Information and Inference: A Journal of the IMA () , 1–15
https://doi.org/

A Better Theory to Prove Program Correctness

YIYANG ZHOU
[]

1. Introduction

You all have learned how to prove that a program is correct. In particular, you know how to use *induction* to justify the correctness of *iterative* and *recursive* programs. However, that theory has a few limitations. For instance, it is very difficult to analyze an iterative program that has an early exit (that is, a *return* or *break* statement) within the loop, or complex control structures like *goto*s. Also, that theory is only good when you have already come up with the program, and it does not help whatsoever during the program development. Furthermore, if the program is flawed, you will not be able to very easily figure out what went wrong. And finally, I am sure that none of you enjoyed the process of proving those inductions!

Today, I will show you a theory that is more flexible, more illuminative during program development, and generally easier to work with.

2. Computer States

Before we begin our journey. Let's adopt a simple model for our computer. A computer has a RAM, and that RAM is divided into many pieces called *state variables*. Those variables have their names (like "x", "n", "$temp$", etc.), and can assume any value within their domains (or *types*). A *state* of a computer is a vector (or tuple) consists of the values of all its state variables. For example, suppose the RAM contains four state variables, an *int*, a *char*, a *boolean*, and a $float$. One of the possible states of the computer may be: $(-2, `A', true, 3.14)$.

A computer may start its execution in any state. This is called the *prestate*, denoted by the Greek letter sigma σ. Computers cannot dictate in which state it will start the execution of a program - even if the prestate does not make sense for the program. After a certain period of time, the computer may halt. We call the state that the computer halts in the *poststate*, denoted by σ'.

By convention, we use the name of the state variables directly when we talk about their initial values. We add an apostrophe to their names when we want to refer to their values after the execution. For instance, n' refers to the value of n after the execution.

There is also a special variable called *time*, usually denoted by t. Unlike normal state variables, the time variable cannot be manipulated by the program directly. It is important to include the time variable as part of the computer state, as it will make it possible for us to argue the termination of executions.

3. Specifications

A specification is a binary expression of prestate-variables and poststate-variables. For every prestate, the specification *specifies* which poststates are satisfactory. The goal of a computer is to deliver a satisfactory poststate when given a prestate as the "input".

For example, suppose there are two state variables x and y. One possible specification might be: $x' = x+1 \wedge y' = y$. For every prestate, this specification specifies that the poststate where x is incremented by 1 and y is unchanged is satisfactory (and there are no other poststate that are satisfactory). Another possible specification might be: $x' > x$. For every prestate, this specification specifies that those poststate where x is changed to a value larger than before are all satisfactory. It is noteworthy that y does not appear in this specification at all, so the computer is free to do whatever it seem fit to y. In our theory, we do not assume the identity of any state variable. If you want to preserve the value of a variable, you have to state it clearly in the specification.

You may be used to deal with specifications as pairs of *preconditions* and *postconditions*. Since they are both *assertions* about prestates and poststates, we can also write them as *binary expressions*. We can connect the two binary expressions using a logical implication to get a single expression. For example, suppose the precondition is "x is less than y", and the postcondition is "Make z to be a number in between x and y". We first translate both sentences into binary expressions. For the precondition, this is easy: $x < y$. For the postcondition, it is a little bit trickier. What we really want to say is "assign a value to z so that it is in between *the initial values of* x and y". We also probably do not want to change the values of x and y. Therefore, the postcondition translates to $x < z' < y \wedge x' = x \wedge y' = y$. Finally, we can connect the precondition and the postcondition with a logical implication: $x < y \longrightarrow x < z' < y \wedge x' = x \wedge y' = y$.

3.1. *Implementability of Specifications*

A specification is *implementable* if for every prestate σ, there is a poststate σ' such that σ and σ' together satisfy the specification.

Not all specifications are implementable. Consider this specification: $x > y \wedge y' = 0$. In order to satisfy it, the initial value of x must be greater than that of y, and y must be made 0, but if x is smaller than y to begin with, there is no mending of it. Remember that computers can only decide which poststate it delivers, but not which prestate it gets, so if the prestate already dissatisfies the specification, the computer cannot do anything to fix it.

4. Programs

You may have noticed that I carefully crafted my words in the previous sections. Instead of saying "the program may halts", I said "the computer may halts". And instead of saying "the poststate of the program", I said "the poststate delivered by the computer". Programs do not do anything by themselves. They are just instructions stored in a file (or, during the old days, on punch cards). It is the computer who is doing all the real work.

Indeed, the thing that specifications specify is not the programs. It is the *behaviors* of the computer. Through specifications, we tell the computer what are the things we want it to do. In other words "what are the desired behaviors". In this sense, programs are no different from specifications. The computer reads the programs, and behaves according to the code. The only difference is that computers know how to satisfy a program by themselves, but they do not necessarily know how to satisfy any random specification.

