Logic for CS Undergraduates: a Sketch

Dennis Hamilton

August 9, 2022

Logic for CS Undergraduates: A Sketch

Position Paper Submitted for LogTeach-22 (2022-07-31), 2022-05-30
2022-08-08 revision for preprint draft

Dennis E. Hamilton
Interoperability Architect
Seattle, WA, USA
dennis.hamilton@acm.org

I find, as a practitioner having grown up in the emergence of Computer Science, that logic is indispensable in useful ways, as is a level of comfort with geometry, algebra, and, to some degree, analysis. I also conclude that, at an undergraduate level, there are elementary approaches to logic that are neither philosophically demanding nor inapplicable in a computational context. I illustrate this with a sketch of logical application about computation and its appreciation in hopefully practical ways. It has served me in the comprehension of computational models, for example. It depends on the utility of distinguishing formal and mathematical entities from those empirical ones perceived in nature and society.

1. Deduction: Propositional Logic and Truth-Value Semantics

Establish symbology and separation from the operational sense found in programming languages: It's useful not to cross that bridge prematurely.

For example,

$p \Leftrightarrow q, p \Rightarrow q, p \vee q, p \wedge q, \neg p$

Keep it classical, emphasizing that we are speaking of logical expressions of deductive theories. E.g., if it is asserted that $p \Leftrightarrow q$ and also $\neg p$, deduction that $\neg q$ holds is a *logical* consequence; if it is asserted that $p \Rightarrow q$ and also p, deduction that q holds is *logical*. The law of excluded middle holds: It is the case that $\neg(p \wedge \neg p)$ and also $\neg(\neg p) \Leftrightarrow p$ as well as $(p \Rightarrow q) \Leftrightarrow (\neg p \vee q)$.

Pitfalls

- Treating propositions as wriggly and not fixed throughout a situation.
- Assuming that one can conclude much given only $p \Rightarrow q$. The language game about being able to prove anything from a false proposition is sleight of hand to be avoided.
- If deduction forms such as $h_1, .., h_n \vdash p$ are introduced (why not?), be clear why $p, \neg p \vdash p \wedge \neg p$ is inconsistent/unsound and not acceptable.

Take-aways

- Understanding the difference between logic being about proper arguments concerning hypothetic cases, not about assertions of fact (by itself)
- Understanding how modus ponens serves as a fundamental form of deduction

Introduce satisfiability. If it's not satisfiable, don't ask. If it is tautologously true, why ask? Determining this by tables and the exponential problem of doing it by enumeration is worth discussion as an interesting problem between logic and computation. Of course, at some point, P ?= NP.

[Note: It is Interesting that *The Art of Computer Programming* does without logic symbology until Volume 4A; and there the forms are defined operationally in terms of bit-fiddling. *Concrete Mathematics* never deals with logic explicitly.]

2. First-Order Logic for Novices: FOL, FOL=, Equational Mostly

Here, the introduction of quantifiers and the consideration of a domain of discourse can be undertaken, along with terms and predicates. Continue having it unlike programming: $\forall x \ \exists y$

Peano arithmetic is useful as a simple case, along with the introduction of equality and non-logical term-expressions. Mathematical induction can be (re-)introduced; that's valuable for later in this context. Other numerical algebraic structures can be illustrative, including modular arithmetic.

The simplification of omitting universal quantifiers that embrace an entire logical "statement" is done in what will typically be equational material in early applications to computational contexts.

The introduction of Boolean algebras is a valuable example as well. Use

$\top$ (nickname "top")
$\perp$ (nickname "bot")

and also $\sim x$ complement, $x \cap y$ "cap", $x \cup y$ "cup", and $x \dot{-} y$ "sep".

In addition to equality, there is also an ordering, $x \subseteq y$ "sub". This is a relation, not an operation, even though there may be a way to compute it.

It will be useful not to rely on T and F and also avoid 1 and 0, so that representations can be addressed more clearly as this journey proceeds.

Explore axioms of a Boolean algebra and some deductions. The special case of a domain consisting of only $\top$ and $\perp$ is interesting for the correlation with propositional logic. The axiom of subordination is demonstrative of both,

$$x \subseteq y \Leftrightarrow x = x \cap y$$

Also there are useful definitions, either by introducing operators or functional notation and cases. E.g., definition of sep (separation)

$$x \dot{-} y = (x \cap \sim y) \cup (\sim x \cap y)$$

with a little about partial ordering and also identity.

