

Complexity Analysis for a Lazy Higher-Order Language*

David Sands[†]

Department of Computing, Imperial College
180 Queens Gate, London SW7 2BZ
email: `ds@uk.ac.ic.doc`

Abstract

This paper is concerned with the time-analysis of functional programs. Techniques which enable us to reason formally about a program's execution costs have had relatively little attention in the study of functional programming. We concentrate here on the construction of equations which compute the time-complexity of expressions in a lazy higher-order language.

The problem with higher-order functions is that complexity is dependent on the cost of applying functional parameters. Structures called *cost-closures* are introduced to allow us to model both functional parameters *and* the cost of their application.

The problem with laziness is that complexity is dependent on *context*. Projections are used to characterise the context in which an expression is evaluated, and cost-equations are parameterised by this context-description to give a compositional time-analysis. Using this form of context information we introduce two types of time-equation: *sufficient-time* equations and *necessary-time* equations, which together provide bounds on the exact time-complexity.

1 Introduction

This paper is concerned with the time-analysis of functional programs. Techniques which enable us to reason formally about a program's execution costs have had relatively little attention in the study of functional programming. There has been some interest in the mechanisation of program cost analysis, perhaps the main examples being [1, 2, 3]. These works describe systems which analyse cost by first constructing (recursive) equations which describe the time-complexity of a functional program in a strict first-order language. A closed form expression for cost is obtained in some cases by mechanised manipulation (transformation) of these equations. The

*This paper appears in the Proceedings of the 1989 Glasgow Functional Programming Workshop, Springer-Verlag Workshops in Computing Series, and in an abridged form in the proceedings of ESOP 90, LNCS 432.

[†]This work was partially supported by ESPRIT Basic Research Action P3124

average-case solution of such equations is considered in [4, 5]. We concentrate here on the *first* part of this process—the construction of equations which compute the time-complexity of a given program. For programs written a first-order strict (*i.e.* call-by-value) language this is very straightforward. In the first part of this paper we show how to deal with a strict higher-order language (a fuller development can be found in [6]). In the remainder of the paper we adapt these ideas to a lazy language. This extension is based on Wadler’s use of *context-analysis* in the construction of time equations for a lazy first-order language [7].

The aim is to develop a calculus that enables us to reason about time-complexity. Given a program (which we will consider to be any expression, plus a set of mutually recursive function definitions), the problem is to find a means of constructing equations which describe the cost (in terms of the number of certain elementary operations) of evaluating any expression. In this paper we choose to express cost in terms of the number of non-primitive function applications. One advantage of deriving cost-equations which are themselves expressed in a functional language is that they are amenable to a rich class of program transformation and analysis techniques *c.f.* [2, 3]—this paper retains the functional flavour of these approaches.

The paper is organised as follows. In section 2 we consider the analysis of first and higher-order strict languages. Section 3 introduces a description of *context* that will be used in the analysis of lazy languages. Section 4 presents *sufficient-time* analysis, an upper-bound analysis for a lazy first-order language, which uses contexts that describe information that is *sufficient* to compute a value. Section 5 presents *necessary-time* analysis, a corresponding lower-bound analysis. Section 6 extends these ideas to a higher-order language.

2 Strict Time Analysis

In this section we consider the analysis of strict languages. A full presentation is given in [6].

2.1 A First Order Language

Firstly we define a simple first-order functional language. We consider a set of mutually recursive function definitions of the form

$$f_i(x_1, \dots, x_{n_i}) = e_i$$

and an expression to be evaluated in the context of these definitions. Expressions have the following syntax:

$$e ::= f(e_1, \dots, e_j) \mid \textit{ident} \mid \textit{const} \mid \textit{if } e_1 \textit{ then } e_2 \textit{ else } e_3$$

Where f is one of the user-defined functions f_i , or a strict primitive function or constructor p .

For each equation of the form $f_i(x_1, \dots, x_{n_i}) = e_i$ it is straightforward to construct an equation taking the same arguments as the original function, which computes the cost (in terms of the number of non-primitive function calls) of applying f_i to a tuple of values. The *cost equation* (or *cost-function*) is defined as:

$$cf_i(x_1, \dots, x_{n_i}) = 1 + \mathcal{T}[[e_i]]$$

where \mathcal{T} is a syntax-directed abstraction given in figure 1. These rules clearly reflect

$\begin{aligned} \mathcal{T}[[const]] &= \mathcal{T}[[ident]] = 0 \\ \mathcal{T}[[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]] &= \mathcal{T}[[e_1]] + \text{if } e_1 \text{ then } \mathcal{T}[[e_2]] \text{ else } \mathcal{T}[[e_3]] \\ \mathcal{T}[[p(e_1 \dots e_n)]] &= \mathcal{T}[[e_1]] + \dots + \mathcal{T}[[e_n]] \\ \mathcal{T}[[f_i(e_1 \dots e_n)]] &= cf_i(e_1 \dots e_n) + \mathcal{T}[[e_1]] + \dots + \mathcal{T}[[e_n]] \end{aligned}$
--

Figure 1: First-Order Strict Cost Definition

the call-by-value evaluation order. For example, in the rule for application, we sum the cost of evaluating the arguments, in addition to the function application. (*NB* We will use infix notation to ease presentation throughout this paper)

Syntax directed derivations of this form, for similar first order languages can be found in [1, 2, 3]. These works focus on some automatic techniques by which the recursive cost-equations can be manipulated to achieve non-recursive equations.

Example

As a simple example of the above scheme, consider the list-append function defined as:

$$\text{append}(x,y) = \text{if null}(x) \text{ then } y \text{ else cons}(\text{hd}(x), \text{append}(\text{tl}(x),y))$$

From this definition, applying \mathcal{T} we obtain the cost-function which computes the number of non-primitive function applications:

$$\begin{aligned} \text{cappend}(x,y) &= 1 + \text{if null}(x) \text{ then } 0 \text{ else cappend}(\text{tl}(x),y) \\ &= 1 + \text{length}(x) \end{aligned}$$

The aim of the systems described in the papers cited above is to derive just such a closed-form expression, by means of program transformation. This paper focuses on the process of obtaining the initial cost-functions, for languages using higher-order functions and laziness—a necessary precursor to the derivation of closed-form equations describing, for example, average-case complexity.

2.2 A Higher-Order Curried Language

In this section we outline a means of deriving cost programs for a higher-order language. The time-equations are derived via two mappings. The first modifies the

original equations so that functional values are augmented with information needed to describe the cost of their application. The second constructs the time-equations using these modified equations.

Firstly we define our language. We have function definitions of the form

$$f_i e_1 \dots e_{n_i} = exp_i$$

along with curried primitive functions p_i (of arity m_i). The syntax of expressions is given in figure 2.

$exp ::= exp\ e$	(application)
$\quad \quad e$	
$e ::= f_i$	(function)
$\quad \quad p$	(primitive function)
$\quad \quad x$	(identifier)
$\quad \quad c$	(constant)
$\quad \quad (exp)$	(parenthesis)
$\quad \quad \text{if } e_1 \text{ then } e_2 \text{ else } e_3$	(conditional)

Figure 2: Expression Syntax

For each definition $f_i x_1 \dots x_{n_i} = exp_i$ we wish to construct a cost function

$$cf_i x_1 \dots x_{n_i} = exp'_i$$

which computes the cost of applying f_i to n_i values.

Suppose we wish to construct a cost-function for an apply function defined as: `apply f x = f x`. The cost function associated with `apply` should have the form:

$$\text{Capply}(f, x) = 1 + \text{the cost of applying } f \text{ to } x.$$

But how do we *syntactically* refer to the cost function associated with `f`?

Cost-Closures

In order to reason about the cost of application of functions, as well as the functions themselves, we introduce structures called *cost-closures*. A cost-closure is a triple (f, cf, a) of a function f , its associated cost-function cf and some arity information a . Together with cost-closures we define two (left associative) infix functions @ and c@ which define the application of cost-closures and the cost of application. Functions @ and c@ satisfy:

$$(f, cf, a) \text{@ } e = \begin{cases} f\ e & \text{if } a = 1 \\ (f\ e, cf\ e, a - 1) & \text{otherwise} \end{cases}$$

$$(f, cf, n) \text{c@ } e = \begin{cases} cf\ e & \text{if } n = 1 \\ 0 & \text{otherwise} \end{cases}$$

The arity component of the cost-closure, and its use in the definition of $c\mathcal{C}$ is explained by the fact that for reasons of efficiency and simplicity, there is no evaluation of the body of a function until the function is supplied with at least the number of arguments in its definition (this avoids the potentially expensive resolution of name clashes, and is thus a feature of most functional language implementations).

