CS40-S13: Functional Completeness

Victor Amelkin ⟨victor@cs.ucsb.edu⟩

April 12, 2013

In class, we have briefly discussed what *functional completeness* means and how to prove that a certain system (a set) of logical operators is functionally complete. In this note, I will review what functional completeness is, how to prove that a system of logical operators is complete as well as how to prove that a system is incomplete.

1 Introduction

When we talk about logical operators, we mean operators like $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\oplus$. Logical operators work on propositions, i.e. if we write $a \wedge b$, then both a and b are propositions. The mentioned propositions may be either *simple* or *compound*, as in

$$a \wedge \underbrace{\neg(x \rightarrow y)}_{b} \qquad \text{– here, } a \text{ is simple, while } b \text{ is compound.}$$

For a logical operator it does not matter whether it operates on simple or compound propositions; the only thing that matters is that all the arguments are propositions, i.e., either True of False.

Since logical operators apply to propositions and "return" either True or False, they can be seen as *boolean functions*. For example, conjunction $\wedge(x_1, x_2) = x_1 \wedge x_2$ is a binary boolean function, i.e., it accepts two arguments (hence, "binary") that can be either True of False and returns either True of False (hence, "boolean"). As another example, negation can be seen as a unary (accepts only one argument) boolean function $\neg(x) = \neg x$.

We, however, rarely use the functional notation $\wedge(x_1, x_2)$ (like in $f(x)$); instead, we usually use the operator notation, e.g., $x_1 \vee x_2$. (Similarly, when you do algebra, you probably write $a + b$ instead of $+(a, b)$, unless you do algebra in Lisp.)

Further, instead of talking about logical operators, we will talk about boolean functions, but, having read the material above, you should understand that these two are the same. Additionally, as it is usually done with boolean functions, instead of using "True" and "False", we will write "1" for "True" and "0" for "False", but this change is just notational.

2 Functional completeness

Now, we can talk about functional completeness of a system (a set) of boolean functions (or logical operators, as you wish).

def. *A system S of boolean functions (or, alternatively, logical operators) is* functionally complete *if every boolean function (or, alternatively, every compound proposition) can be expressed in terms of the functions from S.*

For example, let us look at the following system of boolean functions $S = \{\wedge, \vee, \neg\}$. For this system to be functionally complete, by definition, every possible boolean function (or, alternatively, every compound proposition) should be expressible using only functions from S. For instance, if we have a boolean function (a compound proposition)

$$P(x_1, x_2, x_3) = x_1 \wedge (x_2 \oplus x_3) \wedge (x_1 \rightarrow x_3),$$

we should be able to rewrite P using only $\wedge$, $\vee$, and $\neg$.

In case with P, it is easy to rewrite it using only functions from set S: we know that $x \oplus y \equiv (\neg x \wedge y) \vee (x \wedge \neg y)$ and $x \rightarrow y \equiv \neg x \vee y$, and we can rewrite P using these equivalences. Alternatively, we can build the *truth table* for P and then write down P's *disjunctive normal form (DNF)*. DNF, as you remember, uses only $\wedge$, $\vee$, and $\neg$, so the goal to rewrite P using only functions from S is achieved.

We can use similar reasoning to prove that any other system of boolean functions is functionally complete. For example, in one of the homeworks, you should have proven that system $S = \{\wedge, \neg\}$ is functionally complete. Your proof might have looked as follows:

Proof. For every boolean function (or, alternatively, every compound proposition), we can write down its truth table. Using the truth table, we can construct the DNF. DNF uses only $\wedge$, $\vee$, and $\neg$, all of whom are expressible in terms of $S = \{\wedge, \neg\}$. For $\wedge$ and $\neg$, it is obvious why they are expressible – they are members of S. And the remaining function $\vee$ can be expressed in terms of $\wedge$ and $\neg$ using one of De Morgan's laws:

$$x \vee y = \neg(\neg x \wedge \neg y).$$

Thus, $S = \{\wedge, \neg\}$ is functionally complete.

□

3 Incompleteness

Sometimes, however, the problem is opposite – we may need to prove that a certain system S of boolean functions is *not* functionally complete. According to the definition of functional completeness, it means that *not* every function/proposition can be expressed using only functions from S.

