A Naïve Time Analysis and its Theory of Cost Equivalence

–draft submitted for publication–

David SANDS [1]
DIKU, University of Copenhagen
Universitetsparken 1,
DK-2100 København Ø, DENMARK.
e-mail: `dave@diku.dk`,
tel: +45 35 321408, fax: +45 35 321401

KEYWORDS time analysis, lazy evaluation, operational semantics.

[1]Partially funded by ESPRIT BRA 6809 (Semantique) and the Danish Natural Sciences Research Council (The DART project)

Abstract

Techniques for reasoning about extensional properties of functional programs are well-understood, but methods for analysing the underlying intensional, or operational properties have been much neglected. This paper begins with the development of a simple but useful calculus for time analysis of non-strict functional programs with lazy lists.

One limitation of this basic calculus is that the ordinary equational reasoning on functional programs is not valid. In order to buy back some of these equational properties we develop a non-standard operational equivalence relation called *cost equivalence*, by considering the number of computation steps as an "observable" component of the evaluation process. We define this relation by analogy with Park's definition of bisimulation in CCS. This formulation allows us to show that cost equivalence is a contextual congruence (and thus is substitutive with respect to the basic calculus) and provides useful proof techniques for establishing cost-equivalence laws.

It is shown that basic evaluation time can be derived by demonstrating a certain form of cost equivalence, and we give a axiomatisation of cost equivalence which complete is with respect to this application. This shows that cost equivalence subsumes the basic calculus.

Finally we show how a new operational interpretation of evaluation demands can be used to provide a smooth interface between this time analysis and more compositional approaches, retaining the advantages of both.

Contents

1 Introduction

An appealing property of functional programming languages is the ease with which the *extensional* properties of a program can be understood—above all the ability to show that operations on programs preserve meaning. Prominent in the study of algorithms in general, and central to formal activities such as program transformation and parallelisation, are questions of efficiency, *i.e.* the running-time and space requirements of programs. These are *intensional* properties of a program—properties of *how* the program computes, rather that *what* it computes. The study of intensional properties is not immediately amenable to the algebraic methods with which extensional properties are so readily explored. Moreover, the declarative emphasis of functional programs, together with some of the features that afford expressive power and modularity, namely higher-order functions and lazy evaluation, serve to make intensional properties *more opaque*. In spite of this, relatively little attention has been given to the development of methods for reasoning about the computational cost of functional programs.

As a motivating example consider the following defining equations for insertion sort (written in a Haskell-like syntax)

```
isort [ ]          =  [ ]
isort (h:t)        =  insert h (isort t)

insert x [ ]       =  [x]
insert x (h:t)     =  x:(h:t)        if x ≤ h
                   =  h:(insert x t)  otherwise
```

As expected, isort requires $\mathcal{O}(n^2)$ time to sort a list of length n. However, under lazy evaluation, isort enjoys a rather nice modularity property with respect to time: if we specify a program which computes the minimum of a list of numbers, by taking the head of the sorted list,

$$\mathsf{minimum} = \mathsf{head} \circ \mathsf{isort}$$

then the time to compute minimum is only $\mathcal{O}(n)$. This rather pleasing property of insertion-sort is a well-used example in the context of reasoning about running time of lazy evaluation[1].

By contrast, the following time property of lazy "quicksort" is seldom reported. A typical definition of a functional quicksort over lists might be:

```
qsort [ ]     =  [ ]
qsort (h:t)   =  qsort (below h t) ++ (h:qsort (above h t))
```

where below and above return lists of elements from t which are no bigger, and strictly smaller than h, respectively, and ++ is infix list-append. Functional accounts of quicksort

[1]Originally due to Richard Bird, it appears as an exercise in informal reasoning about lazy evaluation in [BW88][Ch. 6], and in the majority(!) of papers on time analysis of non-strict evaluation.

are also quadratic time algorithms, but conventional wisdom would label quicksort as a better algorithm than insertion sort because of its better average-case behaviour. A rather less pleasing property of lazy evaluation is that by replacing "better" sorting algorithm qsort for isort in the definition of minimum, we obtain an asymptotically *worse* algorithm, namely one which is $\Omega(n^2)$ in the length of the input.

1.1 Overview

In the first part of the paper we consider the problem of reasoning about evaluation time in terms of a very simple measure of evaluation cost. A simple set of *time rules* are derived very directly from a call-by-name operational model, and concern equations on $\langle e\rangle^H$ (the "time" to evaluate expression e to (weak) head normal form) and $\langle e\rangle^N$ (the time to evaluate e to normal form). The approach is naïve in the sense that it is non-compositional (in general, the cost of computing an expression is not defined as a combination of the costs of computing its subexpressions), and does not model graph reduction. However, despite (or perhaps because of) its simplicity, the method appears to be useful as a means of formalising sufficiently many operational details to reason (rigorously, but not necessarily formally) about the complexity of lazy algorithms.

One of the principal limitations of the approach is the fact that the usual meanings of "equality" for programs do not provide equational reasoning in the context of the time rules. This problem motivates development of a nonstandard theory of operational equivalence in which the number of computation steps are viewed as an "observable" component of the evaluation process. We define this relation by analogy with Park's definition of bisimulation between processes. This formulation provides a uniform method for establishing cost-equivalence laws, and together with the key result that cost equivalence is a contextual congruence, provides a useful substitutive equivalence with which the time rules can be extended, since if e_1 is cost equivalent to e_2, then for any syntactic context C, $\langle C[e_1]\rangle^\alpha = \langle C[e_2]\rangle^\alpha$.

In addition we show that the theory of cost equivalence subsumes the time rules, by providing an axiomatisation of cost equivalence which is sound and complete (in a certain sense) with respect to simple evaluation time properties of expressions.

Finally, we return to a significant flaw in the time model, namely of its use of call-by-name rather than call-by-need. We sketch a method to alleviate this problem which provides a smooth integration of the simple time analysis here, and the more compositional call-by-need approaches, with some of the advantages of both.

The development of the theory of cost equivalence is somewhat technical, but the paper is written so that the reader interested primarily in the problem of time analysis of programs in a lazy language should be able to skip the bulk of the technical development, but still take advantage of its results, namely cost equivalence. In the remainder of this introduction we summarise the rest of the paper.

Sections 2 to 5 develop a simple time analysis for a first order language with lazy lists. **Section 2** gives some background describing approaches to the efficiency analysis of lazy functional programs. In **Section 3** we define our language and its operational semantics.

Section 4 defines the notion of time cost over this operational model, and introduces the time-rules which form the basis of the calculus. **Section 5** provides some examples of the use of the time-rules in reasoning about the complexity of simple programs.

Section 6 motivates and develops the theory of cost equivalence. Cost equivalence is based upon a cost simulation preordering which is shown to be preserved by substitution into arbitrary program contexts. It is also shown that it is the largest such relation. **Section 7** gives some variants of the *co-induction* proof principal which are useful for establishing cost equivalences, and presents an axiomatisation of cost equivalence which is complete with respect to the basic time properties of expressions. In **Section 8** we extend the language with higher-order functions. Time-rules are easily added to the new language, and the theory of cost-equivalence is extended in the obvious way by considering an "applicative" cost simulation, which is also shown to have the necessary substitutivity property.

Section 9 presents an example time analysis, illustrating the combined use of time-rules and cost-equivalences.

Section 10 outlines a flexible approach to increasing the compositionality and accuracy of the time analysis with respect to call-by-need evaluation, via the definition of a family of *evaluators* indexed by representations of strictness properties.

To conclude, we consider related work in the area of intensional semantics.

A preliminary version of this paper appeared as [San91b], and summarised [San90a][Ch. 4]. In addition to the inclusion of proofs and additional technical results, Sections 7, 8, 9 and 10 are new, and contain a number of important extensions to the earlier work.

2 Time Analysis: Background

A number of researchers have developed prototype (time) complexity analysis tools in which the algorithm under analysis is expressed as a first-order call-by-value functional program [Weg75,LeM88,Ros89,HC88]. It could be argued that the subject of study in these cases is not functional programming *per se*; the choice of a functional language is motivated by the fact that, for a first-order language with a call-by-value semantics, it is straightforward to mechanically construct functions with the same domain as a given function, which describe (recursively) the number of computation steps required by that function. Although this does not by any means trivialise the problem of finding solutions to these equations in terms of some size-measure of the arguments, it gives a simple but formal reading of the program as a description of computational cost. This is because the cost of evaluating some function application $\mathsf{fun}(E)$ can be understood in terms of the cost of evaluating E, plus the cost of evaluating the application of fun to the value of E.

In the case of a higher-order strict language cost is not only dependent on the simple cost of evaluating the argument E, but also on the possible cost of subsequently applying E, applying the result of an application, and so-forth. Techniques for handling this problem were introduced in [San88], where syntactic structures called *cost-closures* were introduced to enable intensional properties to be carried by functions. Additional techniques for reasoning about higher-order functions which complement this approach are described in [San90a].

A problem in reasoning about the efficiency of programs under lazy evaluation (*i.e.* call-by-name, or more usually, call-by-need, extended to data structures) is that the cost of computing the subexpression E is dependent entirely on the way in which the expression is used in the function fun. More generally, the cost of evaluating some (sub)expression is dependent on the amount of its value needed by its context.

The Compositional Approach

One approach to reasoning about the time-cost of lazy evaluation is to parameterise the description of the cost of computing an expression by a description of the amount of the result that is *needed* by the context in which it appears. This approach is due to Bjerner [Bje89], where a compositional theory for time analysis of the (primitive recursive) programs of Martin-Löf type-theory is developed. A characterisation of "need" (more accurately "not-need") provided by a new form of strictness analysis [WH87] enabled Wadler to give a simpler account of Bjerner's approach [Wad88] in the context of a (general) first-order functional language. The strictness-analysis perspective also gives a natural notion of approximation in the description of context-information, and gives rise, via *abstract interpretation* to a completely mechanisable analysis for reasoning about (approximate) contexts. In [San90b,San90a] the context information available from such an analysis is used to characterise *sufficient-time* and *necessary-time* equations which together provide bounds on the exact time-cost of lazy evaluation, and the method is extended to higher-order functions using a modification of the cost-closure technique.

A problem with these compositional approaches to time analysis remains: the information required about context is itself an uncomputable property in general. The options are to either settle for approximate information via abstract interpretation (or a related approach), or to work with a complete calculus for contexts and hope to find more exact solutions. The former approach, whilst simplifying the task of reasoning about context (assuming that an implementation is available), can lead to unacceptable approximations in time-cost. The latter approach (see [BH89]) can be impractically cumbersome for many relatively simple problems, and is unlikely to extend usefully to higher-order languages.

The Naïve Approach

In the following three sections, we explore a complementary approach which begins with a more direct operational viewpoint. We define a small first-order lazy functional language with lists, and define time-cost in terms of an operational model. (The treatment of a higher-order language in the naïve approach is just as straightforward, but is postponed in order to simplify the exposition of the theory of cost equivalence.) The simplicity of the chosen semantics (a substitution-based call-by-name model) leads to a correspondingly straightforward definition of time-cost, which is refined to give an unsophisticated calculus, in the form of *time-rules* with which we can analyse time-cost. We illustrate the utility of the naïve approach before going on to consider extensions and improvements.

3 A Simple Operational Model

We initially consider a first-order language with lists. For simplicity we present an untyped semantics, but the syntax will be suggestive of a typed version. List construction is sugared with an infix *cons* ":", and lists are examined and decomposed via a case-expression. Programs are closed expressions in the context of function definitions

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

We also assume some strict primitive functions over the atomic constants of the language (booleans, integers *etc.*). Expressions are described by the grammar in figure 1.

$$
\begin{array}{rcll}
e & ::= & f(e_1, \ldots, e_n) & \text{(function call)} \\
 & \mid & p(e_1, \ldots, e_n) & \text{(primitive function call)} \\
 & \mid & \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 & \text{(conditional)} \\
 & \mid & \left(\begin{array}{l} \textsf{case } e_1 \textsf{ of} \\ \quad \begin{array}{rcl} \textsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array} \end{array} \right) & \text{(list-case expression)} \\
 & \mid & e_1 : e_2 & \text{(cons)} \\
 & \mid & x & \text{(identifier)} \\
 & \mid & c & \text{(constant)}
\end{array}
$$

Figure 1: Expression Syntax

3.1 Semantic rules

It is possible to reason about time-complexity of a closed expression by reasoning directly about the "steps" in the evaluation of an expression. The problem with this approach is that it requires us to have the machinery of an operational semantics at our fingertips in order to reason in a formal manner. The degree of operational reasoning necessary can be minimised by an appropriately abstract choice of semantics. In particular, simplicity motivates the choice of a call-by-name calling mechanism—shortcomings and improvements to this model are discussed in section 10. The semantics is defined via *two* types of evaluation rule: one describing evaluation to head-normal-form, and one for evaluation to normal-form. Including rules for evaluation to normal-form is somewhat non-standard for the semantics of a lazy language. To talk about the complete evaluation of programs it is usual to define a *print-loop* to accurately describe the top-level behavior of a program.

