Strictness Analysis Aids Time Analysis

Philip Wadler
Department of Computing Science, University of Glasgow*

October 1987

Abstract

Analysing time complexity of functional programs in a lazy language is problematic, because the time required to evaluate a function depends on how much of the result is "needed" in the computation. Recent results in strictness analysis provide a formalisation of this notion of "need", and thus can be adapted to analyse time complexity.

The future of programming may be in this paradigm: to create software, first write a specification that is clear, and then refine it to an implementation that is efficient. In particular, this paradigm is a prime motivation behind the study of functional programming. Much has been written about the process of transforming one functional program into another. However, a key part of the process has been largely ignored, for very little has been written about assessing the efficiency of the resulting programs.

Traditionally, the major indicators of efficiency are time and space complexity. This paper focuses on the former.

Functional programming can be split into two camps, strict and lazy. In a strict functional language, analysis of time complexity is straightforward, because of the following *compositional rule*:

- The time to evaluate $(f\ (g\ x))$ equals the time to evaluate $(g\ x)$ plus the time to evaluate $(f\ y)$, where y is the value of $(g\ x)$.

However, in a lazy language, this rule only gives an upper bound, possibly a crude one. For example, if f is the *head* function, then $(g\ x)$ need only be evaluated far enough to

*This work was performed at Oxford University, under a research fellowship funded by ICL.

This paper appeared in: *15'th ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

determine the first element of the list, and this may take much less time than evaluating $(g\ x)$ completely.

The key to a better analysis is to describe formally just how much of the result "needs" to be evaluated; we call such a description a *context*. Recent results in strictness analysis show how such contexts can be modelled using the domain-theoretic notion of a *projection* [WH87]. This paper describes how these results can be applied to the analysis of time complexity. The method used was inspired by work of Bror Bjerner on the complexity analysis of programs in Martin-Löf's type theory [Bje87]. The main contribution of this paper is to simplify Bjerner's notation, and to show how contexts can replace his "demand notes".

The language used in this paper is a first-order language. This restriction is made because context analysis for higher-order languages is still under development. An approach to higher-order context analysis is outlined in [Hug87b]. Context analysis is based on backwards analysis, rather than the earlier approach of abstract interpretation; both are discussed in [AH87].

Some work on complexity analysis [Weg75, LeM85] has concentrated on automated analysis: algorithms that derive a closed form for the time complexity of a program. The goal here is less ambitious. We are simply concerned with describing a method of converting a functional program into a series of equations that describe its time complexity. This modest beginning is a necessary precursor to any automatic analysis. The time equations can be solved by tranditional methods, yielding either an exact solution, an upper bound, or an approximate solution. (Incidentally, although [LeM85] claims to analyse a lazy language, the analysis uses exactly the composition rule above, and so is more suited for a strict language.)

The analysis given here involves two kinds of equations. First are equations defining projection transformers that specify how much of a value is "needed". Second are equations that specify the time complexity; these depend on the projection transformers defined by the first equations.

In both cases, we will be more concerned with setting up the equations than with finding their solutions. As already noted, traditional methods may be applied to satisfy the time complexity equations. Solving the projection transformer equations is more problematic. In some cases, we can find an appropriate solution by choosing an appropriate finite domain of projections, and then applying the method of [WH87] to find the solution in this domain. In other cases, no finite domain of solutions is appropriate, and we will find a solution by a more ad-hoc method: guessing one and verifying that it satisfies the required conditions. More work is required to determine what sort of solutions to the projection transformer equations will be most useful for time analysis, and how to find these solutions.

This paper is organized as follows. Section 1 describes the language to be analysed. Section 2 presents the evaluation model. Section 3 gives a form of time analysis suitable for strict evaluation. Section 4 shows how projections can describe what portion of a value is "needed", and introduces projection transformers. Section 5 gives the time analysis for lazy evaluation. Section 6 presents a useful extension to the analysis method. Section 7 concludes.

1 Language

We will use a first-order language with the following grammar:

$$\begin{array}{rcll} e & ::= & x & \text{variables} \\ & \mid & k & \text{constants} \\ & \mid & f\ e_1 \ldots e_n & \text{function applications} \\ & \mid & \textbf{if}\ \ e_0\ \ \textbf{then}\ \ e_1\ \ \textbf{else}\ \ e_2 & \text{conditionals} \end{array}$$

Function definitions have the form

$$f\ x_1 \ldots x_n \quad = \quad e$$

Infixes are allowed as usual; $e_1 + e_2$ is equivalent to $(+)\ e_1\ e_2$, where $(+)$ is a function name.

