Thesis Proposal:
A Theory of Lazy Time and Space

Albert Y. C. Lai

February 27, 2003

Abstract

We propose a thesis on our theory of lazy time and space. It will be along the line of:

1. (Introduction) The convenience of lazy evaluation in programming and the problem of predicting time and space costs of lazy programs. Previous work on this front.
2. (Background) Hehner's theory of programming, on which our theory of lazy time and space will be built as an add-on, at least at the onset.
3. Our add-on theory of lazy time and space: intuition, main idea, examples, comments. (Questions raised in the comments are further elaborated and answered in subsequent chapters.)
4. Some kind of soundness proof of our theory.
5. Porting our theory to other theories of programming than Hehner's.
6. A detailed account of the connection between our theory and previous work on lazy time and space.
7. As much as is appropriate and relevant, perhaps some newly discovered connection among temporal theories, data refinement (simulation), and abstract interpretation. This stems from some observations made in the examples in Chapter 3.
8. Conclusion and new questions.

This proposal motivates the problem, explains our theory of lazy time (the space counterpart is similar), mentions and compares with previous related work, and concludes with an outline of work to do.

1 Introduction

Some programming languages support lazy evaluation, meaning an expression may be left unevaluated until computational decisions and progress rely upon it, and then it will be evaluated *and* the value memoized. Lazy evaulation is important because, on the theoretical side, it approximates the call-by-need or call-by-name aspects of certain lambda calculi, and on the practical side, it offers certain conveniences and simplifications in the design of programs and data structures. But the task of determining the amount of time and space used by a lazy program, even just asymptotically, has been largely a black art.

Here are some of the recent works on the problem (the final thesis will list all or almost all), but there are only a handful, sprinkled here and there over the last decade. Wadler [7] has a theory of lazy time based on domain-theoretic strictness analysis, addressing the call-by-need aspect but not the memoization aspect; its root in domain theory causes it to be somewhat complicated. Bakewell and Runciman [1] has an operational semantics for calculating space usage of Haskell programs; the operational nature implies every prediction has to be a whole-program analysis, making compositional reasoning difficult.

Our theory of lazy time is simpler and more fundamental, requiring only a twist in the mental model of binary relations. It addresses both call-by-need and memoization. It is an add-on to some known theories of programming and correctness, requiring no new logic, analysis method, or proof technique. It supports compositional reasoning as much as the host theory of programming does. Our theory of lazy space is similar to our theory of lazy time, borrowing from the way Hehner's eager space theory resembles his eager time theory [5].

We have conceived the time theory and tried it on a few generalizable examples; the report of these constitutes the bulk of this proposal. To round up a complete account of this theory, including the space part, there is more work we will do: proving its soundness, adapting it to various theories of programming, and connecting our theory with related work to give a sense of unification. In working one of the examples, and in fact during the whole duration of investigation, we seemed to see some connection among temporal theories, data refinement (simulation), and abstract interpretation; we would like to develop this connection and report the results *if relevant and appropriate*. Thus we propose a thesis consisting of the foregoing. Please refer to

the abstract for the proposed structure.

The rest of the proposal is organized as follows. Section 2 summarizes Hehner's theory of programming and time [3, 4], for the purpose of hosting our add-on theory of lazy time. Section 3 describes, exemplifies, and comments on the relevance of our theory of lazy time, noting further work along the way. (It also makes passing mention of space.) Section 4 briefly connects our theory with Wadler's theory and well as backward dataflow analysis. Section 5 concludes and names further work for the thesis.

2 Background

2.1 A Theory of Eager Timing

To unify reasoning of running time with reasoning of program behaviour, Hehner suggests introducing a state variable t into the program in question for time accounting [2]. This variable can be thought of as a ghost variable that exists on paper but not in the computer; or it can be thought of as a counter present in the computer. Either way, its presence permits us to both add values to it at time-consuming points of the program, and postulate and prove the gross change to its value caused by the whole program. For example, to prove that summing n numbers does not use more than n additions, we insert $t{:=}t+1$ near each summing operation in the algorithm, then prove that the whole algorithm increases t by at most n.