Since our motivation is the analysis of time cost, the rules for evaluation to normal-form give a convenient approximation to the printing mechanism (since in the case of non-terminating programs we would need to place them in some "terminating context" to describe their time behaviour anyway).

$$\textbf{Fun} \quad \frac{e_i\{e_1/x_1 \cdots e_{n_i}/x_{n_i}\} \rightharpoonup_{\alpha} u}{f_i(e_1, \ldots, e_{n_i}) \rightharpoonup_{\alpha} u}$$

$$\textbf{Prim} \quad \frac{e_1 \rightharpoonup_{H} c_1 \ \cdots \ e_{n_k} \rightharpoonup_{H} c_k \quad (v = \mathbf{apply}_p(p_i, c_1, \ldots, c_{n_i}))}{p_i(e_1, \ldots, e_{n_k}) \rightharpoonup_{\alpha} v}$$

$$\textbf{Cond} \quad \frac{e_1 \rightharpoonup_{H} \mathsf{true} \quad e_2 \rightharpoonup_{\alpha} u}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rightharpoonup_{\alpha} u} \qquad \frac{e_1 \rightharpoonup_{H} \mathsf{false} \quad e_3 \rightharpoonup_{\alpha} u}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rightharpoonup_{\alpha} u}$$

$$\textbf{Cons} \quad \frac{e_1 \rightharpoonup_{N} v_1\ ,\ e_2 \rightharpoonup_{N} v_2}{e_1 : e_2 \rightharpoonup_{N} v_1 : v_2} \qquad \frac{}{e_1 : e_2 \rightharpoonup_{H} e_1 : e_2}$$

$$\textbf{Const} \quad \frac{}{c \rightharpoonup_{\alpha} c}$$

$$\textbf{Case} \quad \frac{e_1 \rightharpoonup_{H} \mathsf{nil} \quad e_2 \rightharpoonup_{\alpha} u}{\left(\begin{array}{rcl} \mathsf{case}\ e_1\ \mathsf{of} & & \\ \mathsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array}\right) \rightharpoonup_{\alpha} u} \qquad \frac{e_1 \rightharpoonup_{H} e_h : e_t \quad e_3\{e_h/x, e_t/xs\} \rightharpoonup_{\alpha} u}{\left(\begin{array}{rcl} \mathsf{case}\ e_1\ \mathsf{of} & & \\ \mathsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array}\right) \rightharpoonup_{\alpha} u}$$

Figure 2: Dynamic Semantics

We define the operational semantics via rules which allow us to make judgements of the form: $e \rightharpoonup_{N} v$ and $e \rightharpoonup_{H} h$. These can be read as "expression e evaluates to normal form v" and "expression e evaluates to head-normal form[2] h" respectively. There is no rule for evaluating a variable—evaluation is only defined over closed expressions. These rules are presented in figure 2, using meta-variable α to range over labels H and N. Normal-forms, ranged over by v, v_1, v_2 *etc.* (sometimes referred to simply as *values*) are the fully evaluated expressions *i.e.* either fully evaluated lists, or atomic constants (c):

$$v ::= c \mid v_1 : v_2$$

Head-Normal forms, ranged over by h, h_1, h_2 *etc.*, are simply the constants and arbitrary cons-expressions:

$$h ::= c \mid e_1 : e_2$$

[2]The usage of the term *head-normal form* is not to be confused with the corresponding notion in the (pure) lambda calculus; we use this term as a first-order manifestation of the notion of *weak* head-normal form from the terminology of lazy functional languages [Pey87].

A brief explanation of the semantic rules is given below:

- To describe function application we perform direct substitution of parameters. We use the notation $e\{e'/x\}$ to mean expression e with free occurrences of x replaced by the expression e' (see comments below).
- We assume the primitive functions are strict functions on constants, and are given meaning by some partial function $\mathbf{apply}_p$.
- To evaluate a case-expression either to normal or head-normal form, we must evaluate the list-expression e_1 to determine which branch to take. However we do not evaluate the expression any further than the first cons-node.

Notation

We summarise some of the notation used in the remainder of the paper.

Variables and Substitution A list of zero or more variables $x_1, \ldots x_n$ will often be denoted $\vec{x}$, and similarly for a list of expressions.

We use the notation $e\{e_1, \ldots, e_n/x_1, \ldots, x_n\}$ to mean expression e with free occurrences of $x_1, \ldots, x_n$ simultaneously replaced by the expressions $e_1, \ldots, e_n$. In a case expression

$$\begin{array}{rcl} \multicolumn{3}{l}{\mathsf{case}\ e_1\ \mathsf{of}} \\ \mathsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array}$$

the variables x and xs are considered bound in e_3. A formal definition of substitution is omitted, but is standard (see *e.g.* [Bar84]). We will also assume the following substitution property (the standard "substitution lemma"): if variables $\vec{x}$ and $\vec{y}$ are distinct, then

$$p\{\vec{q}/\vec{x}\}\{\vec{r}/\vec{y}\} = p\{\vec{r}/\vec{y}\}\{\vec{q}\{\vec{r}/\vec{y}\}/\vec{x}\}$$

where $\vec{q}\{\vec{r}/\vec{y}\} = q_1\{\vec{r}/\vec{y}\}, \ldots, q_n\{\vec{r}/\vec{y}\}$

The idea of a *context*, ranged over by C, C_1, *etc.* will be used (informally) to denote an expression with a "hole", $[\ ]$, in the place of a subexpression; $C[e]$ is the expression produced by replacing the hole with expression e. Generally we will assume that the expression is closed, and so this notation can be considered shorthand for substitution[3]

Relations If R is a relation, then we will usually write $a\ R\ b$ to mean $(a, b) \in R$. A relation R is a *preorder* if it is transitive and reflexive, and an *equivalence relation* if it is also symmetric. Syntactic equivalence up to renaming of bound variables will be denoted $\equiv$. The maximum relation on closed expression will be denoted by $\top$.

[3]Formal definitions usually allow free variables in e to be captured by C. We will revert to the substitution notation when we need to be more formal, and consider the special case of variable capture explicitly.

4 Deriving Time Rules

We wish to reason about the time-cost of evaluating an expression. For simplicity we express this property in terms of the number of non-primitive function calls occurring in the evaluation of the expression.

For the operational semantics given, the evaluation process is understood in terms of (the construction of) a proof of some judgement according to the semantic rules. The above property of an evaluation corresponds to the number of instances of the rule **Fun** in the proof of $e \rightarrow_\alpha u$ for some closed expression e, whenever such a proof exists for some u. In order to extract rules for reasoning about this property, we rely on some basic properties of the semantics:

- The rules describe deterministic computation: if $e \rightarrow_\alpha u$ and $e \rightarrow_\alpha u'$ then $u \equiv u'$.
- Proofs are unique: if Δ and Δ' are proofs of $e \rightarrow_\alpha u$ then Δ and Δ' are identical.

In the following let $S, S_1 \ldots$ range over judgements of the form $e \rightarrow_\alpha u$, and let $\Delta, \Delta_1 \ldots$ range over proofs of judgements.

Definition **4.1** *Let $T(\Delta)$ be the number of instances of rule* **Fun** *in a given proof* Δ.

$\Box$

Since all proofs are finite[4], assuming the inferences are labeled, we can define T inductively in the structure of the proof, according to the last rule applied:

$$T\left(\frac{\Delta_1, \ldots, \Delta_k}{S}\,\mathbf{r}\right) = \begin{cases} 1 + T(\Delta_1) + \cdots + T(\Delta_k) & \text{if } \mathbf{r} = \mathbf{Fun} \\ T(\Delta_1) + \cdots + T(\Delta_k) & \text{otherwise} \end{cases}$$

To define equations for reasoning about time we can abstract away from the structure of the proof, and express this property in terms of the structure of *expressions*, since the last rule used in the proof of some judgement S is determined largely by the expression-syntax. Using this principal we define equations for $\langle e \rangle^N$, the time to compute the normal-form, and $\langle e \rangle^H$, the time to compute head-normal-form of expression e. The rules for $\langle\rangle^N$ and $\langle\rangle^H$ are given in figure 3.

When we write $\langle e \rangle^\alpha = M$ we mean that this is provable from the time rules, together with standard arithmetic identities. Since we do not include an axiomatisation of integers and their addition, this statement is not completely formal, but will be sufficient for non-automated reasoning.

The rules are adequate in the following sense:

Proposition **4.2** *For all expressions e, if Δ is a proof of $e \rightarrow_\alpha u$, for some u, then*

$$T(\Delta) = n \iff \langle e \rangle^\alpha = n$$

[4]Proofs correspond to terminating computations; our calculus will therefore allow us only to conclude time-properties under a termination assumption. For a further discussion of this point see [San90a].

$$
\begin{array}{rcl}
\langle f_i(e_1,\ldots,e_{n_i})\rangle^\alpha & = & 1 + \langle e_i\{e_1/x_1,\ldots,e_{n_i}/x_{n_i}\}\rangle^\alpha \\[1em]
\langle p(e_1,\ldots,e_k)\rangle^\alpha & = & \langle e_1\rangle^H + \cdots + \langle e_k\rangle^H \\[1em]
\langle \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle^\alpha & = & \langle e_1\rangle^H + \left\{\begin{array}{ll} \langle e_2\rangle^\alpha & \text{if } e_1 \rightarrow_H \mathsf{true} \\ \langle e_3\rangle^\alpha & \text{if } e_1 \rightarrow_H \mathsf{false}\end{array}\right. \\[1em]
\left\langle \begin{array}{rcl} \multicolumn{3}{l}{\mathsf{case}\ e_1\ \mathsf{of}} \\ \mathsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array}\right\rangle^\alpha & = & \begin{array}{l} \langle e_1\rangle^H \\ + \left\{\begin{array}{ll} \langle e_2\rangle^\alpha & \text{if } e_1 \rightarrow_H \mathsf{nil} \\ \langle e_3\{e_h/x, e_t/xs\}\rangle^\alpha & \text{if } e_1 \rightarrow_H e_h : e_t \end{array}\right.\end{array} \\[1em]
\langle e_1 : e_2\rangle^N & = & \langle e_1\rangle^N + \langle e_2\rangle^N \\
\langle e_1 : e_2\rangle^H & = & \langle c\rangle^\alpha \;=\; 0
\end{array}
$$

Figure 3: Time Rules

The proof of this proposition is a straightforward induction in the structure of Δ. Notice that the premise of the proposition makes a termination assumption about the evaluation of e. In this sense the time rules are partially correct. We could then refine this correctness statement further by treating run-time errors separately from nontermination.

5 Direct Time Analysis

The time rules in figure 3, although simple, are sufficient to reason directly about the cost of evaluating closed expressions in this language. We illustrate the utility of this approach with some small examples. The examples are chosen to emphasize features of non-strict evaluation, rather than to present interesting asymptotic analyses, for which the reader may consult a standard text on the analysis of algorithms (*e.g.* [Knu68,AHU74]) or associated mathematical techniques (*e.g.* [GKP89]).

To reason about complexity we consider expressions containing some non-specified *input value* (*i.e.* normal form), which we will denote by a (meta-)variable (written in an italic font). We sometimes also allow meta-variables to range over arbitrary expressions, although usually this is more awkward since the calculus is not compositional.

5.1 Example

Consider the functions over lists given in figure 4.