We take *nil*, *cons*, *head*, *tail*, and *null* to be the usual operations on lists, and we use $[1, 2, 3]$ as an abbreviation for *cons* 1 (*cons* 2 (*cons* 3 *nil*))).

As a running example, we will use an insertion sort program:

```
insert x xs  =  if  null xs
                  then  cons x nil
                  elseif  x ≤ head xs
                          then  cons x xs
                          else  cons (head xs) (insert x (tail xs))

sort xs      =  if  null xs
                  then  nil
                  else  insert (head x) (sort (tail xs))
```

We can compute the minimum of a list with

$$\mathit{minimum}\ \mathit{xs} \quad = \quad \mathit{head}\ (\mathit{sort}\ \mathit{xs})$$

As we shall see, this program is rather silly under strict evaluation (its time complexity is quadratic in the length of *xs*), but acceptable under lazy evaluation (its time complexity is linear).

2 Modelling evaluation

We will use the familiar graph-reduction model of evaluation. Lazy evaluation is modeled by normal order (outermost first) reduction, while strict evaluation is modeled by applicative order (innermost first) reduction.

Here are some of the reduction steps to find the minimum of a list under lazy evaluation:

$$\begin{array}{l}
\mathit{mimimum}\ [3,1,4,2] \\
\rightarrow \mathit{head}\ (\mathit{sort}\ [3,1,4,2]) \\
\rightarrow \mathit{head}\ (\mathit{insert}\ 3\ (\mathit{sort}\ [1,4,2])) \\
\rightarrow \mathit{head}\ (\mathit{insert}\ 3\ (\mathit{insert}\ 1\ (\mathit{sort}\ [4,2]))) \\
\rightarrow \mathit{head}\ (\mathit{insert}\ 3\ (\mathit{insert}\ 1\ (\mathit{insert}\ 4\ (\mathit{sort}\ [2])))) \\
\rightarrow \mathit{head}\ (\mathit{insert}\ 3\ (\mathit{insert}\ 1\ (\mathit{insert}\ 4\ (\mathit{insert}\ 2\ (\mathit{sort}\ [\,]))))) \\
\rightarrow \mathit{head}\ (\mathit{insert}\ 3\ (\mathit{insert}\ 1\ (\mathit{insert}\ 4\ (\mathit{insert}\ 2\ [\,])))) \\
\rightarrow \mathit{head}\ (\mathit{insert}\ 3\ (\mathit{insert}\ 1\ (\mathit{insert}\ 4\ [2]))) \\
\rightarrow \mathit{head}\ (\mathit{insert}\ 3\ (\mathit{insert}\ 1\ (\mathit{cons}\ 2\ (\mathit{insert}\ 4\ [\,])))) \\
\rightarrow \mathit{head}\ (\mathit{insert}\ 3\ (\mathit{cons}\ 1\ (\mathit{cons}\ 2\ (\mathit{insert}\ 4\ [\,])))) \\
\rightarrow \mathit{head}\ (\mathit{cons}\ 1\ (\mathit{insert}\ 3\ (\mathit{cons}\ 2\ (\mathit{insert}\ 4\ [\,])))) \\
\rightarrow 1
\end{array}$$

Strict evaluation would require more reduction steps, and perform them in a different order.

An obvious measure to use for time to evaluate an expression is the number of steps required to reduce the expression to normal form. But exactly what should we count as a reduction step? We will choose to count *call steps*, that is, we count one step each time an application of a non-primitive function is reduced. For example, the above reduction requires ten steps; the final step is not counted because *head* is taken to be primitive. The reader may verify that strict evaluation of the same term requires fourteen call steps.

Note that we are using a *graph reduction* model, so that steps to reduce a common sub-expression are counted only once. Efficient implementation of functional languages based on a graph reduction model are discussed in [Pey87].

3 Strict time analysis

As noted in the introduction, time analysis of programs under strict evaluation is relatively straightforward. The key is to use the composition rule mentioned in the introduction. In this section, we formulate such an analysis more precisely, as a first step toward developing the equivalent analysis for lazy evaluation.

