A Theory of Lazy Imperative Timing

Eric C.R. Hehner
Department of Computer Science
University of Toronto, Canada
hehner@cs.utoronto.ca

We present a theory of lazy imperative timing.

1 Introduction

Lazy evaluation was introduced as a programming language execution strategy in 1976 by Peter Henderson and Jim Morris [3], and by David Turner [7], and is now part of several programming languages, including Gofer, Miranda, and Haskell. It was introduced into the setting of functional programming, and has mainly stayed there, although it is just as applicable to imperative programs [1]. The name lazy evaluation is appropriate in the functional setting, but in the imperative setting it is more appropriately called lazy execution.

The usual, familiar execution of programs is called "eager execution". For example,

$x := 2;\;\; y := 3;\;\; \mathit{print}\ y$

is executed by first executing the assignment $x := 2$, and then the assignment $y := 3$, and then the *print* statement. If this is the entire program, a lazy execution executes only the assignment $y := 3$, and then the *print* statement, because the assignment $x := 2$ is unnecessary.

Here is a more interesting example. Let i be an integer variable, and let fac be an infinite array of integers.

$i := 0;\ fac(0) := 1;\ \textbf{while}\ true\ \textbf{do}\ i := i+1;\ fac(i) := fac(i-1) \times i\ \textbf{od};\ print\,fac(3)$

After initializing i to 0 and $fac(0)$ to 1 , there is an infinite loop that assigns $i!$ (i factorial) to each array element $fac(i)$. Then, after the infinite loop, the value of $fac(3)$ is printed. An eager execution will execute the loop forever, and the final printing will never be done. A lazy execution executes only the first three iterations of the loop, and then prints the desired result. Of course it is easy to modify the program so that the loop is iterated only 3 times in an eager execution: just replace $true$ by $i < 3$. But [4] gives a reason for writing it as above: to separate the producer (initialization and loop) from the consumer (printing). Many programs include a producer and a consumer, and each may be complicated, and it is useful to be able to write them separately. When written as above, we can change the consumer, for example to $print\,fac(4)$, without changing the producer. It is not the purpose of this paper to argue the relative merits of eager and lazy execution, nor to advocate any particular way of programming. The example is intended to show only that lazy execution can reduce execution time, and in the extreme case, it can be reduced from infinite time to finite time.

The analysis of eager execution time is well known; for example, see [2]. Some analysis of lazy execution time has also been done in the functional setting [6]. The purpose of this paper is to present a theory for the analysis of lazy execution time in the imperative setting. This paper is based on part of the PhD thesis of Albert Lai [5], but simplifications have been made to shorten the explanations, and a different measure of time is being used.

J. Derrick, B. Dongol and S. Reeves (Eds.):
Refinement Workshop 2018 (REFINE'18).
EPTCS 282, 2018, pp. 1–9, doi:10.4204/EPTCS.282.1

2 A Practical Theory of Programming

In a Practical Theory of Programming [2], we do not specify programs; we specify computation, or computer behavior. The free variables of the specification represent whatever we wish to observe about a computation, such as the initial values of variables, their final values, their intermediate values, interactions during a computation, the time taken by the computation, the space occupied by the computation. Observing a computation provides values for those variables. When you put the observed values into the specification, there are two possible outcomes: either the computation satisfies the specification, or it doesn't. So a specification is a binary (boolean) expression. If you write anything other than a binary expression as a specification, such as a pair of predicates, or a predicate transformer, you must say what it means for a computation to satisfy a specification, and to do that formally you must write a binary expression anyway.

A program is an implemented specification. It is a specification of computer behavior that you can give to a computer to get the specified behavior. I also refer to any statement in a program, or any sequence or structure of statements, as a program. Since a program is a specification, and a specification is a binary expression, therefore a program is a binary expression. For example, if the program variables are x and y, then the assignment program $x := y+1$ is the binary expression $x' = y+1 \wedge y' = y$ where unprimed variables represent the values of the program variables before execution of the assignment, and primed variables represent the values of the program variables after execution of the assignment.