4.1. *Programming Notations*

The big take-away in this section is that all programs are also specifications. Therefore, they can all be translated into binary expressions of prestates and poststates. Modern programming languages have so many features that it becomes very hard to argue about them. We will only use a minimal amount of programming features today, but they will get the job done.

TABLE 1 *Truth table for* **if** b **then** S **else** R **fi**

TTT	TTF	TFT	TFF	FTT	FTF	FFT	FFF
T	T	F	F	T	F	T	F

4.1.1. ok

The first feature is *ok*, the so-called *pass* statement. It means "Don't do anything." It can be expanded into the binary expression $x' = x \wedge y' = y \wedge z' = z \wedge \ldots$. *ok* is a program.

4.1.2. Assignment

The second feature is $x := e$, the assignment statement. It says "make x to become e, and leave everything else unchanged". It can be translated to the binary expression $x' = e \wedge y' = y \wedge z' = z \wedge \ldots$. Note how this is different from just $x' = e$. The latter does not preserve the values of other variables - the computer is free to alter the values of y and z, because you did not tell it to not to do so. Assignment is a program.

4.1.3. **if then else fi**

The third feature is the **if** b **then** S **else** R **fi** statement. This specification says: "If b is true, do according to S, otherwise do according to R". This specification can be translated to

$(b \to S) \wedge (\neg b \to R)$ or $(b \wedge S) \vee (\neg b \wedge R)$.

but we will use a more concise notation, the eponymous **if then else fi** operator. It is an operator with three operands, and its truth table is given as table 1.

If b can be evaluated by the computer, and both S and R are programs, then **if** b **then** S **else** R **fi** is a program.

4.1.4. sequential composition

The fourth feature we will use is called the"sequential composition". It looks like this: $S.R$. This specification says: "Do according to S, then do according to R". We can express this semantics with binary expressions like this:

$$S.R = \exists x'', y'', \ldots \cdot (< \text{replace } x', y', \ldots \text{ with } x'', y'', \ldots \text{ in } S > \wedge < \text{replace } x, y, \ldots \text{ with } x'', y'', \ldots \text{ in } R >)$$

This expression looks intimidating, but it is actually quite intuitive. It essentially says that there exists an intermediate state (consists of all the double primed variables), such that it can be both a poststate of S and a prestate of R at the same time. A word of caution, though. Before replacing the variables in both specifications, you must expand any programming notation in them. For example, you must expand $x := e$ to $x' = e \wedge y' = y$ before you substitute x'' for x'. Otherwise you might miss some of the "hidden" variables.

Example 1

$$
\begin{aligned}
& x' = x \lor x' = x+1 \;.\; x' = x \lor x' = x+1\\
=& \exists x'' \cdot (x'' = x \lor x'' = x+1) \land (x' = x'' \lor x' = x''+1)\\
=& \exists x'' \cdot (x'' = x \land x' = x'') \lor (x'' = x+1 \land x' = x'') \lor (x'' = x \land x' = x'') \lor (x'' = x+1 \land x' = x''+1)\\
=& \exists x'' \cdot (x'' = x \land x' = x'') \lor \exists x'' \cdot (x'' = x+1 \land x' = x'') \lor \exists x'' \cdot (x'' = x \land x' = x'') \lor \exists x'' \cdot (x'' = x+1 \land x' = x''+1)\\
=& x' = x \lor x' = x+1 \lor x = x' \lor x' = x+2\\
=& x' = x \lor x' = x+1 \lor x' = x+2.
\end{aligned}
$$

The original specification says: “Add 1 to x or leave it unchanged, then add 1 to x or leave it unchanged”. As expected, after simplification, the net result is “add 0, or 1, or 2 to x”.

If both S and R are programs, then $S\ .\ R$ is a program.

Substitution Law A very convenient law that makes sequential compositions much easier to analyze is the Substitution Law.

$$x := e\ .\ P \iff <\text{replace every } x \text{ in } P \text{ with } e>$$

Example 2

$$
\begin{aligned}
& x := x+5 \;.\; y := x+y\\
=& x := x+5 \;.\; (x' = x \land y' = x+y)\\
=& x' = x+5 \land y' = x+5+y
\end{aligned}
$$

As previously mentioned, before the substitution, you must expand any programming notation in P.

