

PART I

Version 2.28: 2004/05/14 13:52:33.595 GMT-5

John Greiner
Ian Barland

This work is produced by The Connexions Project and licensed under the
Creative Commons Attribution License *

Abstract

(Blank Abstract)

part I

1 Why study proofs?

The ancient Greeks loved to sit around and argue. But at the end of the day, they wanted to sit back and decide who had won the argument. When Socrates *claims* that one statement followed from another, is it actually so? Shouldn't there be some set of rules to officially determine when an argument is correct? Thus began the formal study of logic.

ASIDE: *logic* - mid-14c., from O.Fr. *logique*, from L. (*ars*) *logica*, from Gk. *logike* (*technē*) "reasoning (art)," from *logos* "reason, idea, word," from *legein* "speak."

These issues are of course still with us today. And while it might be difficult to codify real-world arguments about (say) gun-control laws, programs *can* be fully formalized, and correctness can be specified. We'll look at three examples where formal proofs are applicable:

- playing a simple game, WaterWorld;
- checking a program for type errors;
- circuit verification.

Many other areas of computer science routinely involve proofs, although we won't explore them here. Manufacturing robots first prove that they can twist and move to where they need to go before doing so, in order to avoid crashing into what they're building. When programming a collection of client and server computers, we usually want to prove that the manner in which they communicate guarantees that no clients are always ignored. Optimizing compilers prove that, within your program, some faster piece of code behaves the same as and can replace what you wrote. With software systems controlling more and more life-critical applications, it's important to be able to *prove* that a program always does what it claims.

*<http://creativecommons.org/licenses/by/1.0>

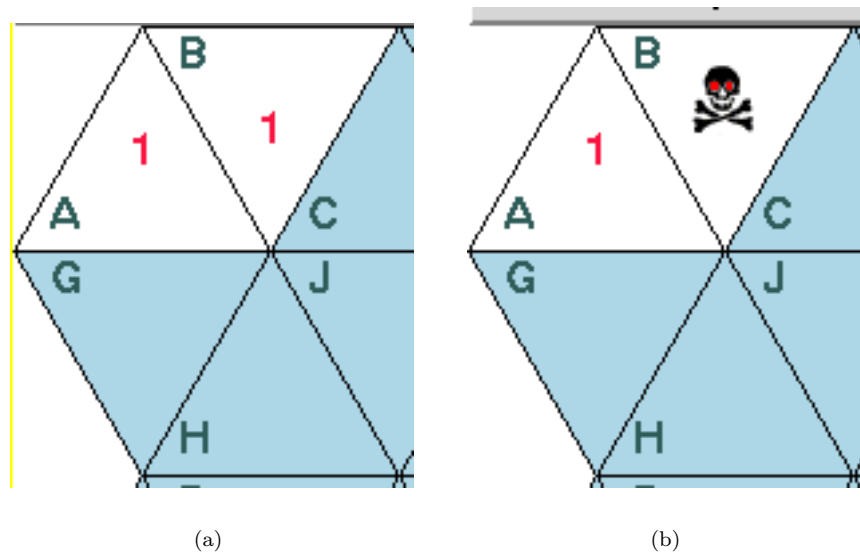


Figure 1: Glimpes of two different WaterWorld boards

1.1 WaterWorld

Consider a game called WaterWorld, where each location is either empty sea or contains pirates. When you enter a location, you must *correctly* anticipate whether or not it contains pirates.

- If you anticipate pirates, you sail recklessly to avoid them.
 - If you are correct, you successfully avoid them.
 - However, if you are incorrect and there were no pirates, then you wreck your ship and drown.
- If you anticipate the location is empty, you let your crew relax.
 - If you are correct, you can measure the pollution content of the water and find out how many neighboring locations have pirates, although not which ones.
 - However, if you are incorrect, and there are pirates, they will keel-haul your ship and enslave you.

(Observe a slight asymmetry: only after exploring a safe location do you get further information.) We wish to know when we have a *guaranteed* safe move.

For instance, in the first board, is there at least one move guaranteed safe? What is your reasoning that they are safe? How about in the second board?