We can connect specifications using any binary operators, even when one or both of the specifications are programs. If A and B are specifications, then $A \Rightarrow B$ says that any behavior satisfying A also satisfies B, where $\Rightarrow$ is implication. This is exactly the meaning of refinement. As an example, again using integer program variables x and y,

$$x := y+1 \quad \Rightarrow \quad x' > y$$

We can say "$x := y+1$ implies $x' > y$", or "$x := y+1$ refines $x' > y$", or "$x := y+1$ implements $x' > y$". When we are programming, we start with a specification that may not be a program, and refine it until we obtain a program, so we may prefer to write

$$x' > y \quad \Leftarrow \quad x := y+1$$

using reverse implication ("is implied by", "is refined by", "is implemented by").

3 Eager Timing

If we are interested in execution time, we just add a time variable t. Then t is the start time, and t' is the finish time, which is ∞ if execution time is infinite. We could decide to account for the real time spent executing a program. Or we could decide to measure time as a count of various operations. In [2] and [5], time is loop iteration count. In this paper, time is assignment count; I make this choice to keep my explanations short, but I could choose any other measure.

Using the same program variables x and y, and time variable t, the empty program ok (elsewhere called $skip$), whose execution does nothing and takes no time, is defined as

$$ok \quad = \quad x' = x \wedge y' = y \wedge t' = t$$

An example assignment is

$$x := y+1 \quad = \quad x' = y+1 \wedge y' = y \wedge t' = t+1$$

The conditional specifications are defined as

$$\textbf{if } a \textbf{ then } b \textbf{ else } c \textbf{ fi} \quad = \quad a \wedge b \vee \neg a \wedge c \quad = \quad (a \Rightarrow b) \wedge (\neg a \Rightarrow c)$$

$$\textbf{if } a \textbf{ then } b \textbf{ fi} \quad = \quad \textbf{if } a \textbf{ then } b \textbf{ else } ok \textbf{ fi}$$

A conditional specification is a conditional program if its parts are programs. The sequential composition $A;B$ of specifications A and B is defined as

$$A;B \;=\; \exists x'',y'',t''\cdot \quad (\text{for } x',y',t' \text{ substitute } x'',y'',t'' \text{ in } A) \;\wedge\; (\text{for } x,y,t \text{ substitute } x'',y'',t'' \text{ in } B)$$

Sequential composition of A and B is mainly the conjunction of A and B, but the final state and time of A are identified with the initial state and time of B. A sequential composition is a program if its parts are programs.

In our example program

$$i := 0;\; fac(0) := 1;\; \textbf{while}\; true\; \textbf{do}\; i := i+1;\; fac(i) := fac(i-1)\times i\; \textbf{od};\; print\, fac(3)$$

to prove that the execution time is infinite, there are two parts to the proof. The first is to write and prove a specification for the loop. Calling the specification $loop$, we must prove

$$loop \;\Leftarrow\; i := i+1;\; fac(i) := fac(i-1)\times i;\; loop$$

The specification we are interested in is

$$loop \;=\; t' = t+\infty$$

The proof uses[1] $\infty+1=\infty$, and is trivial, so we omit it. If we were to try the specification

$$loop \;=\; t' = t+n$$

for any finite number expression n, the proof would fail because $n+1 \neq n$. A stronger specification that succeeds is

$$loop \;=\; (\forall j\le i\cdot fac'(i) = fac(j)) \;\wedge\; (\forall j>i\cdot fac'(j) = fac(i)\times j!\,/\,i!) \;\wedge\; t' = t+\infty$$

but the final values of variables after an infinite computation are normally not of interest (or perhaps not meaningful). The proof uses $(i+1)! = i!\times(i+1)$ and is otherwise easy, so we omit it.