Although this theory was invented in the context of Hehner's theory of program correctness [3], it easily ports to other theories of program correctness. This wide applicability derives from the status of t as just a state variable not unlike other state variables such as the accumulator holding the sum. Whichever theory of correctness is used to prove things done to the accumulator, the same theory can be used to prove things done to t. More importantly, this method of time analysis inherits all the properties of the theory used—strengths, weaknesses, and tradeoffs. In particular, if the theory supports compositional reasoning, then time analysis also enjoys compositional reasoning: static time analysis does not have to degenerate to global analysis. As well, if the theory trades precision for automation, then time analysis can also trade precision for automation.

The previous paragraph muse on the characteristics of the Hehner method of time analysis because we will suggest methods of lazy time analysis that

have some of these characteristics.

2.2 A Theory of Programming

Although the theory of eager timing in the previous subsection is not restricted to any particular theory of programming, our theory of lazy timing will start out being restricted to one. This section reviews Hehner's theory of imperative programming and associated notations, which will be used in the next section to describe our theory of lazy timing. Details can be found in [4, 3].

A specification is a boolean expression relating pre-values of state variables with their post-values.[1] Pre-values are denoted by the names of the variables themselves like x, y, and t, and post-values by adding primes to names like x', y', and t'. Sometimes we also use σ to stand for x, y, t together and σ' for x', y', t' together. As such, the theory is mostly a relational theory, the only difference being notational: we write $\sigma' = \sigma$ when a relation theorist would write $\lambda\sigma, \sigma' \cdot \sigma' = \sigma$ or $\{(\sigma, \sigma') | \sigma' = \sigma\}$. The relational nature is pivotal to our lazy timing theory in the next section.

Some specifications correspond directly to program constructs and are given special synonyms:

$$\begin{aligned} ok &= \sigma' = \sigma \\ x{:=}E &= x' = E \wedge y' = y \wedge t' = t \\ P \,.\, Q &= \exists \sigma'' \cdot (\text{substitute } \sigma'' \text{ for } \sigma' \text{ in } P) \wedge (\text{substitute } \sigma'' \text{ for } \sigma \text{ in } Q) \end{aligned}$$

The first is "do nothing" or the identity relation, the second is an assignment statement or total function, and the third is sequential composition or relational composition.

Refinement is defined as universal implication or relational inclusion: problem P is refined by solution S iff $\forall \sigma, \sigma' \cdot P \Leftarrow S$. For convenience and without loss of soundness, often we just state and prove $P \Leftarrow S$ instead. The solution may use the problem as a component, which translates to recursion or looping in execution.

There will be lists and list operations in the examples in the rest of this paper. Lists may stand for linked lists, cons lists, snoc lists, or arrays. Indexes are zero-based. For a list L, $\#L$ is its length, $L0$ is its head, $L[2;..5]$ is the slice

[1] There is a more general notion of specification, but we will not need it here.

$[L2; L3; L4]$ (in particular $L[1;..\#L]$ is its tail), and $^+$ is the concatenation operator.

3 Our Theory of Lazy Timing

We will describe our theory of lazy timing in imperative programming. Lazy imperative programming is not widely known or available, so it is necessary to define it first. It is quite similar to lazy functional programming. Lazy functional programming means that expressions assigned to formal parameters are evaluated by need and memoized. Lazy imperative programming analogously means that expressions assigned to state variables are evaluated by need and memoized. (The corresponding lazy timing theory for functional programming is not worked out yet, but it shall have the same flavour. Alternatively it may be easier to just say "translate to imperative programming, then use the present theory", as is exemplified in a subsequent subsection.)

Before we go into our theory of lazy timing, we explain how it is inspired by a twist of the meaning of specifications and programs as relations. A program can be modelled as a relation, in that the domain is the input space, the co-domain is the output space, and the relation specifies which input values should lead to which output values. In other words, the direction of information flow is thought to be forward, from the domain to the co-domain. *But this needs not be so.* Direction here is a matter of interpretation, not intrinsic in the mathematics of relations. Indeed, a declarative program for finding an even prime number can be constructed by taking a relation that nondeterministically chooses a prime number, and composing it with a relation that insists on even inputs. While the former stage tells the latter stage of a prime number, it is the latter stage that tells the former stage to choose an even one. Some information may flow forward, and some other information may flow backward; in fact the backward information may constrain the forward information. We will exploit this interpretation for our theory of lazy timing.