For chained assignments, it is best for us to apply the Substitution Law from the right hand side first:

Example 3

$$
\begin{aligned}
& x := 1 \;.\; y := 2 \;.\; x := x+y\\
=& x := 1 \;.\; (y := 2 \;.\; x' = x+y \land y' = y)\\
=& x := 1 \;.\; x' = x+2 \land y' = 2\\
=& x' = 1+2 \land y' = 2\\
=& x' = 3 \land y' = 2
\end{aligned}
$$

If we were to start from the left hand side, we would be stuck after the first application:

Example 4

$$\begin{aligned}
&x := 1\ .\ y := 2\ .\ x := x+y \\
=&x := 1\ .\ x' = x \land y' = 2\ .\ x' = x+y \land y' = y \\
=&x' = 1 \land y' = 2\ .\ x' = 1+y \land y' = y
\end{aligned}$$

And we will have to expand the central sequential composition symbol, which is not very convenient.

4.1.5. Recursion
The last feature is recursion. Recursion is the only way to realize loops in our minimal programming language, but we will have to take a detour to talk about refinement first.

5. Refinement of Specifications

The goal of a programmer is to write a program such that the computer's behavior, when running according to the program, will satisfy the specification given to the programmer. Unlike project managers, who can use any logical symbols to come up with all kinds of crazy specifications (binary expressions), we programmers can only use the five features above - ok, assignments, **if then else fi**, sequential composition, and recursion. We have to figure out a way to tell the computer to do the same thing using only those features.

5.1. *Strength of Specifications*

Two specifications P and Q are equally *strong* if and only if each is satisfied whenever the other is. In other words, P and Q are *logically equivalent* as binary expressions. If we are given a specification, but we find another equally strong specification, we can implement the other specification instead, and the project manager will not notice any difference.

The specification S is said to be *stronger* than P if and only if whenever S is satisfied, so is P. In other words, P is *logically implied by* S as a binary expression. Similarly, if we are able to find a new specification that is stronger than the one given to us, we can implement the new one instead. After all, if the computer's behavior satisfies the stronger specification, it must also satisfy the original one.

The process of finding alternative specifications to implement is called "refinement". When P is refined by S, we write this fact as $P \Longleftarrow S$. The ultimate goal of refinement is to find a specification that is stronger than or equal to the original specification, that is also program.

Here are some examples:

$$\begin{aligned}
&x' > x \Longleftarrow x' = x+1 \land y' = y \\
&x' = x+1 \land y' = y \Longleftarrow x := x+1 \\
&x' = 0 \Longleftarrow \textbf{if}\ x = 0\ \textbf{then}\ ok\ \textbf{else}\ x := x-1\ .\ x' = 0\ \textbf{fi} \\
&x < y \to x < z' < y \land x' = x \land y' = y \Longleftarrow z := (x+y)/2
\end{aligned}$$

5.2. *Recursion*

For any specification that has been refined to a program, we regard it as a program too. $x' > x$ is a program, since through the first and the second refinement in the examples in the previous subsection, we get a stronger version of it that is also a program.

In the third example, the right hand side of the refinement uses the left hand side again in the else-clause. Is $x' = 0$ a program then? We will say that it is. That is, we will allow recursion in our program.

6. Laws of Refinement

Here are some of the laws of refinement. Since specifications are just binary expressions, all of these laws are just binary laws.

6.1. *Refinement by Steps*

6.1.1. Transitivity
Suppose we have $A \Longleftarrow B$ and $B \Longleftarrow C$ as two refinements, then it is true that $A \Longleftarrow C$. This law tells us that, when refining a specification, we do not have to refine it to a program in one step.

6.1.2. Transitivity of Sequential composition
Suppose we have $B \Longleftarrow D$, and $C \Longleftarrow E$. Then it is true that $B\,.\,C \Longleftarrow D\,.\,E$. This law tells us that if we have a refinement $A \Longleftarrow B\,.\,C$, but B or C (or both) has not yet been refined to a program, we do not have to refine the whole sequential composition again. We can refine each of the operands of the sequential composition, and we will get $A \Longleftarrow D\,.\,E$ for free.

6.1.3. Transitivity of If then else fi
This is very similar to the sequential composition one. Suppose we have $C \Longleftarrow E$, and $D \Longleftarrow F$. Then it is true that **if** b **then** C **else** D **fi** $\Longleftarrow$ **if** b **then** E **else** F **fi**. So if $A \Longleftarrow$ **if** b **then** C **else** D **fi**, we get $A \Longleftarrow$ **if** b **then** E **else** F **fi** for free.

6.2. *Refinement by Parts*

6.2.1. Conflation
Suppose we have $A \Longleftarrow B$ and $C \Longleftarrow D$, it is true that $A \wedge C \Longleftarrow B \wedge D$. This law tells us that when we have a specification that is a conjunction of two specifications (e.g., $Z = A \wedge B$), We can refine the two operands separately.