Cost-closures are used in the following way. We define two syntax-directed translation functions \mathcal{V} and \mathcal{T} . The purpose of \mathcal{V} (figure 4) is to modify the original program so that all functional objects are translated into cost-closures, and to perform application via \mathcal{C} . \mathcal{T} (figure 5) defines the cost-functions, using $c\mathcal{C}$. The cost of evaluating any expression exp with respect to definitions $f_i x_1 \dots x_{n_i} = exp_i$, $i = 1, \dots, k$ is then defined by the program given in figure 3. \mathcal{V} is defined on the

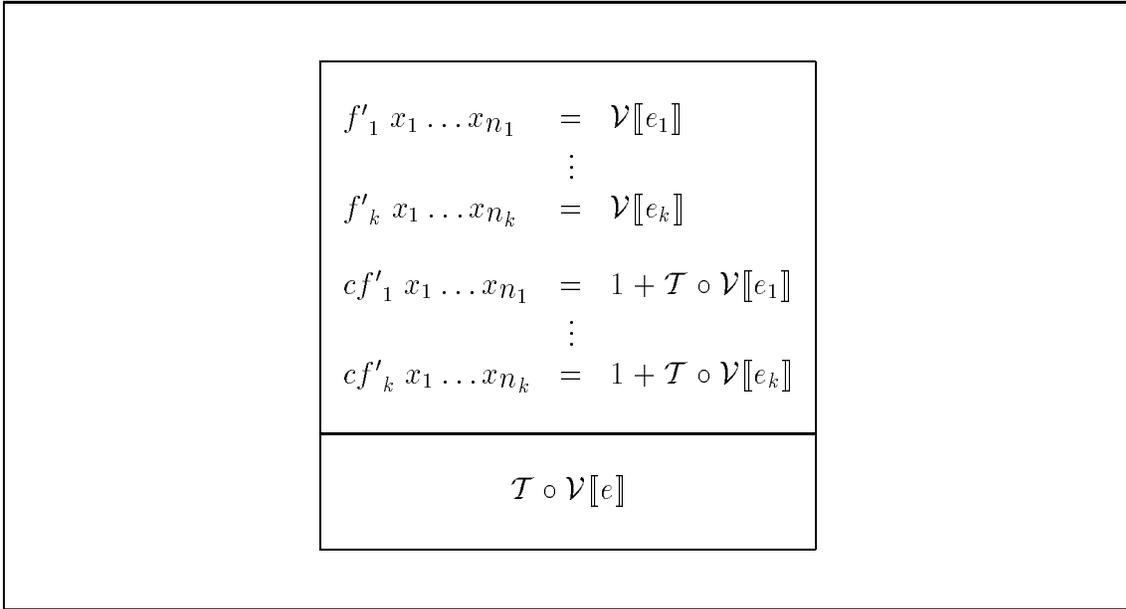


Figure 3: Higher-Order Cost-Program Scheme

structure of expressions exp and e . \mathcal{T} is consequently defined over the syntax of expressions generated by \mathcal{V} .

Some Optimisations

The code derived by the above translation schemes is rather more cumbersome than is necessary. This is because we introduce more \mathcal{C} 's and $c\mathcal{C}$'s than are necessary. Some straightforward optimisations simplify the cost program considerably, and can be defined according to the syntactic structure of expressions [6].

Example

The following simple example illustrates the derivation (and the optimisation):

```
map f x = if (null x) then nil
         else (cons (f (hd x)) (map f (tl x)))
```

$$\begin{aligned}
\mathcal{V}[\text{exp } e] &= \mathcal{V}[\text{exp}] \textcircled{\text{c}} \mathcal{V}[e] \\
\mathcal{V}[\text{if } e_1 \text{ then } e_2 \text{ else } e_3] &= \text{if } \mathcal{V}[e_1] \text{ then } \mathcal{V}[e_2] \text{ else } \mathcal{V}[e_3] \\
\mathcal{V}[(\text{exp})] &= (\mathcal{V}[\text{exp}]) \\
\mathcal{V}[f_i] &= (f'_i, cf_i, n_i) \\
\mathcal{V}[p_i] &= (p_i, cp_i, m_i) \\
\mathcal{V}[c] &= c \\
\mathcal{V}[x] &= x
\end{aligned}$$

Figure 4: Function Modification Map, \mathcal{V}

$$\begin{aligned}
\mathcal{T}[\text{exp}' \textcircled{\text{c}} e'] &= \mathcal{T}[\text{exp}'] + \mathcal{T}[e'] + (\text{exp}' \textcircled{\text{c}} e') \\
\mathcal{T}[\text{if } e'_1 \text{ then } e'_2 \text{ else } e'_3] &= \mathcal{T}[e'_1] + \text{if } e'_1 \text{ then } \mathcal{T}[e'_2] \text{ else } \mathcal{T}[e'_3] \\
\mathcal{T}[(\text{exp}')] &= (\mathcal{T}[\text{exp}']) \\
\mathcal{T}[(p_i, cp_i, m_i)] &= \mathcal{T}[(f_i, cf_i, n_i)] = 0 \\
\mathcal{T}[c] &= \mathcal{T}[x] = 0
\end{aligned}$$

Figure 5: Cost-Expression Construction Map, \mathcal{T}

The cost-function derived from this is:

```

cmap f x = 1 + ((null,cnull,1) c@ x) +
  if ((null,cnull,1) @ x) then nil
  else (((cons,ccons,2) c@ (f @ ((hd,chd,1) @ x)))
+ ((cons,ccons,2)@(f @ ((hd,chd,1)@ x)) c@
  ((map',cmap,2)@ f @ ((tl,ctl,1)@ x)))
+ f c@ ((hd,chd,1)@ x) + ((hd,chd,1)c@ x)
+ ((map',cmap,2)@ f c@ ((tl,ctl,1)@ x))
+ ((map',cmap,2)c@ f) + ((tl,ctl,1)c@ x) )

```

Using simple optimisation schemes, we get the equivalent cost-function definition:

```

cmap f x = 1 + if (null x) then 0
  else (f c@ (hd x)) + (cmap f (tl x))

```

2.3 Correctness

The derived program computes the number of times a certain “step” is performed in the evaluation of the program. In this section we formalise our intuitive model of “evaluation steps” via an *operational semantics*. We prove that the number of steps our derived program computes is correct with respect to the actual operational behaviour of the original program.

Semantics

The (dynamic) operational semantics for our language is defined by an inference system (a set of rules and axioms) in the style of *Natural Semantics* [8].

Step-counting

In order to reason about complexity we define a dynamic semantics so that the evaluation of an expression gives a pair of the value, and the number of reductions of non-primitive functions, (which corresponds to the use of a particular rule in the semantics). In this “step-counting semantics”, given in figure 6 the rules define judgements of the form

$$\rho \vdash_{\phi} exp \xrightarrow{s} \langle v, t \rangle$$

which is read as

Given environment ρ and function environment ϕ , evaluating exp yields value v , with t reductions of non-primitive function applications.

This generalises a standard semantics (not given here) whose judgements are of the form:

$$\rho \vdash_{\phi} exp \rightarrow v$$

Here we sketch some of the details in order to state the correctness theorem.

Environments

The environment to the left of the turnstile is used to map identifiers onto values. This environment is represented by a list, $\langle v_1, \dots, v_n \rangle$ where the i_{th} element, v_i , is the value bound to identifier x_i . ρ is a list of values, where ρ_i denotes the i_{th} element, and $\rho ++ v$ extends ρ with value v .

We will assume that we have constructed a function-environment which maps the function-names to the right-hand-side of their definition. Informally we parameterise the turnstile in the sentences by this environment—it would be straightforward to include the construction of this environment in the semantic rules.

In the following we will use ϕ' to denote the function-environment of the cost-program (see figure 3) corresponding to some function environment ϕ .