If f is a function of n arguments, we let f^T be a function of n arguments such that $f^T\ x_1 \ldots x_n$ is the number of call steps required to evaluate $f\ x_1 \ldots x_n$. For example, we will see that if n is the length of xs, then $\mathit{insert}^T\ x\ xs$ is at most n, and $\mathit{sort}^T\ xs$ is at most $(n+1)(n+2)/2$.

We define f^T as follows. If f is a primitive function (such as *cons* or *head*), we set $f^T\ x_1 \ldots x_n = 0$. Otherwise, f must have a definition of the form

$$f\ x_1 \ldots x_n \quad = \quad e$$

and we set

$$f^T\ x_1 \ldots x_n \quad = \quad 1 + e^T$$

where e^T denotes the number of steps required to evaluate e. That is, evaluating $f\ x_1 \ldots x_n$ requires one step to reduce to e, plus the steps to evaluate e.

It remains to define e^T:

$$\begin{array}{lcl}
x^T & = & 0 \\
k^T & = & 0 \\
(f\ e_1 \ldots e_n)^T & = & (f^T\ e_1 \ldots e_n) + e_1^T + \cdots + e_n^T \\
(\mathbf{if}\ \ e_0\ \ \mathbf{then}\ \ e_1\ \ \mathbf{else}\ \ e_2)^T & = & e_0^T + (\mathbf{if}\ \ e_0\ \ \mathbf{then}\ \ e_1^T\ \ \mathbf{else}\ \ e_2^T)
\end{array}$$

For example, we can derive

$$\begin{array}{lcl}
(f\ (g\ x))^T & = & (f^T\ (g\ x)) + (g\ x)^T \\
 & = & (f^T\ (g\ x)) + (g^T\ x) + x^T \\
 & = & (f^T\ (g\ x)) + (g^T\ x) + 0 \\
 & = & (f^T\ (g\ x)) + (g^T\ x)
\end{array}$$

by applying the rule for $(f\ e_1 \ldots e_n)^T$ twice, and the rule for x^T once. This corresponds to the composition rule: the time to evaluate $(f\ (g\ x))$ equals the time to evaluate $(f\ y)$ plus the time to evaluate $(g\ x)$, where y is the value of $(g\ x)$.

Applying the analysis to the definitions of Section 1 we have:

```
insert^T x xs   =  1 + if  null xs
                       then  0
                       elseif  x ≤ head xs
                            then  0
                            else  insert^T x (tail xs)

sort^T xs       =  1 + if  null xs
                       then  0
                       else  insert^T (head xs) (sort (tail xs))
                            + sort^T (tail xs)

minimum^T xs    =  1 + sort^T xs
```

Note that the definition of $sort^T$ refers to *sort* itself, and to $sort^T$ (so it is recursive), and to $insert^T$.

In general, $f^T\ x_1 \ldots x_n$ will depend on the exact values of $x_1, \ldots, x_n$. For example, if n stands for the length of *xs*, then the value of $insert^T\ x\ xs$ will vary between 1 in the best case (where x is less than the first element of *xs*) and n in the worst case (where x is greater than all elements of *xs*). Similarly, the value of $sort^T\ xs$ will vary between $2n$ in the best case (where *xs* is already in ascending order), and $(n + 1)(n + 2)/2$ in the worst case (where *xs* is in descending order).

Often, analyses as exact as those given above are too precise. For many purposes, it is preferable to give an analysis in a closed form, and to achieve this we may be willing to settle for a worst-case or an approximate analysis (or both). There are well-known methods for deriving such analyses from exact analyses, and we will not deal with such issues in this paper. We simply note that, given the definitions of insert^T and sort^T above, it is straightforward to prove, for instance, that

$$\begin{array}{lcl} \mathit{insert}^T\ x\ \mathit{xs} & = & O(n) \\ \mathit{sort}^T\ \mathit{xs} & = & O(n^2) \end{array}$$

where, again, n denotes the length of xs.

4 Modelling contexts with projections

This section provides a short introduction to contexts as represented by projections. A more detailed introduction can be found in [WH87].