3.1 Basic Theory of Lazy Timing

To account for running time under lazy evaluation, we introduce the time variable t as in Hehner [2], and in addition, for each state variable x, we introduce a corresponding boolean usage variable u_x. An interpretation (but not the only one) of u_x is this. Whereas we think of x as an input and x' as

an output, the roles of u_x and u'_x are reversed: u'_x is an input, using which the program is told whether or not its x' will be needed in the future; and u_x is output, using which the program tells whether or not it needs x from the past. Combining this with sequential composition of programs, which is relational composition in our case, where x' and u'_x of one program is identified with x and u_x of the next, we have a way of propagating usage information *backward* while data and time goes *forward*. That is to say, live variable analysis is a backward dataflow analysis.

Thus, for a program that lazily sets variable x to the result of some computation, we have the means to say "if x' will be used in the future, then this program costs 1 unit; otherwise it is free"[2]:

$$t' = t + \textbf{if } u'_x \textbf{ then } 1 \textbf{ else } 0$$

If the computation is accomplished by reading from another variable y, we also have the means to say "if x' will be used in the future, then this program uses y from the past; otherwise usage of y is inherited directly from the future":

$$u_y = u'_x \vee u'_y$$

Finally, we may add that this program does not need the past value of x:

$$\neg u_x$$

The above forms of expressions are suitable for specifications, but not quite convenient in program code when we want to state "set the pre-value of u_x to false, all other variables are unchanged". We need a notation for backward assignment statements. Thus we introduce the backward assignment operator $=:$ with:

$$u_x =: E' \;=\; u_x = E' \wedge u_y = u'_y \wedge x' = x \wedge y' = y \wedge t' = t$$

Here E' is an expression that mentions no pre-values. This syntactic restriction avoids contradictions like $u_x = \neg u_x$ without losing generality: if we wanted E' to be $u_y \wedge x > 0$, we could write $u'_y \wedge x' > 0$ instead, since we would have $u_y = u'_y$ and $x' = x$ anyway.

The next two subsections give examples of applying this theory. Of course these examples cannot possibly constitute proof of soundness or consistency. Instead they serve as materialization of the foregoing high-level depiction, illustration of the new intuition (afterall it is an unusual mental model), demonstration of the theory in action, and motivation for new questions.

[2]Okay it should also cost 1 unit for building the thunk.

3.2 Basic Example

Here is an example of storing the sum of an array L into s. If subsequently the sum is not needed, the program should take no time. Also, the program does not need the past value of s. The array L is used as a constant (read-only), and we skip stating "L is not changed". We assume an auxiliary global variable n, and we can use it any way we want. For simplicity of presentation, we omit the usage variables u_L and u_n (but it is possible to put them back).

$$
\begin{array}{ll}
 & s' = \Sigma L[0;..\#L] \wedge \neg u_s \wedge t' = t + \textbf{if } u'_s \textbf{ then } \#L \textbf{ else } 0 \\
\Leftarrow & u_s =: \bot \ . \\
 & s := 0 \ . \\
 & n := 0 \ . \\
 & s' = s + \Sigma L[n;..\#L] \wedge u_s = u'_s \wedge t' = t + \textbf{if } u'_s \textbf{ then } \#L - n \textbf{ else } 0
\end{array}
$$

$$
\begin{array}{ll}
 & s' = s + \Sigma L[n;..\#L] \wedge u_s = u'_s \wedge t' = t + \textbf{if } u'_s \textbf{ then } \#L - n \textbf{ else } 0 \\
\Leftarrow & \textbf{if } \#L = 0 \textbf{ then } ok \\
 & \textbf{else } t := t + \textbf{if } u_s \textbf{ then } 1 \textbf{ else } 0 \ . \\
 & \quad s := s + Ln \ . \\
 & \quad n := n + 1 \ . \\
 & \quad s' = s + \Sigma L[n;..\#L] \wedge u_s = u'_s \wedge t' = t + \textbf{if } u'_s \textbf{ then } \#L - n \textbf{ else } 0
\end{array}
$$

