a Theory of Lazy Imperative Timing

Eric C.R. Hehner

Department of Computer Science, University of Toronto

hehner@cs.utoronto.ca

Abstract: We present a theory of lazy imperative timing.

Introduction

Lazy evaluation was introduced as a programming language execution strategy in 1976 by Peter Henderson and Jim Morris [2] 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 [0]. 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,

\[
\begin{align*}
& x := 2; \\
& y := 3; \\
& \text{print } y
\end{align*}
\]

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 \( \text{fac} \) be an infinite array of integers.

\[
\begin{align*}
& i := 0; \\
& \text{fac}(0) := 1; \\
& \text{while } \text{true} \text{ do } i := i + 1; \\
& \text{fac}(i) := \text{fac}(i - 1) \times i; \\
& \text{od} \\
& \text{print } \text{fac}(3)
\end{align*}
\]

After initializing \( i \) to 0 and \( \text{fac}(0) \) to 1, there is an infinite loop that assigns \( i! \) (\( i \) factorial) to each array element \( \text{fac}(i) \). Then, after the infinite loop, the value of \( \text{fac}(3) \) 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 \( \text{true} \) by \( i < 3 \). But [3] 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 \( \text{print } \text{fac}(4) \), 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 [1]. Some analysis of lazy execution time has also been done in the functional setting [5]. 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 [4], but simplifications have been made to shorten the explanations, and a different measure of time is being used.

a Practical Theory of Programming

In a Practical Theory of Programming [1], 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
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 \( x'=y+1 \land y'=y \) 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 \( y \),

\[
\begin{align*}
  x:=y+1 & \implies x'>y \\
\end{align*}
\]

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

\[
\begin{align*}
  x'>y & \iff x:=y+1 \\
\end{align*}
\]

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

**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 \( \infty \) 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 [1] and [4], time is loop iteration count. In this paper, time is assignment count.

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

\[
\begin{align*}
  ok & = x'=x \land y'=y \land t'=t \\
\end{align*}
\]

An example assignment is

\[
\begin{align*}
  x:=y+1 & \iff x'=y+1 \land y'=y \land t'=t+1 \\
\end{align*}
\]

The conditional specifications are defined as

\[
\begin{align*}
  &\text{if } a \text{ then } b \text{ else } c & = & a \land b \lor \neg a \land c \\
  &\text{if } a \text{ then } b \text{ else } & = & \text{if } a \text{ then } b \text{ else } ok \\
\end{align*}
\]

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

\[
\begin{align*}
  A;B & = \exists x'', y'', t''. \ ( \text{for } x', y', t' \text{ substitute } x'', y'', t'' \text{ in } A ) \\
  & \land ( \text{for } x, y, t \text{ substitute } x'', y'', t'' \text{ in } B ) \\
\end{align*}
\]

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

\[
\begin{align*}
  i:=0; \; \text{fac}(0):=1; \; \text{while } \text{true do } i:=i+1; \; \text{fac}(i):=\text{fac}(i-1)\times i; \; \text{od}; \; \text{print fac}(3) \\
\end{align*}
\]

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

\[
\begin{align*}
  loop & \iff i:=i+1; \; \text{fac}(i):=\text{fac}(i-1)\times i; \; loop \\
\end{align*}
\]

The specification we are interested in is

\[
\begin{align*}
  loop & = t'=t+n \\
\end{align*}
\]

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

\[
\begin{align*}
  loop & = t'=t+n \\
\end{align*}
\]
for any finite number expression $n$, the proof would fail because $n+1 \neq n$. A stronger
specification that succeeds is

$$\text{loop} \equiv (\forall i:\text{fac'}(i) = \text{fac}(j)) \land (\forall j:\text{fac'}(j) = \text{fac}(i) \lor i! \lor j!) \land t\!\!t\!t\!t\!t\!t\!t\!t\!t=\infty$$

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

$$t\!\!t\!\!t\!\!t\!t\!t\!t\!t=\infty \iff i=0; \text{fac}(0) := 1; \text{loop}; \text{print fac}(3)$$

This proof is again trivial, and omitted. Eager execution is presented in great detail in [1], and
is not the point of this paper.

**Need Variables**

To calculate lazy execution time, we introduce a time variable and need variables. 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 $y$, we earlier defined

$$\text{ok} \equiv x' = x \land y' = y$$

We now augment this definition with time and 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

$$\text{ok} \equiv x' = x \land y' = y \land t\!\!t\!\!t\!\!t\!t\!t\!t\!t\!t=1 \land \Box x = \Box x' \land \Box y = \Box y'$$

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 $\Box x$(some expression of primed need variables) because the need for an initial value is
determined by the need for final values.