A continuous function α is a *projection* if for every object u,

$$\begin{array}{rcl} \alpha\ u & \sqsubseteq & u \\ \alpha\ (\alpha\ u) & = & \alpha\ u \end{array}$$

The first line says that projections only remove information from an object. The second line says that all the information is removed at once, so applying the projection a second time has no effect. The two lines above can also be written

$$\begin{array}{rcl} \alpha & \sqsubseteq & \mathit{ID} \\ \alpha \circ \alpha & = & \alpha \end{array}$$

where ID is the identity function, defined by $\mathit{ID}\ u = u$ for all u. In this paper, α and β always denote projections.

We will use projections to represent contexts, where the information not removed by the projection is that "needed" by the context. An example is the projection FIRST, defined by

$$\begin{array}{lcl} \mathit{FIRST}\ \mathit{nil} & = & \mathit{nil} \\ \mathit{FIRST}\ (\mathit{cons}\ u\ v) & = & \mathit{cons}\ u\ \bot \end{array}$$

This specifies a context in which only the first element of a list is needed (or the list is empty).

If f is a function of n arguments, we say that *f in an α context is β-safe in its i'th argument* if

$$\alpha\ (f\ u_1 \ldots u_i \ldots u_n) \ \sqsubseteq\ f\ u_1 \ldots (\beta\ u_i) \ldots u_n$$

for all $u_1, \ldots, u_n$. We abbreviate this by writing $f^i : \alpha \Rightarrow \beta$. (The notation f^i represents the i'th argument of f, not f composed with itself i times.) Informally, this says that the portion of $f\ u_1 \ldots u_n$ that is needed by α can be computed given only the portion of u_i that is needed by β. For example, since *head* is the function that returns the first element of a list, we have $\mathit{head}^1 : \mathit{ID} \Rightarrow \mathit{FIRST}$; and if *mapsquare* is the function that squares each element of a list, we have $\mathit{mapsquare}^1 : \mathit{FIRST} \Rightarrow \mathit{FIRST}$.

Strictness. We say that a function f is *strict in its i'th argument* if

$$f\ u_1 \ldots \perp \ldots u_n \quad = \quad \perp$$

for all $u_1, \ldots, u_n$, where $\perp$ appears in the i'th position.

In order to describe strictness with projections, we extend our domains with a new element $\lightning$, pronounced "abort". (The symbol is intended to resemble a lightning bolt.) We require that $\lightning \sqsubseteq \perp$ and that all functions are strict in $\lightning$, although they may or may not be strict in $\perp$; so $f \lightning = \lightning$ for all f, but $f\ \perp = \perp$ only for some f.

(A technical point: Everything, including (:) and **if**, is strict in $\lightning$. Thus, the least fixpoint of a recursive function definition is the constant $\lightning$ function. This is not what we want. For recursive function definitions in our language, we take the least fixpoint above *ABS*, where *ABS* is as defined later.)

The interpretation of $\alpha\ u = \lightning$ is that α requires a value more defined than u. A projection α is said to be *strict* if it requires a value more defined than $\perp$, that is, if $\alpha\ \perp = \lightning$. Thus, a context α is *non-strict* if $\alpha\ \perp = \perp$; whereas a function f is *strict* if $f\ \perp = \perp$.

The largest strict projection is *STR*, defined by

$$\begin{array}{rcll} \mathit{STR}\ u & = & \lightning & \text{if } u = \lightning \vee u = \perp \\ & = & u & \text{otherwise} \end{array}$$

It is straightforward to show that (for all f) f is strict in its i'th argument if and only if $f^i : \mathit{STR} \Rightarrow \mathit{STR}$.

Absence. We say that a function f *ignores its i'th argument* if

$$f\ u_1 \ldots u_i \ldots u_n \quad = \quad f\ u_1 \ldots \perp \ldots u_n$$

for every $u_1, \ldots, u_n$. The projection *ABS* is defined by

$$\begin{array}{rcll} \mathit{ABS}\ u & = & \lightning & \text{if } u = \lightning \\ & = & \perp & \text{otherwise} \end{array}$$

It is straightforward to show that (for all f) f ignores its i'th argument if and only if $f^i : \mathit{STR} \Rightarrow \mathit{ABS}$. Furthermore, $f^i : \mathit{ABS} \Rightarrow \mathit{ABS}$ holds for every f and i.

If it is safe to evaluate a sub-expression in the context *ABS* then the value of the sub-expression is ignored; since we are in a lazy language, no evaluation will be required. As we will see, this makes *ABS* a key to the time analysis method of this paper.