The other part of the eager timing proof is to prove

$$t' = t+\infty \;\Leftarrow\; i := 0;\; fac(0) := 1;\; loop;\; print\, fac(3)$$

This proof is again trivial, and omitted. Eager execution is presented in great detail in [2], and is not the point of this paper.

4 Need Variables

To calculate lazy execution time, we introduce a time variable and need variables. For each variable of a basic (unstructured) type, we introduce a binary (boolean) need variable. If x is an integer variable, then introduce binary need variable $\Box x$ (pronounced "need x"). The value of x may be 7, or any other integer; the value of $\Box x$ may be $true$ or $false$. As always, we use x and x' for the value of this integer variable at the start and end of a program (which could be a simple assignment, or any composition of programs). Likewise we use $\Box x$ and $\Box x'$ for the value of its need variable at the start and end of a program. At the start, $\Box x$ means that the initial value of variable x is needed, either in the computation or following the computation, and $\neg\Box x$ means that the initial value of variable x is not needed for the computation nor following the computation. At the end, $\Box x'$ means that the final value of variable x is needed for something following the computation, and $\neg\Box x'$ means that the final value of variable x is not needed.

With program variables x and y and time variable t, we earlier defined

$$ok \;=\; x'=x \wedge y'=y \wedge t'=t$$

We now augment this definition with need variables. From $x'=x$ we see that the initial value of x is needed if and only if the final value is needed. Likewise for y. So

$$ok \;=\; x'=x \wedge y'=y \wedge t'=t \wedge \Box x=\Box x' \wedge \Box y=\Box y'$$

[1]The arithmetic used here is defined in complete detail in [2, p.233-234].

Although $=$ is a symmetric operator, making $x' = x$ and $x = x'$ equivalent, as a matter of style we write $x' =$ (some expression in unprimed variables) because the final value of a program variable is determined by the initial values of the program variables. But we write $\Box x =$ (someexpression of primed need variables) because the need for an initial value is determined by the need for final values.

We now augment the assignment

$$x := 3 \quad = \quad x' = 3 \;\wedge\; y' = y \;\wedge\; t' = t + 1$$

with need variables. We have a choice. Perhaps the most reasonable option is

$$x := 3 \quad = \quad \textbf{if } \Box x' \textbf{ then } x' = 3 \;\wedge\; t' = t + 1 \textbf{ else } t' = t \textbf{ fi} \;\wedge\; y' = y \;\wedge\; \neg\Box x \;\wedge\; \Box y = \Box y'$$

This says that if the value of x is needed after this assignment, then that value is 3 and the assignment takes time 1 , but if the value of x is not needed afterward, then no final value of x is stated and the assignment takes time 0 because it is not executed. In either case, the value of y is unchanged. The initial value x does not appear, so it is not needed, hence $\neg\Box x$. The last conjunct says that the initial value of y is needed if and only if the final value of y is needed, because y appears in the right side of $y' = y$.

The other option is

$$x := 3 \quad = \quad x' = 3 \;\wedge\; y' = y \;\wedge\; t' = t + \textbf{if } \Box x' \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \;\wedge\; \neg\Box x \;\wedge\; \Box y = \Box y'$$

This option seems less reasonable because it says the final value of x is 3 even if that value is not needed and the assignment is not executed. But if the final value of x is not used, then it doesn't hurt to say it's 3 . This option has the calculational advantage that it untangles the results from the timing. So this is the option we choose. Every assignment has this same timing part, but using the need variable for the variable being assigned.

In the assignment

$$x := x + y \quad = \quad x' = x + y \wedge y' = y \wedge t' = t + \textbf{if } \Box x' \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \wedge \Box x = \Box x' \;\wedge\; \Box y = (\Box x' \;\vee\; \Box y')$$

we see that x appears once, to obtain x' , so $\Box x = \Box x'$. And y appears twice, to obtain x' and y' , so $\Box y = (\Box x' \;\vee\; \Box y')$. Time and need variables can be added automatically, but the algorithm to add them is not presented in this short paper.

