many years, we have amassed a large collection of homework problems, so students see lots of exercises, and working through these exercises is a great way for them to absorb the material, but the exercises are not in the notes. You can think of the lecture notes as condensed notes for the course that are appropriate for someone who knows the material as a study guide. The notes can also be used as a starting point by students, who should mark them up with clarifications as needed when they attend lectures. I advise students to read the lecture notes before class. This way, during class they can focus on the lecture instead of taking notes and they are better prepared to ask for clarifications.

When I started teaching the class, I used the ACL2 book, *Computer-Aided Reasoning, An Approach* by Kaufmann, Manolios and Moore. However, over the years I became convinced that using an untyped first-order logic was not the optimal way of introducing logic and computation to students because they come in with a typed view of the world. That's not to say they have seen type theory; they have not. But, they are surprised when a programming language allows them to subtract a string from a rational number. Therefore, with the help of my Ph.D. student Harsh Chamarthi, I have focused on adding type-like capabilities to ACL2s. Most notably, we added a new data definition framework to ACL2s that supports enumeration, union, product, record, map, (mutually) recursive and custom types, as well as limited forms of parametric polymorphism. We also introduced the `defunc` macro, which allows us to formally specify input and output contracts for functions. These contracts are very general, *e.g.*, we can specify that `/` is given two rationals as input, and that the second rational is not 0, we can specify that `zip` is given two lists of the same length as input and returns a list of the same length as output and so on. Contracts are also checked statically, so ACL2s will not accept a function definition unless it can prove that the function satisfies its contracts and that for every legal input and every possible computation, it is not possible during the evaluation of the function being defined to be in a state where some other function is poised to be evaluated on a value that violates its input contract. I have found that a significant fraction of erroneous programs written by students have contract violations in them, and one of the key things I emphasize is that when writing code, one needs to think carefully about the contracts of the functions used and why the arguments to every function call satisfy the function's contract. Contracts are the first step towards learning how to specify interfaces between systems. With the move to contracts, the ACL2 book became less and less appropriate, which led me to write these notes.

I have distributed these notes to the students in Logic and Computation for several years and they have found lots of typos and have made many suggestions for improvement. Thanks and keep the comments coming!

3

Propositional Logic

The study of logic was initiated by the ancient Greeks, who were concerned with analyzing the laws of reasoning. They wanted to fully understand what *conclusions* could be derived from a given set of *premises*. Logic was considered to be a part of philosophy for thousands of years. In fact, until the late 1800's, no significant progress was made in the field since the time of the ancient Greeks. But then, the field of modern mathematical logic was born and a stream of powerful, important, and surprising results were obtained. For example, to answer foundational questions about mathematics, logicians had to essentially create what later became the foundations of computer science. In this class, we'll explore some of the many connections between logic and computer science.

We'll start with propositional logic, a simple, but surprisingly powerful fragment of logic. Expressions in propositional logic can only have one of two values. We'll use T and F to denote the two values, but other choices are possible, *e.g.*, 1 and 0 are sometimes used.

The expressions of propositional logic include:

1. The *constant expressions true* and *false*: they always evaluate to T and F, respectively.

2. The *propositional atoms*, or more succinctly, *atoms*. We will use p, q, and r to denote propositional atoms. Atoms range over the values T and F.

Propositional expressions can be combined together with the propositional operators, which include the following.

The simplest operator is negation. Negation, $\neg$, is a *unary* operator, meaning that it is applied to a single expression. For example $\neg p$ is the negation of atom p. Since p (or any propositional expression) can only have one of two values, we can fully define the meaning of negation by specifying what it does to the value of p in these two cases. We do that with the aid of the following truth table.

p	$\neg p$
T	F
F	T

What the truth table tells us is that if we negate T we get F and if we negate F we get T.

Negation is the only unary propositional operator we are going to consider. Next we consider *binary* (2-argument) propositional operators, starting with *conjunction*, $\wedge$. The conjunction (and) of p and q is denoted $p \wedge q$ and its meaning is given by the following truth table.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

Each row in a truth table corresponds to an *assignment*, one possible way of assigning values (T or F) to the atoms of a formula. The truth table allows us to explore all relevant assignments. If we have two atoms, there are 4 possibilities, but in general, if we have n atoms, there are 2^n possible assignments we have to consider.

In one sense, that's all there is to propositional logic, because every other operator we are going to consider can be expressed in terms of $\neg$ and $\wedge$, and almost every question we are going to consider can be answered by the construction of a truth table.