6.2.2. Conflation of Sequential Composition
Suppose $A \Longleftarrow B\,.\,C$ and $D \Longleftarrow E\,.\,F$ are true, then it is also true that $A \wedge D \Longleftarrow B \wedge E\,.\,C \wedge F$.

6.2.3. Conflation of If then else fi
Suppose $A \Longleftarrow$ **if** b **then** C **else** D **fi** and $E \Longleftarrow$ **if** b **then** F **else** G **fi** are true, then it is also true that $A \wedge E \Longleftarrow$ **if** b **then** $C \wedge F$ **else** $D \wedge G$ **fi**.

6.3. *Refinement by Cases*

$P \Longleftarrow$ **if** b **then** Q **else** R **fi** if and only if $P \Longleftarrow b \land Q$ and $P \Longleftarrow \neg b \land R$.

Refinement by Cases is not very helpful when refining a specification, but it is useful when we want to prove a refinement involving **if then else fi**.

7. Developing a Program

Let's try to develop a program using this new theory and prove its correctness.

Question 1 *Given two positive natural numbers m and n, such that $n \leq m$. Write a program such that x gets 2^t for some $t \in \mathbf{N}$ and $nx \leq m < 2nx$.*

The first step of any program development is to clearly write the requirement as a specification. One possibility is

$$1 \leq n \leq m \to nx' \leq m < 2nx' \land isp2(x')$$

where $isp2(x)$ is a predicate for saying "x is a power of 2".

Let P refer to the original specification. We first rearrange the inequality

$$P \Longleftarrow 1 \leq n \leq m \to x' \leq \frac{m}{n} < 2x' \land isp2(x')$$

The correctness of this step is pretty obvious, so we will skip the proof of it.

Next, let Q refer to the right hand side of the previous refinement, we have

$$Q \Longleftarrow x := 1\ .\ 1 \leq n \leq m \land x \leq \frac{m}{n} \land isp2(x) \to x' \leq \frac{m}{n} < 2x' \land isp2(x')$$

We will prove the correctness of this refinement.

Proof

$$\begin{aligned}
& RHS \\
\Longleftrightarrow\ & x := 1\ .\ 1 \leq n \leq m \land x \leq \frac{m}{n} \land isp2(x) \to x' \leq \frac{m}{n} < 2x' \land isp2(x') \\
\Longleftrightarrow\ & 1 \leq n \leq m \land 1 \leq \frac{m}{n} \land isp2(1) \to x' \leq \frac{m}{n} < 2x' \land isp2(x') \\
\Longleftrightarrow\ & 1 \leq n \leq m \land True \land True \to x' \leq \frac{m}{n} < 2x' \land isp2(x') \\
\Longleftrightarrow\ & 1 \leq n \leq m \to x' \leq \frac{m}{n} < 2x' \land isp2(x') \\
\Longleftrightarrow\ & LHS. \qquad \square
\end{aligned}$$

The first step employs the Substitution Law to replace every occurrence of x with 1. The second step replace $1 \leq \frac{m}{n}$ with a $True$ as we know this must be the case if $1 \leq n \leq m$. Also, $isp2(1)$ is true by arithmetic. The third step uses the identity of conjunction, and we then have the left hand side.

Next, let $R = 1 \leq n \leq m \wedge x \leq \frac{m}{n} \wedge isp2(x) \to x' \leq \frac{m}{n} < 2x' \wedge isp2(x')$, the new problem arose on the right hand side of previous step. We have

$$R \impliedby \textbf{if } 2x > \frac{m}{n} \textbf{ then } ok \textbf{ else } x := 2x \,.\, R \textbf{ fi}$$

Let's prove the correctness of this refinement too.

First, we use Refinement by Cases to divide the work into two parts:

7.0.1. Proof of $R \impliedby 2x > \frac{m}{n} \wedge ok$
Proof

$$R \impliedby 2x > \frac{m}{n} \wedge ok \tag{7.1}$$

$$\iff (x' \leq \frac{m}{n} < 2x' \wedge isp2(x')) \impliedby 2x > \frac{m}{n} \wedge ok \wedge (1 \leq n \leq m \wedge x \leq \frac{m}{n} \wedge isp2(x)) \tag{7.2}$$

$$\iff (x' \leq \frac{m}{n} < 2x' \wedge isp2(x')) \impliedby 2x > \frac{m}{n} \wedge (x' = x \wedge m' = m \wedge n' = n) \wedge (1 \leq n \leq m \wedge x \leq \frac{m}{n} \wedge isp2(x)) \tag{7.3}$$

$$\impliedby (x' \leq \frac{m}{n} < 2x' \wedge isp2(x')) \impliedby 2x > \frac{m}{n} \wedge (x' = x) \wedge (x \leq \frac{m}{n} \wedge isp2(x)) \tag{7.4}$$