Closures

Closures represent values of function-type. A closure is a triple of the form (f, n, ρ) consisting of a function name, f , an (incomplete) environment, ρ , and an arity n (> 0) representing the number of values required to make the environment complete.

To relate values in the cost-program to values in the original program, (this includes environments, which are by definition lists of values) we define a convenient mapping:

SApp.1	$\frac{\rho \vdash_{\phi} \text{exp} \xrightarrow{s} \langle (f, n', \rho'), n_1 \rangle \quad \rho \vdash_{\phi} e \xrightarrow{s} \langle v', n_2 \rangle}{\rho \vdash_{\phi} \text{exp } e \xrightarrow{s} \langle (f, n' - 1, \rho' ++ v'), n_1 + n_2 \rangle} \text{ if } n' > 1$
SApp.2	$\frac{\rho \vdash_{\phi} \text{exp} \xrightarrow{s} \langle (f_i, 1, \rho'), n_1 \rangle \quad \rho \vdash_{\phi} e \xrightarrow{s} \langle v', n_2 \rangle \quad \rho' ++ v' \vdash_{\phi} \phi(f_i) \xrightarrow{s} \langle v, n_3 \rangle}{\rho \vdash_{\phi} \text{exp } e \xrightarrow{s} \langle v, n_1 + n_2 + n_3 + 1 \rangle}$
SApp.3	$\frac{\rho \vdash_{\phi} \text{exp} \xrightarrow{s} \langle (p_i, 1, \rho'), n_1 \rangle \quad \rho \vdash_{\phi} e \xrightarrow{s} \langle v', n_2 \rangle}{\rho \vdash_{\phi} \text{exp } e \xrightarrow{s} \langle v, n_1 + n_2 \rangle} \text{ if } \mathbf{Apply}(p_i, (\rho' ++ v')) = v$
SCond.1	$\frac{\rho \vdash_{\phi} e_1 \xrightarrow{s} \langle \mathbf{true}, n_1 \rangle \quad \rho \vdash_{\phi} e_2 \xrightarrow{s} \langle v, n_2 \rangle}{\rho \vdash_{\phi} \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \xrightarrow{s} \langle v, n_1 + n_2 \rangle}$
SCond.2	$\frac{\rho \vdash_{\phi} e_1 \xrightarrow{s} \langle \mathbf{false}, n_1 \rangle \quad \rho \vdash_{\phi} e_3 \xrightarrow{s} \langle v, n_2 \rangle}{\rho \vdash_{\phi} \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \xrightarrow{s} \langle v, n_1 + n_2 \rangle}$
SBrac	$\frac{\rho \vdash_{\phi} \text{exp} \xrightarrow{s} \langle v, n \rangle}{\rho \vdash_{\phi} (\text{exp}) \xrightarrow{s} \langle v, n \rangle}$
SUserf	$\rho \vdash_{\phi} f_i \xrightarrow{s} \langle (f_i, n_i, \langle \rangle), 0 \rangle$
SPrimf	$\rho \vdash_{\phi} p_i \xrightarrow{s} \langle (p_i, m_i, \langle \rangle), 0 \rangle \quad \text{if } m_i = \text{arity}(p_i)$
SIdent	$\rho \vdash_{\phi} x_j \xrightarrow{s} \langle \rho_j, 0 \rangle$
SConst	$\rho \vdash_{\phi} c \xrightarrow{s} \langle c, 0 \rangle$

Figure 6: Step-Counting Semantics

DEFINITION 2.1

$$v^{cc} = \begin{cases} \langle v_1^{cc}, \dots, v_k^{cc} \rangle & \text{if } v = \langle v_1, \dots, v_k \rangle \\ \langle (f', n, \rho^{cc}), (cf, n, \rho^{cc}), n \rangle & \text{if } v = (f, n, \rho) \\ v & \text{otherwise} \end{cases}$$

□

Thus function values (closures) are related to a three element list (a cost-closure) in the cost program. (We are taking the semantics of a cost closure to be that of a three element list in order to show that we do not need an extended language to implement them)

The following lemma establishes that $\mathcal{V}[\cdot]$ preserves the meaning of programs (modulo cc):

LEMMA 2.2 *For all expressions exp , if there exists a value v such that*

$$\rho \vdash_{\phi} exp \rightarrow v$$

then

$$\rho^{cc} \vdash_{\phi'} \mathcal{V}[[exp]] \rightarrow v^{cc}$$

The main correctness theorem is as follows:

THEOREM 2.3 *For all expressions exp , and value environments ρ , if there exists a value v such that*

$$\rho \vdash_{\phi} exp \xrightarrow{s} \langle v, t \rangle$$

for some t , then

$$\rho^{cc} \vdash_{\phi'} \mathcal{T} \circ \mathcal{V}[[exp]] \rightarrow t$$

Note that this is not a total correctness (\iff)—it says nothing about nontermination or run-time errors in the evaluation of the original program. It is easy to see that nontermination will be inherited by the cost program, whereas run-time errors (e.g. $\text{hd}(\text{nil})$) may not, and so the cost-program may be more defined than the original.

The proofs of the lemma and the main theorem follow by induction on the structure of the proofs (derivations) in the operational semantics. These are given in full in [6].

3 Lazy Time Analysis: Describing Context

A major obstacle in the time-analysis of lazy languages is the problem of *context sensitivity*: the cost of evaluating an expression depends on the context in which it is used. In order to give a *compositional* treatment of the analysis of lazy-evaluation we must take into account some description of context.

3.1 Modelling Contexts with Projections

The formulation of a context which will be used in our time analysis is that provided by Wadler and Hughes [9] in the analysis of *strictness*. Wadler shows how this formulation of context can be useful for time analysis in [7]. Here we provide an introduction to the use of *projections* to model contexts. For a fuller development the reader is referred to [9]; a more formal development, together with enhanced analysis techniques is given in [10].

The basic problem is, given a function, how much information do we require from the argument in order to determine a certain amount of information about the result. Projections, in the domain theoretic sense, can provide a concise description of both the amount of information which is *sufficient* and the amount which is *necessary*.

DEFINITION 3.1 *A projection, α , is a continuous function from a domain \mathcal{D} onto itself, such that $\alpha \sqsubseteq \text{ID}_{\mathcal{D}}$ and $\alpha \circ \alpha = \alpha$, where $\text{ID}_{\mathcal{D}}$ is the identity function on \mathcal{D} \square*

In other words, given an object u , a projection removes information from that object ($\alpha u \sqsubseteq u$), but once this information has been removed further application has no effect ($\alpha(\alpha u) = \alpha u$). A projection is used to represent a context, where the information removed represents information not needed by that context.

In the following the terms *projection* and *context* will be synonymous, and will be ranged over by α and β .

DEFINITION 3.2 Safe Projections: *Given a (first order) function, f , of n arguments, if*

$$\alpha(f(u_1, \dots, u_n)) = \alpha(f(u_1, \dots, (\beta u_i), \dots, u_n))$$

for all objects u_1, \dots, u_n , then we say that in context α , β is a safe context for the i 'th argument of f . This is abbreviated by $f^i : \alpha \Rightarrow \beta$. \square

Lifted Projections

We will require that projections describe two types of information: what information is *sufficient*, and what information is *necessary*. In order to describe the latter, Wadler and Hughes introduce a new domain element, \perp , called “abort”. The interpretation of $\alpha u = \perp$ is that context α requires a value more defined than u . To make this work, we must have $\perp \sqsubseteq -$ and all functions are naturally-extended to be strict in \perp , i.e., $f(u_1, \dots, \perp, \dots, u_n) = \perp$. These technical devices are explained more formally in [11] in terms of *lifting*.

The Projection Lattice

A projection $\alpha : \mathcal{D}_{\perp} \rightarrow \mathcal{D}_{\perp}$, is called *a projection over \mathcal{D}* . Projections over any domain form a lattice, with ordering \sqsubseteq , containing at least the points given in figure 7.