Now we wish to consider the cost of evaluating the expression

$$\mathsf{head}(\mathsf{reverse}(v))$$

```
append(xs,ys)  =  case  xs of
                        nil ⇒ys
                        h:t ⇒h:append(t,ys)
reverse(xs)    =  case  xs of
                        nil ⇒nil
                        h:t ⇒append(reverse(t), h:nil)
head(xs)       =  case  xs of
                        nil ⇒undefined
                        h:t ⇒h
```

Figure 4: Some list-manipulating functions

which computes the last element of some non-empty list-value $v \equiv v_h : v_t$. Applying the definitions in figure 3

$$\langle \mathsf{head}(\mathsf{reverse}(v)) \rangle^N = 1 + \langle \mathsf{reverse}(v) \rangle^H + \langle e_h \rangle^N$$

where $\mathsf{reverse}(v) \rightarrow_H e_h : e_t$ for some e_h, e_t. It is not hard to show that e_h is a value (using the fact that v is, and by induction on its length), and hence that $\langle e_h \rangle^N = 0$. Now since $v \equiv v_h : v_t$, and hence $v \rightarrow_H v_h : v_t$ we have that

$$\begin{array}{ll}
\multicolumn{2}{l}{\langle \mathsf{reverse}(v) \rangle^H} \\
= & 1 + \langle v \rangle^H + \langle \mathsf{append}(\mathsf{reverse}(v_t),v_h\mathsf{:nil}) \rangle^H \\
= & 1 + \langle \mathsf{append}(\mathsf{reverse}(v_t),v_h\mathsf{:nil}) \rangle^H \\
= & 1 + 1 + \langle \mathsf{reverse}(v_t) \rangle^H \\
 & + \left\{ \begin{array}{l} \langle \mathsf{nil} \rangle^H, \text{if } \mathsf{reverse}(v_t) \rightarrow_H \mathsf{nil} \\ \langle e'_h : \mathsf{append}(e'_t,v_h\mathsf{:nil}) \rangle^H, \\ \text{if } \mathsf{reverse}(v_t) \rightarrow_H e'_h : e'_t \end{array} \right. \\
 & \\
= & 2 + \langle \mathsf{reverse}(v_t) \rangle^H
\end{array}$$

We now have the recurrence equations parameterised by the input value:

$$\begin{array}{lcl}
\langle \mathsf{reverse}(\mathsf{nil}) \rangle^H & = & 1 \\
\langle \mathsf{reverse}(v : vs) \rangle^H & = & 2 + \langle \mathsf{reverse}(vs) \rangle^H
\end{array}$$

whose solution is $\langle \mathsf{reverse}(v) \rangle^H = 1 + 2n$, where n is the length of the list v. Thus we have a total cost of

$$\langle \mathsf{head}(\mathsf{reverse}(v)) \rangle^N = 2(1 + n)$$

where n is the length of the list v, *i.e.* linear time complexity (compare with quadratic complexity for the call-by-value reading, and for $\langle \mathsf{reverse}(v) \rangle^N$).

5.2 Example

This example, originally due to Richard Bird, is well-used in the context of reasoning about lazy evaluation time, since it Here we present the example from the introduction (in the syntax we have defined) which shows the rather pleasing property that one can compute the smallest element of a list in linear time by taking the first element of the (insertion-) sort of the list. The equations in figure 5 define an insertion-sort function (isort).

```
isort(xs)       =  case  xs of
                         nil ⇒nil
                         h:t ⇒insert(h, isort(t))

insert(x, ys)   =  case  ys of
                         nil ⇒x:nil
                         h:t ⇒if x ≤ h
                                then x:(h:t)
                                else h:insert(x,t)
```

Figure 5: Insertion Sort

The time to compute the head-normal form of insertion sort given some list-value (normal form) v is easily calculated from the time rules. First consider the insertion function. Any exhaustive application of the time rules together with a few minor simplifications allow us to conclude that for integer valued expressions e_1, and integer-list valued expressions e_2,

$$\langle \mathsf{insert}(e_1,e_2)\rangle^H = 1 + \langle e_2\rangle^H \\ + \begin{cases} 0 & \text{if } e_2 \to_H \mathsf{nil} \\ \langle e_1\rangle^H + \langle h\rangle^H & \text{if } b \to_H h : t \end{cases}$$

Now consider computing the head normal form of insertion sort applied to some list of integers $v_n : \ldots : v_1 : \mathsf{nil}$ where $n \geq 1$. To aid notation, let V_0 denote the list nil and, for each $i < n$, let V_{i+1} denote the list $v_{i+1} : V_i$.

$$\langle \mathsf{isort}(V_0)\rangle^H = 1$$

$$\begin{aligned} &\langle \mathsf{isort}(V_{i+i})\rangle^H \\ &= 1 + \langle \mathsf{insert}(v_{i+1},\mathsf{isort}(V_i))\rangle^H \\ &= 1 + 1 + \langle \mathsf{isort}(V_i)\rangle^H \\ &+ \begin{cases} 0 & \text{if } \mathsf{isort}(V_i) \to_H \mathsf{nil} \\ \langle v_{i+1}\rangle^H + \langle h\rangle^H & \text{if } \mathsf{isort}(V_i) \to_H h : t \end{cases} \end{aligned}$$

Clearly $\langle v_{i+1}\rangle^{H} = 0$. A simple induction in i establishes that if $\mathsf{isort}(V_i) \rightarrow_{H} h : t$ then $\langle h\rangle^{H} = 0$ also. This leaves us with the simple recurrence

$$\begin{aligned} \langle \mathsf{isort}(V_0)\rangle^{H} &= 1 \\ \langle \mathsf{isort}(V_{i+i})\rangle^{H} &= 2 + \langle \mathsf{isort}(V_i)\rangle^{H} \end{aligned}$$

Giving $\langle \mathsf{isort}(V_n)\rangle^{H} = 2n+1$.

5.3 Example

Consider the following (somewhat non-standard[5]) definition of Fibonacci:

$$\begin{aligned} \mathsf{fib(n)} &= \mathsf{f(n,0)} \\ \mathsf{f(n,\ r)} &= \mathsf{if\ n{=}0\ then\ 1} \\ &\ \mathsf{else\ r + f(n\text{-}1,\ f(n\text{-}2,0))} \end{aligned}$$

Consider the time to compute an instance of fib:

$$\begin{aligned} &\langle \mathsf{fib}(k)\rangle^{N} = 1 + \langle \mathsf{f}(k,\ 0)\rangle^{N} \\ &\langle \mathsf{f}(k,\ 0)\rangle^{N} \\ &= \ 1 + \left\langle \begin{array}{l} \mathsf{if}\ k{=}0\ \mathsf{then}\ 1 \\ \mathsf{else}\ 0 + \mathsf{f}(k\text{-}1,\ \mathsf{f}(k\text{-}2,0)) \end{array} \right\rangle^{N} \\ &= \ 1 + \left\{ \begin{array}{ll} \langle 1\rangle^{N} & \text{if } k{=}0 \rightarrow_{N} \mathsf{true} \\ \langle 0 + \mathsf{f}(k\text{-}1,\ \mathsf{f}(k\text{-}2,0))\rangle^{N} & \text{if } k{=}0 \rightarrow_{N} \mathsf{false} \end{array} \right. \\ &= \ 1 + \left\{ \begin{array}{ll} 0 & \text{if } k{=}0 \rightarrow_{N} \mathsf{true} \\ \langle \mathsf{f}(k\text{-}1,\ \mathsf{f}(k\text{-}2,0))\rangle^{N} & \text{if } k{=}0 \rightarrow_{N} \mathsf{false} \end{array} \right. \end{aligned}$$

Instantiating k we have

$$\begin{aligned} \langle \mathsf{f}(0,\ 0)\rangle^{N} &= 1 && (1) \\ \langle \mathsf{f}(1,\ 0)\rangle^{N} &= 1 + \langle \mathsf{f}(0,\ \mathsf{f}(1\text{-}2,0))\rangle^{N} \\ &= 2 && (2) \\ \langle \mathsf{f}(k+2,\ 0)\rangle^{N} &= 1 + \langle \mathsf{f}(k+1,\ \mathsf{f}(k,0))\rangle^{N} \\ &= 1 + 1 + \langle \mathsf{f}(k,0)\rangle^{N} && (3) \\ &\quad + \langle \mathsf{f}(k,\ \mathsf{f}(k-1,0))\rangle^{N} && (4) \end{aligned}$$

Now $\langle \mathsf{f}(k+1,\ 0)\rangle^{N} = 1 + \langle \mathsf{f}(k,\ \mathsf{f}(k-1,0))\rangle^{N}$, so

$$\langle \mathsf{f}(k+2,\ 0)\rangle^{N} = 1 \begin{array}{ll} + & \langle \mathsf{f}(k,0)\rangle^{N} \\ + & \langle \mathsf{f}(k+1,\ 0)\rangle^{N} \end{array} \qquad (5)$$

Equations 1, 2 and 5 give a linear recurrence-relation that can be solved (exactly) using standard techniques (see *e.g.*, [GKP89]); the asymptote is

$$\langle \mathsf{f}(k,\ 0)\rangle^{N} = \Theta(k^{(1+\sqrt{5})/2})$$

[5]Note that under a call-by-value semantics fib is divergent for any $\mathsf{n} > 0$.

6 A Theory of Cost Equivalence

The previous example subtly illustrates some potential problems in reasoning about cost using the equations for $\langle\cdot\rangle^H$ and $\langle\cdot\rangle^N$. The use of the time rules in the previous examples follows a simple pattern of case analysis (instantiation) and simplification, leading to the the construction of a recurrence by a simple syntactic matching. In the simplification process, it is tempting to make simplifications which are not directly justifiable from the time rules.

The potential problems stem from the fact that if we know that two expressions are extensionally equivalent, $e_1 = e_2$, it is clearly *not* the case that $\langle e_1\rangle^N = \langle e_2\rangle^N$ in general since we expect, with any reasonable definition of extensional equivalence, that an expression and it's normal-form (supposing one exists) will be equivalent. More generally given any context C, we expect that $C[e_1] = C[e_2]$, but **not** that $\langle C[e_1]\rangle^\alpha = \langle C[e_2]\rangle^\alpha$ in general, so ordinary equational reasoning is not valid within the time-rules. Simiarly if $\langle e_1\rangle^H = \langle e_2\rangle^H$ then we cannot expect in general that $\langle C[e_1]\rangle^H = \langle C[e_2]\rangle^H$.

However, even in the last example above we *have* used simple equalities such as (line 4) $(k+1)$ - $1 = k$ in precisely this way (albeit benignly) to simplify cost-expressions in order to construct a recurrence.[6] In this instance the simplification is obviously correct, but with the current calculus we cannot justify it.

This section is devoted to developing a stronger notion of equivalence of expressions which respects cost, and allows a richer form of equational reasoning on expressions within the calculus.

What is needed is an appropriate characterisation of (the weakest) equivalence relation $=_{\langle\rangle}$ which satisfies

$$e =_{\langle\rangle} e' \Longrightarrow \langle C[e]\rangle^\alpha = \langle C[e']\rangle^\alpha$$

To develop this general "contextual congruence" relation, we use a notion of *simulation* similar to the various simulations developed in process algebras such as Milner's calculus of communicating systems [Mil83]. In the theory of concurrency a central idea is that processes that cannot be distinguished by observation should be identified. This "observational" viewpoint is adopted in the "lazy" λ-calculus [Abr90], where an equivalence called *applicative bisimulation* is introduced. In the lazy λ-calculus, the observable properties are just the convergence of untyped lambda-terms. For our purposes we need to treat cost as an observable component of the evaluation process, and so we develop a suitable notion of *cost-(bi)simulation*.

6.1 Cost-Simulation

The partial functions $\rightarrow_H$ and $\rightarrow_N$ together with $\langle\rangle^N$ and $\langle\rangle^H$ are not sufficient to completely characterise the cost-behaviour of expressions in all contexts, since we need to

[6]Spelling this simplification out, we have an application of a primitive function (subtraction) to the (meta) constant "$k+1$" (*i.e.*, the constant one larger than the meta-constant k), which we simplify to "k".

characterise possibly infinite "observations" on expressions which arise in our language because of the non-strict list-constructor (*c.f.* untyped weak head-normal forms in [Abr90]).

Roughly speaking, the notion of equivalence we want satisfies:

> *e and e' are equivalent iff $\langle e\rangle^H = \langle e'\rangle^H$ and their head-normal-forms are either identical constants, or they are cons-expressions whose corresponding components are equivalent.*

Unfortunately, although this is a property that we would like our equivalence to obey, it does not constitute a definition (to see why, note that we not only wish to relate expressions having normal-forms, but also those which are "infinite"), so following [Mil83] we use a technique due to Park [Par80] for identifying processes—the notion of a *bisimulation* and its related proof technique. We will develop the equivalence relation we require in terms of preorders called *cost-simulations*—we will then say that two expressions are cost-equivalent if they simulate each other.