For each structured variable in a program, there is a need variable structured exactly the same way. For example, if $x : [int, int]$ is a pair of integers, then $\Box x : [bin, bin]$ is a pair of binaries (booleans); the value of $\Box x$ is $[true, true]$ or $[true, false]$ or $[false, true]$ or $[false, false]$. And if y is an integer variable, then

$$\begin{aligned}
x(0) := 2 \quad &= \quad x'(0) = 2 \;\wedge\; x'(1) = x(1) \;\wedge\; y' = y \;\wedge\; t' = t + \textbf{if } \Box x'(0) \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \\
&\qquad \wedge\, \neg\Box x(0) \;\wedge\; \Box x(1) = \Box x'(1) \;\wedge\; \Box y = \Box y' \\
x(0) := x(1) \quad &= \quad x'(0) = x(1) \;\wedge\; x'(1) = x(1) \;\wedge\; y' = y \;\wedge\; t' = t + \textbf{if } \Box x'(0) \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \\
&\qquad \wedge\, \neg\Box x(0) \;\wedge\; \Box x(1) = (\Box x'(0) \;\vee\; \Box x'(1)) \;\wedge\; \Box y = \Box y' \\
x(0) := y \quad &= \quad x'(0) = y \;\wedge\; x'(1) = x(1) \;\wedge\; y' = y \;\wedge\; t' = t + \textbf{if } \Box x'(0) \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \\
&\qquad \wedge\, \neg\Box x(0) \;\wedge\; \Box x(1) = \Box x'(1) \;\wedge\; \Box y = (\Box x'(0) \;\vee\; \Box y')
\end{aligned}$$

If we define datatype $tree$ recursively as

$$tree \quad = \quad [\,] \mid [tree, int, tree]$$

then a tree is either the empty list, or it is a list of three components, the first component being the left subtree, the middle component being the root value, and the last component being the right subtree. This requires us to define datatype

$$\Box tree \quad = \quad bin \mid [\Box tree, bin, \Box tree]$$

for need variables. If we have variable x of type $tree$, we also have need variable $\Box x$ of type $\Box tree$. If $x = [\,]$, then $\Box x$ is either $true$ or $false$. If $x = [[\,], 3, [[\,], 5, [\,]]]$, then $\Box x$ may be $[true, true, [true, true, true]]$ or 31 other values.

Returning to integer variables x and y , here is an example conditional program.

$$\begin{aligned}
& \textbf{if } x=0 \textbf{ then } y:=0 \textbf{ else } x:=0 \textbf{ fi} \\
= \quad & x'=\textbf{if } x=0 \textbf{ then } x \textbf{ else } 0 \textbf{ fi} \ \wedge\ y'=\textbf{if } x=0 \textbf{ then } 0 \textbf{ else } y \textbf{ fi} \\
& \wedge\ t'=t+\textbf{if } x=0 \ \wedge\ \Box y' \textbf{ then } 1 \textbf{ else if } x\neq 0 \ \wedge\ \Box x' \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi fi} \\
& \wedge\ \Box x=(\Box x' \ \vee\ \Box y') \ \wedge\ \Box y=\Box y'
\end{aligned}$$

We see that x occurs in the right sides of both x' and y', so $\Box x=(\Box x' \vee \Box y')$. We see that y occurs in the right side of only y', so $\Box y=\Box y'$. We have added the need variables in accordance with the rules, as we would expect a compiler to do. But we can do better by using some algebra. Notice that $\textbf{if } x=0 \textbf{ then } x \textbf{ else } 0 \textbf{ fi} = 0$, so the results part can be stated equivalently as

$$x'=0 \ \wedge\ y'=\textbf{if } x=0 \textbf{ then } 0 \textbf{ else } y \textbf{ fi}$$

which results in the need part

$$\Box x=\Box y' \ \wedge\ \Box y=\Box y'$$

We find that $\Box x$ has been strengthened, making lazier execution possible. But a compiler would not be expected to make this improvement.