DEFINITION 3.3 *A strict projection is any projection α such that $\alpha(-) = \perp$ \square*

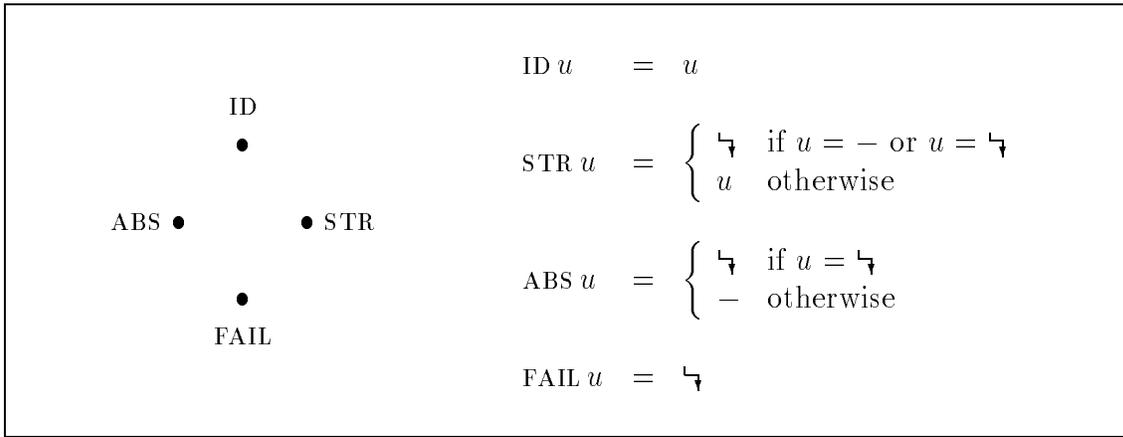


Figure 7: The Projection Lattice

The largest of such projections is STR, giving us an alternative definition of strict projections: a projection α is strict if and only if $\alpha \sqsubseteq \text{STR}$. Of the non-strict projections, the smallest is the projection ABS. This context is important since if it is safe to evaluate an expression in the context ABS, then the value of the expression will not be needed. FAIL is the unsatisfiable context.

There may be infinitely many projections, which are all either strict ($\text{FAIL} \sqsubseteq \alpha \sqsubseteq \text{STR}$) or non-strict ($\text{ABS} \sqsubseteq \alpha \sqsubseteq \text{ID}$). The four projections above will be used “polymorphically” to represent the corresponding projection over the appropriate domain.

Given two projections $\alpha \sqsubseteq \beta$, α represents a more precise description of a context than β . The context ID is therefore the least informative. Furthermore, if it is safe to evaluate some expression in a context α , then it is always safe to evaluate in a context β , $\alpha \sqsubseteq \beta$.

Contexts for Lists

The following projections are useful for building contexts over the non-flat domain of lists \mathcal{D}^* of elements from some domain \mathcal{D} :

$$\text{NIL } u = \begin{cases} \text{nil} & \text{if } u = \text{nil} \\ \Downarrow & \text{otherwise} \end{cases}$$

$$\text{CONS } \alpha \beta u = \begin{cases} \text{cons}(\alpha x)(\beta xs) & \text{if } u = \text{cons } x \ xs \\ \Downarrow & \text{otherwise} \end{cases}$$

NIL is the context which requires an empty-list, and $\text{CONS } \alpha \beta$ is the context which requires a non-empty list whose head is needed in context α and whose tail is needed in context β . For example, the context $(\text{CONS } \text{STR } \text{ABS})$ requires a non-empty list whose first element is needed, and the rest is not.

Context Analysis

Analysing context is a *backwards analysis*: given a context α for a function f , what can we say about the contexts of the arguments? We need to propagate the information about the result of a function *backwards* to its arguments. *i.e.*, given a function f of arity n , and a context α we need to find each β_i such that $f^i : \alpha \Rightarrow \beta_i$.

Ideally we need to find the smallest β_i , since these describe the contexts most precisely. In order to give a computable approximation we may settle for *some* β_i satisfying the above property.

Projection Transformers

A function of α yielding such a β_i is called a *projection transformer*. We will adopt the following notation: The projection-transformer written $f^{\#i}$ is a function satisfying $f^i : \alpha \Rightarrow f^{\#i} \alpha$. *NB* Strictly speaking we should distinguish between the syntactic objects—the program defining f , and the semantic objects—the projections, and the denotations given by some semantic function. Following the style of [9] we will mix these entities for notational convenience.

Rules for defining recursive equations for the projection transformers are given in [9]—an important result here is that a solution to these equations can be determined *automatically* if we work with finite lattices of projections, although it is not difficult to modify the equations to give more accurate projection equations (which are harder to solve).

4 Sufficient-Time Analysis

In this section we show how context information can be used to aid the time analysis of a lazy first-order language; Sufficient-time analysis (with some minor differences) corresponds to the time analysis presented in [7]. The information obtained by the backwards analysis is used to derive equations which compute an upper bound to the precise cost of a given program. This upper-bound is obtained by using information which tells us what values are *sufficient* to compute an expression. We call the resulting analysis a *sufficient-time* analysis.

4.1 Context-Parameterised Cost Functions

As in the first-order time analysis of section 2, we will define a *cost-function*, cf_i , for each function f_i defined in the original program. As before the cost functions will take as parameters the original arguments to the functions, but in addition they will be parameterised by a *context*, representing the context in which the functions are evaluated.

How can cost-functions make use of context ?

We know that any expression in the context ABS will be ignored, so the cost in this context is zero. In any other context the cost of a function application will be

$$\begin{aligned}
\mathcal{T}_s[[c]]\alpha &= \mathcal{T}_s[[x]]\alpha = 0 \\
\mathcal{T}_s[[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]]\alpha &= \alpha \hookrightarrow_s \mathcal{T}_s[[e_1]]\text{ID} + \text{if } e_1 \text{ then } \mathcal{T}_s[[e_2]]\alpha \text{ else } \mathcal{T}_s[[e_3]]\alpha \\
\mathcal{T}_s[[p(e_1 \dots e_n)]]\alpha &= \mathcal{T}_s[[e_1]](p^{\#1}\alpha) + \dots + \mathcal{T}_s[[e_n]](p^{\#n}\alpha) \\
\mathcal{T}_s[[f_i(e_1 \dots e_n)]]\alpha &= \mathcal{T}_s[[e_1]](f_i^{\#1}\alpha) + \dots + \mathcal{T}_s[[e_n]](f_i^{\#n}\alpha) \\
&\quad + cf_i(e_1 \dots e_n, \alpha)
\end{aligned}$$

Figure 8: Definition of \mathcal{T}_s

(approximated above by) 1 + the cost of evaluating the body of the function, in that context.

We define the cost functions associated with each function $f_i(x_1 \dots x_{n_i}) = e_i$ to be

$$cf_i(x_1, \dots, x_{n_i}, \alpha) = \alpha \hookrightarrow_s 1 + \mathcal{T}_s[[e_i]]\alpha$$

where we introduce the notation $\alpha \hookrightarrow_s e$ to abbreviate cost e “guarded” by context α :

$$\alpha \hookrightarrow_s e = \begin{cases} 0 & \text{if } \alpha = \text{ABS} \\ e & \text{otherwise} \end{cases}$$

The syntactic map \mathcal{T}_s defined in figure 8 is very similar to that defined in figure 1, but is defined with respect to a particular context. $\mathcal{T}_s[[e]]\alpha$ defines the cost of evaluating expression e in context α . It makes use of the context transformers $f_i^{\#1} \dots f_i^{\#n_i}$ defined for each function f_i , which satisfy the required safety criterion. In particular it will be appropriate to set $f^{\#m}(\text{ABS}) = \text{ABS}$, since if the result of a function is not needed, then neither are its arguments.

The rule for function application tells us that the cost of evaluating a function application is the associated cost-function applied to the arguments (and the context) plus the sum of evaluating the arguments in the contexts prescribed by the context-transformers.

The conditional expression, like any other, has zero cost in the context ABS (guaranteed by the use of \hookrightarrow_s). Otherwise we sum the cost of evaluating the condition (which may or may not be evaluated, hence the safe-context for boolean values ID, *c.f.* [7]) plus either the cost of the alternate or the consequent, depending on the *value* of the condition.