Initially, s is cut off from its past ($\neg u_s$ in the specification and $u_s =: \bot$ in the code). In the loop, s takes on many incarnations, each depending on its previous, so an incarnation is needed iff its previous is ($u_s = u'_s$ in the loop specification and no change to u_s in the loop body). So at the end, if the final incarnation of s (the result) is needed, then all incarnations are, which take time $\#L$ altogether; otherwise, all incarnations are skipped. (The same story applies to u_n if we put it back. As for u_L, we have roughly $u_L = u'_s \vee u'_L$ due to the statement $s := s + Ln$; this is a very coarse-grain account of the usage of L, and a fine-grain account is in the subsection after next.)

Here is a main program making use of the above, showing that the lazy timing is right. It invokes the above routine to sum L and store in s, then invokes a similar routine to sum M and store in s (thus erasing the sum of L), then uses the latter result. The running time should be $\#M$, not $\#L + \#M$,

and the following is therefore provable:

$$
\begin{array}{ll}
& s' = \Sigma L[0;..\#L] \wedge \neg u_s \wedge t' = t + \textbf{if } u'_s \textbf{ then } \#L \textbf{ else } 0 \ . \\
& s' = \Sigma L[0;..\#M] \wedge \neg u_s \wedge t' = t + \textbf{if } u'_s \textbf{ then } \#M \textbf{ else } 0 \ . \\
& u_s =: \top \\
\Rightarrow & t' = t + \#M
\end{array}
$$

We should remark that although this main program uses the sum of M just once, further uses do not incur extra costs. This is consistent with memoization.

To demonstrate compositional reasoning, let us change the implementation of the summing routine. We use a non-tail recursion. We also use n in a different way.

$$
\begin{array}{ll}
& s' = \Sigma L[0;..\#L] \wedge \neg u_s \wedge t' = t + \textbf{if } u'_s \textbf{ then } \#L \textbf{ else } 0 \\
\Leftarrow & n := 0 \ . \\
& s' = \Sigma L[n;..\#L] \wedge n' = n \wedge \neg u_s \wedge t' = t + \textbf{if } u'_s \textbf{ then } \#L - n \textbf{ else } 0
\end{array}
$$

$$
\begin{array}{ll}
& s' = \Sigma L[n;..\#L] \wedge n' = n \wedge \neg u_s \wedge t' = t + \textbf{if } u'_s \textbf{ then } \#L - n \textbf{ else } 0 \\
\Leftarrow & \textbf{if } \#L = n \textbf{ then } u =: \bot \ . \ s := 0 \\
& \textbf{else } n := n + 1 \ . \\
& \qquad s' = \Sigma L[n;..\#L] \wedge n' = n \wedge \neg u_s \wedge t' = t + \textbf{if } u'_s \textbf{ then } \#L - n \textbf{ else } 0 \ . \\
& \qquad n := n - 1 \ . \\
& \qquad t := t + \textbf{if } u_s \textbf{ then } 1 \textbf{ else } 0 \ . \\
& \qquad s := Ln + s
\end{array}
$$

We dive into the recursion and cut off the past of s when we hit the bottom. As we return, we build up incarnations of s. Each incarnation is needed iff its next is, so the recursion specification does not constrain u'_s (it lets the future decide), but the code after each recursive call carefully ensures $u_s = u'_s$ so that the recursive call can see the future.

After proving these new refinements, this new implementation becomes a drop-in replacement for the old one. There is no need to re-visit the reasoning of the main program, as details not mentioned in the specification (such as the fate of n) were never used.

As noted, the above method is too coarse-grained for aggregate data structures such as arrays and lists. A fine-grained variation is shown in the next subsection.