Next, we consider *disjunction*. The disjunction (or) of p and q is denoted $p \vee q$ and its meaning is given by the following truth table.

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

In English usage, "p or q" often means p or q, but not both. Consider the mother who tells her child:

> You can have ice cream or a cookie.

The child is correct in assuming this means that she can have ice cream or a cookie, but not both.

As you can see from the truth table for disjunction, in logic "or" always means at least one.

We can write more complex formulas by using several operators. An example is $\neg p \vee \neg q$, where we use the convention that $\neg$ binds more tightly than any other operator, hence we can only parse the formula as $(\neg p) \vee (\neg q)$. We can construct truth tables for such expressions quite easily. First, determine how many distinct atoms there are. In this case there are two; that means we have four rows in our truth table. Next we create a column for each atom and for each operator. Finally, we fill in the truth table, using the truth tables that specify the meaning of the operators.

p	q	$\neg p$	$\neg q$	$\neg p \vee \neg q$
T	T	F	F	F
T	F	F	T	T
F	T	T	F	T
F	F	T	T	T

Next, we consider implication, $\Rightarrow$. This is called logical (or material) implication. In $p \Rightarrow q$, p is the antecedent and q is the consequent. Implication is often confusing to students because the way it is used in English is quite complicated and subtle. For example, consider the following sentences.

> If Obama invented the Internet, then the inhabitants of Boston are all dragons.

Is it true?
What about the following?

> If Obama was elected president, then the inhabitants Tokyo are all descendants of Godzilla.

Logically, only the first is true, but most English speakers will say that if there is no connection between the antecedent and consequent, then the implication is false.

Why is the first logically true? Because here is the truth table for implication.

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Here are two ways of remembering this truth table. First, $p \Rightarrow q$ is equivalent to $\neg p \vee q$. Second, $p \Rightarrow q$ is false only when p is T, but q is F. This is because you should think of $p \Rightarrow q$ as claiming that if p holds, so does q. That claim is true when p is F. The claim can only be invalidated if p holds, but q does not.

As a final example of the difference between logical implication (whose meaning is given by the above truth table) and implication as commonly used, consider a father telling his child:

> If you behave, I'll get you ice cream.

The child rightly expects to get ice cream if she behaves, but also expects to *not* get ice cream if she doesn't: there is an implied threat here.

The point is that the English language is subtle and open for interpretation. In order to avoid misunderstandings, mathematical fields, such as Computer Science, tend to use what is often called "mathematical English," a very constrained version of English, where the meaning of all operators is clear.

Above we said that $p \Rightarrow q$ is equivalent to $\neg p \vee q$. This is the first indication that we can often reduce propositional expressions to simpler forms. If by simpler we mean less operators, then which of the above is simpler?

Can we express the equivalence in propositional logic? Yes, using equality of Booleans, $\equiv$, as follows $(p \Rightarrow q) \equiv (\neg p \vee q)$.

Here is the truth table for $\equiv$.

p	q	$p \equiv q$
T	T	T
T	F	F
F	T	F
F	F	T

How would you simplify the following?

1. $p \wedge \neg p$

2. $p \vee \neg p$

3. $p \equiv p$

Here is one way.

1. $(p \wedge \neg p) \equiv \mathit{false}$

2. $(p \vee \neg p) \equiv \mathit{true}$

3. $(p \equiv p) \equiv \mathit{true}$

The final binary operator we will consider is $\oplus$, xor. There are two ways to think about xor. First, note that xor is e*x*clusive *or*, meaning that exactly one of its arguments is true. Second, note that xor is just the Boolean version of not equal. Here is the truth table for $\oplus$.

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

To avoid using too many parentheses, from now on we will follow the convention: $\neg$ binds tightest, followed by $\{\wedge, \vee\}$, followed by $\Rightarrow$, followed by $\{\oplus, \equiv\}$. Hence, instead of

$$((p \vee (\neg q)) \Rightarrow r) \oplus ((\neg r) \Rightarrow (q \wedge (\neg p)))$$

we can write

$$p \vee \neg q \Rightarrow r \oplus \neg r \Rightarrow q \wedge \neg p$$

We will also consider a *ternary* operator, *i.e.*, a operator with three arguments. The operator is *ite*, which stands for "if-then-else," and means just that: if the first argument holds, return the second (the then branch), else return the third (the else branch). Since there are three arguments, there are eight rows in the truth table.

p	q	r	$\mathit{ite}(p, q, r)$
T	T	T	T
T	T	F	T
T	F	T	F
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	F