The cost of evaluating any expression in the context ABS is zero, so we have:

PROPOSITION 4.1 *For every expression e , $\mathcal{T}_s[[e]]\text{ABS} = 0$*

PROOF Straightforward structural induction in e □

A Small Example

Consider the expression: `hd(cons(not(true), exp))`, where

$$\text{not}(x) = \text{if } x \text{ then false else true}$$

and exp represents some arbitrary expression. The cost-function for `not` is

$$\begin{aligned} \text{cnot}(\mathbf{x}, \alpha) &= \alpha \hookrightarrow_s 1 + (\alpha \hookrightarrow_s 0 + \text{if } \mathbf{x} \text{ then } 0 \text{ else } 0) \\ &= \alpha \hookrightarrow_s 1 \end{aligned}$$

We assume a boolean-valued program is evaluated in the context `STR`, and so the cost program is defined by: $\mathcal{T}_s[\text{hd}(\text{cons}(\text{not}(\text{true}), exp))]$ `STR`, which is, by definition

$$\text{cnot}(\text{true}, \text{cons}^{\#1}(\text{hd}^{\#1}(\text{STR}))) + 0 + \mathcal{T}_s[exp] \text{cons}^{\#2}(\text{hd}^{\#1}(\text{STR}))$$

The context transformers for the primitive functions satisfy

$$\text{hd}^{\#1}(\alpha) = \text{CONS } \alpha \text{ ABS} \quad \text{cons}^{\#1}(\text{CONS } \alpha \beta) = \alpha \quad \text{cons}^{\#2}(\text{CONS } \alpha \beta) = \beta$$

and so the cost is $\text{cnot}(\text{true}, \text{STR}) + \mathcal{T}_s[exp] \text{ABS} = 1$, for any expression exp .

4.2 Approximation and Safety

What are the precise properties of the cost programs? Here we consider the approximation and correctness properties of the “lazy” cost-program.

Approximation

The expression $\mathcal{T}_s[e]\alpha$ gives an upper-bound estimate to the cost of lazy evaluation of e in context α . The cost expressions formed by \mathcal{T}_s are refinements of the call-by-value cost-expressions (section 2) in which subexpressions whose values are not needed do not contribute to the cost equation. Since the safety condition for projections does not specify that we require the smallest possible projection, the context `ABS` may be approximated by any larger projection. This approximation is reflected in the cost-program as an over-estimation of cost. (In the extreme case the context transformers are such that the context `ABS` is *never* derived in the cost-program, and so the value of the cost program is the same as that given by the strict derivation of figure 1.) Note also that in computing the cost of a function application $f(e)$ in context α the cost due to e will only be counted *once*. The context of e , $f^{\#1}(\alpha)$ will be the net context of the possible contexts in which e is shared, and so the process properly models call-by-*need*.

Safety

Whenever the cost-program terminates yielding a value, that value is indeed an upper bound to the time cost of evaluating the program lazily. A problem with this analysis method is that there are cases when the cost-program does not yield a value when it should do so. Firstly the cost-program may not terminate even when the program does—non-terminating cost expressions can be thought of as “computing” the worst possible upper-bound to the cost. However the approximation in the cost-program can lead to arbitrary run-time errors (*i.e.* not just nontermination). In the next section we introduce *necessary-time* equations which allow us to place a lower-bound on the precise complexity and which have better termination properties.

5 Necessary-Time Analysis

So far we have outlined the use of contexts to derive equations which can give an upper-bound to the time-complexity of an expression in a particular context. As mentioned previously, this idea is based on [7]. The cost-functions which compute this sufficient-complexity are only partially correct in the sense that if they compute a value, then that value is indeed an upper-bound to the time-cost of a program. There is potentially much more information about context using the projections described: *strict* contexts allow us to describe the amount of information which is *necessary* to compute a value. In this section we show how the use of this information can give us equations which describe a lower bound to the precise time-cost (the *necessary-time*) and which overcome the termination deficiencies of sufficient-time analysis. The key to sufficient-time analysis is the use of the context ABS to deduce that an expression will not be evaluated. The key to *necessary-time* is the operational interpretation of the *strict* projections.

5.1 Necessary-Cost Functions

In order to construct functions which compute the necessary-cost of evaluating a function in a particular context, we make the following operational connection between expressions which can be safely evaluated in a strict context, and their operational behaviour.

- If it is safe to evaluate an expression of the form $f(e_1, \dots, e_n)$ in a strict context, then operationally, we know that this outermost application must be reduced.

Conversely, if an expression is evaluated in a non-strict context then that expression *may or may not* be reduced (only the context ABS allows us to conclude that it *definitely* will not).

Motivated by this observation, we now define the necessary-cost. The cost of evaluating an expression e in a context α is given by $\mathcal{T}_n[[e]]\alpha$ where \mathcal{T}_n is once again a mapping defined over the syntax of expressions, and assuming some safe context transformers for the user-defined functions.

For each function definition of the form

$$f_i(x_1 \dots x_{n_i}) = e_i$$

we will define an associated necessary-cost-function

$$cf_i(x_1, \dots, x_{n_i}, \alpha) = \alpha \hookrightarrow_n 1 + \mathcal{T}_n[[e_i]]\alpha$$

where we use the notation $\alpha \hookrightarrow_n e$ to abbreviate necessary-cost e modulo context α :

$$\alpha \hookrightarrow_n e = \begin{cases} e & \text{if } \alpha \sqsubseteq \text{STR} \\ 0 & \text{otherwise} \end{cases}$$

The definition of \mathcal{T}_n is given in figure 9. The rules are very similar to the definitions for \mathcal{T}_s but we use \hookrightarrow_n in place of \hookrightarrow_s . The only other difference is in the translation for the conditional expression.

$$\begin{aligned}
\mathcal{T}_n[\mathit{const}]\alpha &= \mathcal{T}_n[\mathit{ident}]\alpha = 0 \\
\mathcal{T}_n[\mathit{if } e_1 \mathit{ then } e_2 \mathit{ else } e_3]\alpha &= \alpha \hookrightarrow_n \mathcal{T}_n[e_1]\mathit{STR} \\
&\quad + \mathit{if } e_1 \mathit{ then } \mathcal{T}_n[e_2]\alpha \mathit{ else } \mathcal{T}_n[e_3]\alpha \\
\mathcal{T}_n[p(e_1 \dots e_n)]\alpha &= \mathcal{T}_n[e_1](p^{\#1}\alpha) + \dots + \mathcal{T}_n[e_n](p^{\#n}\alpha) \\
\mathcal{T}_n[f_i(e_1 \dots e_n)]\alpha &= \mathcal{T}_n[e_1](f_i^{\#1}\alpha) + \dots + \mathcal{T}_n[e_n](f_i^{\#n}\alpha) \\
&\quad + cf_i(e_1 \dots e_n, \alpha)
\end{aligned}$$

Figure 9: Definition of \mathcal{T}_n

PROPOSITION 5.1 *For all contexts α , if $\alpha \sqsubseteq \mathit{STR}$ then*

$$\alpha(\mathit{if } u_1 \mathit{ then } u_2 \mathit{ else } u_3) = \alpha(\mathit{if } \mathit{STR}(u_1) \mathit{ then } u_2 \mathit{ else } u_3)$$

PROOF Straightforward by cases according, $u_1 \sqsupset -$ and $u_1 \sqsubseteq -$ □

This tells us that in any strict context it is safe to evaluate the condition in the context STR , and thus gives us the appropriate context for determining the cost due to the condition in the conditional expression.