3.3 Advanced Theory and Example: Lazy Infinite List

A major application of lazy evaluation is the construction of infinite data structures to be consumed finitely. For example, an infinite cons list is created by a fixpoint equation, and then only the first six items are used ever. The computer should spend no more time than is necessary for the construction of the needed prefix.

An infinite cons list of 0's may be created by a program like

$$(\forall i \cdot L'i = 0) \Leftarrow (\forall i \cdot L'i = 0)\ .\ L{:=}\,[0]^+L$$

The unusual position of the recursion is derived from a Haskell program for the same task:

```
p :: () -> [Int]
p = ([0]++) . p
```

i.e., functional composition $f \circ g$ typically becomes sequential composition $g\ .\ f$.

The lazy timing of this program cannot possibly be controlled by one single boolean usage u_L—it is too imprecise. The most precise method is to use an infinite list U of usage variables: for each index i, $U'i$ is true iff $L'i$ will be used. The time taken up by the program should be the supremum of $i+1$ over those i's with $U'i$ true, or 0 if there is no such i; this is the amount of work necessary for building the prefix $L[0;..i+1]$ to satisfy the demand. (If some items in this prefix are not needed, it is still alright and necessary to include them, as this is a cons list. Alright, the cons cells, not the items themselves, are constructed.)

The program augmented with timing information is then:

$$\begin{array}{ll} & (\forall i \cdot L'i = 0) \wedge t' = t + \textbf{if } \neg\exists i \cdot U'i \textbf{ then } 0 \textbf{ else } \mathrm{SUP}\, i : U'i \cdot i+1 \\ \Leftarrow & (\forall i \cdot L'i = 0) \wedge t' = t + \textbf{if } \neg\exists i \cdot U'i \textbf{ then } 0 \textbf{ else } \mathrm{SUP}\, i : U'i \cdot i+1\ . \\ & L{:=}\,[0]^+L\ . \\ & U{=:}\,U'[1;..\infty]\ . \\ & t{:=}\,t + \textbf{if } \neg\exists i \cdot U'i \textbf{ then } 0 \textbf{ else } 1 \end{array}$$

The loop body passes backward just the tail of U, since the past furnishes just the tail of L. The timing in the loop body says: if any item of L is used, be it the one I contribute or one I inherit from the past, then I cost

1 unit, since my item will have to be constructed either way; otherwise, I cost nothing. The above refinement can be proved by going through all four cases: $L'0$ is or is not used, and some item of the tail $L'[1;..\infty]$ is or is not used. (Two of the cases can be merged.)

The method is precise but can be tedious; for example the proof for the above expands to at least three cases, and lots of quantifiers abound. We can lose some precision and still prove the same conclusion. We do not need to know the usage of each item; all we need to know here is the maximum index. So we now switch back to using one single usage u_L, but instead of a boolean, it is now a non-negative integer standing for the length of the prefix constructed, replacing

$$\textbf{if } \neg\exists i \cdot Ui \textbf{ then } 0 \textbf{ else } \mathrm{SUP}\, i : Ui \cdot i + 1$$

Basically, the partial order of single booleans is not precise enough, the partial order of infinite lists of booleans is exact but too large, and the partial order of single numbers is just right, which can be seen as an abstraction of the exact partial order. That is to say, the effectiveness of an analysis relies on an effective abstract interpretation. The program becomes:

$$\begin{array}{ll} & (\forall i \cdot L'i = 0) \wedge t' = t + u'_L \\ \Leftarrow & (\forall i \cdot L'i = 0) \wedge t' = t + u'_L \ . \\ & L{:=}\,[0]^+ L \ . \\ & u_L{=:}\,\textbf{if } u'_L = 0 \textbf{ then } 0 \textbf{ else } u'_L - 1 \ . \\ & t{:=}\,t + \textbf{if } u_L = 0 \textbf{ then } 0 \textbf{ else } 1 \end{array}$$

If a certain prefix of L needs to be constructed, the loop body costs one unit for the one item it contributes, and its past is just responsible for the remaining cost. Otherwise, all is free. The proof of this refinement involves just two cases, and each case is trivial arithmetic.

