Proving Implications

James Wilcox and Kevin Zatloukal

August 2024

A function is implemented correctly if, for every input that satisfies the **precondition**, it produces an output that satisfies the **postcondition**. We can confirm that this is the case in two steps.

First, reading through our code carefully, we can collect the set of facts that must hold at each return statement. Some of these facts come from the precondition and others from the code itself.

Once we have collected those facts, it remains prove that the facts known at the return statement guarantee that the facts required by the postcondition must also hold. If we an do that for all return statements, then we can be confident that the code is correct.

In these notes, we will focus on the second of those two steps: showing that a set of given facts guarantee that another fact or facts must also hold. Demonstrating such a relationship is proving an "implication". We say that the proven fact "follows from" or "is implied by" the known facts.

Three techniques for proving implications will be sufficient for all of our reasoning in this course. Our primary technique is "proof by calculation". We will also need "proof by cases" and "structural induction". The latter two work by turning the complex implication we want to prove into two (or more) simpler implications. Once the implications are simple enough, we can prove them by calculation. For this reason, calculation is our workhorse technique.

Facts

Our "facts" are statements, believed to be true, about the **variables** in the code.

To prove implications, we will work in our math notation. We will use a mathematical variables to represent the variable of the same name in the code. For example, the variable `x` in the code, will be x in our math. Furthermore, in this setting, each variable comes with a data type. For example, if we know that x is an integer in the code (`bigint`), then we know that $x : \mathbb{Z}$ in our math notation.

Our facts about variables will typically be equations or inequalities, though the negation of a equality ($\neq$) is also allowed. For example, if our variables are $x, y, z : \mathbb{Z}$, then we could have the facts

$$x = 3,\ y < 10,\ \text{and}\ z \neq 0$$

The ","s in this list are logical "and"s.[1] We can also write two-sided bounds as a single fact, e.g., $5 \leq y < 10$ is shorthand for the two facts $5 \leq y$ and $y < 10$.

Equalities make sense for all the types of our math notation, but inequalities only make sense for numbers, that is, types $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$. We will generally assume integers unless stated otherwise. If $x : \mathbb{N}$, we can think of this as an integer $x : \mathbb{Z}$ with the additional known fact that $x \geq 0$.

[1] In rare cases, we will need an "or", but we will try to avoid that.

Calculation

Given a set of facts about the variables, our primary tool for proving new facts is calculation. To prove a new equality or inequality, we start with an expression and then use the known facts to establish its relationship with other expressions. For example, suppose that we know $x = y$ and $z < 10$. Then, we can calculate

$$\begin{aligned} x + z &= y + z && \text{since } x = y \\ &< y + 10 && \text{since } z < 10 \end{aligned}$$

The text on the right explains why the relationship on that line holds.

The first line says that $x + z = y + z$ holds because we know that $x = y$ holds. Specifically, since $x = y$, we can *replace* the "x" in "$x + z$" with a "y", giving us "$y + z$". The keyword **since** thus indicates that the stated equation holds by substituting the right side of some equation (in this case, "$x = y$") in the expression at the beginning of the line. Here, the line begins with "$x + z$" and substituting the right side of "$x = y$" (y) for the left (x) gives us "$y + z$".

We can also use **since** to substitute the left side of an inequality ($\leq$ or $<$) for the right side. For example, if we instead knew that $x < y$, then replacing the x in "$x + z$" with the larger value y would make the whole expression larger[2]. Thus, we could see that $x + z < y + z$ *since* $x < y$.

Note that, in our calculation block above, the second line has nothing to the left of "$<$"! In that case, the left-hand side is implicitly the right-hand side of the previous line. In this case, the previous line ends with $y + z$, so the second-line is saying $y + z < y + 10$ since $z < 10$.

The calculation above proves that $x + z < y + 10$. This follows from the facts that $x + z = y + z$ and $y + z < y + 10$. More generally, any chain of equalities establishes that the first expression equals the last one. A chain of equalities and "$\leq$" inequalities, proves that the first expression is "$\leq$" the last one. A chain of equalities and "$\leq$" inequalities and "$<$" inequalities, proves that the first expression is "$<$" the last one.

Note that a calculation with "$\leq$ on one line and "$\geq$" on another would not prove anything about how the first and last expressions relate to each other. For that reason, calculations that use "$\leq$" and "$<$" should only use those two types of inequalities, and calculations that use "$\geq$" and "$>$" should only use those two. (Both types can include equalities, "$=$", as well though.)