5.2 Example

As a example of necessary-time analysis we use insertion-sort (as in [7]). The definitions are given in figure 10. The necessary-time equations constructed according to

```

insert(x,xs) = if null(xs) then cons(x,nil)
              else if x < hd(xs) then cons(x,xs)
              else cons(hd(xs),insert(x,tl(xs)))
sort(xs)     = if null(xs) then nil
              else insert(hd(xs),sort(tl(xs)))
min(xs)      = hd(sort(xs))

```

Figure 10: Insertion Sort

\mathcal{T}_n are given in figure 11: In this example we wish to consider the cost of evaluating min in a strict context. We are not particularly concerned here with the techniques for deriving the safe projection transformers. We note however that the projection transformers needed in this example are members of the finite domains for lists (and integers) described in [9] for the purpose of strictness analysis, and as such can be determined mechanically by fixpoint iteration. The equations we require are:

$$\begin{aligned}
\mathit{hd}^{\#1}(\mathit{STR}) &= \mathit{CONS } \mathit{STR } \mathit{ABS} \\
\mathit{cons}^{\#2}(\mathit{CONS } \mathit{STR } \mathit{ABS}) &= \mathit{ABS} \\
\mathit{insert}^{\#2}(\mathit{CONS } \mathit{STR } \mathit{ABS}) &= \mathit{CONS } \mathit{STR } \mathit{ABS}
\end{aligned}$$

$$\begin{aligned}
\text{cinsert}(x, \text{xs}, \alpha) &= \alpha \hookrightarrow_n 1 + \text{if null}(\text{xs}) \text{ then } 0 \\
&\quad \text{else if } x < \text{hd}(\text{xs}) \text{ then } 0 \\
&\quad \quad \text{else cinsert}(x, \text{tl}(\text{xs}), \text{cons}^{\#2}(\alpha)) \\
\text{csort}(\text{xs}, \alpha) &= \alpha \hookrightarrow_n 1 + \text{if null}(\text{xs}) \text{ then } 0 \\
&\quad \text{else cinsert}(\text{hd}(\text{xs}), \text{sort}(\text{tl}(\text{xs})), \alpha) + \\
&\quad \quad \text{csort}(\text{tl}(\text{xs}), \text{insert}^{\#2}(\alpha)) \\
\text{cmin}(\text{xs}, \alpha) &= \alpha \hookrightarrow_n 1 + \text{csort}(\text{xs}, \text{hd}^{\#1}(\alpha))
\end{aligned}$$

Figure 11: Necessary-Cost Functions

Now we examine the cost of `min`:

$$\begin{aligned}
\text{cmin}(\text{xs}, \text{STR}) &= \text{STR} \hookrightarrow_n 1 + \text{csort}(\text{xs}, \text{hd}^{\#1}(\text{STR})) \\
&= 1 + \text{csort}(\text{xs}, \text{CONS STR ABS})
\end{aligned}$$

$$\begin{aligned}
&\text{csort}(\text{xs}, \text{CONS STR ABS}) \\
&= 1 + \text{if null}(\text{xs}) \text{ then } 0 \\
&\quad \text{else cinsert}(\text{hd}(\text{xs}), \text{sort}(\text{tl}(\text{xs})), \text{CONS STR ABS}) + \\
&\quad \quad \text{csort}(\text{tl}(\text{xs}), \text{insert}^{\#2}(\text{CONS STR ABS})) \\
&= 1 + \text{if null}(\text{xs}) \text{ then } 0 \\
&\quad \text{else cinsert}(\text{hd}(\text{xs}), \text{sort}(\text{tl}(\text{xs})), \text{CONS STR ABS}) + \\
&\quad \quad \text{csort}(\text{tl}(\text{xs}), \text{CONS STR ABS})
\end{aligned}$$

$$\begin{aligned}
&\text{cinsert}(y, \text{ys}, \text{CONS STR ABS}) \\
&= 1 + \text{if null}(\text{ys}) \text{ then } 0 \\
&\quad \text{else if } y < \text{hd}(\text{ys}) \text{ then } 0 \\
&\quad \quad \text{else cinsert}(y, \text{tl}(\text{ys}), \text{cons}^{\#2}(\text{CONS STR ABS})) \\
&= 1 + \text{if null}(\text{ys}) \text{ then } 0 \\
&\quad \text{else if } y < \text{hd}(\text{ys}) \text{ then } 0 \\
&\quad \quad \text{else cinsert}(y, \text{tl}(\text{ys}), \text{ABS}) \\
&= 1
\end{aligned}$$

and so

$$\begin{aligned}
\text{csort}(\text{xs}, \text{CONS STR ABS}) &= 1 + \text{if null}(\text{xs}) \text{ then } 0 \\
&\quad \text{else } 1 + \text{csort}(\text{tl}(\text{xs}), \text{CONS STR ABS})
\end{aligned}$$

This simple recurrence has the exact solution $1 + 2 * \text{length}(\text{xs})$ and so

$$\text{cmin}(\text{xs}, \text{STR}) = 2 + 2 * \text{length}(\text{xs})$$

In this example the sufficient-time equations derive the same result, since the contexts (`CONS STR ABS`) and `STR` are very precise (*i.e.* they are the smallest safe projections). Therefore we can conclude that this is the *exact* time complexity.

5.3 Approximation and Safety

The expression $\mathcal{T}_n[[e]]\alpha$ gives a lower-bound estimate to the cost of the lazy evaluation of e in context α . For a non-strict context α the lower bound must be zero since an expression in such a context *may or may not* need to be evaluated. Proposition 5.2 below establishes this property.

PROPOSITION 5.2 *For every expression e , $\text{ABS} \sqsubseteq \alpha \sqsubseteq \text{ID} \Rightarrow \mathcal{T}_n[[e]]\alpha = 0$*

PROOF Structural induction in e . □

Safety

The necessary-cost programs enjoy better termination properties than the sufficient-cost programs, being at least as well defined as the original program. We state this property in the following way:

THEOREM 5.3 *Given mutually recursive functions f_1, \dots, f_m , defined by equations:*

$$f_i(x_1, \dots, x_{n_i}) = e_i, \quad i = 1 \dots m$$

then for all objects u_1, \dots, u_{n_i} , and contexts α

$$\alpha(f_i(u_1, \dots, u_{n_i})) \sqsupset - \Rightarrow cf_i(u_1, \dots, u_{n_i}, \alpha) \sqsupset -$$

where cf_i is defined by the equation $cf_i(x_1, \dots, x_{n_i}, \alpha) = \alpha \hookrightarrow_n \mathcal{T}_n[[e_i]]\alpha$

PROOF Omitted—a fixed point induction over the functions and cost-functions simultaneously. □

6 Higher-Order Lazy Time-Analysis

In this section we develop an extension to the techniques for lazy time analysis to incorporate higher-order functions. This is achieved by adaptation of the higher-order analysis given in section 2, illustrated with a conservative extension to the context information available for first-order functions.

6.1 Context Information

The extension of lazy-time analysis to higher-order functions also needs context information. Here we immediately run into some problems. The techniques which we have assumed so far, concerning the form and derivation of context transformers, cannot be directly extended to higher-order functions. Consider, for example, an instance of the apply function, `apply f x`, in some context α . The problem here is that there is no useful context information that can be propagated to \mathbf{x} (by any context function `apply#2`) which is *independent* of the function f .

Wray's thesis [12] shows how to handle a “second order” language (for strictness analysis) by additional parameterization of the context transformers to include the

context transformers for functional arguments. An approach to fully higher-order backwards analysis is outlined in [13]. This is based on a mixture of abstract interpretation (forwards analysis) and first-order backwards analysis. For the purposes of this section it will not be necessary to introduce these devices. Instead we will demonstrate our methods with a sufficient-time analysis using a very simple extension of the context information to higher-order functions. It is expected that the information provided by a full development of context analysis for higher-order functions could be accommodated in the time analysis we present here.

The language we use here is defined by the same grammar as that of the higher-order language in section 2.

6.2 The Projection Transformers

The method we shall describe for constructing the time equations will require the use of the same style of projection transformers that are used for the first-order analysis—for each function definition f_i we will require projection transformers $f_i^{\#k}$ such that $f_i^k \alpha \Rightarrow (f_i^{\#k} \alpha)$.

Since we are working with a higher-order language, we may expect expressions of the form

$$f_i \ e_1 \dots \ e_{n_i} \ e_{n_i+1} \dots \ e_m$$

Here the contexts propagated to expressions $e_1 \dots e_{n_i}$ are determined by the projection transformers of f_i . For a conservative estimate we know it is safe to propagate the context ID to the expressions $e_{n_i+1} \dots e_m$. In fact, the analysis we present will be able to use more precise information in this instance.

Objects of function type will also require projections to describe the context in which they are needed. A projection of a function gives a function which has less defined results on some of its arguments. For the purpose of time analysis it is sufficient to use the four-point context domain to describe the amount of evaluation of a functional argument (*i.e.* all or nothing). In an expression of the form $exp \ e$ in a context α , we can safely set the context for exp to be a mapping of α into the four-point domain for functions. For convenience we define a functional \diamond to perform this task:

$$\diamond \alpha = \begin{cases} \text{FAIL} & \text{if } \alpha = \text{FAIL} \\ \text{ABS} & \text{if } \alpha = \text{ABS} \\ \text{STR} & \text{if } \text{FAIL} \sqsubseteq \alpha \sqsubseteq \text{STR} \\ \text{ID} & \text{if } \text{ABS} \sqsubseteq \alpha \sqsubseteq \text{ID} \end{cases}$$

6.3 Accumulating Cost-Functions

As in the strict higher-order language we will define for each function in the language a cost function, constructed via two syntactic maps. The first, \mathcal{V}_L , plays the same rôle as that of \mathcal{V} in the higher-order strict language — it constructs cost-closures and makes their application explicit via an apply function \mathcal{O}_L . The second, \mathcal{T}_L , is used to define the cost-expressions. In the following we use the term *cost-expression*

to refer to objects of type $context \rightarrow cost$. The definitions of \mathcal{V}_L and \mathcal{T}_L are given in figures 12 and 13. These definitions will be explained in the following sections.