Sequential composition remains the same with need variables added.

$$\begin{aligned}
A;B \quad = \quad \exists x'',y'',t'',\Box x'',\Box y''\cdot\ & (\text{for } x',y',t',\Box x',\Box y' \text{ substitute } x'',y'',t'',\Box x'',\Box y'' \text{ in } A\,) \\
\wedge\ & (\text{for } x,y,t,\Box x,\Box y \text{ substitute } x'',y'',t'',\Box x'',\Box y'' \text{ in } B\,)
\end{aligned}$$

At the end of an entire program, we put $stop$, defined as

$$stop \quad = \quad x'=x \ \wedge\ y'=y \ \wedge\ t'=t \ \wedge\ \neg\Box x \ \wedge\ \neg\Box y$$

Like ok, its execution does nothing and takes no time. Since this is the end of the whole program, $\neg\Box x \ \wedge\ \neg\Box y$ says there is no further need for the values of any variables.

5 Example

We now have all the theory we need. Let us apply it to our example program

$$i:=0;\ fac(0):=1;\ \textbf{while } true \textbf{ do } i:=i+1;\ fac(i):=fac(i-1)\times i \textbf{ od};\ print\,fac(3);\ stop$$

To begin, we need a specification for the loop, which we call $loop$. With a number on each line for reference,

$$\begin{aligned}
loop \quad = \quad & (\forall j\le i\cdot fac'(j)=fac(j)) && 0 \\
& \wedge\ (\forall j>i\cdot fac'(j)=fac(i)\times j!/i!) && 1 \\
& \wedge\ t'=t+\textbf{if } \Box i' \textbf{ then } \infty && 2 \\
& \qquad\qquad \textbf{else if } \exists j\ge i\cdot\Box fac'(j) \textbf{ then } 2\times((max\ j\ge i\cdot\Box fac'(j)\cdot j)-i) \textbf{ else } 0 \textbf{ fi} \\
& \qquad\qquad \textbf{fi} \\
& \wedge\ \Box i=(\exists j>i\cdot\Box fac'(j)) && 3 \\
& \wedge\ (\forall j<i\cdot\Box fac(j)=\Box fac'(j)) && 4 \\
& \wedge\ \Box fac(i)=(\exists j\ge i\cdot\Box fac'(j)) && 5 \\
& \wedge\ (\forall j>i\cdot\neg\Box fac(j)) && 6
\end{aligned}$$

Lines 0 and 1 are the same as in the stronger version of the eager specification presented earlier. For eager execution lines 0 and 1 are not necessary because the loop execution is nonterminating, but for lazy execution they are necessary. Line 2 is the timing. It says that if the final value of i is needed, then the loop takes forever; otherwise, if the final value of $fac(j)$ is needed for any $j\ge i$, then the loop time is twice the difference between the largest such j and i, because there are two assignments in each iteration; otherwise the loop takes 0 time because it will not be executed[2]. Line 3 says that the loop needs an initial value for i if and only if a final value of $fac(j)$ is needed for any $j > i$. Line 4 says that for $j < i$, $fac(j)$ must have an initial value if and only if its final value is needed. Line 5 says that $fac(i)$ needs an initial value if and only if the final value of $fac(j)$ is needed for any $j \geq i$. And line 6 says that for $j > i$, the initial value of $fac(j)$ is not needed. In this paragraph, the words "initial" and "final" are used to mean relative to the entire loop execution: initially before the first iteration (if there is one), and finally after the last iteration (if there is one).

There can be more than one specification that's correct in the sense that it makes the proofs succeed. For example, if the type of variable i allows it, we could add the line $i' = \infty$, and then line 3 would be $\Box i = (\Box i' \lor \exists j > i \cdot \Box fac'(j))$, but since line 2 says that if we need i' then execution time is infinite, these additions really don't matter.