$$\iff (x' \leq \frac{m}{n} < 2x' \wedge isp2(x')) \impliedby 2x' > \frac{m}{n} \wedge (x' = x) \wedge (x' \leq \frac{m}{n} \wedge isp2(x')) \tag{7.5}$$

$$\impliedby (x' \leq \frac{m}{n} < 2x' \wedge isp2(x')) \impliedby 2x' > \frac{m}{n} \wedge (x' \leq \frac{m}{n} \wedge isp2(x')) \tag{7.6}$$

$$\iff True. \qquad \square$$

The step from 7.1 to 7.2 uses the Portation Law[1] to bring the antecedent of the left hand side to the right. The step from 7.2 to 7.3 expands *ok*. 7.3 to 7.4 uses the Specialization Law[2] to remove some of the unnecessary conjuncts. 7.4 to 7.5 replaces x with x' on the right hand side since we have $x' = x$ as a conjunct. Finally 7.5 to 7.6 uses the Specialization Law again to get the LHS.

[1] Portation Law: $a \wedge b \to c \iff a \to (b \to c)$
[2] Specialization Law: a$\wedge b \to a$

7.0.2. Proof of $R \Longleftarrow 2x \le \frac{m}{n} \wedge (x := 2x \,.\, R)$
Proof

$$R \Longleftarrow 2x \le \frac{m}{n} \wedge (x := 2x \,.\, R) \tag{7.7}$$

$$\Longleftrightarrow R \Longleftarrow 2x \le \frac{m}{n} \wedge (1 \le n \le m \wedge 2x \le \frac{m}{n} \wedge isp2(2x) \to x' \le \frac{m}{n} < 2x' \wedge isp2(x')) \tag{7.8}$$

$$\Longleftrightarrow R \Longleftarrow 2x \le \frac{m}{n} \wedge (2x \le \frac{m}{n} \to (1 \le n \le m \wedge isp2(2x) \to x' \le \frac{m}{n} < 2x' \wedge isp2(x'))) \tag{7.9}$$

$$\Longleftrightarrow R \Longleftarrow 2x \le \frac{m}{n} \wedge (1 \le n \le m \wedge isp2(2x) \to x' \le \frac{m}{n} < 2x' \wedge isp2(x'))) \tag{7.10}$$

$$\Longleftarrow x' \le \frac{m}{n} < 2x' \wedge isp2(x') \Longleftarrow 2x \le \frac{m}{n} \wedge (1 \le n \le m \wedge isp2(x)) \wedge \tag{7.11}$$
$$(1 \le n \le m \wedge isp2(2x) \to x' \le \frac{m}{n} < 2x' \wedge isp2(x'))$$

$$\Longleftrightarrow x' \le \frac{m}{n} < 2x' \wedge isp2(x') \Longleftarrow 2x \le \frac{m}{n} \wedge (1 \le n \le m \wedge isp2(2x)) \wedge \tag{7.12}$$
$$(1 \le n \le m \wedge isp2(2x) \to x' \le \frac{m}{n} < 2x' \wedge isp2(x'))$$

$$\Longleftrightarrow x' \le \frac{m}{n} < 2x' \wedge isp2(x') \Longleftarrow 2x \le \frac{m}{n} \wedge (x' \le \frac{m}{n} < 2x' \wedge isp2(x')) \tag{7.13}$$

$$\Longleftarrow x' \le \frac{m}{n} < 2x' \wedge isp2(x') \Longleftarrow x' \le \frac{m}{n} < 2x' \wedge isp2(x') \tag{7.14}$$

$$\Longleftrightarrow True. \qquad \square$$

7.7 to 7.8 expands R on the right hand side and uses the Substitution Law to replace every x with $2x$. 7.8 to 7.9 uses the Portation Law in reverse to pull the $2x \le \frac{m}{n}$ out. 7.9 to 7.10 uses the Discharge Law[3] to get rid of the $2x \le \frac{m}{n}$. 7.10 to 7.11 uses the Portation Law to bring the antecedent of the LHS to the right. 7.11 to 7.12 replaces $isp2(x)$ with $isp2(2x)$ on the RHS, as if x is a power of 2, so is $2x$. 7.12 to 7.13 uses the Discharge Law again to simplify the RHS. Finally, 7.13 to 7.14 uses the Specialization Law to make both sides the same. And then we reach $True$.

Thus, we have proven that $R \Longleftarrow$ **if** $2x > \frac{m}{n}$ **then** *ok* **else** $x := 2x \,.\, R$ **fi**, and we are done with refinement for this problem.