User-defined functions

For each function defined $f_i x_1 \dots x_{n_i} = e_i$ we define a sufficient-cost function to be

$$cf_i \langle x_1, c_1 \rangle \dots \langle x_{n_i}, c_{n_i} \rangle \alpha = \alpha \hookrightarrow_s 1 + \mathcal{T}_L \circ \mathcal{V}_L \llbracket e_i \rrbracket \alpha + c_1(f_i^{\#1} \alpha) + \dots + c_{n_i}(f_i^{\#n_i} \alpha)$$

In addition to the context-transformers, the cost functions require modified versions of functions themselves: $f'_i x_1 \dots x_{n_i} = \mathcal{V}_L \llbracket e_i \rrbracket$

Application and it's Cost

The cost-functions defined above now have additional parameterisation in the form of cost-expressions paired with each argument. We will explain this choice by considering the cost associated with function application $exp e$.

In the higher-order strict language, application is first translated to $exp' \ @ \ e'$ (were exp' is defined according to \mathcal{V}) and the cost of evaluation is

$$\mathcal{T} \llbracket exp' \rrbracket + \mathcal{T} \llbracket e' \rrbracket + exp' \ c@ \ e'$$

Suppose we begin by re-using \mathcal{V} , and we attempt to define (with respect to some context β) a lazy version of \mathcal{T} , \mathcal{T}_L .

In the rule for $\mathcal{T}_L \llbracket exp' \ @ \ e' \rrbracket \beta$ we must propagate the context β to the appropriate cost-expressions. We can map β into a four-point domain (overloading \diamond) to get a safe context for the function exp' . We do not know the appropriate context for e' , but we can always safely use the context ID and set

$$\mathcal{T}_L \llbracket exp' \ @ \ e' \rrbracket \beta = \mathcal{T}_L \llbracket exp' \rrbracket \diamond \beta + \mathcal{T}_L \llbracket e' \rrbracket \text{ID} + (exp' \ c@ \ e') \beta$$

Two major problems make this rule unsatisfactory.

- (i) No useful context information is propagated to e' . The information we have available is the projection transformers, but this is not used since we do not in general know which projection transformer is appropriate.
- (ii) If we have a partial application, for example if exp' is $(\text{cons}, \text{ccons}, 2)$ then e may not be evaluated at all.

We solve both of these problems by passing both the argument, *and* the cost-expression to the cost function. It is then the cost function's task to apply the appropriate context (which is determined by the projection transformers of the function) to these cost expressions—see the cost-function scheme above. We introduce new versions of $@$ and $c@$ to accommodate these requirements.

Cost-closures and the apply function

For these reasons we need to define a new version of \mathcal{V} and a different version of the function \mathfrak{C} . The “lazy” version of \mathcal{V} , \mathcal{V}_L is defined in figure 12. Because, in the rule for application, the cost-closure $\mathcal{V}_L[[exp]]$ is applied to the cost-expression $\mathcal{T}_L[[e]]$, we need a new version of the \mathfrak{C} function which satisfies:

$$(f, cf, a) \mathfrak{C}_L \langle e, ce \rangle = \begin{cases} f e & \text{if } a = 1 \\ (f e, cf \langle e, ce \rangle, a - 1) & \text{otherwise} \end{cases}$$

Note that cost-closures retain the same *function–costfunction–arity* structure.

Defining the cost-expressions

Figure 13 also defines cost-expressions via a mapping \mathcal{T}_L . A significant difference here is that we do not make the definition with respect to a particular context. This is because we wish to pass cost-expressions (functions $context \rightarrow cost$) to the cost-functions without applying them to a particular context.

To define \mathcal{T}_L we define a couple of useful functions:

- Addition of cost expressions: we use a specialised addition operator, $\diamond+$, which (for the left operand) maps the context into the four-point projection domain of the left operand: $(ce_1 \diamond+ ce_2) \alpha = (ce_1 (\diamond \alpha)) + (ce_2 \alpha)$. By allowing \diamond to be polymorphic, $\diamond+$ is associative.
- The null cost-expression: the function $\bar{0}$ gives zero-cost in any context, so $\bar{0} \alpha = 0$ for any α .

Consider the rule for application:

$$\mathcal{T}_L[[exp' \mathfrak{C}_L \langle e', ce' \rangle]] = \mathcal{T}_L[[exp']] \diamond+ (exp' \mathfrak{C}_L \langle e', ce' \rangle)$$

If we apply this expression to a context β , we get

$$\mathcal{T}_L[[exp']] \diamond \beta + (exp' \mathfrak{C}_L \langle e', ce' \rangle) \beta$$

To ensure \mathfrak{C}_L gives us a cost expression, only a small change is needed in the definition of \mathfrak{C}

$$(f, cf, n) \mathfrak{C}_L \langle e, ce \rangle = \begin{cases} cf \langle e, ce \rangle & \text{if } n = 1 \\ \bar{0} & \text{otherwise} \end{cases}$$

Primitive functions

The cost-function associated with a primitive function p_i of arity m_i is

$$cp_i \langle x_1, c_1 \rangle \dots \langle x_{m_i}, c_{m_i} \rangle \alpha = c_1(p_i \#^1 \alpha) + \dots + c_{m_i}(p_i \#^{m_i} \alpha)$$

$$\begin{aligned}
\mathcal{V}_L[\text{exp } e] &= \mathcal{V}_L[\text{exp}] \mathfrak{C}_L \langle \mathcal{V}_L[e], \mathcal{T}_L \circ \mathcal{V}_L[e] \rangle \\
\mathcal{V}_L[\text{if } e_1 \text{ then } e_2 \text{ else } e_3] &= \text{if } \mathcal{V}_L[e_1] \text{ then } \mathcal{V}_L[e_2] \text{ else } \mathcal{V}_L[e_3] \\
\mathcal{V}_L[(\text{exp})] &= (\mathcal{V}_L[\text{exp}]) \\
\mathcal{V}_L[[f_i]] &= (f'_i, cf_i, n_i) \\
\mathcal{V}_L[[p_i]] &= (p_i, cp_i, m_i) \\
\mathcal{V}_L[[c]] &= c \\
\mathcal{V}_L[[x]] &= x
\end{aligned}$$

Figure 12: The function modification map

$$\begin{aligned}
\mathcal{T}_L[\text{exp}' \mathfrak{C}_L \langle e', ce' \rangle] &= \mathcal{T}_L[\text{exp}'] \diamond_+ (\text{exp}' \mathfrak{C}_L \langle e', ce' \rangle) \\
\mathcal{T}_L[\text{if } e'_1 \text{ then } e'_2 \text{ else } e'_3] &= \mathcal{T}_L[e'_1] \diamond_+ \text{if } e'_1 \text{ then } \mathcal{T}_L[e'_2] \text{ else } \mathcal{T}_L[e'_3] \\
\mathcal{T}_L[(\text{exp}')] &= (\mathcal{T}_L[\text{exp}']) \\
\mathcal{T}_L[(p_i, cp_i, m_i)] &= \mathcal{T}_L[(f_i, cf_i, n_i)] = \bar{0} \\
\mathcal{T}_L[[c]] &= \mathcal{T}_L[[x]] = \bar{0}
\end{aligned}$$

Figure 13: The cost-function construction map

Applying the above schemes in the construction of time-equations requires that we remove (partially evaluate) unnecessary instances of \mathfrak{C}_L , and \mathfrak{C}_L , as we outlined in section 2.2. In addition we need to specialise functions to remove unnecessary parameters—this is because of the additional parameterisation involved in both modified functions, and cost-functions. The (somewhat lengthy) examples have been omitted, but it is worth noting that the process could benefit from some simple mechanical support.