$\perp \subseteq y$ (hence "bot")
$x \subseteq \top$ (hence "top")
$x = y \Leftrightarrow x \subseteq y \wedge y \subseteq x$

Take-aways

- Exposure to schemes different from arithmetic and number theory, also keeping notations straight. This is helpful in what follows.
- Further employment of notation to be taken as those that may-have-been/will-be seen in programming languages and taken as operational rather than about abstract (mathematical) entities
- The idea of primitives, including constant terms and defined functions/predicates from which the elements of a domain are characterized: axiomatics.

3. First-Order (mostly) Structures

It is useful to now speak of mathematical structures. The goal is to arrive at a notion of computational interpretation of structures. Computational theory and models of computation do not have to be addressed in any detail. The door is kept open for later on. Category theory is also unnecessary. That door is also open.

It is important to relate among structures as a foundation for interpretations among structures. For this purpose, it is useful that a structure ‹σ› is taken to consist of a triple,

$$‹\sigma› = \langle \sigma\mathrm{D}, \sigma\mathrm{F}, \sigma\mathrm{T} \rangle$$

where σD signifies the domain of discourse, the individuals of interest. Consider that there is a set, σF, of all the functions over σD. (We can consider all the predicates too, without being too fussy.) It is not expected to exhibit σF, although it is useful to talk about it and also to consider what's in it. Finally, σT is the theoretical characterization of ‹σ›, using FOL, usually FOL=.

This arrangement deviates from how mathematical structures are typically depicted. This is done because we want to speak about the rather invisible nature of σF and also be clear which theory, σT, we are operating in. This sets up depiction of relationships between structures.

Focus on denumerability of σD and having canonical forms for the individuals. The distinction of decimal numerals from the mathematical natural numbers is instructive in this case, including how we learn arithmetic as manipulation of those forms. That there are other canonical forms is also relevant, whether binary notation, hexadecimal, or verbalized numbers (e.g, "one hundred forty-five"). It will become apparent that we do not have a way to exhibit mathematical entities, only talk about them even though certain ones, identified as constants, are singled out by their being distinguished axiomatically (0 and 1, T and $\perp$) .

The definitions of functions happen in the usual way by (recursive) cases of equations, and some functions are established axiomatically whether by name or introduction as operators. (E.g., the introduction of Boolean algebra functions.)

An important take-away in consideration of such structures has to do with extensionality. If two function, *f* and *g* are characterized in σT such that for all *x*, *f*(*x*) = *g*(*x*), it is useful to say that the definitions/characterizations of *f* and *g* determine the same function in σF. It is valuable to avoid saying they "are" the same function or saying they are equivalent functions. This is an important consideration also in contrast with the mis-appropriation of terms such as **function** (**integer** and **real**) for computational purposes.

Many interesting structures relevant to computation have denumerable σD such as all finite strings over a fixed alphabet, or such strings having a context-free grammar, with recursive definition by cases. Admission of structural induction with regard to the characteristics of defined functions and determining that two definitions determine the same function becomes valuable to appreciate.

4. Interpretations and Representations

By **interpretation** of one structure in another is meant the usual model-theoretic arrangement, although the interpretation need not be a model in the official sense.

The simple case is obtained by simply specifying how the primitives/axioms of structure ‹α› are associated with functions and predicates determined and exhibited in structure ‹β›, it being clear that whatever is logically deducible in ‹α› holds *for the counterpart* in ‹β›. Using the valuable distinction of Thomas Forster, If equality in ‹α› is sent to equality in ‹β›, interpretation is an implementation, and otherwise it is a simulation with equality taken to a different identity, perhaps one of equivalence classes.

There are some useful simplifying moves, a kind of mathematical engineering, that can be demonstrated here. The existence of duals should not be given metaphysical significance, and while one can interpret the prototypical Boolean algebra in number theory (and vice versa), it is economical to map 0 with $\perp$ and anything else (but 1 canonical) with $\top$. The inquisitive student might find Boole's *The Laws of Thought* informative at this point.