The calculation above proves that $x + z < y + z$ follows from the given facts $x = y$ and $z < 10$. This means that $x + z < y + z$ must be true *for any* values of the variables x, y, and z that satisfy the conditions $x = y$ and $z < 10$. For example, it holds when $x = y = 50$ and $z = 5$, but also when $x = y = z = 1$ and *infinitely* many other values.

In mathematical logic, proving that a claim holds for all values of the variables in this manner (as opposed to using the techniques we will see later on) is called a "direct proof". We will continue to call it "calculation" since we will establish facts solely through the use of calculation blocks like the one above.

Using Definitions

Each line in the definition of a function is an equality, so they can also be used in a calculation blocks. For example, suppose that we have the following definition of a function $f : \mathbb{Z} \to \mathbb{Z}$:

$$f(x) := 2x + 1$$

This tells us that $f(x) = 2x + 1$, so we can substitute one side of this for the other. For example, we have

$$f(x) + z = 2x + 1 + z \quad \text{def of } f$$

Note that, in the explanation, we said "def of" rather than "since". The former indicates that we are using the definition of the named function rather than a specific fact that we happen to know. If we did not have

[2] If y appeared negated or in the denominator, then this substitution would make the expression smaller, but usually substituting the left of "$\leq$" gives us a "$\leq$" result.

a definition of f but happened to know that $f(4) = 10$, then we would instead say

$$f(4) + z = 10 + z \quad \text{since } f(4) = 10$$

This "since" indicates that the fact listed after it is a specific fact that was given to us about $f(4)$.

Finally, note that we can apply the definition of the function to any value of the appropriate type. For example, if we know that f has the definition above, then we can write

$$f(5) + z = 2 \cdot 5 + 1 + z \quad \text{def of } f$$

The fact that was substituted was $f(5) = 2 \cdot 5 + 1 = 11$, not the general definition $f(x) = 2x + 1$. We will assume that, for functions defined by pattern matching, the reader can perform this substitution in their head, so it is not necessary to write out the fact that $f(5) = 2 \cdot 5 + 1$ (or fact that $2 \cdot 5 + 1 = 11$ because the arithmetic is so simple).

Proof By Cases

In some cases, it may not be able to establish a fact using a calculation block without additional information.

Suppose, for example, that $f : \mathbb{Z} \to \mathbb{Z}$ had the following definition:

$$\begin{aligned} f(x) &:= 2x + 1 && \text{if } x \geq 0 \\ f(x) &:= 0 && \text{if } x < 0 \end{aligned}$$

In that case, we cannot use the fact that $f(x) = 2x + 1$ unless we happen to know that $x \geq 0$.

Even if we do not know anything about the value of x, we still may be able to prove facts about $f(x)$. To do so, we can split our analysis into **cases** that tell us which line in the definition of f applies.

For example, suppose we want to show that $f(x) > x$ is always true. As just noted, we can't replace $f(x)$ by either $2x + 1$ or 0 unless we know more about x.

If we happened to know that $x \geq 0$, then we could see that this claim is true as follows:

$$\begin{aligned} f(x) &= 2x + 1 && \text{def of } f \text{ (since } x \geq 0) \\ &= x + x + 1 \\ &\geq x + 1 && \text{since } x \geq 0 \\ &> x && \text{(since } 1 > 0) \end{aligned}$$

and if we happened to know that $x < 0$, then we could say that it is true as follows

$$\begin{aligned} f(x) &= 0 && \text{def of } f \text{ (since } x < 0) \\ &> x && \text{since } 0 > x \end{aligned}$$

Since one of the two conditions "$x \geq 0$" and "$x < 0$" must hold (they are exhaustive), we can conclude that $f(x) > x$ must be true in general. This approach of splitting up the proof into separate, exhaustive cases is called "proof by cases" and is our second technique for proving implications.

Before we more on, note that, in applying the definition of f above, we wrote "def of f (since $x \geq 0$)" in the first case. The second part "(since $x \geq 0$)" is necessary because we are not only using the definition of f but also the fact that $x \geq 0$. This happens whenever we define a function using side conditions. With pure pattern matching, this will usually not be necessary, but it is always necessary with side conditions. This is another reason to prefer pattern matching: it makes reasoning about functions easier.

As we just saw, proof by cases allows us to break up the proof of a difficult-to-prove claim into two (or more) claims that are easier to prove because we have more information in each case. Our next (and last) technique also has this form: it allows us to break up a hard claim into simpler claims that can usually then be proven by calculation.

Structural Induction

So far, we have seen only equalities between atomic types (namely, integers); however, equalities are also sensible facts about compound types, so we need to know how to prove those as well.