1.2 Type Checking

When writing a program, we'd like to simply look at the program and determine whether it has any bugs, without having to run it. We'll see in the future, however, that such a

general problem cannot be solved. Instead, we focus on finding more limited kinds of errors. **Type checking** determines whether all functions are called with the correct *type* of inputs. E.g., the function `+` should be called with numbers, not Booleans, and a function which a programmer has declared to return an integer really should always return an integer. Consider the following program:

```
int SEEMS_LIKE_FOREVER = 24; // Time enough to procrastinate.

// goSeeMovie?:
// When invited to see a movie, do you accept?
// Depends on how many hours until homework is due,
// and on how much time is needed to do the homework.
//
// Current implementation: If the homework is due far in the
// future, we'll ignore it even if it will take a long time.
//
bool goSeeMovie?( int hrsTilHwDue, int timeNeeded ) {
    if (hrsTilHwDue >= SEEMS_LIKE_FOREVER)
        true;
    else
        (hrsTilHwDue >=timeNeeded);
}

// Example:
// goSeeMovie?( 8, 8 ) = false
// goSeeMovie?( 48, 48 ) = true
```

One reason programmers are required to declare the intended type of each variable is so that the computer (the compiler) can *prove* that certain errors won't occur. How can you or the compiler prove, in the above, that `goSeeMovie?` really always returns a Boolean value as promised? For now, you can presume that it is called with integers, as indicated, but given an entire program, you'd want to prove that was the case.

Consider this variant:

```
int SEEMS_LIKE_FOREVER = 24;

bool goGetRootCanal?( int hrsTilHwDue, int timeNeeded ) {
    if (hrsTilHwDue >= SEEMS_LIKE_FOREVER)
        true;

    else if (hrsTilHwDue > SEEMS_LIKE_FOREVER) // This branch never reached.
        timeNeeded;
    else
        false;
}
```

Most compilers will reject this code, since it appears that the highlighted code can return an integer, instead of a Boolean. However, using some math knowledge about what `>=` and `>` mean, we could prove the second conditional branch is *never* taken. Thus, the code is perfectly safe, although admittedly a bit strange. So we can see that the compiler's type checking is overly conservative.

ASIDE: In general, compilers tend to focus on simple compile-time checking for type errors and other relatively simple bugs. We'll see later¹ there is a good reason for this.

1.3 Circuit Verification

Given a circuit's blueprints, will it work as advertised? In 1994, Intel had to recall 5 million of its Pentium processors, due to a mistake in the circuit: on some input numbers, it didn't divide correctly. This cost 475 million dollars, lots of bad publicity, and it happened *after* intensive testing. Could it have been possible to have a program try to prove the chip's correctness or uncover an error?

Software and hardware companies are increasingly turning to the use of automated proofs, rather than semi-haphazard testing, to verify (parts of) large products correct. However, it is a formidable task, and how to do this is also an active area of research.

There are of course many more examples; one topical popular concern is verifying certain security properties of electronic voting slates (often provided by vendors who keep their source software a proprietary secret).

Having proofs of correctness is not just comforting; it allows us to save effort (less time testing, and also able to make better optimizations), and prevent recall of faulty products. But: who decides a proof is correct – the employee with best SAT scores?!? Is there some trusted way to verify proofs, besides careful inspection by a skilled, yet still error-prone, professional?

2 Just what is a proof? (informal)

Example 1:

The following submission from an anonymous engineer to the January, 1902 edition of Popular Mechanics caught my eye. Seems like something every Boy/Girl Scout and Architect should know.

"HOW TO USE THE WATCH AS A COMPASS: Very few people are aware of the fact that in a watch they are always provided with a compass, with which, when the sun is shining, the cardinal points can be determined. All one has to do is to point the hour hand to the sun and south is exactly half way between the hour and the figure 12 on the watch. This may seem strange to the average reader, but it is easily explained. While the sun is passing over 180 degrees (east to west) the hour hand of the watch passes over 360 degrees (from 6 o'clock to 6 o'clock). Therefore the angular movement of the sun in one hour corresponds to the angular movement of the hour hand in half an hour; hence, if we point the hour hand toward the sun the line from the point midway between the hour hand and 12 o'clock to the pivot of the hands will point to the south. – Engineer."

They give an argument of correctness; is that really a proof? Well, it has some unstated assumptions: e.g., the sun is at its highest (northernmost) point of its transit at noon. Is this actually true? Does it depend on the time of year? I'm not exactly sure (and will have to sit down and scratch my head and draw pictures of orbits, to convince myself). Certainly there are at least a couple of caveats: even beyond account for Daylight Savings Time, the solar-time and clock-time only align at time-zone boundaries, and they drift up to an hour apart, before the next boundary rectifies the difference.

¹<http://cnx.rice.edu/content/m10719/latest/>

The intent of this anecdote was to give enough evidence to convince you; not necessarily to be a complete, stand-alone self-contained proof. But be careful to distinguish between something which sounds reasonable, and something that you're certain of (e.g. assertions of how burning oil raises to atmospheric CO_2 levels, vs assertions of how big worldwide petroleum reserves are). That's fine for casual emails, but what about when you need to be sure?

2.1 An argument by form

How can we tell true proofs from false ones? What, exactly, are the rules of a proof? These are the questions which will occupy us.

Proofs are argument by form. We'll illustrate this with three parallel examples of a particular proof form called **sylogism**.