Here are some very useful ways of characterizing propositional formulas. Start by constructing a truth table for the formula and look at the column of values obtained. We say that the formula is:

♦ *satisfiable* if there is at least one T

- ♦ *unsatisfiable* if it is not satisfiable, *i.e.*, all entries are F
- ♦ *falsifiable* if there is at least one F
- ♦ *valid* if it is not falsifiable, *i.e.*, all entries are T

We have seen examples of all of the above. For example, $p \wedge q$ is satisfiable, since the assignment that makes p and q both T results in $p \wedge q$ also being T. This example is also falsifiable, as evidenced by the assignment that makes p F and q T. An example of an unsatisfiable formula is $p \wedge \neg p$. If you construct the truth table for it, you will notice that every assignment makes it F (so it is falsifiable too). Finally, an example of a valid formula is $p \vee \neg p$.

Notice that if a formula is valid, then it is also satisfiable. In addition, if a formula is unsatisfiable, then it is also falsifiable.

Validity turns out to be really important. A valid formula, often also called a *theorem*, corresponds to a correct logical argument, an argument that is true regardless of the values of its atoms. For example $p \Rightarrow p$ is valid. No matter what p is, $p \Rightarrow p$ always holds.

3.1 P = NP

A natural question arises at this point: Is there an algorithm that given a propositional logic formula returns "yes" if it is satisfiable and "no" otherwise?

Here is an algorithm: construct the truth table. If we have the truth table, we can easily decide satisfiability, validity, unsatisfiability, and falsifiability.

The problem is that the algorithm is inefficient. The number of rows in the truth table is 2^n, where n is the number of atoms in our formula.

Can we do better? For example, recall that we had an inefficient recursive algorithm for `sum-n` (the function that given a natural number n, returns $\sum_{i=0}^{n} i$). The function required n additions, which is exponential in the number of bits needed to represent n ($\log n$ bits are needed). However, with a little math, we found an algorithm that was efficient because it only needed 3 arithmetic operations.

Is there an efficient algorithm for determining Boolean satisfiability? By efficient, we mean an algorithm that in the worst case runs in polynomial time. Gödel asked this question in a letter he wrote to von Neumann in 1956. No one knows the answer, although this is one of the most studied questions in computer science. In fact, most of the people who have thought about this problem believe that no polynomial time algorithm for Boolean satisfiability exists.

3.2 The Power of Xor

Let us take a short detour, I'll call "the power of xor."

Suppose that you work for a secret government agency and you want to communicate with your counterparts in Europe. You want the ability to send messages to each other using the Internet, but you know that other spy agencies are going to be able to read the messages as they travel from here to Europe.

How do you solve the problem?

Well, one way is to have a shared secret: a long sequence of F's and T's (0's and 1's if you prefer), in say a code book that only you and your counterparts have. Now, all messages are really just sequences of bits, which we can think of as sequences of F's and T's, so you take your original message m and xor it, bit by bit, with your secret s. That gives rise to coded message c, where $c \equiv m \oplus s$. Notice that here we are applying $\equiv$ and $\oplus$ to sequences of Boolean values, often called *bit-vectors*.

Anyone can read c, but they will have no idea what the original message was, since s effectively scrambled it. In fact, with no knowledge of s, an eavesdropper can extract no information about the contents of m from c, except for the length of the message, which can be partially hidden by padding the message with extra bits.

But, how will your counterparts in Europe decode the message? Notice that some propositional reasoning shows that $m = c \oplus s$, so armed with your shared secret, they can determine what the message is.

This is one of the most basic encryption methods. It provides extremely strong security but is difficult to use because it requires sharing a secret. While sharing a key might be feasible for government agencies, it is *not* feasible for you and all the companies you buy things from on the Internet.

The shared secret should be a random sequence of bits and once bits of the secret key are used, they should never be used again. Why? This method is called the one-time pad method.

Exercise 3.1 *Show that this scheme is secure. Here's how. Show that for any coded message c of length l, if an adversary only knows c (but not m and not s), then for* any *m (of length l), there exists a secret s (of length l) such that $c = m \oplus s$.*

Exercise 3.2 *If s, the secret key, is not a random sequence, why is this a bad idea? For example, what if s is all 0's or all 1's?*

Exercise 3.3 *If you keep reusing s, the secret key, why is this a bad idea?*

Is this a reasonable way to exchange information? Well you have probably seen movies with the "red telephone" that connects the Pentagon with the Kremlin. While a red telephone was never actually used, there *was* a system in place to allow Washington to directly and securely communicate with Moscow. The original system used encrypted teletype messages based on one-time pads. The countries exchanged keys at their embassies.