All in all, we have

$$1 \le n \le m \to nx' \le m < 2nx' \wedge isp2(x') \Longleftarrow 1 \le n \le m \to x' \le \frac{m}{n} < 2x' \wedge isp2(x') \tag{7.15}$$

$$1 \le n \le m \to x' \le \frac{m}{n} < 2x' \wedge isp2(x') \Longleftarrow x := 1 \,.\, R \tag{7.16}$$

$$R \Longleftarrow \textbf{if } 2x > \frac{m}{n} \textbf{ then } ok \textbf{ else } x := 2x \,.\, R \textbf{ fi} \tag{7.17}$$

[3] Discharge Law: $a \wedge (a \to b) \Longleftrightarrow a \wedge b$

And we can translate this refinement into a C code. One of the possibility is

```c
int function(m, n)
{
   int x = 1;
   while 2*x <= (float)m/n
     {
       x = 2*x;
     }
   return x;
}
```

You might notice that all of our intermediate specifications disappear after we translate the refinement into C code. This is reason why it is difficult to prove the correctness of the program after it has been written in code instead of during development.

8. Time

Attentive readers may have noticed that our proof of correctness is incomplete - We have only proven the *partial correctness* of our program. That is, if the computer halts, it will produce a satisfactory poststate. But we have never justified why it must halt in the first place. To fix the flaw, we must also analyze the time it takes to run our program.

There are many ways to measure the running time of a program. After all, the time variable t is unitless. In our case, the only way that the computer might stuck running our program is to stuck in a loop. Therefore, we adopt a timing policy where each recursive call must increment the time variable by 1, before the call.

We argue that, if the preconditions are met, then the time it takes to run our program is at most $\log(\frac{m}{n})$. The revised specification we need to refine is

$$[1 \le n \le m \to nx' \le m < 2nx' \wedge isp2(x')] \wedge [1 \le n \le m \to t' \le t + \log(\frac{m}{n})]$$

where the first bracket talks about the partial correctness of our program, and the second bracket talks about the upper bound of execution time.

Luckily, we do not have to completely redo our work. Laws of Refinement by Parts tell us that we can refine each conjunct of the specification separately, and we have already done the first part.

I will state three (true) refinements first, and I will explain why we care about them

$$1 \le n \le m \to t' \le t + \log(\frac{m}{n}) \Longleftarrow 1 \le n \le m \to t' \le t + \log(\frac{m}{n}) \tag{8.1}$$

$$1 \le n \le m \to t' \le t + \log(\frac{m}{n}) \Longleftarrow x := 1 \,.\, 1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx}) \tag{8.2}$$

$$\begin{aligned} &1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx}) \Longleftarrow \\ &\textbf{if } 2x > \frac{m}{n} \textbf{ then } ok \textbf{ else } x := 2x \,.\, t := t+1 \,.\, 1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx}) \textbf{ fi} \end{aligned} \tag{8.3}$$

The first two refinements are pretty trivial - 8.1 is an identity, and both sides of 8.2 can easily be simplified to the same thing after one application of the Substitution Law, but the third one is not. We will prove that one in a moment.

Refinement 8.1, when combined with refinement 7.15 using Refinement by Parts gives us

$$P \wedge [1 \le n \le m \to t' \le t + \log(\frac{m}{n})] \impliedby Q \wedge [1 \le n \le m \to t' \le t + \log(\frac{m}{n})] \tag{8.4}$$

Refinement 8.2, when combined with refinement 7.16 gives us

$$Q \wedge [1 \le n \le m \to t' \le t + \log(\frac{m}{n})] \impliedby (x := 1) \wedge (x := 1) \,.\, R \wedge [1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx})$$

which can be simplified to

$$Q \wedge [t' \le t + \log(\frac{m}{n})] \impliedby x := 1 \,.\, R \wedge [1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx})] \tag{8.5}$$

Let's first modify 7.17 by a little bit. We add a time increment just before the recursion call

$$R \impliedby \textbf{if } 2x > \frac{m}{n} \textbf{ then } ok \textbf{ else } x := 2x \,.\, t := t + 1 \,.\, R \textbf{ fi}$$

We know that this modified version is still a correct refinement since R does not care about the time variable t at all, so we can eliminate the extra $t := t + 1$ term with one application of the Substitution Law.