For example, a system $S = \{\wedge, \vee\}$ is *not* functionally complete, i.e., some boolean functions cannot be rewritten using only $\wedge$ and $\vee$. We can guess that $\neg$ is probably one of those functions not expressible in terms of $\wedge$ and $\vee$, but the problem is to prove it.

3.1 A programming contest metaphor

Prior to plunging into proving and disproving functional completeness, imagine a team of students participating in a programming contest. In this contest, the problems require knowledge of some (maybe even all) of the following:

- knowledge of C++,
- knowledge of search algorithms,
- good understanding of the material given in CS40 at UCSB (in Spring, 2013).

We have a team of three students, each of whom knows some of the mentioned topics. Let us display this information in a table form:

	knows C++	knows search	aced CS40
Alice	+	+	−
Bob	−	−	+
Clive	+	−	−

In the table above, for every team member, we specify whether he or she is familiar with each topic. For example, Bob apparently does not know C++ and has no idea how to search, so we put − in the corresponding cells. However, Bob has aced CS40; thus, we put + in the corresponding cell.

It is clear that if a certain programming problem requires knowledge of C++, then at least one team member should know C++, so that the team can solve this problem. It is easy to see that for every of three mentioned topics (C++, search, CS40), there is at least one team member familiar with it (in every column of our table, there is at least one +).

Thus, we can claim that regardless of what problem this team is presented with, as long as this problem only requires knowledge of C++ and/or search and/or CS40, then this team can solve it.

This sounds very similar to the definition of functional completeness ("regardless of what function we are presented with, we can express it using the functions from the system"). We will see that testing functional completeness is very similar to what we did with the programming contest team.

3.2 Back to functions

Indeed, things are very similar when we deal with functional completeness. Instead of a team of students, we have a team (a system) of functions. And instead of three topics that students may know (C++, search, CS40), we will deal with five properties functions may or may not have. The completeness test itself will be similar to identifying whether a team has at least one member knowing each topic.

First, let us describe those five properties of boolean functions.

Property 1: We say that boolean function f *preserves zero*, if on the 0-input it produces 0. By the 0-input we mean such an input, where every input variable is 0 (this input usually corresponds to the first row of the truth table). We denote the class of zero-preserving boolean functions as T_0 and write $f \in T_0$.

Here are the truth tables for a few boolean functions we are familiar with:

x	$\neg x$
0	1
1	0

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

We see that on 0-input, both $\wedge$ and $\vee$ produce 0, while $\neg$ produces 1. Thus, $\wedge$ and $\vee$ preserve zero, and $\neg$ does not. We write $\wedge \in T_0$ and $\vee \in T_0$, but $\neg \notin T_0$.

Property 2: Similarly to T_0, we say that boolean function f *preserves one*, if on 1-input, it produces 1. The 1-input is the input where all the input variables are 1 (this input usually corresponds to the last row of the truth table). We denote the class of one-preserving boolean functions as T_1 and write $f \in T_1$.

Looking at the truth tables of out three functions, we see that $\neg$ does not preserve one since it produces 0 on 1-input. However, both $\wedge$ and $\vee$ preserve 1, since their values on 1-input are both 1.

x	$\neg x$
0	1
1	0

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

Thus, $\wedge \in T_1$ and $\vee \in T_1$, but $\neg \notin T_1$.

Property 3: We say that boolean function f is *linear* if one of the following two statements holds for f:

- For every 1-value of f, the number of 1's in the corresponding input is *odd*, and for every 0-value of f, the number of 1's in the corresponding input is *even*.

  or

- For every 1-value of f, the number of 1's in the corresponding input is *even*, and for every 0-value of f, the number of 1's in the corresponding input is *odd*.

If one of these statements holds for f, we say that f is linear[1]. We denote the class of linear boolean functions with L and write $f \in L$.

Now, let us check whether our three functions are linear.

x	$\neg x$
0	1
1	0

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1

In the case of $\neg$, for every 1-value of this function, the number of 1's among the input variables is always even. Actually, there is only one 1-value of $\neg$, and the number of 1's in the corresponding input is 0, which is an even number. Similarly, for every 0-value of $\neg$ (and there is only one such value), the number of 1's in the input is odd (we have the only input variable, and it is 1 in the corresponding row). Thus, $\neg$ is linear.