3.3 Useful Equalities

Here are some simple equalities involving the constant *true*.

1. $p \vee true \equiv true$

2. $p \wedge true \equiv p$

3. $p \Rightarrow true \equiv true$

4. $true \Rightarrow p \equiv p$

5. $p \equiv true \equiv p$

6. $p \oplus \mathit{true} \equiv \neg p$

Here are some simple equalities involving the constant *false*.

1. $p \vee \mathit{false} \equiv p$
2. $p \wedge \mathit{false} \equiv \mathit{false}$
3. $p \Rightarrow \mathit{false} \equiv \neg p$
4. $\mathit{false} \Rightarrow p \equiv \mathit{true}$
5. $p \equiv \mathit{false} \equiv \neg p$
6. $p \oplus \mathit{false} \equiv p$

Why do we have separate entries for $p \Rightarrow \mathit{false}$ and $\mathit{false} \Rightarrow p$, above, but not for both $p \vee \mathit{false}$ and $\mathit{false} \vee p$? Because $\vee$ is commutative. Here are some equalities involving commutativity.

1. $p \vee q \equiv q \vee p$
2. $p \wedge q \equiv q \wedge p$
3. $p \equiv q \equiv q \equiv p$
4. $p \oplus q \equiv q \oplus p$

What about $\Rightarrow$. Is it commutative? Is $p \Rightarrow q \equiv q \Rightarrow p$ valid? No. By the way, the right-hand side of the previous equality is called the *converse*: it is obtained by swapping the antecedent and consequent.

A related notion is the *inverse*. The inverse of $p \Rightarrow q$ is $\neg p \Rightarrow \neg q$. Note that the inverse and converse of an implication are equivalent.

Even though a conditional is not equivalent to its inverse, it is equivalent to its *contrapositive*:

$$(p \Rightarrow q) \equiv (\neg q \Rightarrow \neg p)$$

The contrapositive is obtained by negating the antecedent and consequent and then swapping them.

While we're discussing implication, a very useful equality involving implication is:

$$(p \Rightarrow q) \equiv (\neg p \vee q)$$

Also, we often want to replace $\equiv$ by $\Rightarrow$, which is possible due to the following equality:

$$(p \equiv q) \equiv [(p \Rightarrow q) \wedge (q \Rightarrow p)]$$

Here are more equalities.

1. $\neg\neg p \equiv p$
2. $\neg \mathit{true} \equiv \mathit{false}$
3. $\neg \mathit{false} \equiv \mathit{true}$

4. $p \wedge p \equiv p$
5. $p \vee p \equiv p$
6. $p \Rightarrow p \equiv \mathit{true}$
7. $p \equiv p \equiv \mathit{true}$
8. $p \oplus p \equiv \mathit{false}$
9. $p \wedge \neg p \equiv \mathit{false}$
10. $p \vee \neg p \equiv \mathit{true}$
11. $p \Rightarrow \neg p \equiv \neg p$
12. $\neg p \Rightarrow p \equiv p$
13. $p \equiv \neg p \equiv \mathit{false}$
14. $p \oplus \neg p \equiv \mathit{true}$

Here's one set of equalities you have probably already seen: DeMorgan's Laws.

1. $\neg(p \wedge q) \equiv \neg p \vee \neg q$
2. $\neg(p \vee q) \equiv \neg p \wedge \neg q$

Here's another property: associativity.

1. $((p \vee q) \vee r) \equiv (p \vee (q \vee r))$
2. $((p \wedge q) \wedge r) \equiv (p \wedge (q \wedge r))$
3. $((p \equiv q) \equiv r) \equiv (p \equiv (q \equiv r))$
4. $((p \oplus q) \oplus r) \equiv (p \oplus (q \oplus r))$

We also have distributivity:

1. $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
2. $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

We also have transitivity:

1. $[(p \Rightarrow q) \wedge (q \Rightarrow r)] \Rightarrow (p \Rightarrow r)$
2. $[(p \equiv q) \wedge (q \equiv r)] \Rightarrow (p \equiv r)$

Last, but not least, we have absorption:

1. $p \wedge (p \vee q) \equiv p$
2. $p \vee (p \wedge q) \equiv p$

Let's consider absorption more carefully. Here is a simple calculation:

Proof

$\quad p \wedge (p \vee q)$

$\equiv$ { Distribute $\wedge$ over $\vee$ }

$\quad (p \wedge p) \vee (p \wedge q)$

$\equiv$ { $(p \wedge p) \equiv p$ }

$\quad p \vee (p \wedge q)$ □

The above proof shows that $p \wedge (p \vee q) \equiv p \vee (p \wedge q)$, so if we show that $p \wedge (p \vee q) \equiv p$, we will have also shown that $p \vee (p \wedge q) \equiv p$.