Example 2:

- 1.All people are mortal.
- 2.Socrates is a person.
- 3.Therefore, Socrates is mortal.

Example 3:

- 1.All [substitution ciphers] are [vulnerable to brute-force attacks].
- 2.The [Julius Caesar cipher] is a [substitution cipher].
- 3.Therefore, the [Julius Caesar cipher] is [vulnerable to brute-force attacks].

Note that you don't need to know anything about cryptography to know that the conclusion follows from the two premises. (Are the premises indeed true? That's a different question.)

Example 4:

- 1.All griznoxes chorble happily.
- 2.A floober is a type of griznox.
- 3.Therefore, floobers chorble happily.

You don't need to be a world-class floober expert to evaluate this argument, either.

ASIDE: Lewis Carroll, a logician, has developed many whimsical examples² of syllogisms and simple reasoning. (Relatedly, note how the social context of Carroll's examples demonstrates some feminist issues in teaching logic³.)

As you've noticed, the *form* of the argument is the same in all these. If you are assured that the first two premises are true, then, without any true understanding, you (or a computer) can automatically come up with the conclusion. A syllogism is one example of a **inference rule** – that is, a rule form that a computer can use to deduce new facts from known ones.

²<http://home.earthlink.net/~lfdean/carroll/puzzles/logic.html>

³<http://www.indiana.edu/~koertge/rfemlog.html>

2.2 Some non-proofs

Of course, not all arguments are valid proofs. Identifying invalid proofs is just as interesting as identifying valid ones.

Homer: Ah, not a bear in sight. The Bear Patrol must be working.

Lisa: That's specious reasoning, Dad.

Homer: Thank you, honey.

Lisa: By your logic, this rock keeps tigers away.

Homer: Oh? How does it work?

Lisa: It doesn't work.

Homer: Uh-huh.

Lisa: It's just a stupid rock.

Homer: Uh-huh.

Lisa: But I don't see any tigers around here, do you?

[pause]

Homer: Lisa, I want to buy your rock!

[A moment's hesitation; then, money changes hands.]

Example 5:

1. Warm cola tastes bad.
2. Warm salt-water tastes bad.
3. Therefore, mixing them together tastes bad.

The conclusion is true, I can assure you. But does this really follow? Here is an argument with a similar form:

1. Ice-cold coke tastes good.
2. Ice coffee tastes good.
3. Therefore, mixing them together tastes good.

Unfortunately, this time the conclusion is *not* true. Either the first proof wasn't valid, or it was somehow relying on an assumed property of mixing-together-bad-tasting-things (a property which doesn't hold for good-tasting things). A chef can argue against the latter: you can sometimes mix together bad-tasting things to get good-tasting things – for example, mix a pile of salt and lots of leavened dry crunchy bread to get saltines.

ASIDE: Of course, this is all assuming taste is universal, or even for a single person is consistent over time. Prepend "This morning, I thought that ..."

The point illustrated, is that often real-world arguments incorrectly imply that their result follows from the *form* of the argument, when in fact the form is not valid in the way a syllogism is. This fallacy can be illuminated by finding a different domain in which the argument fails. (The original argument, if its conclusion was indeed true, must be patched either by adding the unspoken assumptions or fixing the invalid form.) The practice of searching for domains which invalidate the argument can help both sides of a debate hone in on bringing the unspoken assumptions to light.

Exercise 1:

Mistakes in syllogisms are hard to make: what are the only two ways to have an error in a syllogism?

Solution:

1. The argument isn't actually in syllogism form. An *incorrect* syllogism:

(a) All people don't know my file's password.

ASIDE: Equivalent to "Nobody knows my file's password". Why did we phrase this sentence awkwardly? Because a syllogism requires the first premise to be of the form "All somethings have someproperty".

(b) All hackers are people.

(c) Therefore, my file is secure from hackers.

WARNING: Not the syllogism's conclusion.

To be a syllogism, the conclusion would have to be "all hackers don't know my file's password." The file may or may not be secure, but the above doesn't prove it.

2. One of the two premises is wrong.

(a) All people don't know my file's password.

WARNING: Is this really true?

(b) All hackers are people.

WARNING: Is this really true?

(c) Therefore, all hackers don't know my file's password.

This proof fails of course, if some hackers are non-people (e.g., programs), or if some people know the password. (In fact, presumably *you* know the password!)

Of course, even if a proof fails, the conclusion might be true for other reasons. An incorrect argument doesn't prove the conclusion's opposite!

2.3 Other Inference rules

Of course, there are more ways to deduce things, beyond a syllogism.

- Who decides what the valid inference rules are?
- Is it always clear when people have used the inference rules correctly?