Using the same reasoning, we can show that neither $\wedge$ nor $\vee$ is linear. $\wedge$ is not linear because on 0 outputs, the number of 1's in the corresponding inputs is sometimes even and sometimes odd. Thus, neither of two conditions from our definition can hold for $\wedge$. Similarly, $\vee$ is not linear because on 1-outputs, the number of 1's in the corresponding inputs is sometimes even and sometimes odd.

Thus, $\neg \in L$, but $\wedge \notin L$ and $\vee \notin L$.

[1]You may wonder why such functions are called "linear", yet there is no sign of linearity as we know it from calculus. The terrible truth about the given "definition" of linear boolean functions is that it is not a definition, but a criterion (i.e., a necessary and sufficient condition for linearity of a boolean function). The definition of a linear boolean function, yet, clearly manifests linearity, is too complicated, and, as such, has been omitted. If, however, you are still curious: a boolean function is linear by definition (this time, real definition) if its Zhegalkin polynomial is linear.

Property 4: We say that boolean function f is *monotone* if for every input, switching any input variable from 0 to 1 can *only* result in the function's switching its value from 0 to 1, and *never* from 1 to 0. We denote the class of monotone boolean functions with M and write $f \in M$.

Yet the given definition looks slightly intimidating, it is actually very easy to check monotonicity by looking at the following *Hasse diagrams*:

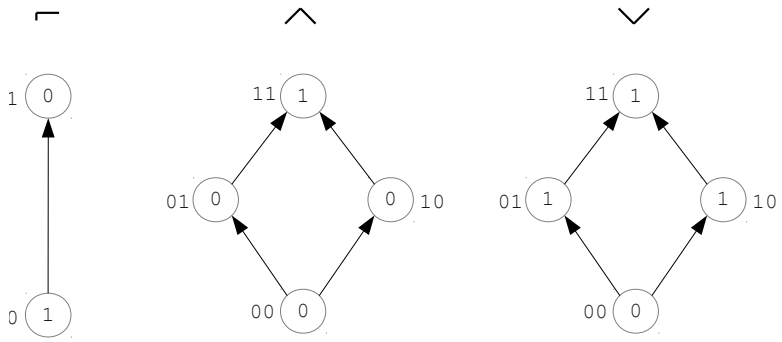

Figure 1: Checking monotonicity with Hasse diagrams

Here is how these diagrams are built. For a function you want to check for monotonicity, draw a vertex (or a circle, as it is done on the diagrams above) for every row in the function's truth table. Next to each vertex, write the corresponding input from the truth table. Inside each vertex, write the value of the function on the corresponding input. Finally, draw arrows using the following rule: if input B can be obtained from input A by switching exactly *one* input variable from 0 to 1, then draw an arrow from the vertex corresponding to input A to the vertex corresponding to input B.

For example, the middle diagram is built for $\wedge$. Its truth table contains 4 rows, thus there are 4 vertices (circles) on the diagram. Next to each vertex, we write input values and connect vertices with arrows using the mentioned rule. (For example, 01 can be obtained from 00 by switching *exactly* one input variable from 0 to 1. Thus, there is an arrow from 00 to 01. However, 11 cannot be obtained from 00 by switching *exactly* one input variable from 0 to 1. Thus, there is no arrow from 00 to 11.)

Now, to check whether a function is monotone, follow every path on this function's Hasse diagram from 0-input (all input variables are 0) to 1-input (all input variables are 1). The boolean function is monotone if on every such path, it *never decreases its value (never goes from* 1 *to* 0*)*, yet, it can remain constant or grow (i.e., switch from 0 to 1).

For example, for $\wedge$, there are two paths from 00-input to 11-input, and on both these paths, $\wedge$ is either constant or it changes from 0 to 1. Thus, $\wedge$ is monotone. The same is true for $\vee$ – there are two paths from 00 to 11 on its Hasse diagram, and on both of these paths $\vee$ never decreases.

The things are different, however, in case of $\neg$. There is only one path from 0-input to 1-input, and on this path, $\neg$ *decreases* (switches its value from 1 to 0). Thus, $\neg$ is not monotone.