3.4 Proof Techniques

Let's try to show that $p \wedge (p \vee q) \equiv p$. We will do this using a proof technique called case analysis.

Case analysis: If f is a formula and p is an atom, then f is valid iff both $f|_{((p\ \mathit{true}))}$ and $f|_{((p\ \mathit{false}))}$ are valid. By $f|_{((p\ x))}$ we mean, substitute x for p in f.

Proof

$\quad p \wedge (p \vee q) \equiv p$

$\equiv$ { Case analysis }

$\quad \mathit{true} \wedge (\mathit{true} \vee q) \equiv \mathit{true}$ and $\mathit{false} \wedge (\mathit{false} \vee q) \equiv \mathit{false}$

$\equiv$ { Basic Boolean equalities }

$\quad \mathit{true} \equiv \mathit{true}$ and $\mathit{false} \equiv \mathit{false}$

$\equiv$ { Basic Boolean equalities }

$\quad \mathit{true}$ □

Another useful proof technique is *instantiation*: If f is a valid formula, then so is $f|_\sigma$, where σ is a *substitution*, a list of the form:

$$((\mathit{atom}_1\ \mathit{formula}_1) \cdots (\mathit{atom}_n\ \mathit{formula}_n)),$$

where the atoms are "target atoms" and the formulas are their images. The application of this substitution to a formula uniformly replaces every free occurrence of a target atom by its image.

Here is an example of applying a substitution. $(a \vee \neg(a \wedge b))|_{((a\ (p \vee q))(b\ a))} = ((p \vee q) \vee \neg((p \vee q) \wedge a))$.

Here is an application of instantiation. $a \vee \neg a$ is valid, so therefore, so is $(a \vee \neg a)|_{((a\ (p \vee (q \wedge r))))} = (p \vee (q \wedge r)) \vee \neg(p \vee (q \wedge r))$.

3.5 Normal Forms and Complete Boolean Bases

We have seen several propositional operators, but do we have any assurance that the operators are complete? By complete we mean that the propositional operators we have can be used to represent any Boolean function.

How do we prove completeness?

Consider some arbitrary Boolean function f over the atoms $x_1, \ldots, x_n$. The domain of f has 2^n elements, so we can represent the function using a truth table with 2^n rows. Now the question becomes: can we represent this truth table using the operators we already introduced?

Here is the idea of how we can do that. Take the disjunction of all the assignments that make f true. The assignments that make f true are just the rows in the truth table for which f is T. Each such assignment can be represented by a *conjunctive clause*, a conjunction of *literals*, atoms or their negations. So, we can represent each of these assignments. Now to represent the function, we just take the disjunction of all the conjunctive clauses.

Consider what happens if we try to represent $a \oplus b$ in this way. There are two assignments that make $a \oplus b$ true and they can be represented by the conjunctive clauses $a \wedge \neg b$ and $\neg a \wedge b$, so $a \oplus b$ can be represented as the disjunction of these two conjunctive clauses: $(a \wedge \neg b) \vee (\neg a \wedge b)$.

Notice that we only need $\neg, \vee$, and $\wedge$ to represent any Boolean function!

The formula we created above was a disjunction of *conjunctive clauses*. Formulas of this type are said to be in *disjunctive normal form* (DNF). If each conjunctive clause includes all the atoms in the formula, then we say that the formula is in *full disjunctive normal form.* Another type of normal form is *conjunctive normal form* (CNF): each formula is a conjunction of *disjunctive clauses*, where a disjunctive clause is a disjunction of literals. Disjunctive clauses are also just called *clauses.* If each clause includes all the atoms in the formula, then we say that the formula is in *full conjunctive normal form.*

Can you come up with a way of representing an arbitrary truth table in full CNF? (Hint: Consider what we did for DNF.)

Any formula can be put in DNF or CNF. In fact, the input format for modern SAT solvers is CNF, so if you want to check the satisfiability of a Boolean formula using a SAT solver, you have to transform the formula so that it is in CNF.