To simplify our presentation we add some notation:

Definition **6.1** *If $\mathcal{R}$ is a binary relation on closed expressions, then $\mathcal{R}^{:}$ is the binary relation on head-normal-forms such that*

$$(h\,\mathcal{R}^{:}\,h') \iff \begin{array}{ll} \textit{either} & h = h' = c \\ \textit{or} & h = e_1 : e_2,\ h' = e'_1 : e'_2 \textit{ and} \\ & e_1\,\mathcal{R}\,e'_1,\ \textit{and } e_2\,\mathcal{R}\,e'_2 \end{array}$$

□

Definition **6.2** *The cost-labeled transition, $\xrightarrow{t}_{\alpha}$, $t \in \mathbf{N}$ is defined*

$$e \xrightarrow{t}_{\alpha} u \overset{\text{def.}}{=} e \rightarrow_{\alpha} u \textit{ and } \langle e\rangle^{\alpha} = t$$

□

Now we define a basic notion of cost-simulation, by analogy with Park's (bi)simulation:

Definition **6.3 (Cost-Simulation)**
A binary relation $\mathcal{R}$ on closed expressions is a **cost-simulation** *if, whenever $e\,\mathcal{R}\,e'$*

$$e \xrightarrow{t}_{H} h \implies (\ e' \xrightarrow{t}_{H} h' \textit{ and } h\,\mathcal{R}^{:}\,h')$$

□

Definition **6.4** *For each relation Q on closed expressions, define $\mathcal{F}(Q)$ to be the relation on closed expressions that relates e and e' exactly when*

$$e \xrightarrow{t}_{H} h \implies (\ e' \xrightarrow{t}_{H} h' \ \textit{ and } h\,Q^{:}\,h')$$

□

Now we can easily see that

- $\mathcal{F}$ is monotonic, *i.e.*, $\mathcal{R} \subseteq \mathcal{S} \Longrightarrow \mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{S})$
- $\mathcal{S}$ is a cost-simulation iff $\mathcal{S} \subseteq \mathcal{F}(\mathcal{S})$

Definition **6.5** *Let $\preceq$ denote the maximum cost-simulation*

$$\bigcup\{\mathcal{S} : \mathcal{S} \subseteq \mathcal{F}(\mathcal{S})\}$$

□

Proposition **6.6** *$\preceq$ is the maximal fixed-point of $\mathcal{F}$.*

proof Follows from the Knaster-Tarski fixed point theorem, by the fact that $\mathcal{F}$ is a monotone function on a complete lattice. □

With these results we have the following useful proof technique: to show that $e \preceq e'$ it is necessary and sufficient to exhibit *any* cost-simulation containing (*i.e.* relating) the pair (e, e'). This technique will be illustrated later in the proof that $\preceq$ is a precongruence.

6.2 Expressing $\rightarrow_N$ in terms of $\rightarrow_H$

The above definition of cost-simulation is described in terms of evaluation to head-normal-form only. For this to be sufficient to describe properties of evaluation to normal-form we need some properties relating $\rightarrow_N$ and $\rightarrow_H$. The following property allows us to factor evaluation to normal form through evaluation to head normal form, whilst preserving cost behaviour:

Proposition **6.7** *For all closed e, e'*

- $e \rightarrow_N c \iff e \rightarrow_H c$
- $e \xrightarrow{t}_N v \iff e \xrightarrow{t_1}_H h$ *and* $h \xrightarrow{t_2}_N v$ *and* $t_1 + t_2 = t$ *for some* h.

And finally we have

Lemma **6.8** *If $e \preceq e'$ then if $e \xrightarrow{t}_N u$ then $e' \xrightarrow{t}_N u$*

The proofs are outlined in Appendix A

Remark The (first) implication in the lemma cannot be reversed. For example, if I is an identity function, then the expressions I(nil):nil and nil:I(nil) take the same time to reach identical normal forms but are not cost-simulation comparable.

6.3 Precongruence

Now we are ready to prove the key property that we demand of cost-simulation: cost-simulation is a precongruence, *i.e.* it is substitutive preordering (the fact that $\preceq$ is a preorder is easily established).

Some notation: for convenience we abbreviate some indexed family of expressions $\{e_j : j \in J\}$ by $\tilde{e}$. Similarly we will abbreviate the substitution $\{e_j/x_j : j \in J\}$ by $\{\tilde{e}/\tilde{x}\}$, and when, for all $j \in J$, $(e_j \; Q \; e'_j)$ for some relation Q, we write $(\tilde{e} \; Q \; \tilde{e}')$.

Definition **6.9** *$\Re$ is defined to be the relation*

$$\Re = \{(e\{\tilde{e}/\tilde{x}\}, e\{\tilde{e}'/\tilde{x}\}) \mid \tilde{x} \subseteq FV(e), \tilde{e} \preceq \tilde{e}'\}$$

□

Lemma **6.10** *$\Re$ is a cost-simulation.*

proof Assume that $\tilde{e} \preceq \tilde{e}'$, for some closed expressions $\tilde{e}$, $\tilde{e}'$. Abbreviate substitutions $\{\tilde{e}/\tilde{x}\}$ and $\{\tilde{e}'/\tilde{x}\}$ by σ and σ' respectively, and assume that e is any expression containing at most variables $\tilde{x}$. Assume that $e\sigma \xrightarrow{t}_H h$. The lemma now requires us to prove that $e\sigma' \xrightarrow{t}_H h'$ for some h' such that $h \; \Re^{\bullet} \; h'$. We establish this by induction on the structure of the proof of $e\sigma \rightarrow_H h$, and by cases according to the structure of expression e. We give a couple of illustrative cases:

$\boxed{e \equiv x}$ Observe that $\preceq$ is contained in $\Re$, and the result follows.

$\boxed{e \equiv f(e_1 \ldots e_n)}$
Assume that f is defined by $f(y_1 \ldots y_n) = e_f$. Since $f(e_1 \ldots e_n)\sigma \equiv f(e_1\sigma \ldots e_n\sigma)$, the last rule in the above inference must be an instance of **Fun**, and so we must have $e_f\{e_1\sigma/y_1 \cdots e_n\sigma/y_n\} \xrightarrow{t-1}_H h$. We can take variables in $\tilde{x}$ to be distinct from $y_1 \ldots y_n$, and so

$$e_f\{e_1\sigma/y_1 \cdots e_n\sigma/y_n\} \equiv (e_f\{e_1/y_1 \cdots e_n/y_n\})\sigma.$$

Now since $(e_f\{e_1/y_1 \cdots e_n/y_n\})\sigma \xrightarrow{t-1}_H h$ by a smaller proof, the inductive hypothesis gives

$$(e_f\{e_1/y_1 \cdots e_n/y_n\})\sigma' \xrightarrow{t-1}_H h' \text{ where } h \; \Re^{\bullet} \; h'.$$

So by rule **Fun**, together with definition 4.1 we can conclude that

$$f(e_1\sigma', \ldots, e_n\sigma') \xrightarrow{t}_H h'$$

with $h \; \Re^{\bullet} \; h'$ as required. □

Theorem **6.11 (Precongruence)** *If $\tilde{e} \preceq \tilde{e}'$ for some commonly indexed families of closed expressions $\tilde{e}, \tilde{e}'$, then for all expressions e containing at most variables $\tilde{x}$*

$$e\{\tilde{e}/\tilde{x}\} \preceq e\{\tilde{e}'/\tilde{x}\}$$

proof The relation $\Re$, given above, is such that $(e\{\tilde{e}/\tilde{x}\}, e\{\tilde{e}'/\tilde{x}\}) \in \Re$ whenever $\tilde{e} \preceq \tilde{e}'$ and e contains at most variables $\tilde{x}$. Lemma 6.10 (above) establishes that $\Re$ is a cost-simulation, *i.e.* that $\Re \subseteq \preceq$, and so we must also have $(e\{\tilde{e}/\tilde{x}\}, e\{\tilde{e}'/\tilde{x}\}) \in \preceq$. □

Although in this case we can see that $\Re$ is identically $\preceq$, we express the proof in this way since it illustrates a general method for establishing cost simulations. Now we can define our notion of cost-equivalence to be the equivalence relation:

DEFINITION **6.12 (cost-equivalence)** $(=_{\langle\rangle}) \equiv (\preceq \cap \preceq^{-1})$, *i.e.*

$$(e_1 =_{\langle\rangle} e_2) \iff (e_1 \preceq e_2) \,\&\, (e_2 \preceq e_1)$$

□

So two expressions are *cost-equivalent* if they cost-simulate each other. Now we have as the main corollary of precongruence

COROLLARY **6.13** *For all contexts C, and closed expressions e and e', if $e =_{\langle\rangle} e'$, then*

$$\langle C[e] \rangle^{\alpha} = \langle C[e'] \rangle^{\alpha}$$

whenever $C[e]\downarrow_{\alpha}$

PROOF $\alpha = H$, immediate from theorem 6.11 and the definition of $\preceq$, viewing a context as an expression containing a single free variable; $\alpha = N$, immediate from theorem 6.11 and lemma 6.8 □

Open Expressions In the obvious way we can extend cost simulation to open expressions by saying that $e \preceq e'$ if for all closing substitutions σ, $e\sigma \preceq e\sigma'$. As a consequence we can show that

$$e \preceq e' \,\&\, e_1 \preceq e_2 \Rightarrow e\{x/e_1\} \preceq e'\{x/e_2\}$$

We can extend the congruence property to open expressions (where free variables may be captured by the context) by showing that, for any expressions e_1 and e_2 containing at most free variables x and xs, and for any closed expressions e and e' such that $e_1 \preceq e_2$

$$\left(\begin{array}{lcl} \mathsf{case}\ e\ \mathsf{of} & & \\ \qquad\quad \mathsf{nil} & \Rightarrow & e' \\ \quad x : xs & \Rightarrow & e_1 \end{array}\right) \preceq \left(\begin{array}{lcl} \mathsf{case}\ e\ \mathsf{of} & & \\ \qquad\quad \mathsf{nil} & \Rightarrow & e' \\ \quad x : xs & \Rightarrow & e_2 \end{array}\right).$$

Open Endedness A statement of cost equivalence involving some function symbol f naturally assumes a particular defining equation, so strictly speaking, cost equivalence should be parameterised by a set of function definitions. The semantic rule for function application is really a rule schema, but in the proof that cost simulation is a precongruence, it is not necessary to assume a particular set of definitions. As a result, adding a new function definition (*i.e.*, a defining equation for a new function name) does not invalidate earlier cost equivalences; furthermore, the maximality results of the next section imply that such an extension of the language must be conservative with respect to cost equivalence.

6.4 Cost simulation as the largest contextual cost congruence

We have shown that cost simulation is a precongruence, which was sufficient for it to be substitutive with respect to the time rules. A remaining question is whether it is the largest possible precongruence with respect to the time rules. *i.e.* Are there expression pairs e, e' such that in all contexts C, $\langle C[e]\rangle^\alpha = \langle C[e]\rangle^\alpha$ but for which $e \not\precsim e'$?

In this section we outline the result that, under a mild condition on the constructs of the language (outlined below), $\precsim$ is indeed the largest such relation, *i.e.*

$$(\forall C.\ C[e]{\downarrow_\alpha} \Rightarrow \langle C[e]\rangle^\alpha = \langle C[e']\rangle^\alpha) \Rightarrow e \precsim e'.$$

Roughly speaking, we say that two expressions e, e' are *cost-distinguishable* whenever there exists a context C such that $\langle C[e]\rangle^\alpha \neq \langle C[e']\rangle^\alpha$. A necessary (and, as we will show, sufficient) condition for the above implication to hold is that every pair of distinct constants in the language are cost-distinguishable. We refer to this as the *CD condition*. The CD condition is not a particularly strong one since it is satisfied if, for example, we assume a primitive function providing an equality test over the constants.

We prove the above result by exploring the relationship between cost simulation and various contextual congruences. We summarise these results below, and refer the reader to Appendix B for more details.

- We define a cost congruence preorder $\leq_c$ between closed expressions such that $e_1 \leq_c e_2$ if and only if for all contexts C, $C[e_1]\ \mathcal{F}(\top)\ C[e_2]$. *i.e.* if the results of evaluation to head-normal form are produced in the same number of steps, and the results have the same "outermost" form (see definition 6.4).
- We show that $\precsim = \leq_c$.
- We define a pure cost congruence preorder $\leq_{pc}$ which does not take into account the actual head-normal forms produced: $e_1 \leq_{pc} e_2$ if and only if for all contexts C, whenever $C[e_1] \xrightarrow{t}_H u_1$ then there exists u_2 such that $C[e_2] \xrightarrow{t}_H u_2$.
- Assuming the CD condition, we show that $\leq_c = \leq_{pc}$. As a corollary, we have that $(\forall C.\ C[e_1]{\downarrow_H} \Rightarrow \langle C[e_1]\rangle^H = \langle C[e_2]\rangle^H) \Rightarrow e \precsim e'$. The extension of this to include evaluation to normal-form is straightforward.

7 Proof Principles and an Axiomatisation of Cost Equivalence

The definition of cost simulation comes with a useful proof technique for establishing instances. In the first part of this section we outline some simple variations of this technique.

In the second part of this section we show that cost equivalence subsumes the time rules (*i.e.*can be viewed as the basis of a time calculus independently) by giving a complete axiomatisation of cost equivalence with respect to the cost-labeled transition $\xrightarrow{t}_H$.