We have $\wedge \in M$ and $\vee \in M$, but $\neg \notin M$.

Property 5: We say that boolean function $f(x_1, \ldots, x_n)$ is *self-dual* if

$$f(x_1, \ldots, x_n) = \neg f(\neg x_1, \ldots, \neg x_n).$$

The function on the right in the equality above (the one with negations) is called *the dual of* f. We will call the class of self-dual boolean functions S and write $f \in S$.

Finally, something that can be easily checked. Let us start with negation. The dual for $\neg x$ is $\neg(\neg(\neg x)) = \neg x$. Thus, it is clear that $\neg$ is self-dual.

The things are different for $\wedge$ and $\vee$. The dual for $x \wedge y$ is $\neg((\neg x) \wedge (\neg y))$, and the latter, according to one of De Morgan's laws is $x \vee y$. It is clear that $x \wedge y$ and its dual $x \vee y$ have different values on some inputs and, thus, are not equivalent. Thus, $\wedge$ is not self-dual. Using the same reasoning, we can show that $\vee$ is also not self-dual.

Thus, $\neg \in S$, but $\wedge \notin S$ and $\vee \notin S$.

We are done with properties and ready to state the criterion of functional completeness of a system of boolean functions.

Theorem. *A system of boolean functions is functionally complete if and only if this system does* not entirely *belong to any of* T_0, T_1, L, M, S.

The mentioned theorem is called *Post's completeness criterion* and is due to Emil Post. What this criterion says is that in order for a system of boolean functions to be functionally complete, this system should have

- at least one function that does *not* preserve zero (i.e. it is *not* in T_0), and
- at least one function that does *not* preserve one (i.e. it is *not* in T_1), and
- at least one function that is *not* linear (i.e. it is *not* in L), and
- at least one function that is *not* monotone (i.e. it is *not* in M), and
- at least one function that is *not* self-dual (i.e. it is *not* in S).

Of course, one function may combine some of these roles, e.g., a function may *not* preserve one and, at the same time, *not* be monotone.

Next, we will see how to use this criterion in practice.

4 Testing completeness

Let us now apply Post's criterion to test completeness of some systems of boolean functions. We will specify which function belongs to which classes and display the results in a table as we did with the students and skills for the programming contest. Fortunatelly, we have already established what classes each of $\wedge$, $\vee$, and $\neg$ belongs to, so we will just collect these results.

	not in T_0	not in T_1	not in L	not in M	not in S
$\wedge$	$-$	$-$	$+$	$-$	$+$
$\vee$	$-$	$-$	$+$	$-$	$+$
$\neg$	$+$	$+$	$-$	$+$	$-$

According to Post's criterion, for a system $\{\wedge, \vee, \neg\}$ of boolean functions to be functionally complete, there should be at least one $+$ in every column of the table above, which is obviously true. Thus, by Post's criterion, the system $\{\wedge, \vee, \neg\}$ is functionally complete. But we have already knew it, so let us now try to look at different systems of functions.

Let us look at an abridged system $\{\wedge, \neg\}$.

	not in T_0	not in T_1	not in L	not in M	not in S
$\wedge$	$-$	$-$	$+$	$-$	$+$
$\neg$	$+$	$+$	$-$	$+$	$-$

Still, every column of the table contains at least one $+$. Thus, system $\{\wedge, \neg\}$ is also functionally complete. This result should match the result you should have obtained in your homework (though, you should have proven completeness of this system differently).

Now, how about system $\{\wedge, \vee\}$?

	not in T_0	not in T_1	not in L	not in M	not in S
$\wedge$	$-$	$-$	$+$	$-$	$+$
$\vee$	$-$	$-$	$+$	$-$	$+$

The columns corresponding to classes T_0, T_1, and M have no $+$'es, which means that system $\{\wedge, \vee\}$ is *not* functionally complete.

An easy way to understand Post's criterion is as follows: if all of the functions in our system of boolean functions, say, preserve zero, i.e., all of them are in T_0, then regardless of how we combine our functions, the resulting function will also preserve zero. This is true for each of the mentioned classes of functions. If *all* functions of our system are inside one of the five mentioned classes, then every function expressible in terms of these functions *also* belongs to this class. Consequently, functions outside of this class cannot be expressed in term of the functions of our system.