The first proof is the loop refinement. We must prove

$$loop \quad \Leftarrow \quad i := i+1;\ fac(i) := fac(i-1) \times i;\ loop$$

For the proof, we first replace each of the sequentially composed programs with their binary equivalent, including time and need variables. Then we use the sequential composition rule, and use one-point laws to eliminate the quantifiers. And we make any simplifications we can along the way. The proof is in the Appendix.

Then to prove that the overall execution time is 9, we must prove

$$t' = t+9 \quad \Leftarrow \quad i := 0;\ fac(0) := 1;\ loop;\ printfac(3);\ stop$$

For $printfac(3)$, we suppose it is like an assignment $print := fac(3)$, except that $print$ is not a program variable. This proof is also in the Appendix.

6 Execution versus Proof

In a lazy execution, the value of a variable may not be evaluated at various times during execution. Nonetheless, the value that would be evaluated if the execution were eager can still be used in the proof of lazy execution time. For example, in the loop specification line 3, we see the conjunct

$$\Box i = (\exists j > i \cdot \Box fac'(j))$$

If there is no $j > i$ for which $fac(j)$ is needed after the loop, then the value of i is not needed before the loop. The value of i is used in the proof to say whether the value of i is needed in execution.

If we change the print statement to $printfac(0)$, then the loop is not executed at all. The initialization $fac(0)$ is still required, but $i := 0$ is not. The theory tells us that the execution time is 2. The theory still requires that the assignment $i := 0$ produces $i' = 0$, but the execution does not.

7 Conclusion

We have presented a theory of lazy imperative timing. The examples presented are small enough so we know what the right answers are without using the theory; that enables us to see whether the theory is working. But the theory is not limited to small, easy examples.

Time and need variables are added according to a syntactic formula, and that can be automated. But in some cases, that formula does not achieve maximum laziness. To achieve maximum laziness may

[2] I confess that I did not get the lazy $loop$ specification right the first time I wrote it; my error was in the time line 2. The $loop$ specification is used in two proofs (below), and any error prevents one of the proofs from succeeding. That is how an error is discovered. Fortunately, the way the proof fails gives guidance on how to correct the specification.

require some further algebra. The proofs can also be automated, but the prover needs to be given domain knowledge.

References

[1] Walter Guttmann (2010): *Lazy UTP*. In: *Symposium on Unifying Theories of Programming*, Springer LNCS 5713, pp. 82–101, doi:10.1007/978-3-642-14521-6_6.

[2] Eric C.R. Hehner (1993): *A Practical Theory of Programming*. Springer, doi:10.1007/978-1-4419-8596-5. Available at `http://www.cs.utoronto.ca/~hehner/aPToP`.

[3] Peter Henderson & James H. Morris (1976): *A Lazy Evaluator*. In: *ACM Symposium on Principles of Programming Languages*, pp. 95–103, doi:10.1145/800168.811543.

[4] John Hughes (1989): *Why Functional Programming Matters*. *Computer Journal* 32(2), pp. 98–107, doi:10.1093/comjnl/32.2.98.

[5] Albert Y.C. Lai (2013): *Eager, Lazy, and Other Executions for Predicative Programming*. Ph.D. thesis, University of Toronto.

[6] David Sands (1990): *Complexity Analysis for a Lazy Higher-Order Language*. In: *European Symposium on Programming*, Springer LNCS 432, pp. 361–376, doi:10.1007/3-540-52592-0_74.

[7] David A. Turner (1979): *A New Implementation Technique for Applicative Languages*. *Software: Practice and Experience* 9(1), pp. 31–49.