7.1 Co-induction Principles

We motivated the theory of cost equivalence with a need for substitutive laws (*i.e.* cost-equivalence schemas) with which to augment the time-rules. Some example laws are given below:

PROPOSITION **7.1**

$$\begin{array}{lrcl}
(i) & p(c_1,\ldots,c_n) & =_{\langle\rangle} & c \textit{ if } \mathbf{apply}(p, c1, \ldots, c_n) = c \\
(ii) & (e_1+e_2)+e_3 & =_{\langle\rangle} & e_1+(e_2+e_3) \\
(iii) & \multicolumn{3}{l}{\mathsf{if}\ (\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4} \\
 & & =_{\langle\rangle} & \\
\multicolumn{4}{l}{\quad \mathsf{if}\ e_0\ \mathsf{then}\ (\mathsf{if}\ e_1\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4)} \\
\multicolumn{4}{l}{\quad \mathsf{else}\ (\mathsf{if}\ e_2\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4)}
\end{array}$$

The proof of theorem 6.11 illustrates a general technique for establishing cost-equivalence laws such as the above, where we construct a suitable relation (*i.e.* containing all instances of the law), and show that it is a cost simulation. Recall from the previous section the functional $\mathcal{F}(\_)$: For each relation Q on closed expressions,

$$e\ \mathcal{F}(Q)\ e' \iff e \xrightarrow{t}_{H} h \implies (\ e' \xrightarrow{t}_{H} h' \ \ \textit{and}\ h\ Q^{\cdot}\ h')$$

The definition of $\preceq$ as the maximal fixed point of $\mathcal{F}(\_)$ comes with the following useful proof technique, which following [MT91b] we call *Co-induction*:

> To show that $e \preceq e'$ it is necessary and sufficient to exhibit *any* relation R containing (*i.e.* relating) the pair (e, e') and such that R is a cost simulation (*i.e.* $R \subseteq \mathcal{F}(R)$).

Some minor variations of this technique also turn out to be useful.

PROPOSITION **7.2** *To prove R is a cost simulation, it is sufficient to prove either of the following conditions (*cost simulation modulo S*, and* cost simulation up to cost equivalence *respectively):*

(i) $R \subseteq \mathcal{F}(R \cup S)$ *for some cost simulation S.*

(ii) $R \subseteq \mathcal{F}(=_{\langle\rangle}; R; =_{\langle\rangle})$.

PROOF

(i) $R \subseteq \mathcal{F}(R \cup S)$ implies

$$\begin{array}{rcll}
R \cup \preceq & \subseteq & \mathcal{F}(R \cup S) \cup \preceq & \\
 & = & \mathcal{F}(R \cup S) \cup \mathcal{F}(\preceq) & (\preceq \text{ is a f. p.}) \\
 & \subseteq & \mathcal{F}(R \cup S \cup \preceq) & (\text{monotonicity}) \\
 & = & \mathcal{F}(R \cup \preceq) & (S \subseteq \preceq)
\end{array}$$

which implies that $(R \cup \preceq) \subseteq \preceq$ and hence $R \subseteq \preceq$.

(ii) For arbitrary relations A and B it is not hard to show that $\mathcal{F}(A);\mathcal{F}(B)\subseteq\mathcal{F}(A;B)$. Using the fact that $=_{\langle\rangle}$ is transitive, and a fixed point of $\mathcal{F}(\_)$ we can show that

$$(=_{\langle\rangle};\mathcal{F}(=_{\langle\rangle};R;=_{\langle\rangle});=_{\langle\rangle})\subseteq\mathcal{F}(=_{\langle\rangle};R;=_{\langle\rangle})$$

Now $R\subseteq\mathcal{F}(=_{\langle\rangle};R;=_{\langle\rangle})$ implies

$$\begin{array}{rcl} =_{\langle\rangle};R;=_{\langle\rangle} & \subseteq & =_{\langle\rangle};\mathcal{F}(=_{\langle\rangle};R;=_{\langle\rangle});=_{\langle\rangle} \\ & \subseteq & \mathcal{F}(=_{\langle\rangle};R;=_{\langle\rangle}) \end{array}$$

Hence $(=_{\langle\rangle};R;=_{\langle\rangle})\subseteq\preceq$, and since $=_{\langle\rangle}$ is greater than the identity relation, $R\subseteq\preceq$.

□

Method (i) is typically used with S taken to be the relation of syntactic equivalence. Method (ii) can be viewed as a "semantic" co-induction principle. For example, part (iii) of Proposition 7.1 is proved by showing that the relation containing all instances is a cost simulation modulo syntactic equivalence. This goes through by a simple case analysis on the possible outcomes of the conditionals, and is left as an exercise.

7.2 An Axiomatisation of Cost Equivalence

The language is too expressive to expect a complete set of cost equivalence laws. However we can give a set which is complete with respect to the time rules in a sense that we will make precise below.

The key to this axiomatisation is the use of an identity function to represent a single "tick" of computation time.

Definition 7.3 *Let I be an identity function given by a program definition $\mathsf{I}(\mathsf{x}) = \mathsf{x}$. For any integer $n \geq 0$, write $\mathsf{I}^n(exp)$ for the expression given by n applications of the function I to exp:*

$$\underbrace{\mathsf{I}(\cdots\mathsf{I}(}_{n}\ exp)\cdots).$$

We will write $\mathsf{I}^1(exp)$ as simply $\mathsf{I}(exp)$.

□

In figure 6 we state a set K of cost equivalence laws.

We write

$$\vdash_K e_1 =_{\langle\rangle} e_2$$

if $e_1 =_{\langle\rangle} e_2$ is provable from the cost equivalence laws K together with the facts that $=_{\langle\rangle}$ is a congruence relation (*i.e.* reflexivity, transitivity and substitutivity rules) and the following tick-elimination rule:

$$\textbf{I-elim}\ \ \frac{\mathsf{I}(e_1) =_{\langle\rangle} \mathsf{I}(e_2)}{e_1 =_{\langle\rangle} e_2}\ .$$

Now we have the following soundness and completeness results for the system $\vdash_K$, with respect to the cost-labeled transition relation $\xrightarrow{t}_H$:

$$
\begin{array}{lrcl}
\textbf{Fun.}\mathsf{I} & f_i(e_1,\ldots,e_{n_i}) & =_{\langle\rangle} & \mathsf{I}(e_i\{e_1/x_1\cdots e_{n_i}/x_{n_i}\}) \\[1em]
\textbf{Prim.}\mathsf{I} & p_i(e_1,\ldots,\mathsf{I}(e_j),\ldots,e_{n_k}) & =_{\langle\rangle} & \mathsf{I}(p_i(e_1,\ldots,e_j,\ldots,e_{n_k})) \\[1em]
\textbf{Prim} & p_i(c_1,\ldots,c_{n_k}) & =_{\langle\rangle} & v \;\; \mathsf{if}\ v = \mathbf{apply}_p(p_i,c_1,\ldots,c_{n_i}) \\[1em]
\textbf{Cond.}\mathsf{I} & \mathsf{if}\ \mathsf{I}(e_1)\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & =_{\langle\rangle} & \mathsf{I}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) \\[1em]
\textbf{Cond.true} & \mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3 & =_{\langle\rangle} & e_2 \\[1em]
\textbf{Cond.false} & \mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3 & =_{\langle\rangle} & e_3 \\[1em]
\textbf{Case.}\mathsf{I} & \left(\begin{array}{rcl} \multicolumn{3}{l}{\mathsf{case}\ \mathsf{I}(e_1)\ \mathsf{of}} \\ \mathsf{nil} & \Rightarrow & e_2 \\ x:xs & \Rightarrow & e_3 \end{array}\right) & =_{\langle\rangle} & \mathsf{I}\left(\begin{array}{rcl} \multicolumn{3}{l}{\mathsf{case}\ e_1\ \mathsf{of}} \\ \mathsf{nil} & \Rightarrow & e_2 \\ x:xs & \Rightarrow & e_3 \end{array}\right) \\[1em]
\textbf{Case.nil} & \left(\begin{array}{rcl} \multicolumn{3}{l}{\mathsf{case\ nil\ of}} \\ \mathsf{nil} & \Rightarrow & e_2 \\ x:xs & \Rightarrow & e_3 \end{array}\right) & =_{\langle\rangle} & e_2 \\[1em]
\textbf{Case.cons} & \left(\begin{array}{rcl} \multicolumn{3}{l}{\mathsf{case}\ e_h : e_t\ \mathsf{of}} \\ \mathsf{nil} & \Rightarrow & e_2 \\ x:xs & \Rightarrow & e_3 \end{array}\right) & =_{\langle\rangle} & e_3\{e_h/x, e_t/xs\}
\end{array}
$$

Figure 6: Cost equivalence laws K

THEOREM **7.4 (completeness)**
For all closed expressions e, if $e \overset{m}{\rightarrow}_{H} h$ then $\vdash_K e =_{\langle\rangle} \mathsf{I}^m(h)$.

The proof is given in Appendix A

THEOREM **7.5 (soundness)** *For all closed expressions e, and head normal forms h_1, if*

$$\vdash_K e =_{\langle\rangle} \mathsf{I}^m(h_1)$$

then $e \overset{m}{\rightarrow}_{H} h_2$ for some h_2 such that $\vdash_K h_1 =_{\langle\rangle} h_2$.

PROOF By definition of I, and the fact that h_1 is a head normal form, $\mathsf{I}^m(h_1) \overset{m}{\rightarrow}_{H} h_1$. By definition of cost equivalence it follows that $e \overset{m}{\rightarrow}_{H} h_2$ for some h_2 such that $h_1 =_{\langle\rangle} h_2$. It remains to show that this cost equivalence is provable:

$$\begin{array}{rcll} e \overset{m}{\rightarrow}_{H} h_2 & \Rightarrow & \vdash_K \mathsf{I}^m(h_2) =_{\langle\rangle} e & \text{(Theorem 7.4)} \\ & \Rightarrow & \vdash_K \mathsf{I}^m(h_1) =_{\langle\rangle} \mathsf{I}^m(h_2) & \\ & \Rightarrow & \vdash_K h_1 =_{\langle\rangle} h_2 & (\mathsf{I}\text{-elim}) \end{array}$$

□

8 Higher Order Functions

One advantage of a simple operational approach to reasoning about programs is the relative ease with which we can handle higher order functions. In this section we show how the time rules can be easily extended to cope with the incorperation of the terms and evaluation rules of the lazy lambda calculus [Abr90]. The only potentially difficult part is the extension of the theory of cost equivalence to cope with lambda terms. We sketch how the precongruence proof for cost simulation can be extended, with few modifications, to handle lambda terms and their application.

8.1 The Lazy Lambda Calculus

We consider an extension to the language with the terms and evaluation rules of the lazy lambda calculus[Abr90,Ong88]. The lazy lambda calculus, Λ, shares the syntax of the pure untyped lambda calculus, but has an operational semantics which is consistent with implementations of higher order functional languages, *viz.* there is no evaluation "under a lambda".

The usual definitions of free and bound variables in lambda terms apply, and we do not repeat the definitions here. The evaluation rules for application and lambda terms are given below:

$$\textbf{lambda}\ \lambda x.e \rightarrow_{\alpha} \lambda x.e$$

$$\textbf{apply}\frac{e_1 \rightarrow_{H} \lambda x.e \quad e\{e_2/x\} \rightarrow_{\alpha} u}{e_1\ e_2 \rightarrow_{\alpha} u}$$

Remark Notice that we do not evaluate under a lambda even in the case of evaluation to "normal form". This is consistant with the printing mechanisms provided for higher order functional languages that allow functions to be the top-level results of programs. However, in the sequel we focus purely on the $\rightarrow_H$ relation.

In the analysis of cost we choose to additionally count the number of times we invoke the **apply**-rule in the evaluation of a term. The extension of the time rules is completely obvious:

$$\begin{array}{rcl} \langle \lambda x.e \rangle^{\alpha} & = & 0 \\ \langle e_1\ e_2 \rangle^{\alpha} & = & 1 + \langle e_1 \rangle^{H} + \langle e\{e_2/x\} \rangle^{\alpha}, \text{ if } \ e_1 \rightarrow_H \lambda x.e \end{array}$$

8.2 Applicative Cost Simulation

We will sketch the following:

- the extension of the definition of cost simulation to *applicative* cost simulation;
- the proof that applicative cost simulation is a precongruence.

The extension of the definition of cost simulation to handle the case where an expression evaluates to a lambda expression follows the definition of applicative (bi)simulation [Abr90].