Exercise 3.4 *You are given a Boolean formula and your job is to put it in CNF and DNF. How efficiently can this be done?*

Hint: Consider formulas of the form

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$$

$$(x_1 \vee y_1) \wedge (x_2 \vee y_2) \wedge \cdots \wedge (x_n \vee y_n)$$

Back to completeness. We saw that these three operators are already complete: $\neg, \vee, \wedge$. Can we do better? Can we get rid of some of them?

We can do better because $\vee$ can be represented using only $\wedge, \neg$. Similarly $\wedge$ can be represented using only $\vee, \neg$. How?

Exercise 3.5 *Can we do better yet? No. $\neg$ is not complete; neither is $\wedge$; neither is $\vee$. Prove it.*

Next, think about this claim: you can represent all the Boolean operators using just *ite* (and the constants *false*, *true*). If we can represent $\neg$ and $\vee$, then as the previous discussion shows, we're done.

$$\neg p \equiv ite(p, false, true)$$

$$p \vee q \equiv ite(p, true, q)$$

Exercise 3.6 *Represent the rest of the operators using ite.*

Exercise 3.7 *There are 16 binary Boolean operators. Are any of them complete? If so, exhibit such a operator and prove that it is complete. If not, prove that none of the operators is complete.*

3.6 Decision Procedures

When a formula is not valid, it is falsifiable, so there exists an assignment that makes it false. Such an assignment is often called a *counterexample* and can be very useful for debugging purposes. Since ACL2s is a programming language, we can use it to write our own decision procedure that provides counterexamples to validity.

Exercise 3.8 *Write a decision procedure for validity in ACL2s.*

While we are on the topic of decision procedures, it is worth pointing that we characterized formulas as satisfiable, unsatisfiable, valid, or falsifiable. Let's say we have a decision procedure for one of these four characterizations. Then, we can, rather trivially, get a decision procedure for any of the other characterizations.

Why?

Well, consider the following.

Proof

Unsat f

$\equiv$ { By the definition of sat, unsat }

not (Sat f)

$\equiv$ { By definition of Sat, Valid }

Valid $\neg f$

$\equiv$ { By definition of valid, falsifiable }

not (Falsifiable $\neg f$) □

How do we use these equalities to obtain a decision procedure for either of unsat, sat, valid, falsifiable, given a decision procedure for the other?

Well, let's consider an example. Say we want a decision procedure for validity given a decision procedure for satisfiability.

Valid f

$\equiv$ { not (Sat f) $\equiv$ $Valid \neg f$, by above }

not (Sat $\neg f$)

What justifies this step? Propositional reasoning and instantiation.

Let p denote "(Sat f)" and q denote "(Valid $\neg f$)." The above equations tell us $\neg p \equiv q$, so $p \equiv \neg q$.

If more explanation is required, note that $(\neg p \equiv q) \equiv (p \equiv \neg q)$ is valid. That is, you can transfer the negation of one argument of an equality to the other.

Make sure you can do this for all 12 combinations of starting with a decision procedure for sat, unsat, valid, falsifiable, and finding decision procedures for the other three characterizations.

There are two interesting things to notice here.

First, we took advantage of the following equality:

$$(\neg p \equiv q) \equiv (p \equiv \neg q)$$

There are lots of equalities like this that you should know about, so study the provided list.

Second, we saw that it was useful to extract the propositional skeleton from an argument. We'll look at examples of doing that. Initially this will involve word problems, but later it will involve reasoning about programs.

3.7 Propositional Logic in ACL2s

This class is about logic from a computational point of view and our vehicle for exploring computation is ACL2s. ACL2s has *ite*: it is just `if`!

Remember that ACL2s is in the business of proving theorems. Since propositional logic is used everywhere, it would be great if we could use ACL2s to reason about propositional logic. In fact, we can.

Consider trying to prove that a propositional formula is valid. We would do that now, by constructing a truth table. We can also just ask ACL2s. For example, to check whether the following is valid

$$(p \Rightarrow q) \equiv (\neg p \vee q)$$

We can ask ACL2s the following query

```
(thm (implies (and (booleanp p) (booleanp q))
              (iff (implies p q) (or (not p) q))))
```

Try it in Beginner mode.

In fact, if a propositional formula is valid (that is, it is a theorem) then ACL2s will definitely prove it. We say that ACL2s is a decision procedure for propositional validity. A *decision procedure* for propositional validity is a program that given a formula can decide whether or not it is valid. We saw that ACL2s indicates that it has determined that a formula is valid with "Q.E.D." [1]

What if you give ACL2s a formula that is not valid? Try it with an example, say:

$$(p \oplus q) \equiv (p \vee q)$$

We can ask ACL2s the following query

```
(thm (implies (and (booleanp p) (booleanp q))
              (iff (xor p q) (or p q))))
```

As you can see, ACL2s also can provide counterexamples to false conjectures.