7 Conclusions

We have presented a method of analysing the time complexity of a lazy higher-order functional language. The techniques for a strict higher-order language are more fully developed in [6]. We have extended of these ideas to give a treatment of lazy higher-order languages, based upon [7]: projections are used to characterise the context in which an expression is evaluated, and cost-equations are parameterised on this context-description. We have introduced two types of time-equation: *sufficient-time* equations (corresponding to the equations in [7]), and *necessary-time* equations, which together provide bounds on the exact time-complexity.

7.1 Related Work

Higher-Order Functions

Analysing the time-complexity of higher-order functions is considered by Shultis [14]. He begins with a non-standard denotational semantics which models both value and cost. A slightly less cumbersome logic is then defined for reasoning about cost by the direct manipulation of expression syntax. The logic is “tested” against the model by using an implementation of the semantics—no formal connection is provided between the logic and the model (in fact correctness could not be established without first restricting the language to typable expressions, although this is not mentioned in the paper).

A means of analysing higher-order functions, rather more in the “functional” style of the approach taken here, can be found in [15]. Le Métayer’s solution involves defining a family of cost-functions for each function in the original script, for which the i^{th} cost-function computes the cost of applying the function to its i^{th} argument (in this respect it is closer to Shultis’ approach). The syntax-directed rules for obtaining the cost functions require that cost function definitions are constructed dynamically. In addition, unlike the techniques presented here the analysis cannot handle lists of functions, or non-polymorphically typable functions.

In [16] Talcott is concerned with providing tools for reasoning about intensional properties of programs (like cost). To this end, *computation structures*, *derived properties* and *derived programs* are introduced. These simply correspond to a *proof* or derivation in an operational semantics, *properties of the proof* (computed by the step-counting semantics), and *cost-programs* respectively. This framework enables derived programs computing cost to be constructed mechanically, but only for a first-order subset of the lisp-like language used. We reason about the intensional properties of programs in a more straightforward way, without the need for any extra machinery other than the relatively familiar “symbol pushing” involved in operational semantics (see [17]).

Lazy Evaluation

A (non-compositional) means of analysing a call-by-name language is considered in [15]. Le Métayer’s solution involves transforming a call-by-name program into a strongly equivalent one with call-by-value semantics. The call-by-value program can be analysed using “strict” techniques (such as those presented in section 2). The translation, however, makes the program significantly more complex, and it not clear that the translation preserves the number of steps that are being counted in the analysis.

Bjerner’s time analysis for programs in the language of Martin-Löf type-theory [18] has relevance to the analysis of first-order lazy functional languages, and provided inspiration for Wadler’s work. His operational model of contexts, *evaluation degrees* could form an alternative basis for the work presented here. More recently, Bjerner and Holmström [19] have adapted the ideas in [18] to give a calculus for the time analysis of a first-order functional language. The equations used to describe

context are *precise*, thus specifying an *exact* time-analysis. The problem here is that the equations cannot be solved mechanically. The main correctness theorem developed (independently) in [19] (apart from the correctness of the context equations) corresponds very closely to theorem 5.3—if we view their model of context (called “demands”) as projections, we get a class of projections for which necessary and sufficient times will always be equal. Equations for this class of “exact” projections can be derived with a straightforward modification of the projection equations in [9].

7.2 Further Work

Higher-Order Context Information

The use of first-order context analysis in the analysis of a higher-order language means that, even though cost-expressions are passed as arguments so they are applied to the appropriate context, there are many cases where the contexts derived for higher-order functions are not sufficiently precise. Consider the following function definition: For satisfiable contexts α , the `apply` function (`apply f x = f x`) has the following projection transformers: `apply#1 α = ◇α`, and `apply#2 α = ID`.

Without knowing about the context of the function `apply`, the context for x is approximated by the least informative context `ID`.

The sufficient-time equation constructed with these projection transformers is

$$\text{capply } \langle f, \text{fc} \rangle \langle x, \text{xc} \rangle \alpha = \alpha \hookrightarrow_s 1 + \text{fc}(\diamond\alpha) + \text{xc ID} + (f \text{ c@}_L \langle x, \bar{0} \rangle) \alpha$$

The lack of accurate projection transformers means that the cost-expression `xc` is applied to the imprecise context `ID`—it is not difficult to construct examples where this gives an unsatisfactory time analysis. Context-analyses for higher-order languages are not well-developed. As mentioned before, Wray’s strictness analysis handles “second order” functions—projection equations can be extended to handle such functions, and the resulting context descriptions can be used by cost-functions presented here. Fully higher-order analyses still present problems for the construction of both approximate and precise context equations.

An alternative solution to this problem further utilises the technique of “passing” cost-expressions. The expression bound to `x` in the function `apply` above is evaluated in the context of the function bound to `f`, so we can pass the cost expression on to the cost function associated with `f` as follows:

$$\text{capply } \langle f, \text{fc} \rangle \langle x, \text{xc} \rangle \alpha = \alpha \hookrightarrow_s 1 + \text{fc}(\diamond\alpha) + (f \text{ c@}_L \langle x, \text{xc} \rangle) \alpha$$

To generalise this technique we must check that any parameter whose cost-expression we wish to propagate is not shared (*i.e.* it is not required in more than one context). For a sufficient-time analysis we could propagate to all contexts, while in a necessary-time analysis we could choose to propagate the cost expression to a single context. In addition we need to determine when the propagation is necessary, since unnecessary propagation (*i.e.* when the context information is sufficiently precise) decreases the compositionality of cost-functions with no additional benefit.

Acknowledgements

Thanks to Jesper Andersen, Chris Hankin, Sebastian Hunt and Daniel Le Métayer for their useful suggestions relating to earlier drafts of this paper.

References

- [1] B. Wegbreit. Mechanical program analysis. *C.ACM*, 18:528–539, September 1975.
- [2] D. LeMétayer. Mechanical analysis of program complexity. In *ACM SIGPLAN 85 Symposium*, July 1985.
- [3] M. Rosendahl. Automatic complexity analysis. In *Functional Programming Languages and computer architecture, conference proceedings*. ACM press, 1989.
- [4] T. Hickey and J. Cohen. Automating program analysis. *J. ACM*, 35:185–220, January 1988.
- [5] P. Flajolet. Mathematical methods in the analysis of algorithms and data structures. Repport 400, INRIA, Le Chesnay, France, May 1985.
- [6] D. Sands. Complexity analysis for a higher order language. Technical Report DOC 88/14, Imperial College, October 1988.
- [7] P. Wadler. Strictness analysis aids time analysis. In *15th ACM Symposium on Principals of Programming Languages*, January 1988.
- [8] G. Kahn. Natural semantics. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer Verlag, 1987. LNCS 247.
- [9] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *1987 Conference on Functional Programming and Computer Architecture*, Portland, Oregon, September 1987.
- [10] K. Davis and P. Wadler. Backwards strictness analysis: Proved and improved. In *Proceedings of Glasgow Workshop on Functional Programming*, August 1989.
- [11] G.L. Burn. A relationship between abstract interpretation and projection analysis (extended abstract). In *17th ACM Symposium on Principals of Programming Languages*, January 1990.
- [12] S. C. Wray. Programming techniques for functional languages. Technical Report 92, University of Cambridge Computer Laboratory, June 1986.
- [13] R. J. M. Hughes. Backwards analysis of functional programs. DoC Research Report CSC/87/R3, University of Glasgow, March 1987.
- [14] J. Shultis. On the complexity of higher-order programs. Technical Report CU-CS-288, University of Colorado, Febuary 1985.

- [15] D. LeMétayer. Analysis of functional programs by program transformation. In *Second France–Japan Artificial Intelligence and Computer Science Symposium*. North–Holland, 1988.
- [16] C. Talcott. Derived properties and derived programs. Technical report, Stanford, 1985.
- [17] G. D. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, Denmark, September 1981.
- [18] B. Bjerner. *Time Complexity of Programs in Type Theory*. PhD thesis, Chalmers University of Technology, 1989.
- [19] B. Bjerner and S. Holmström. A compositional approach to time analysis of first order lazy functional programs. In *Functional Programming Languages and computer architecture, conference proceedings*. ACM press, 1989.