The point is demonstration how much representation is applicable in computing and Computer Science, even when it is not undertaken with disciplined application of (mathematical) logic. These are clearly simplified cases and how greater complexity is addressed is perhaps to be appreciated, but not undertaken at this level. That one can view devising software fit for some purpose as a case of (extremely informal) theory building (after Peter Naur) can be introduced.

Successful **representation** is achieved when there is a straightforward determination of α*D* individuals corresponding to functionally-determined β*D* individuals (perhaps with canonical forms) in the particular ‹β› interpretation of ‹α›.

5. Computational Interpretations, Models and Stored Programs

It is useful to consider that computational interpretations are transitive in the sense that if structure ‹α› has an interpretation in structure ‹β›, and structure ‹β› has a computational interpretation, that serves as one for ‹α›.

A structure ‹µ› is an abstract (stored-program) **computational model** if there is a µ*T*-determined universal function, say µ.App(*p*,*x*), such that for every computable function *f(x)* in µ*F,*

$$\exists p\ \forall x\ [\mu.\mathrm{App}(p,x) = f(x)]$$

is the case for µ*D* denumerable.

µ.App(*p*,*x*) is a direct computational interpretation of ‹µ› in this case. It is an abstraction of stored-program computation and there are useful examples in practical applications of computers (compiling, interpreting, parsing, etc.).

Church-Turing universality is established by explanation of exactly what the thesis is and that other CT-recognized models are demonstrable b simulations in a ‹µ›. It is not necessary to undertake such an effort. It is more about understanding that there are such recognized cases.

That one can, in such cases, have programs that compute programs, is a foundational notion in CS and this is a simplified glimpse of that. One can also point out Turing Machines and the idea of the UTM without going very far down this road. There might be opportunity to introduce the notion that there are (notionally) functions in µ F that are not computable, not determinable by any finite formulation in µT.

6. Interpretations, Models, and Natural Occurrence

In all of this so far, consideration is confined to a world of abstract (mathematical) entities.

It requires extra-logical interpretation of a computational model in terms of the operation of a mechanism, a computer, that is in fact operated in the natural world. Mathematical structures are not mechanisms.

This is not a new difficulty. It applies as much to physics as it does to the connection of computation to mathematics and logic when we take computations to be about something that is other than the direct computational interpretation.

Even the 0s and 1s considered particles of computer representations are abstractions engineered and represented with extraordinary dependability. We might say better that the 0s and 1s are manifest (represented) in an intended interpretation.

There is an informal treatment of these logico-mathematical structures that is a valuable illustration of something fundamental in computation. Software development and Computer Science involve models of structures in the world -- that is, nature exhibits to our experience phenomena that we can take as interpretations of theories. (Consider the Pythagoreans seeing number in everything.) Finding empirically-dependable interpretations in the world does not comprehend the world, and we need to understand that concerning the theories that are embodied in computers (e.g., object-oriented approaches) and how that does not capture the world entire (sometimes not even a little bit). And empirical suitability matters and must be confirmable.

We must neither foster nor embrace notions that our world is apprehended by a mechanism.

I would hope that a treatment of logic relevant to Computer Science will address this distinction as well as shed light on the marvelous ways computers have become dependable for implementing and simulating structures of importance in our lives. From a scientific perspective, I think Albert Einstein said it clearly (https://orcmid.com/blog/2010/02/abstraction-einstein-on-theoryreality.asp).

As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality.

[Einstein1921]

References

[Einstein1921]
Einstein, Albert. Geometry and Experience: An expanded form of an address to the Prussian Academy of Sciences in Berlin on January 27, 1921. Pp. 25-56 in [Einstein1922]

[Einstein1922]
Einstein, Albert. *Sidelights on Relativity*. G. B. Jeffrey and W. Perret, translators. E. P. Dutton (New York: 1922); Dover edition (New York: 1983) ISBN 0-486-24511-X pbk.

Contains Ether and Relativity (1920) and Geometry and Experience (1921). *Sidelights on Relativity* is reprinted, with commentary of the editor, on pp. 235-262 of [Hawking2007]

[Hawking2007]
Hawking, Stephen (ed.). *A Stubbornly Persistent Illusion: The Essential Scientific Works of Albert Einstein*. Running Press (Philadelphia: 2007). ISBN 0-7624-3003-6.