Then let's combine 8.3 with this modified version of 7.17. We get

$$R \wedge [1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx})] \impliedby$$

$$\textbf{if } 2x > \frac{m}{n} \textbf{ then } ok \wedge ok \textbf{ else } (x := 2x \,.\, t := t + 1 \,.\, R) \wedge (x := 2x \,.\, t := t + 1 \,.\, [\ldots]) \textbf{ fi}$$

The $ok \wedge ok$ is obviously just ok, but the else-clause is a bit trickier. We will use Refinement by Parts again, in particular, the Conflation of Sequential Composition[4] to get

$$\textbf{else}\text{-clause} \impliedby (x := 2x \,.\, t := t + 1) \wedge (x := 2x \,.\, t := t + 1) \,.\, R \wedge [1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx})])$$

which can be simplified to

$$\textbf{else}\text{-clause} \impliedby (x := 2x \,.\, t := t + 1) \,.\, R \wedge [1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx})])$$

Altogether, we have

$$R \wedge [1 \le n \le m \wedge x \le \frac{m}{n} \wedge isp2(x) \to t' \le t + \log(\frac{m}{nx})] \impliedby \textbf{if } 2x > \frac{m}{n} \textbf{ then } ok \textbf{ else } x := 2x \,.\, t := t + 1 \,.\, R \wedge [\ldots] \textbf{ fi} \tag{8.6}$$

Combining 8.4, 8.5, and 8.6, we get

[4] Recall: If $A \impliedby B \,.\, C$ and $D \impliedby E \,.\, F$, then $A \wedge D \impliedby B \wedge E \,.\, C \wedge F$

$$P \wedge [1 \le n \le m \to t' \le t + \log(\frac{m}{n})] \Longleftarrow Q \wedge [1 \le n \le m \to t' \le t + \log(\frac{m}{n})]$$

$$Q \wedge [1 \le n \le m \to t' \le t + \log(\frac{m}{n})] \Longleftarrow x := 1 \,.\, R \wedge [1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx})]$$

$$R \wedge [1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx})] \Longleftarrow \textbf{if } 2x > \frac{m}{n} \textbf{ then } ok \textbf{ else } x := 2x \,.\, t := t+1 \,.\, R \wedge [\ldots] \textbf{ fi}$$

which is exactly the refinement we want for our new specification that includes time.

In conclusion, to add time to our proof, we don't have to redo all proofs again, but there are four things we must do. First, we choose a suitable timing policy and add increments of the time variable to our code accordingly. Second, we come up with a suitable specification for the time upper bound at each step of the original refinement. Three, we prove those time specifications using the same refinement structures as we did for the partial correctness, so that we can finally; Merge the two sets of refinements using Laws of Refinement by Parts.

The only thing remains to be done is the proof of 8.3, which we will do right now.

8.0.1. Proof of $1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx}) \Longleftarrow 2x > \frac{m}{n} \wedge ok$
This part is rather trivial. First, notice that after expanding *ok*, we get $t' = t$ on RHS. Then, since we have $x \le \frac{m}{n}$ on LHS, we know that $\log(\frac{m}{nx}) \ge 0$. Therefore the whole implication holds.

8.0.2. Proof of $1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx}) \Longleftarrow 2x \le \frac{m}{n} \wedge (x := 2x \,.\, t := t+1 \,.\, 1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx})$

Proof

$$RHS \tag{8.7}$$

$$\Longleftrightarrow 2x \le \frac{m}{n} \wedge (x := 2x \,.\, 1 \le n \le m \wedge x \le \frac{m}{n} \to t' \le t + 1 + \log(\frac{m}{nx})) \tag{8.8}$$

$$\Longleftrightarrow 2x \le \frac{m}{n} \wedge (1 \le n \le m \wedge 2x \le \frac{m}{n} \to t' \le t + 1 + \log(\frac{m}{2nx})) \tag{8.9}$$

$$\Longleftrightarrow 2x \le \frac{m}{n} \wedge (1 \le n \le m \wedge 2x \le \frac{m}{n} \to t' \le t + 1 + \log(\frac{m}{nx}) - 1) \tag{8.10}$$

$$\Longleftrightarrow 2x \le \frac{m}{n} \wedge (1 \le n \le m \wedge 2x \le \frac{m}{n} \to t' \le t + \log(\frac{m}{nx})) \tag{8.11}$$

$$\Longleftrightarrow 2x \le \frac{m}{n} \wedge (1 \le n \le m \to t' \le t + \log(\frac{m}{nx})) \tag{8.12}$$

$$LHS \Longleftarrow RHS \tag{8.13}$$

$$\Longleftrightarrow t' \le t + \log(\frac{m}{nx}) \Longleftarrow (1 \le n \le m \wedge x \le \frac{m}{n}) \wedge 2x \le \frac{m}{n} \wedge (1 \le n \le m \to t' \le t + \log(\frac{m}{nx})) \tag{8.14}$$

$$\Longleftarrow t' \le t + \log(\frac{m}{nx}) \Longleftarrow 1 \le n \le m \wedge (1 \le n \le m \to t' \le t + \log(\frac{m}{nx})) \tag{8.15}$$