A Appendix

Proof of the loop refinement

$$loop \quad \Leftarrow \quad i := i+1;\ fac(i) := fac(i-1)\times i;\ loop$$

starting with the right side:

$$i := i+1;\ fac(i) := fac(i-1)\times i;\ loop$$

Replace each statement by its definition.

$$
\begin{aligned}
= \quad & i' = i+1 \\
& \wedge\ (\forall j \cdot fac'(j) = fac(j)) \\
& \wedge\ t' = t+1 \\
& \wedge\ \Box i = \Box i' \\
& \wedge\ (\forall j \cdot \Box fac(j) = \Box fac'(j)); \\
\\
& i' = i \\
& \wedge\ fac'(i) = fac(i-1)\times i \\
& \wedge\ (\forall j \neq i \cdot fac'(j) = fac(j)) \\
& \wedge\ t' = t+1 \\
& \wedge\ \Box i = (\Box i' \ \vee\ \Box fac'(i)) \\
& \wedge\ (\forall j < i-1 \cdot \Box fac(j) = \Box fac'(j))\ \wedge\ \Box fac(i-1) = (\Box fac'(i)\ \vee\ \Box fac'(i-1)) \\
& \wedge\ \neg \Box fac(i) \\
& \wedge\ (\forall j > i \cdot \Box fac(j) = \Box fac'(j));
\end{aligned}
$$

$$\begin{aligned}
& (\forall j \le i \cdot fac'(j) = fac(j)) \\
\wedge\ & (\forall j > i \cdot fac'(j) = fac(i) \times j!/i!) \\
\wedge\ & t' = t + \textbf{if } \Box i' \textbf{ then } \infty \textbf{ else if } \exists j \ge i \cdot \Box fac'(j) \textbf{ then } 2 \times ((max\, j \ge i \cdot \Box fac'(j) \cdot j) - i) \textbf{ else } 0 \textbf{ fi fi} \\
\wedge\ & \Box i = (\exists j > i \cdot \Box fac'(j)) \\
\wedge\ & (\forall j < i \cdot \Box fac(j) = \Box fac'(j)) \\
\wedge\ & \Box fac(i) = (\exists j \ge i \cdot \Box fac'(j)) \\
\wedge\ & (\forall j > i \cdot \neg \Box fac(j))
\end{aligned}$$

Eliminate the first semi-colon.

$$\begin{aligned}
= \quad & i' = i + 1 \\
\wedge\ & fac'(i+1) = fac(i) \times (i+1) \\
\wedge\ & (\forall j \ne i+1 \cdot fac'(j) = fac(j)) \\
\wedge\ & t' = t + 2 \\
\wedge\ & \Box i = (\Box i' \ \vee\ \Box fac'(i+1)) \\
\wedge\ & (\forall j < i \cdot \Box fac(j) = \Box fac'(j)) \ \wedge\ \Box fac(i) = (\Box fac'(i+1) \ \vee\ \Box fac'(i)) \\
\wedge\ & \neg \Box fac(i+1) \\
\wedge\ & (\forall j > i+1 \cdot \Box fac(j) = \Box fac'(j));
\end{aligned}$$

$$\begin{aligned}
& (\forall j \le i \cdot fac'(j) = fac(j)) \\
\wedge\ & (\forall j > i \cdot fac'(j) = fac(i) \times j!/i!) \\
\wedge\ & t' = t + \textbf{if } \Box i' \textbf{ then } \infty \textbf{ else if } \exists j \ge i \cdot \Box fac'(j) \textbf{ then } 2 \times ((max\, j \ge i \cdot \Box fac'(j) \cdot j) - i) \textbf{ else } 0 \textbf{ fi fi} \\
\wedge\ & ni = (\exists j > i \cdot \Box fac'(j)) \\
\wedge\ & (\forall j < i \cdot \Box fac(j) = \Box fac'(j)) \\
\wedge\ & \Box fac(i) = (\exists j \ge i \cdot \Box fac'(j)) \\
\wedge\ & (\forall j > i \cdot \neg \Box fac(j))
\end{aligned}$$

Eliminate the last semi-colon. This step uses $(i+1)! = i! \times (i+1)$.