The lattice of projections. Projections form a lattice, with *ID* at the top, and *FAIL* at the bottom, where *FAIL* is defined by $\mathit{FAIL}\ u = \lightning$ for all u. For the projections mentioned so far, we have

$$\begin{tikzcd} & \mathit{ID}\ \bullet & \\ \mathit{ABS}\ \bullet & & \bullet\ \mathit{STR} \\ & \mathit{FAIL}\ \bullet & \end{tikzcd}$$

There are an infinite number of other projections α, which are all either strict ($\mathit{FAIL} \sqsubseteq \alpha \sqsubseteq \mathit{STR}$) or non-strict ($\mathit{ABS} \sqsubseteq \alpha \sqsubseteq \mathit{ID}$).

Usually, the bottom element of a lattice represents no information, and the top element represents contradictory information. For the lattice of projections, the opposite is true. The top element represents no information: *ID* denotes the context that may or may not require a value; and the bottom element represents contradictory information: *FAIL* denotes the context satisfied by no value. In general, $\alpha \sqsubseteq \beta$ if α represents *more* (rather than less) information than β.

The constant context. The projection *CONST* v, defined by

$$\begin{array}{lcll} \mathit{CONST}\ v\ u & = & v & \text{if } u = v \\ & = & \lightning & \text{otherwise} \end{array}$$

denotes a context that requires exactly the value v. For instance, if *square* is the squaring function, and a context requires *square* x to be 4, then x must be 2 or -2; so $\mathit{square}^1 : \mathit{CONST}\ 4 \Rightarrow \mathit{CONST}\ 2 \sqcup \mathit{CONST}\ (-2)$. Thus, projections can convey quite precise information, although for many purposes it is sufficient to use projections (such as *STR*) that are less precise.

List contexts. The projections *NIL* and *CONS* $\alpha\ \beta$, defined by

$$\begin{array}{lcl} \mathit{NIL}\ u & = & \mathit{STR}\ (u \sqcap \mathit{nil}) \\ \mathit{CONS}\ \alpha\ \beta\ u & = & \mathit{STR}\ (u \sqcap \mathit{cons}\ (\alpha\ (\mathit{head}\ u))\ (\beta\ (\mathit{tail}\ u))) \end{array}$$

denote the context that requires the empty list, and the context that requires a cons cell with head in context α and tail in context β. For example,

$$\mathit{CONS}\ (\mathit{CONST}\ 5)\ (\mathit{CONS}\ \mathit{ABS}\ \mathit{NIL})$$

denotes the context that requires a list of exactly length two, the first element of which must be 5, and the second element of which will be ignored.

Again, for many purposes it is sufficient to use less precise projections, such as *FIRST* or *ALL*. Recall that *FIRST* denotes the context which requires an empty list or only the first element of a list; it may be defined by

$$\mathit{FIRST} \ = \ \mathit{NIL} \sqcup \mathit{CONS}\ \mathit{ID}\ \mathit{ABS}$$

The projection *ALL*, defined by

$$\mathit{ALL} \ = \ \mathit{NIL} \sqcup \mathit{CONS}\ \mathit{STR}\ \mathit{ALL}$$

denotes the context that requires every element of a list.

Projection transformers. The problem of context analysis is this: given f, i, and α we wish to find a β such that $f^i : \alpha \Rightarrow \beta$. By a small abuse of notation, we also let f^i stand for a function of α that yields such a β. That is, f^i stands for a function over projections such that $f^i : \alpha \Rightarrow (f^i\ \alpha)$, for every projection α over the range of f. We call f^i a *projection transformer*.

Of course, we could just set $f^i\ \alpha = ID$ for every f, i, and *alpha*, since we always have $f^i : \alpha \Rightarrow ID$. But if possible we would like to find a smaller projection, since a smaller value for $f^i\ \alpha$ will allow for a more precise time analysis.

Ideally, we would like to find the *smallest* β satisfying $f^i : \alpha \Rightarrow \beta$, but there are two problems with this. First, it is not clear that a smallest such β always exists. Second, even if it did exist, it would not be computable. It is easy to show that $f^i : STR \Rightarrow FAIL$ holds iff f diverges for every value of its i'th argument; thus if we could always determine the smallest β then we could solve the halting problem. Therefore, we must settle for letting $f^i\ \alpha$ be *some* β satisfying $f^i : \alpha \Rightarrow \beta$, but not necessarily the smallest such β.