3.8 Word Problems

Next, we consider how to formalize word problems using propositional logic.

Consider formalizing and analyzing the following.

[1] Q.E.D. is abbreviation for "quod erat demonstrandum," Latin for "that which was to be demonstrated."

> Tom likes Jane if and only if Jane likes Tom. Jane likes Bill. Therefore, Tom does not like Jane.

Here's the kind of answer I expect you to give.

> Let p denote "Tom likes Jane"; let q denote "Jane likes Tom"; let r denote "Jane likes Bill."
>
> The first sentence can then be formalized as $p \equiv q$.
>
> We denote the second sentence by r.
>
> The third sentence contains the claim we are to analyze, which can be formalized as $((p \equiv q) \wedge r) \Rightarrow \neg p$.
>
> This is not a valid claim. A truth table shows that the claim is violated by the assignment that makes p, q, and r *true*. This makes sense because r (that Jane likes Bill) does not rule out q (that "Jane likes Tom"), but q requires p (that "Tom likes Jane").

Consider another example.

> A grade will be given if and only if the test is taken. The test has been taken. Was a grade given?

Anything of the form "a iff b" is formalized as $a \equiv b$. The problem now becomes easy to analyze.

> John is going to the party if Mary goes. Mary is not going. Therefore, John isn't going either.

How do we formalize "a if b"? Simple: $b \Rightarrow a$. Finish the analysis.

> John is going to the party only if Mary goes. Mary is not going. Therefore, John isn't going either.

How do we formalize "only if"? A simple way to remember this is that "if" is one direction of "if and only if" and "only if" is the other direction. Thus, "a only if b" is formalized as $a \Rightarrow b$.

Try this one.

> John is going to the party only if Mary goes. Mary is going. Therefore, John is going too.

One more.

> Paul is not going to sleep unless he finishes the carrot hunt on Final Fantasy XII. Paul went to sleep. Therefore, he finished the carrot hunt on Final Fantasy XII.

How do we formalize "a unless b"? It is $\neg b \Rightarrow a$. Why? Because "a unless b" says that a has to be *true*, except when (unless) b is *true*, so when b is *true*, a can be anything. The only assignment that violates "a unless b" is when a is *false* and b is *false*. So, notice that "a unless b" is equivalent to "a or b".

One more example of unless.

> You will not get into NEU unless you apply.

is the same as

> You will not get into NEU if you do not apply.

which is the same as

> You will not get into NEU or you will apply.

So, the hard part here is formalizing the problem. After that, even ACL2s can figure out if the argument is valid.

3.9 The Declarative Approach to Design

Most of the design paradigms you have seen so far require you to describe how to solve problems algorithmically. This is true for functional, applicative, and object-oriented programming paradigms. A rather radically different approach to design is to use the declarative paradigm. The idea is to specify *what* we want, not *how* to achieve it. A satisfiability solver is then used to find a solution to the constraints.

3.9.1 Avionics Example

Let us consider an example from the avionics domain.

We have a set of *cabinets* $C = \{C_1, C_2, \ldots, C_{20}\}$. Cabinets are physical locations on an airplane that provide network access, battery power, memory, CPUs, and other resources.

We also have a set of *avionics applications* $A = \{A_1, A_2, \ldots, A_{500}\}$. The avionics applications are software programs and include applications such as navigation, control, collision detection, and collision avoidance.

Our job is to map each application to one cabinet subject to a large number of constraints.

Instead of us figuring out how to achieve this mapping, by using the declarative approach we will instead just specify the constraints we have on the mapping and we will let the declarative system we are using figure out a solution for us. This allows us to operate at a much higher level of abstraction than is the case with functional, imperative, or object-oriented approaches.

To make the idea concrete, we consider one simple example of a constraint: applications A_1, A_2, and A_3 have to be separated. What this means is that no pair of them can reside on the same cabinet. Here is how we might express this constraint in the declarative language CoBaSA. Assume that we have defined `A`, the array of 500 applications and `C`, the array of 20 cabinets.

```
Map AC A C
For_all cab in C {AC(1,cab) implies ((not AC(2,cab)) and (not AC(3,cab)))}
For_all cab in C {AC(2,cab) implies (not AC(3,cab))}
```

The first line tells us that `AC` is a *map*, a function from `A` to `C`. When we define a map, we get access to *indicator variables*. Such variables are Boolean variables of the form `AC(app,cab)`, where `AC(app,cab)` = *true* iff `AC(app)` = `cab`, *i.e.*, map `AC` applied to application `app` returns `cab`.