DEFINITION **8.1** *If $\mathcal{R}$ is a binary relation on closed expressions, then $\mathcal{R}^{\lambda}$ is the binary relation on lambda expressions such that $(\lambda x.e_1\ \mathcal{R}^{\lambda}\ \lambda y.e_2)$ if and only if for all closed expressions e, $(\lambda x.e_1)e\ \mathcal{R}\ (\lambda y.e_2)e$*

□

As in definition 6.5 we define applicative cost-simulation as the maximal fixed point of a monotone function: For each binary relation R on closed expressions, define the relation $\mathcal{A}(R)$ by

$$\begin{array}{lll} e\ \mathcal{A}(R)\ e' \iff & \textit{if} & e \xrightarrow{t}_H h \\ & \textit{then} & e' \xrightarrow{t}_H h' \textit{ for some } h' \\ & & \text{such that } h(\mathcal{R}^{\cdot} \cup \mathcal{R}^{\lambda})h' \end{array}$$

Now we say that a relation S is an *applicative cost-simulation* if $S \subseteq \mathcal{A}(S)$.

DEFINITION **8.2 (Applicative Cost Simulation)**
Let $\sqsubseteq$ denote the largest applicative cost-simulation, the maximum fixed point of $\mathcal{A}(\cdot)$, given by $\bigcup\{S : S \subseteq \mathcal{A}(S)\}$

□

It is again straightforward to show that $\sqsubseteq$ is a preorder. We prove that $\sqsubseteq$ is preserved by substitution into arbitrary (closed) contexts by a direct extension of the proof of that for $\preceq$ (lemma 6.10 and theorem 6.11). As before we construct a relation which contains $\sqsubseteq$ and all closed substitution instances and show that it is a cost-simulation.

THEOREM **8.3 (Precongruence II)** *If $\tilde{e} \sqsubseteq \tilde{e}'$ for some commonly indexed families of closed expressions $\tilde{e}, \tilde{e}'$, then for all expressions e containing at most variables $\tilde{x}$*

$$e\{\tilde{e}/\tilde{x}\} \sqsubseteq e\{\tilde{e}'/\tilde{x}\}$$

The proof is sketched in Appendix A.

Again the use of the term *congruence* could be challenged since we do not consider open expressions who's free variables are captured by the context. As before we can extend applicative cost simulation to open expressions e and e', by saying $e \sqsubseteq e'$ if $e\sigma \sqsubseteq e\sigma'$ for all closing substitutions σ. It is then easy to show that, for example, $\lambda x.e \sqsubseteq \lambda x.e'$.

9 A Further Example

In this section we present a final example. It gives a good illustration of the use (and proof) of a cost equivalence in the derivation of a time property. The reader is invited to attempt an analysis without the use of cost equivalence.

The following equations (figure 7) define a simple functional version of quicksort (qs) using a filter, and append (as defined earlier) written here as an infix function ++. Primitive functions for integer comparison have also been written infix to aid readability.

```
qs(xs)        =  case  xs of
                       nil ⇒nil
                       h:t ⇒qs(filter(λx.x ≤ h, t))
                            ++(h : qs(filter(λx.x > h, t)))

filter(p,ys)  =  case  ys of
                       nil ⇒nil
                       h:t ⇒if p h then h : filter(p,t)
                            else filter(p,t)
```

Figure 7: Functional Quicksort

For this example we will show that quicksort exhibits its worst-case $\mathcal{O}(n^2)$ behaviour even when we only require the first element of the list to be computed, in contrast to the earlier insertion sort example which always takes linear time to compute the first element of the result. First consider the general case:

$$\langle \mathsf{qs}(e)\rangle^H = 1 + \langle e\rangle^H + \begin{cases} 0 & \text{if } e \rightarrow_H \mathsf{nil} \\ \left\langle \begin{array}{l} \mathsf{qs(filter(}\lambda\mathsf{x.x} \leq \mathsf{y,\ z))} \\ +\!\!+\mathsf{(y{:}qs(filter(}\lambda\mathsf{x.x} > \mathsf{y,\ z)))} \end{array} \right\rangle^H & \text{if } e \rightarrow_H \mathsf{y{:}z} \end{cases} \tag{6}$$

From the time rules and the definition of append, this simplifies to

$$\langle \mathsf{qs}(e)\rangle^H = 1 + \langle e\rangle^H + \begin{cases} 0 & \text{if } e \rightarrow_H \mathsf{nil} \\ 1 + \langle \mathsf{qs(filter(}\lambda\mathsf{x.x} \leq \mathsf{y,\ z))}\rangle^H & \text{if } e \rightarrow_H \mathsf{y{:}z} \end{cases}$$

It is not surprising that we will use non-increasing lists to show that $\langle \mathsf{qs}(v)\rangle^H = \Omega(n^2)$. Towards this goal, fix an arbitrary family of integer values $\{v_i\}_{i>0}$ such that $v_i \leq v_j$ whenever $i \leq j$. Now define the family of non-increasing lists $\{A_i\}_{i\geq 0}$ by induction on i:

$$\begin{array}{rcl} A_0 & = & \mathsf{nil} \\ A_{k+1} & = & v_{k+1} : A_k \end{array}$$

Now we will show that $\langle \mathsf{qs}(R_n)\rangle^H$ is quadratic in n. The key to this derivation is the identification of a cost equivalence which allows us to simplify an instance of filter. Define the family of lists $\{A_i^a\}_{i\geq 0, a\geq 0}$ inductively as follows[7]:

$$\begin{array}{rcl} A_0^a & = & \mathsf{I}^a(\mathsf{nil}) \\ A_{k+1}^a & = & \mathsf{I}^{2a}(v_{k+1} : A_k^a) \end{array}$$

Proposition **9.1** *For all $a \geq 0$, and i, j such that $0 \leq j \leq i$,*

$$\mathsf{filter}(\lambda x.x \leq v_i,\ A_j^a) =_{\langle\rangle} A_j^{a+1}.$$

PROOF We construct a family of relations containing all instances of the proposed cost-equivalence, and show that each member (and hence their union) is a cost-simulation. For each $i > 0$ $a \geq 0$, let X_i^a be the symmetric closure of the following relation:

$$\{(\mathsf{filter}(\lambda x.x \leq v_i,\ A_j^a), A_j^{a+1}) \mid j \leq i\}$$

Now it is sufficient to show that each X_i^a is a cost simulation, since this implies that their union is also a cost simulation. We show that X_i^a is a cost simulation modulo identity, i.e., that $X_i \subseteq \mathcal{F}(X_i^a \cup \equiv)$. Let $p = \lambda x.x \leq v_i$. Each pair of related elements in X_i^a has the form $(\mathsf{filter}(p,\ A_j^a), A_j^{a+1})$ (or vice-versa) We proceed by cases according to the value of j. Suppose $j = 0$. Then from the definitions we have that $\mathsf{filter}(p,\ A_0^a) \overset{a+1}{\rightarrow}_H \mathsf{nil}$ and $A_j^{a+1} \overset{a+1}{\rightarrow}_H \mathsf{nil}$, and we are done.

[7]Recalling from section 7.2 that I is just the identity function $\mathsf{I(x) = x}$, and $\mathsf{I}^n(e)$ denotes n applications of the identity function to e, with the convention that $\mathsf{I}^0(e)$ is just e.

Suppose $j = k + 1$ for some $k \geq 0$. Then by calculation from the definitions

$$\begin{array}{rcl} \mathsf{filter}(p,\ A^a_{k+1}) & \overset{2+2a}{\longrightarrow}_H & v_{k+1} : \mathsf{filter}(p,\ A^a_k) \\ A^{a+1}_{k+1} & \overset{2(a+1)}{\longrightarrow}_H & v_{k+1} : A^{a+1}_k \end{array}$$

Now the heads are related by the identity, and the tails by X^a_i, so the results are related by $(X^a_i \cup \equiv)^{:}$ and we are done. □

Remark A proof can be given without explicit use of the evaluation relation at all, by using the cost-equivalence laws given in section 7.2, together with an argument using mathematical induction on j. However, the proof method of constructing a cost simulation is more powerful since it allows the proposition be generalised to include possibly infinite non-increasing lists (although this is not relevant in the context of sorting).

Now we can return to quicksort. We consider the more general case of $\left\langle \mathsf{qs}(A^a_j) \right\rangle^H$. Considering the cases when $j = 0$ and $j = k + 1$, and instantiating the general time equation 6 gives:

$$\begin{array}{rcl} \langle \mathsf{qs}(A^a_0) \rangle^H & = & 1 + \langle A^a_0 \rangle^H \\ & = & 1 + a \end{array}$$

$$\begin{array}{rcl} \left\langle \mathsf{qs}(A^a_{k+1}) \right\rangle^H & = & 1 + \left\langle A^a_{k+1} \right\rangle^H + 1 \\ & & + \langle \mathsf{qs}(\mathsf{filter}(\lambda \mathsf{x}.\mathsf{x} \leq v_{k+1}\ , A^a_k)) \rangle^H \\ & = & 2 + 2a \\ & & + \langle \mathsf{qs}(\mathsf{filter}(\lambda \mathsf{x}.\mathsf{x} \leq v_{k+1}\ , A^a_k)) \rangle^H \end{array}$$

We can simplify the right-hand side by the proposition, to give

$$\left\langle \mathsf{qs}(A^a_{k+1}) \right\rangle^H = 2 + 2a + \left\langle \mathsf{qs}(A^{a+1}_k) \right\rangle^H$$

Again the recurrence is easily solved; a simple induction is sufficient to check that

$$\langle \mathsf{qs}(A^a_n) \rangle^H = n^2 + 2na + 2n + a + 1.$$

Since the A_n are just A^0_n, we finally have

$$\langle \mathsf{qs}(A_n) \rangle^H = n^2 + 2n + 1.$$

10 Call-By-Need and Compositionality

The calculus is betrayed by its simple operational origins because it describes a *call-by-name* evaluation mechanism, when most actual implementations of lazy evaluation use *call-by-need*. For example, consider the definition

$$\mathsf{average(xs) = divide(sum(xs),length(xs))}$$

where divide is a primitive function, and sum and length are the obvious functions on lists. In reasoning about the evaluation of an instance, average(e), our method will overestimate the evaluation time because of the duplication of expression e and the subsequent duplication of the evaluation of the spine of e.

One solution is to work with an operational model that takes into account the sharing of expressions and evaluations (*e.g.* [BvEG$^+$87]). Unfortunately this may tend to overly complicate the calculus, and is likely to be impractical—although there are some promising (less general) approaches to modeling sharing and storage, [AA91,MT91a], which may be prove usable.

Another solution is to move towards the compositional approaches mentioned in the introduction. A suitable interface between the compositional approach in [San90b] (which differs from that of [Wad88] in its use of genuine *strictness* rather than *absence* information) from and the operational approach of this paper is via Burn's notion of an *evaluator* [Bur91]. An evaluator is an operational concept which provides a link from information provided by strictness analysis, to the operational semantics. In particular, strictness analysis (see, *e.g.* [BHA86]) will tell us that when an application of average is being evaluated, it is safe to evaluate the argument to normal-form (since this evaluation will occur anyway). In terms of our calculus, by taking into account (in advance) the amount of evaluation that must be performed on the argument, we get a more compositional analysis, and a better approximation to call-by-need, using:

$$\langle \mathsf{average}(e)\rangle^H = \langle e\rangle^N + \langle \mathsf{average}(v)\rangle^H, \text{where } \; e \to_N v \; .$$

In this section we describe a new formalisation of evaluators appropriate for providing a smooth interface between compositional and noncompositional (call-by-need and call-by-name) approaches to time analysis. The development is for the first order language, although it can be used within higher-order programs.

10.1 Demands

Burn's formalisation of evaluators is couched in terms of *reduction strategies* in a typed lambda calculus with constants. An evaluator is defined, relative to a particular (Scott) closed set of denotations, as any reduction strategy which fails to terminate exactly when the denotation of the term is in the set. For example, the leftmost outermost reduction strategy fails to terminate if and only if the denotation of the term is in the Scott-closed set $\{\perp\}$.

However, the definition of an evaluator is not constructive. Given a Scott-closed set, Burn does not provide an operational definition of an evaluator for that set.

In the approach we have taken to the operational semantics of the language, issues of reduction strategy are internalised by the evaluation relations. In order to use the evaluator concept in reasoning about evaluation cost we need a constructive definition of evaluators. We will define a family of evaluators, and show how the relations $\to_N$ and $\to_H$ can be viewed as instances. The starting point of this definition is a language of *demands*.

We interpret a demand as a specification of a degree of evaluation, and define a demand-parameterised evaluation relation which realises these demands. Bjerner and Hölmstrom [BH89] use a particularly simple language of demands in the context of compositional time analysis. Their interpretation of a demand only makes sense relative to a particular expression: a demand on an expression is a representation of an approximation to its denotation. Their language of demands is too "precise" for our purposes. The language of demands which we use will be closer to that of, for example, Lindström's lattice of demands [Lin86] or Dybjer's formal opens [Dyb87], and our interpretation of demand, as in [Dyb87], will be closely related to strictness. The operational interpretation of demand as an evaluator is in turn reminiscent to Bjerner's definition of computing an expression to one of its *proper evaluation degrees* [Bje89].