The agreement between the simpler program using the numeric u_L and the exact program using the list U is not by kludge and luck; it is a necessary consequence. One way to see it is, as hinted above, an abstract interpretation from the partial order over U to the partial order over u_L. Another way to see it is a data refinement (simulation) between U and u_L using the representation invariant

$$u_L = \textbf{if } \neg\exists i \cdot Ui \textbf{ then } 0 \textbf{ else } \mathrm{SUP}\, i : Ui \cdot i + 1$$

which is again hinted above. (Okay, so we confess there is some hacking in discovering a candidate abstraction and invariant; but once the abstraction is validated and the data refinement worked out, the agreement between the two programs must follow.) These realizations suggest interesting questions connecting temporal theories, data refinements (simulations), and abstract interpretations. We do not have time presently to write down our modest progress on this connection, but we do intend to work it out and include selected results in the final thesis.

Lists are not the only interesting lazy data structure. We would like to generalize as much as possible to (co)algebraic data structures in the final thesis.

3.4 Relevance and Applicability

We now address some itching issues on the relevance and applicability of our theory.

3.4.1 Applicability to Other Theories of Correctness

Hehner's theory of eager timing is applicable to all imperative theories of correctness. Our theory of lazy timing does not seem to be as widely applicable. Our theory exploits relational theories, so all we can say is our theory is applicable to relational theories of correctness.

There is a further constraint. Most relational theories of correctness (essentially except Hehner's [3] and a version of Hoare and He's [6]) insist on total correctness and termination under eager evaluation. In doing so, their notions of composition and legal relations forbid backward information flow and lazy evaluation in an essential way. Thus our theory of lazy timing is not that widely applicable afterall.

But this is a shortcoming of theories insisting on total correctness, not a shortcoming of any add-on theory of lazy timing. Correctness can and should be factored into partial correctness and timeliness. Partial correctness concerns *what* might be delivered, and it is a declarative and safety notion, insensitive to execution strategies. Timeliness concerns *when* and *whether* there is a [usable] delivery, and it is an operational and liveness notion, sensitive to execution strategies. If a theory couples tightly the two notions together, it necessarily over-commits itself to one particular execution strategy in a rather unique manner, and nothing could possibly be done

to adapt it to a different execution strategy, short of a complete rewrite. But if a theory carefully keeps the two factored, such as Hehner's and Hoare and He's, then all we need to do is to pull out the chapter on eager timing and drop in a chapter on lazy timing.

From this we can see an unnoticed applicability of our theory to predicate transformer semantics. Instead of starting with weakest preconditions, we may be able to start with weakest liberal preconditions and add a lazy notion of termination and time. This is completely a conjecture now, but it is possible to work it out and include it in the final thesis.

3.4.2 Theory vs. Reality

We observe two discrepancies between this lazy time theory with real lazy implementations, and we briefly explain why such discrepancies are tolerable for the sake of keeping the theory fairly simple.

The first discrepancy is that the result of the computation is always delivered in the theory, whereas it is seldom delivered in reality if it is never used. This is analogous to the Hehner theory of eager timing [2], in which when it comes to a non-terminating program, the theory says the output is delivered at time infinity, while no implementation delivers at all. The defense for the eager theory is that the hypothetical delivery cannot be observed even in principle, so it is as good as no delivery. We can employ a similar defense for the lazy theory: if the result is not used, whether it is delivered or not makes no difference to the rest of the program or the observer, so delivering it is as good as not delivering it.

The second discrepancy is that, in some implementations, time is not spent in running the part of a program that delivers a computational result, but rather at the end of the program where all needed results are used, or forced; and in some other implementations, although time is spent in parts, the direction of execution and time consumption is backward. Our theory does not attempt to temporally simulate execution processes of some or all implementations. It only tries to be an accounting theory, a way of distributing or amortizing costs to parts; its sole validation or invalidation lies in its predictions of end-to-end running times.

That last point calls for some kind of soundness proof of our theory, e.g., that it should be consistent with some conceivable implementation. This is not done yet, but can be done in the final thesis.