In particular, we can give projection transformers for *head*, *tail*, and *cons* as follows:

$$\begin{array}{lcl} head^1\ \alpha & = & CONS\ \alpha\ ABS \\ tail^1\ \alpha & = & CONS\ ABS\ \alpha \\[1ex] cons^1\ \alpha\ u & = & \bigsqcup_{v \in D^\star} head\ (\alpha\ (cons\ u\ v)) \\ cons^2\ \alpha\ v & = & \bigsqcup_{u \in D} tail\ (\alpha\ (cons\ u\ v)) \end{array}$$

where D is a domain of list elements, and $D^\star$ is the domain of lists over D.

As another example, we might choose projection transformers for *sort* and *insert* such that

$$\begin{array}{lcl} sort^1\ ALL & = & ALL \\ insert^1\ ALL & = & STR \\ insert^2\ ALL & = & ALL \\[1ex] sort^1\ FIRST & = & ALL \\ insert^1\ FIRST & = & STR \\ insert^2\ FIRST & = & FIRST \end{array}$$

The last three lines say that to find the first element of *sort xs* one must evaluate all of xs, but to find the first element of *insert x xs* one need only evaluate x and the first element of xs.

Rules for deriving projection transformers are beyond the scope of this paper, but are given in [WH87]. The technique there is divided into two parts: deriving equations that projection transformers must satisfy, and constructing appropriate finite domains of projections. The equations are then solved by the usual fixpoint iteration technique; since the domain is finite, convergence is guaranteed. The finite domains given in that paper appear to be appropriate for strictness analysis, but not for time analysis. In general, for each time analysis one may need to choose different finite domains. In this case, one must note that $FIRST$ and ALL are the relevant contexts (the finite domains for lists discussed in [WH87] include ALL but not $FIRST$). Once the finite domains have been chosen, the equations may be solved automatically.

5 Lazy time analysis

We now generalize the previous method of time analysis, by adding a parameter that specifies the context in which a function or expression is to be evaluated. By using projection transformers, we can determine which sub-expressions will be demanded in context ABS, and hence require no evaluation.

We generalize $f^T\ x_1 \ldots x_n$ to $f^T\ x_1 \ldots x_n\ \alpha$, which denotes the number of call steps required to evaluate $f\ x_1 \ldots x_n$ in context α. Similarly, we generalize e^T to $e^T\ \alpha$, which denotes the number of call steps to evaluate e in context α.

As before, if f is primitive then we set $f^T\ x_1 \ldots x_n\ \alpha = 0$. Otherwise, f must have a definition of the form

$$f\ x_1 \ldots x_n \quad = \quad e$$

and we set

$$\begin{array}{rcll} f^T\ x_1 \ldots x_n\ \alpha & = & 0 & \text{if } \alpha = ABS \\ & = & 1 + e^T\ \alpha & \text{otherwise} \end{array}$$

We define $e^T\ \alpha$ by setting $e^T\ ABS = 0$, and if $\alpha \neq ABS$,

$$\begin{array}{lcl} x^T\ \alpha & = & 0 \\ k^T\ \alpha & = & 0 \\ (f\ e_1 \ldots e_n)^T\ \alpha & = & (f^T\ e_1 \ldots e_n\ \alpha) + (e_1^T\ (f^1\ \alpha)) + \cdots + (e_n^T\ (f^n\ \alpha)) \\ (\mathbf{if}\ \ e_0\ \ \mathbf{then}\ \ e_1\ \ \mathbf{else}\ \ e_2)^T\ \alpha & = & e_0^T\ STR + (\mathbf{if}\ \ e_0\ \ \mathbf{then}\ \ e_1^T\ \alpha\ \ \mathbf{else}\ \ e_2^T\ \alpha) \end{array}$$

This corresponds directly to the strict version, the key difference being the use of projection transformers f^i to derive the contexts in which $e_1, \ldots, e_n$ are evaluated when evaluating $f\ e_1 \ldots e_n$.