Definition **10.1** *The language of demands, $d \in D$ is given by*

$$D = \epsilon \mid \kappa \mid D_1{:}D_2 \mid \kappa + D_1{:}D_2 \mid \mu x.(D) \mid x$$

□

The demand ϵ is the zero-demand which is satisfied by any expression. The demand κ is satisfied by any expression which evaluates to a constant, including nil (we could easily extend the demands to also include demands for each individual constant). The cons-demand $D_1{:}D_2$ is satisfied by any expression which evaluates to a cons, and whose head and tail satisfy D_1 and D_2 respectively. We have a restricted form of disjunctive demand for either a constant or a cons, and finally we add a recursive demand useful for specifying demands over lists.

This informal reading of a demand can easily be formalised, and the set of expressions which satisfy a demand can be shown to be *open* with respect to the usual operational preorder (*c.f.*[Dyb87]). We do not pursue this formalisation here, but just focus on the operational interpretation of demands as evaluators.

Definition **10.2** *For each closed demand d we define an associated evaluator $\Longrightarrow_d$. The family of evaluators is defined by inductively as the least relations between closed expressions that satisfy the following rules:*

$$\frac{}{e \Longrightarrow_\epsilon e} \qquad \frac{e \rightarrow_H c}{e \Longrightarrow_\kappa c}$$

$$\frac{e \rightarrow_H e_1 : e_2 \quad e_1 \Longrightarrow_{d_1} e_1' \quad e_2 \Longrightarrow_{d_1} e_2'}{e \Longrightarrow_{d_1{:}d_2} e_1' : e_2'}$$

$$\frac{e \Longrightarrow_\kappa e'}{e \Longrightarrow_{\kappa+d_1{:}d_2} e'} \qquad \frac{e \Longrightarrow_{d_1{:}d_2} e'}{e \Longrightarrow_{\kappa+d_1{:}d_2} e'}$$

$$\frac{e \Longrightarrow_{d\{\mu x.(d)/x\}} e'}{e \Longrightarrow_{\mu x.(d)} e'}$$

□

Note that the evaluators are defined using the basic reduction engine, $\rightarrow_H$. It is not difficult to show that evaluators are deterministic (*i.e.*, each $\Longrightarrow_d$ is a partial function) and that proofs of evaluation judgements are unique. As for $\rightarrow_\alpha$, we write $e \overset{t}{\Longrightarrow}_d e'$ when $e \Longrightarrow_d e'$ and its proof uses $t \geq 0$ instances of rule **Fun**.

Now we can show that the previous evaluation relations can be viewed as instances of evaluators:

PROPOSITION **10.3**

(i) $e \overset{t}{\Longrightarrow}_{\kappa+\epsilon::\epsilon} e' \iff e \overset{t}{\rightarrow}_H e'$

(ii) $e \overset{t}{\Longrightarrow}_{\mu x.(\kappa+x::x)} e' \iff e \overset{t}{\rightarrow}_N e'$

PROOF

(i) The first part is simple. ($\Rightarrow$) follows from the definition of evaluators, ($\Leftarrow$) by considering the two cases for e'.

(ii) ($\Rightarrow$) Induction on the size of the proof of $e \overset{t}{\Longrightarrow}_{\mu x.(\kappa+x::x)} e'$. Abbreviate demand $\mu x.(\kappa + x::x)$ by A. Clearly either $e' = c$, and hence $e \overset{t}{\rightarrow}_H c$, in which case $e \overset{t}{\rightarrow}_N c$ follows from proposition 6.7, or $e' = e_1 : e_2$, and the proof has the form

$$\dfrac{\dfrac{\dfrac{e \overset{t'}{\rightarrow}_H e_1' : e_2' \quad e_1' \overset{t1}{\Longrightarrow}_A e_1 \quad e_2' \overset{t2}{\Longrightarrow}_A e_2}{e \overset{t}{\Longrightarrow}_{A::A} e_1 : e_2}}{e \overset{t}{\Longrightarrow}_{\kappa+A::A} e_1 : e_2}}{e \overset{t}{\Longrightarrow}_A e_1 : e_2}$$

where $t = t' + t_1 + t_2$. By the induction hypothesis, $e_1' \overset{t1}{\rightarrow}_N e_1$ and $e_2' \overset{t2}{\rightarrow}_N e_2$. By the rules for evaluation to normal form, $e_1' : e_2' \overset{t_1+t_2}{\rightarrow}_N e_1 : e_2$ and so by proposition 6.7, since $t = t' + t_1 + t_2$ we can conclude $e \overset{t}{\rightarrow}_N e_1 : e_2$, as required.

($\Leftarrow$) follows by a routine induction on the structure of normal form e', appealing again to proposition 6.7. The details are omitted.

□

Now we can give a definition of time rules $\langle \_ \rangle^d$ which extend the definitions for $\langle \_ \rangle^H$ (and subsume the definitions for $\langle \_ \rangle^N$). The rules are obvious, but we include them for completeness in figure 8. It is also possible to show, (although we do not provide details here) that cost equivalence is congruence with respect to the demanded time rules,

$$e \preceq e' \Rightarrow \forall C. \langle C[e] \rangle^d = \langle C[e'] \rangle^d, \text{whenever } C[e]{\downarrow_d}.$$

$$
\begin{array}{rcl}
\langle e\rangle^{\epsilon} & = & 0 \\[2ex]
\langle e\rangle^{\kappa} & = & \langle e\rangle^{H} \text{ if } e \rightharpoondown_{H} c \\[2ex]
\langle e\rangle^{d_1:d_2} & = & \langle e\rangle^{H} + \langle e_1\rangle^{d_1} + \langle e_2\rangle^{d_2} \text{ if } e \rightharpoondown_{H} e_1 : e_2 \\[2ex]
\langle e\rangle^{\kappa+d_1:d_2} & = & \langle e\rangle^{H} + \begin{cases} 0 & \text{if } e \rightharpoondown_{H} c \\ \langle e_1\rangle^{d_1} + \langle e_2\rangle^{d_2} & \text{if } e \rightharpoondown_{H} e_1 : e_2 \end{cases}
\end{array}
$$

Figure 8: Demand Time Rules

10.2 Demand Use

The weakness in the model with respect to call-by-need computation is that the substitution operation duplicates expressions, and consequently does not "share" any evaluations of that expression to head-normal form (and subsequent evaluations of sub-expressions).

Let $\langle e\rangle^{d}_{\text{need}}$ informally denote the call-by-need cost of evaluating expression e up to demand d.

Suppose we know that the following condition (Δ) holds:

> Whenever an application of some function f is to be evaluated up to demand d, then its argument must satisfy some demand d'.

In other words, for all arguments e, if $\mathsf{f}(e) \Longrightarrow_{d} e'$ for some e', then there exists a u such that $e \Longrightarrow_{d'} u$. When this is the case, we can *refine* our call by name model as follows.

$$\langle \mathsf{f}(e)\rangle^{d} \geq (\langle e\rangle^{d'} + \langle \mathsf{f}(u)\rangle^{d}) \geq \langle \mathsf{f}(u)\rangle^{d}_{\text{need}}\,.$$

To attempt a rigorous justification of the second inequality, we would naturally need to formalise what we mean by call-by-need evaluation. This nontrivial task is beyond the scope of this paper, and we leave it as an open problem.

Here is an informal explanation of the first inequality. Firstly, it is not too difficult to show that when condition (Δ) holds for f that there exists an expression u such that $e \Longrightarrow_{d'} u$, and that $\mathsf{f}(u) \Longrightarrow_{d} e'$.[8] Now, since e must satisfy d' whenever $\mathsf{f}(e)$ satisfies d, it must be the case that each sub-proof of a judgement of the form $a \rightharpoondown_{H} b$ in the proof of $e \Longrightarrow_{d'} u$ must also occur *at least once* in the proof of $\mathsf{f}(e) \Longrightarrow_{d} e'$. This is because no backtracking is needed in proof construction, and so the condition implies that $\Longrightarrow_{d}$ must perform as much computation on e as $\Longrightarrow_{d'}$. Now consider the proof of $\mathsf{f}(u) \Longrightarrow_{d} e'$. This proof will have essentially the same structure as the proof of the evaluation of $\mathsf{f}(e)$ up

[8]This is a *constructive* version of Burn's evaluation transformer theorem—further investigations will be presented elsewhere.

to d, except that each of the aforementioned sub-proofs will be replaced by sub-proofs of the form $b \rightarrow_H b$, which are just instances of axioms, and hence zero-cost. The *in*equality arises because the costs of each syntactically distinct sub-proof $a \rightarrow_H b$ in $e \Longrightarrow_{d'} u$ is counted only once in $\langle e \rangle^{d'}$.

We have not found a satisfactory proof based on this sketch (or otherwise). It would be nice to have an "algebraic" proof which does not mention proof-trees explicitly. Cost equivalence and related tools such as *improvement* preorderings [San91a] may be useful here.

10.3 Demand Propagation

In order to use this method for refining the call-by-name model we need to establish for some context $C[\;]$, the demand an expression placed in its hole satisfy must satisfy, given some demand which must be satisfied by the composite expression. Of course, such demands on the hole are not unique. For example, any expression placed in the hole must satisfy the trivial demand ϵ. However this trivial demand does not allow us to refine the call-by-name cost model. In general we want to determine as "large" a demand on the hole as possible.

In this paper we shall not pursue the problem of demand propagation, but mention some of the connections with the much-studied strictness analysis problem.

As mentioned earlier, demand propagation can be formalised by modelling a demand as the set of expressions which satisfy it. These sets can be shown to be open (right-closed) under an operational preordering on expressions, and so the problem can be viewed as an inverse-image analysis problem [Dyb87], albeit expressed in terms of an operational model rather than a denotational one. The corresponding problem in a higher order language is more easily tackled by a forwards analysis in which information about the *complement* of a demand (all the elements which do not satisfy it) are propagated forwards from sub-expressions to their context. This corresponds to the higher-order strictness analysis described in [BHA86]. So in principle the approach described here can be extended to a higher-order language, but it is not obvious how a higher-order demand can be given an operational interpretation, so there is still an inherently noncompositional aspect (*c.f.*. the higher-order approach described in [San90a], Ch. 5).

10.4 Comparison with Earlier Compositional Approaches

Here we place the approach outlined in this section in comparison with the compositional methods for time analysis overviewed in Section 2. The use of demands to refine call-by-name evaluation time to give a better approximation to call-by-need is consistent with the use of strictness information in *necessary time analysis* [San89]. The key difference is that strictness information is used by necessary-time analysis to give a lower bound to call-by-need computation cost, whereas it is used here to give an upper-bound (which in turn is bounded above by call-by-name cost). In both cases the quality of demand information determines the tightness of the bound. We would argue that the method here is more useful.

The time analysis described by Wadler [Wad88] (which is the first order instance of *sufficient-time analysis* in [San89]) also gives an upper-bound to call-by-need time, but in a rather different manner: it uses information about "absence" (constancy) which describes what parts of an expression will *not* be evaluated. Unfortunately in this case although the quality of this upper bound is determined by the quality of this "absence" information[9] , this upper-bound may not be well-defined even when the program is.

The approach described by Bjerner and Hölmstrom [BH89] is a fully compositional method for reasoning about first order functional programs with lazy lists. This approach is harder to compare because the "demand" information is *exact*; the call-by-need time analysis is therefore exact as well (although, as here, the treatment of call-by-need is not formalised). This precision, as the authors note, is also a serious drawback in reasoning about programs, since as a first step one must decide what "demand" to make on the output. Since the language of demands is so precise, in order to reason about computation of a program to normal form one must begin with the demand that *exactly* describes that normal form—in other words (a representation of) the normal form itself. It is not immediately obvious how to introduce "approximate" demands in this approach without running into the definedness problems of Wadler's method.

To summarise, the approach sketched in this section has the following advantages:

- it provides a safe (well-defined) time-bound lying between call-by-need and call-by-name costs;
- it allows flexibility in the use of demand information so it can be targeted to where either
  - (i) more compositionality is needed to decompose the time analysis of a large program, or
  - (ii) where (it is suspected that) the call-by-name model is too crude.

Although our method and exposition were inspired by formulating a constructive (operational) version of Burn's evaluators, in retrospect the approach is much closer to Bjerner's original work on time analysis of programs of type theory, because of it's operational basis. The connection to evaluators does not occur in Bjerner's work because of the absence of nonterminating computations in that language.

11 Related Work

Here we review work related to the theory of cost-simulation.

[9]Quality should not be confused with *safety*—we assume that the information is always *safe* in the sense that whenever it predicts a property of a program, the property does indeed hold.