3.5 Space Theory

In the foregoing, we have only covered time. This subsection covers space, and it is brief because the major hurdle has been overcome by the usage variables.

Hehner's theory of space [5] adds yet another variable s to count the amount of space in use. It is incremented at program points of memory allocation, and decremented at program points of memory deallocation. The value of $s' - s$ over the whole program gives the amount of space leak. To further determine the maximum amount of space ever occupied, simply introduce another variable m, and do

$$m := \max\ m\ s$$

at every point of increment of s.

Our lazy theory of space simply needs to adopt the same additional variables s and m. There is nothing original in this regard, and nothing original is needed in this regard. The original idea is adding the usage variables and using them backward; once we have that, the rest is trivial. We increment and decrement s according to future usage, just like we incremented the time variable t according to future usage.

4 Related Work

As hinted in the previous section when describing our theory of lazy timing, there is some resemblance of our theory to live variable analysis. Indeed, the live variable problem is precisely about which assignment statements can be skipped, and its solution is precisely a backward propagation of dataflow information. But live variable analysis approximates blindly whenever there are branches and loops, and it does not embed dataflow variables into the program. As a result, it also has to keep track of usage information per assignment statement, not just per state variable. Our theory can be as imprecise or precise as the user desires and the hosting theory of programming allows, and by embedding usage information right next to assignment statements, we only need one usage variable per state variable, which is simpler.

Wadler's theory [7] treats call-by-need timing by using domain theory and defining "contexts/projections" and "projection transformers", but there is a simpler way of explaining it in light of our theory. The problem is to

time $f\,(g\,x)$. The solution is to define "projection", a way to specify how much of a parameter is needed (so this is like the possible values stored into our usage variables); and a "projection transformer" h^1 for each caller h, a function mapping projections to projections for doing this: if the result of $h\,y$ is to be used in a way specified by projection α, then $h^1\,\alpha$ gives the projection that specifies how y will be used by h, which means to specify how the use of outputs affects the use of inputs (so this is like our $u{=:}\,E'$ backward assignments). The amount of time $h\,y$ takes under projection α depends on both y and α (so this is like our $t' = t + \textbf{if } u' \textbf{ then } \#L \textbf{ else } 0$ specifications), and part of it involves finding out how much time the callee y takes under projection $h^1\,\alpha$ (so this is like our passing information backwards through composition). Thus to time $f\,(g\,x)$ under the projection (usage) α, the backward expression $g^1\,(f^1\,\alpha)$ is involved, just like in our theory. To constrast Wadler's theory with ours, we note that our theory (or our exposition) presents the backward flow more intuitively and directly, is simpler (projects are functions themselves), introduces fewer definitions, handles memoization, and covers nondeterministic specifications and programs. We also begin to give an affirmative answer to a question in Wadler's paper: can his theory be adapted to space analysis?

Our theory can be likened to old methods and theories, and in this section we have exactly done so. But in all cases, our theory is simpler and more general, or even better, our theory *explains* old theories.

5 Conclusion

We have described and illustrated a simple theory of lazy timing. To understand it takes no more than an unusual understanding of relations, and to use it takes just the ordinary mathematics of relations or whichever theory of programming is used to host our theory.

We still have to extend this theory to more infinite data structures. We still have to bring it to more theories of imperative programming. We still have to determine the best way of bringing it to functional programming. We still have to prove it sound. We propose to do all these in the final thesis.

References

[1] A. Bakewell and C. Runciman. A space semantics for core Haskell. In *Proceedings of the Haskell Workshop*, September 2000.

[2] Eric C. R. Hehner. Termination is timing. In J. L. A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 36–47, Groningen, The Netherlands, June 1989. Springer.

[3] Eric C. R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14(2,3):133–158, October 1990.

[4] Eric C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer, 1993.

[5] Eric C. R. Hehner. Formalization of time and space. *Formal Aspects of Computing*, 10:290–306, 1998.

[6] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1998.

[7] Philip Wadler. Strictness analysis aids time analysis. In *15'th ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.