This property of our five classes is called closedness, and, hence, the classes T_0, T_1, L, M, and S are usually called the *closed classes of boolean functions*. If we are inside one of these classes, there is "no escape". To "escape", at least one of our functions should be outside.

5 Sheffer stroke and Peirce's arrow

As a finishing touch, let us look at another boolean function (or, alternatively, logical operator). It is a binary boolean function called *Sheffer stroke* or *NAND*, denoted with | (or, sometimes, ↑)

x_1	x_2	$x_1 \mid x_2$
0	0	1
0	1	1
1	0	1
1	1	0

By looking at the truth table of Sheffer stroke, it becomes clear why it also goes under the name NAND – it is equivalent to *not-and*, i.e., $\neg(x_1 \wedge x_2)$.

The surprising fact about Sheffer stroke is that it posesses the ultimate expressive power, that is, the system of boolean functions consisting of only Sheffer stroke is functionally complete. In other words, regardless of what boolean function we have, it can be rewritten using only Sheffer stroke. Let us check whether it is true using Post's criterion.

We need to check whether | belongs to any of the classes T_0, T_1, L, M, S. According to Post's criterion, the system $\{|\}$ consisting only of Sheffer stroke will be functionally complete iff Sheffer stroke does *not* belong to any of T_0, T_1, L, M, S, i.e., all the cells in the following table should be filled with +.

	not in T_0	not in T_1	not in L	not in M	not in S
|	?	?	?	?	?

By looking at the truth table of |, we see that it does not preserve zero, since it is 1 on the 0-input. Similarly, it does not preserve one. Thus, we have

	not in T_0	not in T_1	not in L	not in M	not in S
|	+	+	?	?	?

The linearity conditions cannot hold for |, since on 1-outputs, the number of 1's among the input variables is sometimes even and sometimes odd. Thus, | is not linear:

	not in T_0	not in T_1	not in L	not in M	not in S
|	+	+	+	?	?

To check monotonicity, let us draw a Hasse diagram for |:

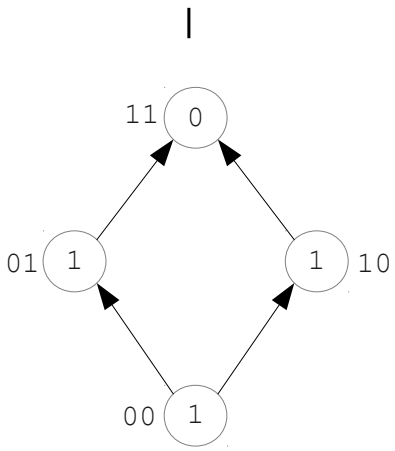

Figure 2: Checking monotonicity of |

On both paths from 00 to 11, function | decreases its value from 1 to 0 when moving from 01 or 10 to 11. Thus, | is not monotone:

	not in T_0	not in T_1	not in L	not in M	not in S
$\mid$	+	+	+	+	?

Finally, let us check self-duality:

$$dual(x_1|x_2) = \neg(\neg x_1|\neg x_2) = \neg(\neg(\neg x_1 \wedge \neg x_2)) = \neg(x_1 \vee x_2) = x_1 \downarrow x_2.$$

The boolean function $\downarrow$ we obtained as the dual for | is called *Peirce's arrow* or *NOR* (guess why?) and it is clearly not the same as |. Thus, | is not self-dual.

	not in T_0	not in T_1	not in L	not in M	not in S
$\mid$	+	+	+	+	+

We see that every column in the obtained table contains at least one +, which means that the system consisting of single function | is functionally complete.

To resolve any doubts, let us see how $\wedge$ and $\neg$ can be expressed using only Sheffer stroke:

$$\neg x = x|x,$$
$$x_1 \wedge x_2 = (x_1|x_2)|(x_1|x_2).$$

As an excercise, check whether a system of boolean functions consisting only of Peirce's arrow[2] is functionally complete.

[2] $x_1 \downarrow x_2 = \neg(x_1 \vee x_2)$