For example, since $cons^1\ FIRST = STR$ and $cons^2\ FIRST = ABS$, we have

$$\begin{array}{cl} \multicolumn{2}{l}{(cons\ (head\ xs)\ (insert\ x\ (tail\ xs)))^T\ FIRST} \\ = & cons^T\ (head\ xs)\ (insert\ x\ (tail\ xs))\ FIRST \\ & \quad + (head\ xs)^T\ (cons^1\ FIRST) \\ & \quad + (insert\ x\ (tail\ xs))^T\ (cons^2\ FIRST) \\ = & 0 + (head\ xs)^T\ STR + (insert\ x\ (tail\ xs))^T\ ABS \\ = & 0 + 0 + 0 \end{array}$$

Note that this expression appears as the second branch of the conditional in the definition of *insert*. If *insert x xs* is evaluated in context *FIRST* no recursive call of *insert* will be required, because the recursive call appears in the context *ABS*.

Applying the analysis to the definitions of Section 1 we have:

$$
\begin{array}{lcl}
\mathit{insert}^T\ x\ \mathit{xs}\ \mathit{FIRST} & = & 1 \\[1em]
\mathit{sort}^T\ \mathit{xs}\ \mathit{FIRST} & = & 1 + \textbf{if}\ \ \mathit{null}\ \mathit{xs} \\
 & & \qquad \textbf{then}\ \ 0 \\
 & & \qquad \textbf{else}\ \ \mathit{insert}^T\ (\mathit{head}\ \mathit{xs})\ (\mathit{sort}\ (\mathit{tail}\ \mathit{xs}))\ \mathit{FIRST} \\
 & & \qquad\qquad + \mathit{sort}^T\ (\mathit{tail}\ \mathit{xs})\ \mathit{FIRST} \\[1em]
\mathit{minimum}^T\ \mathit{xs}\ \mathit{STR} & = & 1 + \mathit{sort}^T\ \mathit{xs}\ \mathit{FIRST}
\end{array}
$$

The equation for *minimum* was derived using the fact that $\mathit{head}^1\ \mathit{STR} = \mathit{CONS}\ \mathit{STR}\ \mathit{ABS} \sqsubseteq \mathit{FIRST}$.

We can solve the above equations to show that

$$
\begin{array}{lcl}
\mathit{sort}^T\ \mathit{xs}\ \mathit{FIRST} & = & 2n + 1 \\
\mathit{minimum}^T\ \mathit{xs}\ \mathit{STR} & = & 2n + 2
\end{array}
$$

where n is the length of *xs*. This fulfills our promise to show that the time complexity of *minimum* is linear under lazy evaluation. Note that in this case it is easy to obtain an exact analysis in closed form, because the number of call steps to find the minimum depends only on the length of the list, and not on the data in it.

It is also easy to show that $\mathit{insert}^T\ x\ \mathit{xs}\ \mathit{ALL}$ and $\mathit{sort}^T\ \mathit{xs}\ \mathit{ALL}$ yield definitions exactly equivalent to those yielded by the analysis for strict evaluation in Section 3. This is exactly what we would expect, since *ALL* forces all elements of the list to be evaluated, and so lazy and strict evaluation require equal time.

6 Extended projection transformers

The form of projection transformer used in the preceding throws away too much information for some analyses. This section briefly outlines an extension to remedy this problem.

Say that we wish to analyse the time complexity of

$$\mathit{take}\ m\ (\mathit{sort}\ \mathit{xs})$$

evaluated in the context *ALL*, where *take m ys* returns the first m elements of the list *ys*. To do this, we must use the fact that *sort xs* is evaluated in the context *FIRST m* (a generalisation of the *FIRST* used previously) that requires the first m elements of a list; it is defined by

$$
\begin{array}{lcl}
\mathit{FIRST}\ 0 & = & \mathit{ABS} \\
\mathit{FIRST}\ (m+1) & = & \mathit{NIL} \sqcup \mathit{CONS}\ \mathit{STR}\ (\mathit{FIRST}\ m)
\end{array}
$$

Using the projection transformer for *take*, we could presumably express the desired relation among contexts by writing

$$\mathit{take}^2\ \mathit{ALL}\ =\ \mathit{FIRST}\ m$$

Unfortunately, this statement is meaningless! The problem is that m does not appear as an argument to take^2.

To fix this, we extend the projection transformers f^i so that, like f^T, they take all the arguments to f as well as the context in which it is called. The safety criterion is that if $f^i\ u_1 \dots u_n\ \alpha = \beta$ then we must have

$$\alpha\ (f\ u_1 \dots u_i \dots u_n) \ \sqsubseteq\ f\ u_1 \dots (\beta\ u_i) \dots u_n$$

For example, it is easy to show that

$$\mathit{take}^2\ m\ \mathit{xs}\ \mathit{ALL} \ = \ \mathit{FIRST}\ m$$

satisfies the safety criterion. Note that we pass both m and xs to take^2, although the value of xs is (in this case) ignored.