$$\Longleftarrow t' \le t + \log(\frac{m}{nx}) \Longleftarrow 1 \le n \le m \wedge (t' \le t + \log(\frac{m}{nx})) \tag{8.16}$$

$$\Longleftarrow t' \le t + \log(\frac{m}{nx}) \Longleftarrow t' \le t + \log(\frac{m}{nx}) \tag{8.17}$$

$$\Longleftrightarrow True. \qquad \square$$

8.7 to 8.9 uses the Substitution Law twice. 8.9 to 8.11 uses the arithmetic of logarithm. 8.11 to 8.12 uses the Specialization Law. 8.13 to 8.14 uses the Portation Law. 8.14 to 8.15 uses the Specialization Law. 8.15 to 8.16 uses the Discharge Law. 8.16 to 8.17 again uses the Specialization Law. And then we have the same thing on both sides.

Now, we have officially finished the proof of correctness of our program.

9. Reverse Engineering a Program

We have learned how to prove the correctness of a program that we developed, but sometimes we need to justify other people's work, and not all of them are smart enough to use our theory and make others' lives easier.

Here is the another solution to the problem in the previous section

```
int function(m, n)
{
   int x = 1;
   int y = 2*n;
   while y <= m
     {
        y = 2*y;
        x = 2*x;
     }
   return x;
}
```

And the programmer who wrote this did not care to also write their refinements down, so we are forced to guess what they had in their mind.

First, we both start with the same specification. As a reminder, here is the original specification again

$$P = 1 \leq n \leq m \rightarrow nx' \leq m < 2nx' \land isp2(x')$$

The programmer then initialized two variables x and y, so we know the frst refinement should look something like

$$P \impliedby x := 1 \,.\, y := 2n \,.\, Q$$

where Q is an unknown specification.

Since Q takes the form of a while-loop, it is reasonable to guess that Q could be refined like this

$$Q \impliedby \textbf{if } y > m \textbf{ then } ok \textbf{ else } y := 2y \,.\, x := 2x \,.\, Q \textbf{ fi}$$

The only work for us to do is to make an educated guess about Q so that both refinements above hold true. There is no algorithmic way to do this, and you will have to be observant and use your experience. For a while-loop like this, usually it involves conjuncting the *loop invariant* to the antecedent of the original specification, but that is not guaranteed to work every time. I hope you can see why it is important to write down your refinements along with your code in the future.

Anyway, here is a suitable specification for Q I come up with

$$Q = 1 \leq n \leq m \land y = 2nx \land nx \leq m \land is2p(x) \rightarrow nx' \leq m < 2nx' \land isp2(x')$$

You can verify that the two refinements are correct, using techniques similar to the ones we have used. We then add time analysis to the specification to finish the proof

Let $T = 1 \leq n \leq m \land y = 2nx \land nx \leq m \rightarrow t' \leq t + \log(\frac{m}{nx})$. We can prove the following two refinements

$$1 \leq n \leq m \rightarrow t' \leq t + \log(\frac{m}{n}) \impliedby x := 1 \,.\, y := 2n \,.\, T$$

$$T \impliedby \textbf{if } y > m \textbf{ then } ok \textbf{ else } y := 2y \,.\, x := 2x \,.\, t := t + 1 \,.\, T \textbf{ fi}$$

10. Key Takeaways

Even if you did not follow everything, there are a few takeaways I want you to remember.

First, a computer has many *states*. During a computation, it transits from one state to another. The state in which it starts its execution is called the *prestate*. The state in which it halts is called the *poststate*.

A specification is a binary expression of prestates and poststates. It specifies the behavior of computers. For each prestate, it specifies which poststate(s) the computer must halt in. A computer behavior satisfies the specification if for all prestates, the computer produces a satisfactory poststate.

A program is a special kind of specification. It can only use a certain set of symbols and notations (called the programming language) to specify the computer's behavior. The goal of a programmer is to use the programming language to write a specification that is stronger or equal to the original specification. They can use the technique called *refinement* to nudge the original specification towards a program.

To prove the correctness of a program is to prove the refinement from the original specification to that final program. We first prove its *partial correctness*, and then conjoin an upper bound of the time variable to the specification to complete the proof.

Finally, if you use this theory to develop program in the future, always write down the refinements, even if you do not have time to do the proofs rigorously. That way, even if your program is flawed, you can easily find out which step went wrong.

11. Acknowledgement

The theory presented in this report is discussed in detail in the book *a Practical Theory of Programming* by Professor Eric Hehnner. You can find free copies of the book along with lecture recordings at https://www.cs.toronto.edu/~hehner/aPToP/. The problem in section 7 is adapted from the first question of assignment 2 of CSCB36 Fall 2020.