$$\begin{aligned}
= \quad & (\forall j \le i \cdot fac'(j) = fac(j)) \\
\wedge\ & (\forall j > i \cdot fac'(j) = fac(i) \times j!/i!) \\
\wedge\ & t' = t + \textbf{if } \Box i' \textbf{ then } \infty \textbf{ else if } \exists j \ge i \cdot \Box fac'(j) \textbf{ then} 2 \times ((max\, j \ge i \cdot \Box fac'(j) \cdot j) - i) \textbf{ else } 0 \textbf{ fi fi} \\
\wedge\ & \Box i = (\exists j > i \cdot \Box fac'(j)) \\
\wedge\ & (\forall j < i \cdot \Box fac(j) = \Box fac'(j)) \\
\wedge\ & \Box fac(i) = (\exists j \ge i \cdot \Box fac'(j)) \\
\wedge\ & (\forall j > i \cdot \neg \Box fac(j))
\end{aligned}$$

$$= \quad loop$$

Proof of

$$t' = t + 9 \quad \Leftarrow \quad i := 0;\ fac(0) := 1;\ loop;\ print fac(3);\ stop$$

starting with the right side:

$$i := 0;\ fac(0) := 1;\ loop;\ print fac(3);\ stop$$

Replace each statement by its definition.

$$\begin{aligned}
= \quad & i' = 0 \\
\wedge\ & (\forall j \cdot fac'(j) = fac(j)) \\
\wedge\ & t' = t + \textbf{if } \Box i' \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \\
\wedge\ & \neg \Box i \\
\wedge\ & (\forall j \cdot \Box fac(j) = \Box fac'(j));
\end{aligned}$$

$$\begin{aligned}
& i' = i \\
\wedge\ & fac'(0) = 1 \\
\wedge\ & (\forall j > 0 \cdot fac'(j) = fac(j)) \\
\wedge\ & t' = t + \textbf{if } \Box fac'(0) \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \\
\wedge\ & \Box i = \Box i' \\
\wedge\ & \neg\Box fac(0) \\
\wedge\ & (\forall j > 0 \cdot \Box fac(j) = \Box fac'(j));
\end{aligned}$$

$$\begin{aligned}
& (\forall j \le i \cdot fac'(j) = fac(j)) \\
\wedge\ & (\forall j > i \cdot fac'(j) = fac(i) \times j!/i!) \\
\wedge\ & t' = t + \textbf{if } \Box i' \textbf{ then } \infty \textbf{ else if } \exists j \ge i \cdot \Box fac'(j) \textbf{ then } 2 \times ((max\, j \ge i \cdot \Box fac'(j) \cdot j) - i) \textbf{ else } 0 \textbf{ fi fi} \\
\wedge\ & \Box i = (\exists j > i \cdot \Box fac'(j)) \\
\wedge\ & (\forall j < i \cdot \Box fac(j) = \Box fac'(j)) \\
\wedge\ & \Box fac(i) = (\exists j \ge i \cdot \Box fac'(j)) \\
\wedge\ & (\forall j > i \cdot \neg\Box fac(j));
\end{aligned}$$

$$\begin{aligned}
& print = fac(3) \\
\wedge\ & i' = i \\
\wedge\ & (\forall j \cdot fac'(j) = fac(j)) \\
\wedge\ & t' = t + 1 \\
\wedge\ & \Box i = \Box i' \\
\wedge\ & \Box fac(3) \\
\wedge\ & (\forall j \ne 3 \cdot \Box fac(j) = \Box fac'(j));
\end{aligned}$$

$$\begin{aligned}
& i' = i \\
\wedge\ & (\forall j \cdot fac'(j) = fac(j)) \\
\wedge\ & t' = t \\
\wedge\ & \neg\Box i \\
\wedge\ & (\forall j \cdot \neg\Box fac(j))
\end{aligned}$$

Eliminate the semi-colons and simplify.

$$\begin{aligned}
= \quad & print = 6 \\
\wedge\ & (\forall j \cdot fac'(j) = j!) \\
\wedge\ & t' = t + 9 \\
\wedge\ & \neg\Box i \\
\wedge\ & (\forall j \cdot \neg\Box fac(j))
\end{aligned}$$

Use specialization.

$$\Rightarrow \quad t' = t + 9$$