For simple compound types, this is not much more complicated. Two tuples or records are equal if and only if their corresponding components are equal, so equality of these types can always be restated in terms of equality of simpler types. For example, if we want to prove that $(f(x), y) = (3, 5)$, then we just need to show that $f(3) = 3$ and $y = 5$. Those are both equalities between numbers, so we can prove them by calculation or cases.

Proving an equality between two values of an inductive data type is more complex. Unlike tuples and records, values of these types (such as lists) can be arbitrarily large, so they don't just reduce to a finite number of equalities between numbers. Instead, we need a new proof technique. Thankfully, each inductively defined data type comes with a new way to reason about it, which is referred to as *structural induction*. (Just as we use "structural recursion" to calculate with inductive data types, we use "structural induction" to reason about them.)

The structure of an inductive argument follows the structure of the inductive data type. We first show the fact holds for all the basis elements of the type (by calculation) and then show that the fact holds in the recursive cases *assuming* that the fact is true for arguments of the constructors with the same type.

As an example, recall the List type which is defined inductively as follows:

$$\textbf{type } \mathsf{List} := \mathsf{nil} \mid \mathsf{cons}(x\colon \mathbb{Z},\ L\colon \mathsf{List})$$

We can define the function $\mathsf{sum} : \mathsf{List} \to \mathbb{Z}$ as follows:

$$\begin{aligned} \mathsf{sum}(\mathsf{nil}) &:= 0 \\ \mathsf{sum}(x :: L) &:= x + \mathsf{sum}(L) \end{aligned}$$

and the function $\mathsf{twice} : \mathsf{List} \to \mathsf{List}$ as follows:

$$\begin{aligned} \mathsf{twice}(\mathsf{nil}) &:= \mathsf{nil} \\ \mathsf{twice}(x :: L) &:= 2x :: \mathsf{twice}(L) \end{aligned}$$

Note that both functions are structurally recursive.

Now, suppose that we want to prove that $\mathsf{sum}(\mathsf{twice}(L)) = 2\,\mathsf{sum}(L)$ for all lists L. We can write a structural induction argument of this fact.

In the "base case", we prove that this claim holds for the basis element of the List type, nil. This means we need to prove the claim $\mathsf{sum}(\mathsf{twice}(\mathsf{nil})) = 2\,\mathsf{sum}(\mathsf{nil})$. This is simple enough that we can do it by calculation.

In the "inductive step", we prove that the claim holds for the recursively built elements of the List type, which are of the form $x :: L$ (i.e., $\mathsf{cons}(x, L)$). Here, we must prove $\mathsf{sum}(\mathsf{twice}(x :: L)) = 2\,\mathsf{sum}(x :: L)$.

When we do the second proof, structural induction allows us to assume that we have already proven that the claim hold for all the arguments to the constructor (cons) that also have type List. In the case of $x :: L$, the second argument L has type List, so we get to assume that we have already proven $\mathsf{sum}(\mathsf{twice}(L)) = 2\,\mathsf{sum}(L)$. This last fact is called the "Inductive Hypothesis". It is the most critical fact of any structural induction argument, so we write it out explicitly and cite it explicitly in any calculation that uses it.

Here is the complete proof in full detail:

Base Case We can calculate

$$\begin{aligned} \mathsf{sum}(\mathsf{twice}(\mathsf{nil})) &= \mathsf{sum}(\mathsf{nil}) && \text{def of } \mathsf{twice} \\ &= 0 && \text{def of } \mathsf{sum} \\ &= 2 \cdot 0 \\ &= 2 \cdot \mathsf{sum}(\mathsf{nil}) && \text{def of } \mathsf{sum} \end{aligned}$$

Inductive Hypothesis Suppose that $\mathsf{sum}(\mathsf{twice}(L)) = 2\,\mathsf{sum}(L)$ holds.

Inductive Step We can calculate

$$\begin{aligned}
\mathsf{sum}(\mathsf{twice}(x :: L) &= \mathsf{sum}(2x :: \mathsf{twice}(L)) && \text{def of } \mathsf{twice} \\
&= 2x + \mathsf{sum}(\mathsf{twice}(L)) && \text{def of } \mathsf{sum} \\
&= 2x + 2\,\mathsf{sum}(L) && \text{Ind. Hypothesis} \\
&= 2(x + \mathsf{sum}(L)) \\
&= 2\,\mathsf{sum}(x :: L) && \text{def of } \mathsf{sum}
\end{aligned}$$