The equations for deriving projection transformers presented in [WH87] can be modified, in a straightforward way, to apply to extended projection transformers. However, it is not straightforward to solve the equations so derived, because we are no longer dealing with finite domains. (The projections *FIRST* m form an infinite family.) A simple ad-hoc method is to guess a solution and verify that it satisfies the safety criterion. This is what we did above.

The equations to compute $e^T\ \alpha$ require only a simple change:

$$(f\ e_1 \dots e_n)^T\ \alpha \ = \ (f^T\ e_1 \dots e_n\ \alpha) + (e_1^T\ (f^1\ e_1 \dots e_n\ \alpha)) + \cdots + (e_n^T\ (f^n\ e_1 \dots e_n\ \alpha))$$

The other rules remain the same.

Using the extended anlysis, it is straightforward to demonstrate that

$$(\mathit{take}\ m\ (\mathit{sort}\ \mathit{xs}))^T\ \mathit{ALL} \ = \ O(mn)$$

where n is the length of xs.

7 Conclusions

This paper has presented a method of analysing the time complexity of functional programs under lazy evaluation. The method uses *projections* to characterise the context in which an expression is evaluated—that is, how much of the value is “needed”. The analysis itself breaks into two parts. The first derives *projection transformers* that specify the context in which the argument of a function is evaluated in terms of the context in which the result is evaluated. The second derives *time equations* that specify the time required to compute a function in terms of its arguments and the context in which it is evaluated.

Projection analysis was first developed in connection with strictness analysis—determining which arguments of a function are guaranteed to be “needed” at run-time. It is this concern with knowing how much of a value is “needed” that forms the connection between strictness analysis and time analysis.

Strictness analysis and time analysis place different demands on projection analysis. For strictness analysis, it appears that one can find adequate solutions to the equations using the family of finite domains for lists described in [WH87]. For time analysis, the situation is less satisfactory. This paper has used two methods to solve the projection transformer equations. The first is to carefully choose a finite domain of solutions, and then to apply the methods of [WH87]. The second is to guess a solution and verify that it satisfies the safety criterion.

Further work is required to determine what sort of solutions to the projection transformer equations will be most useful for time analysis, and how to find these solutions. Two pieces of work that may be relevant are [Hug85], which uses algebraic and heuristic techniques to solve context equations, and [HW87], which uses "strictness patterns" that are similar to projections and has a solution space consisting of all strictness patterns representable by finite graphs.

Although the method outlined here appears to have general applicability, more examples of its use are required to verify whether this is the case. The issue of to what extent time analysis might be automated remains to be explored.

Clearly, many open questions remain. To conclude, here is another: Since projection analysis aids time analysis, can it similarly aid space analysis?

Acknowledgements. As mentioned previously, this paper owes much to Bror Bjerner. It also owes something to Richard Bird, who provided the insertion sort example.

References

[AH87] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987, to appear.

[Bje87] B. Bjerner. Complexity analysis of programs in type theory. Programming Methodology Group, Chalmers University of Technology, Göteborg, Sweden, 1987.

[Hug85] R. J. M. Hughes. Strictness detection in non-flat domains. In N. Jones and H. Ganzinger, editors, *Workshop on Programs as Data Objects*, Springer-Verlag, Copenhagen, October 1985. LNCS 217.

[Hug87b] R. J. M. Hughes. Backwards analysis of functional programs. University of Glasgow research report CSC/87/R3, March 1987.

[HW87] C. V. Hall and D. S. Wise. Compiling strictness into streams. In *14'th ACM Symposium on Principles of Programming Languages*, pages 132–143, Munich, January 1987.

[LeM85] D. Le Metayer. Mechanical analysis of program complexity. In *ACM SIGPLAN Symposium on Language Issues in Programming Environments*, Seattle, Washington, June 1985.

[Pey87] S. L. Peyton Jones. *Implementing Functional Languages using Graph Reduction.* Prentice-Hall, 1987.

[WH87] P. L. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *3'rd International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, September 1987.

[Weg75] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9): 528–539, September 1975.