We now augment the assignment

$$x:=3 \quad \equiv \quad x'=3 \land y'=y$$

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

$$x:=3 \quad \equiv \quad \text{if } \Box x' \text{ then } x'=3 \land t\!\!t\!\!t\!\!t\!t\!t\!t\!t=1 \text{ else } t\!\!t\!\!t\!\!t\!t\!t\!t\!t=1$$

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

$$x:=3 \quad \equiv \quad x'=3 \land y'=y \land t\!\!t\!\!t\!\!t\!t\!t\!t\!t=1 \text{ if } \Box x' \text{ then } 0 \text{ else } 0$$

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. And this option has the calculational advantage that it untangles the
results from the timing. So this is the choice we make. Every assignment has this same timing
part, but using the need variable for the variable being assigned.

In the assignment

$$x:=x+y \quad \equiv \quad x'=x+y \land y'=y \land t\!\!t\!\!t\!\!t\!t\!t\!t\!t=1 \text{ if } \Box x' \text{ then } 1 \text{ else } 0 \text{ if } \Box x=\Box x' \land \Box y=\Box y'$$

we see that $x$ appears once, to obtain $x'$, so $\Box x = \Box x'$. And $y$ appears twice, to obtain $x'$.
We now have all the theory we need. Let us apply it to our example program

\[ t := 0; \quad \text{fac}(0) := 1; \quad \text{while} \quad \text{true} \quad \text{do} \quad t := i + 1; \quad \text{fac}(i) := \text{fac}(i-1) \times i \quad \text{od}; \quad \text{print} \text{fac}(3); \quad \text{stop} \]

To begin, we need a specification for the loop, which we call \textit{loop}. With a number on each line for reference,
whether the theory is working. But the theory is not limited to small, easy examples.

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.

\[
\text{loop } = (\forall j \leq i \cdot \text{fac}(j) = \text{fac}(j)) \\
\land (\forall j > i \cdot \text{fac}(j) = \text{fac}(j) \times j / i!)
\]

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 \(\text{fac}(j)\) is needed for any \(j \geq i\), 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. Line 3 says that the loop needs an initial value for \(i\) if and only if a final value of \(\text{fac}(j)\) is needed for any \(j > i\). Line 4 says that for \(j < i\), \(\text{fac}(j)\) must have an initial value if and only if its final value is needed. Line 5 says that \(\text{fac}(i)\) needs an initial value if and only if the final value of \(\text{fac}(j)\) is needed for any \(j \geq i\). And line 6 says that for \(j > i\), the initial value of \(\text{fac}(j)\) 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, and finally after the last iteration.

I confess that I did not get the lazy \text{loop} specification right the first time I wrote it; my error was in the time line 2. The \text{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.

The first proof is the loop refinement. We must prove

\[
\text{loop } \iff i := i + 1; \text{fac}(i) := \text{fac}(i - 1) \times i; \text{loop}
\]

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

\[
\text{i}' = i + 9 \iff i := 0; \text{fac}(0) := 1; \text{loop}; \text{print fac}(3); \text{stop}
\]

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

**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 \(\Box i = (3 > i \cdot \Box \text{fac}(j))\). If there is no \(j > i\) for which \(\text{fac}(j)\) 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 \text{print fac}(0), then the loop is not executed at all. The initialization \text{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.

**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 easily automated. But in some cases, that formula does not achieve maximum laziness. To achieve maximum laziness may require some further algebra. The proofs can also be automated, but the prover needs to be given domain knowledge.

References


Appendix

Proof of the loop refinement

```plaintext
loop ⇔ i := i+1; fac(i) := fac(i-1)×i; loop
```

starting with the right side:

```plaintext
i := i+1; fac(i) := fac(i-1)×i; loop
```

replace each statement

```plaintext
i' = i+1
∧ (V j: fac'(j) = fac(j))
∧ t' = t+1
∧ □ i = □ i'
∧ (V j: □ fac(j) = □ fac'(j));
```

```plaintext
i' = i
∧ fac'(i) = fac(i-1)×i
∧ (V j≠i: fac'(j) = fac(j))
∧ t' = t+1
∧ □ i = □ i' v □ fac'(i)
∧ (V j<i-1: □ fac(j) = □ fac'(j))
∧ □ fac(i-1) = (□ fac’(i) v □ fac’(i-1))
∧ ¬ □ fac(i)
∧ (V j>i: □ fac(j) = □ fac'(j));
```

```plaintext
(V j≥i: fac'(j) = fac(j))
∧ (V j≥i: fac'(j) = fac(i)×j!/i!)
∧ t' = t+1 if □ i' then = else if (V j≥i: □ fac'(j) then 2×((max j≥i: □ fac'(j))–i) else 0 fi fi
∧ □ i = (V j≠i: □ fac'(j))
∧ (V j<i: □ fac(j) = □ fac'(j))
∧ □ fac(i) = (V j≥i: □ fac'(j))
∧ (V j>i: ¬ □ fac(j))
```