Consider the following argument:

1. (A) is next to one pirate; and
2. (A) has only one explored neighbor;
3. Conclusion: If you are a square next to (A), then you contain a pirate.

This conclusion is not true: (B) is certainly next to (A), but it doesn't contain a pirate. Alternately, consider that same proof applied to this board: Here, we know that (G) is safe, contrary to the conclusion. (In real life, I make this mistake all the time, when playing WaterWorld, arrggh! –The Author.)

The problem is that the author of the argument presumably meant to conclude "all explored neighbors of (A) contain a pirate".

Before we can study exact proofs, we need a way of writing exactly what we mean. This will occupy us for the next module.

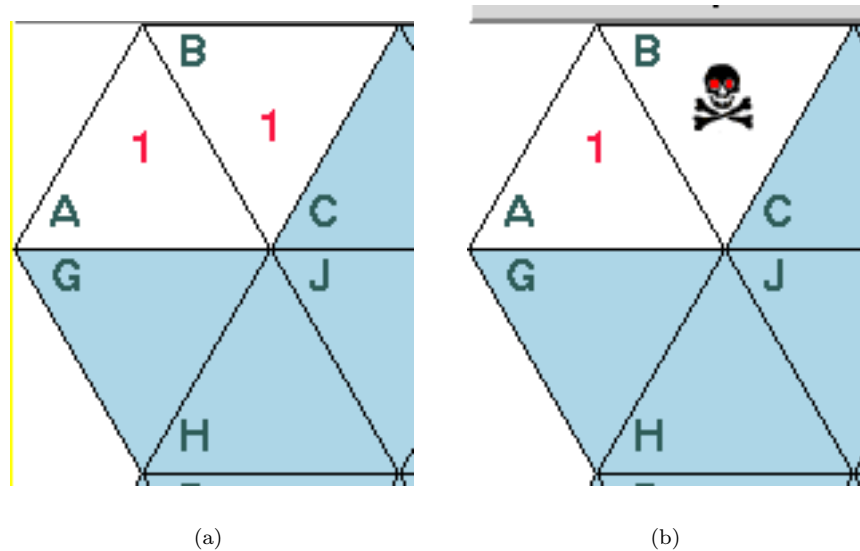


Figure 2: Glimpes of two different WaterWorld boards

2.4 The need for a precise language

These previous glitches in the WaterWorld arguments both arise, of course, because we were sloppy about what each sentence meant exactly. We used informal English – a fine language for humans, who can cope with remarkable amounts of ambiguity – but not a good language for specifying arguments.

ASIDE: Laws and contracts are really written in a separate language from English – legalese – full of technical terms with specific meanings. This is done because, while some ambiguity is tolerable in 99% of human interaction, sometimes the remaining 1% is problematic. However, legalese still contains intentionally ambiguous terms. When is a punishment cruel and unusual? What exactly is the Community Standard of indecency? The legal system tries to simultaneously be formal about laws, yet also be flexible to allow for unforeseen situations and situation-specific latitude. (And the result of this tension is the position of Judge.) In this course, we are trying to be entirely specific, with no ambiguity.

Consider, from a previous example⁴, the statement “. . .[this is something] all boy scouts and engineers should know”. Does this mean all people who are *both*, or everybody who is at least one or the other? Genuinely ambiguous, in English! (In English, people often use “and/or” to mean “one or the other or possibly both”.)

We’ll next look at a way to specify some concepts non-ambiguously, at least for WaterWorld. We need to be more careful about how we state our facts and how we use these known facts to deduce other facts. Remember, faulty reasoning might not just mean losing a silly game. Hardware and software bugs can lead to significant bodily harm (Imagine soft-

⁴<http://cnx.rice.edu/content/m10714/latest/#watch-as-compass>

ware bugs in an airplane autopilot or surgical robot system), security loopholes(Netscape⁵, IE⁶) or expensive recalls⁷.

One reaction to the above arguments is "Well, big deal – somebody made a mistake (mis-interpreting or mis-stating a claim); that's their problem. (And sheesh, they sure are dolts!)" But as a programmer, that's not true: Writing large systems, human programmers *will* err, no matter how smart or careful they are. Thus we are looking for systematic ways to prove code correct or incorrect.

ASIDE: Other professions have checklists, protocols, and regulations to minimize human error; programming is no different, except that the industry is still working on exactly what the checklists or training should be.

In our study of formal logic, we'll need two things: a precise syntax and vocabulary for expressing things without ambiguity, and rules to infer new facts from old. We'll first work on the syntax and vocabulary part, by developing a formal system for the game of WaterWorld.

⁵<http://wp.netscape.com/security/notes/>

⁶<http://www.microsoft.com/technet/security/current.asp>

⁷<http://cnx.rice.edu/content/m10714/latest/#intel-bug>