3.9.2 Solving Declarative Constraints

In this section, we will get a glimpse into the process by which the three lines of CoBaSA constraints above get turned into a formula in propositional logic that is then given to a SAT solver.

We start by writing the constraints above using standard mathematical notation, starting with the second constraint.

$$\langle \forall c \in C :: AC_1^c \implies \neg AC_2^c \wedge \neg AC_3^c \rangle$$

The $\forall$ symbol is a *universal quantifier*. It states that for all cabinets ($c \in C$) if application A_1 gets mapped to the cabinet (the indicator variable AC_1^c) then neither application A_2 nor application A_3 get mapped to the same cabinet. We can actually rewrite this using propositional logic as follows:

$$\bigwedge_{c \in C} AC_1^c \implies \neg AC_2^c \wedge \neg AC_3^c$$

So, universal quantification can be thought of as conjunction. Now, if we expand this out, we wind up with the following:

$$\begin{gathered}(AC_1^1 \implies \neg AC_2^1 \wedge \neg AC_3^1) \wedge \\ (AC_1^2 \implies \neg AC_2^2 \wedge \neg AC_3^2) \wedge \\ \dots \\ (AC_1^{20} \implies \neg AC_2^{20} \wedge \neg AC_3^{20})\end{gathered}$$

So, we are almost at the point where we can give this to a SAT solver. One issue, however, is that most SAT solvers require their input to be in CNF (Conjunctive Normal Form). What we have is a conjunction, but the conjuncts are not clauses. Here is how to turn them into clauses. We show how to do this for the first conjunct only, since the rest follow the same pattern. The first conjunct above gets turned into the following two clauses.

$$\begin{gathered}\neg AC_1^1 \vee \neg AC_2^1 \\ \neg AC_1^1 \vee \neg AC_3^1\end{gathered}$$

Notice that these two clauses are semantically equivalent to the conjunct they correspond to.

There is one more issue to deal with before we can use a SAT solver: SAT solvers tend to require DIMACS file format. In the DIMACS format, variables are represented by positive integers, negated variables by negative integers, and each clause is a list of integers that ends with a 0 and a newline. So, the first thing to do is to come up with a mapping from indicator variables into the positive integers so that no two indicator variables get mapped to the same number. Here is one way of doing that.

$$var(AC_a^c) = 20(a-1) + c$$

With this mapping, the translation of the 2^{nd} CoBaSA constraint to DIMACS format gives us the following 40 disjuncts:

```
-1 -21 0
-1 -41 0
-2 -22 0
-2 -42 0
...
-20 -40 0
-20 -60 0
```

The third CoBaSA constraint gets translated in a similar way. What about the first constraint?

Well, here is one way of thinking of the `map` constraint using logic.

$$\langle \forall a \in A :: \langle \exists c \in C :: AC_a^c \rangle\rangle$$

This says that for every application a, there exists some cabinet c (this is the meaning of the existential quantifier $\exists$), such that the indicator variable AC_a^c holds (is *true*). Notice that we could have said that there exists a *unique* cabinet c such that the indicator variable AC_a^c holds. This would have been a more faithful translation, but it turns out that if more than one indicator variable is *true*, then all that means is that we have a choice as to where to place a, so we prefer to have fewer constraints and not insist on uniqueness. Now, just as universal quantification can be thought of as conjunction, existential quantification can be thought of a disjunction, so we can rewrite the above constraint using only propositional operators as:

$$\bigwedge_{a \in A} \bigvee_{c \in C} AC_a^c$$

We proceed as previously by expanding this out with the goal of generating a CNF formula.

$$AC_1^1 \vee AC_1^2 \vee AC_1^3 \cdots \vee AC_1^{20}$$

$$AC_2^1 \vee AC_2^2 \vee AC_2^3 \cdots \vee AC_2^{20}$$

$$\ldots$$

$$AC_{500}^1 \vee AC_{500}^2 \vee AC_{500}^3 \cdots \vee AC_{500}^{20}$$

Finally, we apply *var* to transform the indicator variables into numbers in order to obtain the DIMACS version of the above formula.

```
1 2 3 ... 20 0
21 22 23 ... 40 0
...
9980 9981 9982 ... 10000 0
```