eliminate first semi-colon
2017-12-28 a Theory of Lazy Imperative Timing

\[\begin{align*}
& i' = i+1 \\
& \lambda \text{fac}'(i+1) = \text{fac}(i) \times (i+1) \\
& \lambda (\forall j \neq i+1 \cdot \text{fac}'(j) = \text{fac}(j))
\end{align*}\]

\[\begin{align*}
& t = t+2 \\
& \square i = (\square i' \lor \square \text{fac}'(i+1)) \\
& \lambda \text{fac}(i) = (\square \text{fac}'(i+1) \lor \square \text{fac}(i)) \\
& \lambda \square \text{fac}(i+1) \\
& \lambda (\forall j \neq i+1 \cdot \square \text{fac}(j) = \square \text{fac}'(j));
\end{align*}\]

\[\begin{align*}
& (\forall j \leq i \cdot \text{fac}'(j) = \text{fac}(j)) \\
& \lambda (\forall j > i \cdot \text{fac}'(j) = \text{fac}(i) \times i! / j!)
\end{align*}\]

\[\begin{align*}
& t = t+1 \text{if } \square i' \text{ then } \approx \text{ else if } \exists j \leq i \cdot \square \text{fac}'(j) \text{ then } 2 \times \text{max}(j \leq j \leq \text{fac}'(j) - i) \text{ else } 0 \text{ fi fi} \\
& \lambda n = (\exists j > i \cdot \square \text{fac}'(j)) \\
& \lambda (\forall j < i \cdot \square \text{fac}(j) = \square \text{fac}'(j)) \\
& \lambda \square \text{fac}(i) = (\exists j < i \cdot \square \text{fac}'(j)) \\
& \lambda (\forall j > i \cdot \neg \square \text{fac}(j))
\end{align*}\]

\[\text{eliminate last semi-colon. this step uses } (i+1)! = i! \times (i+1)\]

\[\begin{align*}
= & \text{loop}
\end{align*}\]

Proof of

\[i' = i+9 \iff i := 0; \text{fac}(0) := 1; \text{loop}; \text{print fac}(3); \text{stop}\]

starting with the right side:

\[i := 0; \text{fac}(0) := 1; \text{loop}; \text{print fac}(3); \text{stop}\]

replace each statement

\[\begin{align*}
& i' = 0 \\
& \lambda (\forall j \text{ fac}'(j) = \text{fac}(j)) \\
& \lambda t = t+1 \text{if } \square i' \text{ then } 1 \text{ else } 0 \text{ fi} \\
& \text{eliminate last semi-colon. this step uses } (i+1)! = i! \times (i+1) \\
& \lambda (\forall j > 0 \cdot \text{fac}'(j) = \text{fac}(j)) \\
& \lambda t = t+1 \text{if } \square \text{fac}'(0) \text{ then } 1 \text{ else } 0 \text{ fi} \\
& \lambda \square i = \square i' \\
& \lambda \neg \square \text{fac}(0) \\
& \lambda (\forall j > 0 \cdot \square \text{fac}(j) = \square \text{fac}'(j));
\end{align*}\]

\[\begin{align*}
& (\forall j \leq i \cdot \text{fac}'(j) = \text{fac}(j)) \\
& \lambda (\forall j > i \cdot \text{fac}'(j) = \text{fac}(i) \times i! / j!)
\end{align*}\]

\[\begin{align*}
& t = t+1 \text{if } \square i' \text{ then } \approx \text{ else if } \exists j \leq i \cdot \square \text{fac}'(j) \text{ then } 2 \times \text{max}(j \leq j \leq \text{fac}'(j) - i) \text{ else } 0 \text{ fi fi} \\
& \lambda \square i = (\exists j > i \cdot \square \text{fac}'(j)) \\
& \lambda (\forall j < i \cdot \square \text{fac}(j) = \square \text{fac}'(j)) \\
& \lambda \square \text{fac}(i) = (\exists j < i \cdot \square \text{fac}'(j)) \\
& \lambda (\forall j > i \cdot \neg \square \text{fac}(j));
\end{align*}\]
\begin{align*}
\text{print} &= \text{fac}(3) \\
\land \ i' &= i \\
\land \ (\forall j \cdot \text{fac}'(j) = \text{fac}(j)) \\
\land \ t' &= t+1 \\
\land \ \square i &= \square i' \\
\land \ \square \text{fac}(3) \\
\land \ (\forall j \neq 3 \cdot \square \text{fac}(j) &= \square \text{fac}'(j)); \\
&= \\
\land \ (\forall j \cdot \text{fac}'(j) = \lfloor \frac{j}{3} \rfloor) \\
\land \ t' &= t+9 \\
\land \ \neg \square i \\
\land \ (\forall j \cdot \neg \square \text{fac}(j)) \\
\Rightarrow \ i' &= t+9
\end{align*}