11.1 Other Non-standard Theories of Equivalence

Talcott [Tal85] introduced a broad semantic theory of side-effect free Lisp-like languages, notable for its treatment of both extensional and intensional aspects of computation. In particular a class of preorderings called *comparison relations* were introduced for a side-effect free Lisp derivative. The method of their definition is similar to the definition of operational approximation as a Park-style simulation, although the cost-aspects are not built into this definition directly as they are here. Nonetheless, the class of comparison relations could be said to contain relations analogous to the cost equivalence relation considered here (this is just the observation that cost-simulation can be viewed as a refinement of a pure simulation not involving time properties), and indeed it is suggested (as a topic for further work) that certain comparison relations could be developed to provide soundness and improvement proofs for program transformation laws. However, only the "maximal" comparison relations (essentially, the more usual operational approximation and equivalence relations) and their application are considered in detail. Following on from other aspects of this work, Mason [Mas86] sketches the definition a family of equivalence relations involving a variety of operation execution-counts. However, as this is for a pure language with neither higher-order functions nor lazy data-structures, the relations have a relatively uninteresting structure.

Moggi's categorical semantics of computation [Mog89] is intended to be suitable for capturing broader descriptions of computation than just input-output behaviour. Gurr [Gur91] has studied extensions of denotational semantics to take account of resource-use, and has shown how Moggi's approach can be used to model computation in such a way that program equivalence also captures equivalence of resource-requirements. Gurr extends Moggi's λ_c-calculus (a formal system for reasoning about equivalence) with sequents for reasoning about the resource properties *directly* (although the ability to do this depends on certain "representability" issues, not least of all that the resource itself should correspond to a type in the metalanguage). The resulting calculus is dubbed "λ_{com}". We can compare this to the approach taken here, where cost-equivalence (a "resource" equivalence) is used in conjunction with a set of *time rules* which are used to reason about the cost property directly. There are further analogies in the details of Gurr's approach: he defines rules for sequents of the form

$$\Gamma \vdash_{\lambda_{com}} \langle e, v, t\rangle : \tau$$

which says that expression e has value v (of type τ) and consumes resource t. He goes on to show that these rules are redundant since their information can be expressed in the λ_c calculus (with the addition of some specific axioms). In particular the above entailment would imply

$$\Gamma \vdash_{\lambda_c} \text{let } x = t \text{ in } v : \tau$$

where t is the canonical term (of unit type) which consumes time "t". Notice the similarity with our axiomatisation of cost equivalence, which uses the identity function in much the same manner as the computational let is used in the λ_c calculus.

An important difference is that in our approach these concepts are derived from (and hence correct with respect to) the operational model, so we may argue, at least, that an operational approach provides a more appropriate starting point for a semantic study of efficiency—although the deeper connections between these approaches deserves some further study[10]. Another important difference is that we consider a lazy recursive data structure. One possible method for dealing with this in the context of Moggi's framework would be to combine it with Pitts' co-induction principal for recursively defined domains [Pit92]. Other aspects of Gurr's work are concerned with giving a semantic framework for some aspects of (asymptotic) complexity, which is outside the scope of this work.

Another, perhaps more abstract category-theoretic approach to intensional semantics is presented by Brookes and Geva [BG92]. This work places a heavy emphasis on the relationship between the intesional semantics and its underlying extensional one. The key structure is the use of a co-monad, in contrast to Gurr's use of Moggi's monadic style, and it's suitability for formulating the kind of intensional semantics described here is perhaps less obvious.

11.2 Program Improvement and Generalisations

The theory of cost simulation is significant in its own right since there are many potentially interesting relations (preorders and equivalences) involving time, that can be investigated in a similar manner. For example, consider a *program refinement* relation, $\trianglerighteq$, which is the largest relation such that whenever $e_1 \trianglerighteq e_2$

$$\begin{aligned} e_1 \xrightarrow{t}_H h \Rightarrow\ & e_2 \xrightarrow{t'}_H h', \text{for some } h, h' \text{ such that } t \geq t' \\ & \text{and } (h \trianglerighteq^{:} h'). \end{aligned}$$

Using the same approach to that of cost simulation, it can be shown that $\trianglerighteq$ is a (pre)congruence[11] , and consequently that for all contexts C,

$$e \trianglerighteq e' \Longrightarrow \langle C[e] \rangle^{\alpha} \geq \langle C[e'] \rangle^{\alpha},$$

which can be restated as "e' is at least as efficient as e in any context". Using the same proof technique as illustrated in Section 7, we have a systematic means of verifying refinement laws, such as

$$\mathsf{append}(\mathsf{append}(e_1,e_2),e_3) \trianglerighteq \mathsf{append}(e_1,\mathsf{append}(e_2,\ e_3)).$$

The notion of refinement is a possible semantic criterion (albeit a somewhat exacting one) for the intensional correctness of "context free" program transformations.

The main foundation for such theories of improvement and equivalence is the definition of an appropriate simulation relation, together with a proof that it satisfies the substitutivity property. So for each variation in the language (such as the addition of new operators) and each new definition of what improvement means, we require these

[10]Interestingly, Gurr gives an operational semantics (call-by-value) to terms of the λ_{com} calculus, but leaves the property corresponding to our Theorem 7.5 as a conjecture.

[11]Note that if we consider a strict refinement relation where we replace $\geq$ by $>$ in the definition of $\trianglerighteq$, then it will *not* be a contextual congruence.

constructions. This is somewhat tedious, so in a separate study[San91a] the problem of finding a more general formulation of the theory of improvement relations is addressed. For a general class of lazy languages, it is shown how a preorder on computational properties (an "improvement" ordering) induces a simulation-style preordering on expressions (the definitions of cost simulation and program refinement are simple instances). Borrowing some syntactic conventions and semantic techniques from [How89], some *improvement extensionality* conditions on the operators of the language are given which guarantee that the improvement ordering is a precongruence. The conditions appear relatively easy to check. Furthermore, the higher-order language given here is studied as a special case. In this context a *computational property* is considered to be a function from the proof of an evaluation judgement (the computation) to a preordered set (the set of properties, ordered by "improvement"). The main result of this generalisation is that the simulation preorder induced by any computational property is guaranteed to be a congruence whenever the property satisfies a simple monotonicity requirement with respect to the rules of the operational semantics. This result can be generalised to Howe's class of structured computation systems [How91].

12 Conclusions

This paper has presented a direct approach to reasoning about time cost under lazy evaluation. The calculus takes the form of *time rules* extracted from a suitably simple operational semantics, together with some equivalence laws which are substitutive with respect to these rules.

The aim of this calculus is to reveal enough of the "algorithmic structure" of operationally opaque lazy functional programs to permit the use of the more traditional techniques developed in the context of the analysis of imperative programs (see *e.g.* [AHU74]), and initial experiments with this calculus suggest that it is of both practical and pedagogical use.

A desire for substitutive equivalences for the time-rules led to a theory of cost equivalence, via a non-standard notion of operational approximation called *cost-simulation*. Cost equivalence provides useful extensions to the time rules (although, as we showed, from a technical point of view they subsume them) but is also interesting in it's own right since it suggests an operationally-based route to the study of intensional semantics. Initial investigations of this area are reported in [San91a].

Finally, we proposed an interface of this calculus with a more compositional method which also improves the accuracy of the analysis with respect to call-by-need evaluation, but is able to retain the simplicity of the naïve approach where appropriate. The compositionality is based on an operational interpretation of evaluators to re-order computation on the basis of strictness-like properties. This model can also be used to provide a more constructive formulation of the evaluation transformer theorem [Bur91], which formally connects information from strictness analysis with its associated optimisations. Further work is needed to strengthen the relationship between cost-equivalence and the compositional approach. The idea of a *context dependent bisimulation* between processes, as studied by Larsen (originating with [Lar86]), seems appropriate here since it suggests the introduction of context (= demand)-dependent cost-simulation.

Acknowledgements

Thanks to Sebastian Hunt for the numerous discussions which prompted the development of a general theory cost-equivalence. Thanks also to Bent Thomsen for helpful discussions on proofs of congruence, to Chris Hankin for useful comments on an earlier draft, and to various members of the *Semantique* project for numerous suggestions and constructive criticisms; in particular to John Hughes for suggesting that time analysis might be completely expressed in terms of cost equivalences. Thanks to Richard Bird, who originally suggested to me that a naïve approach could actually be useful for reasoning about call-by-name evaluation, and also suggested the `reverse` example.

References

[AA91] Z. M. Ariola and Arvind. A syntactic approach to program transformation. In *Proceedings of the symposium on Partial Evaluation and Semantics-Based Program Manipulation.* ACM press, SIGPLAN notices, 26(9), September 1991.

[Abr90] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming.* Addison Wesley, 1990.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* The Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Company, London, 1974.

[Bar84] H.P. Barendregt. *The Lambda Calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science Publishers B.V., 2nd edition, 1984.

[BG92] S. Brookes and S. Geva. Computational comonads and intensional semantics. Technical report, CMU, 1992.

[BH89] 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.

[BHA86] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. *Science of Computer Programming*, 7:249–278, 1986.

[Bje89] B. Bjerner. *Time Complexity of Programs in Type Theory.* PhD thesis, Chalmers University of Technology, 1989.

[Bur91] G. L. Burn. The evaluation transformer model of reduction and its correctness. In *TAPSOFT '91 (also Imperial College report DOC 90/19)*, 1991.

[BvEG$^+$87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *PARLE '87 volume II*, number 259 in LNCS, pages 191–231. Springer Verlag, 1987.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[CC79] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.

[Dyb87] P. Dybjer. Inverse image analysis. In *Proceedings of the 14th International Colloquium on Automata, Languages and Programming*, Karlsruhe, Germany, July 1987. Springer-Verlag LNCS 267.

[GKP89] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, 1989.

[Gur91] D. Gurr. *Semantic Frameworks for Complexity.* PhD thesis, Department of Computer Science, Edinburgh, 1991. (Available as reports CST-72-91 and ECS-LFCS-91-130).

[HC88] T. Hickey and J. Cohen. Automating program analysis. *J. ACM*, 35:185–220, January 1988.

[How89] D. J. Howe. Equality in lazy computation systems. In *Fourth annual symposium on Logic In Computer Science.* IEEE, 1989.

[How91] D. J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Sixth annual symposium on Logic In Computer Science*, 1991.

[Knu68] D. E. Knuth. *Volume 1: Fundamental Algorithms.* The Art of Computer Programming. Addison-Wesley, 1968.

[Lar86] K. G. Larsen. *Context-Dependent Bisimulation Between Processes.* PhD thesis, Department of Computing, University of Edinburgh, 1986.

[LeM88] D. LeMétayer. An automatic complexity evaluator. *ACM ToPLaS*, 10(2):248–266, April 1988.

[Lin86] G. Lindstrom. Static evaluation of functional programs. In *Symposium on Compiler Construction.* SIGPLAN, 1986.

[Mas86] I. Mason. *The Semantics of Destructive Lisp.* Number 5 in CSLI Lecture Notes. CSLI, 1986.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Fourth annual symposium on Logic in Computer Science*, 1989.

[MT91a] I. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[MT91b] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[Ong88] C.-H. Luke Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming.* PhD thesis, Imperial College, University of London, 1988.

[Par80] D. Park. Concurrency and automata on infinite sequences. In *5th GI conference on Theoretical Computer Science.* LNCS 104, Springer Verlag, 1980.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd, London, 1987.

[Pit92] A. Pitts. A co-induction principal for recursively defined domains. Technical report, University of Cambridge Computer Laboratory, 1992.

[Ros89] M. Rosendahl. Automatic complexity analysis. In *Functional Programming Languages and computer architecture, conference proceedings.* ACM press, 1989.

[San88] D. Sands. Complexity analysis for a higher order language. Technical Report DOC 88/14, Imperial College, October 1988.

[San89] D. Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of Glasgow Workshop on Functional Programming*, Workshop Series. Springer Verlag, August 1989.

[San90a] D. Sands. *Calculi for Time Analysis of Functional Programs.* PhD thesis, Imperial College, September 1990.

[San90b] D. Sands. Complexity analysis for a lazy higher-order language. In *Proceedings of the Third European Symposium on Programming*, number 432 in LNCS. Springer-Verlag, May 1990.

[San91a] D. Sands. Operational theories of improvement in functional languages (extended abstract). In *Proceedings of the Fourth Glasgow Workshop on Functional Programming*, pages 298–311, Skye, August 1991. Springer Workshop Series.

[San91b] D. Sands. Time analysis, cost equivalence and program refinement. In *Proceedings of the Eleventh Conference on Foundations of Software Technology and Theoretical Computer Science*, number 560 in LNCS, pages 25–39. Springer-Verlag, December 1991.

[Tal85] C. L. Talcott. *The Essence of Rum, A Theory of the intensional and extensional aspects of Lisp-type computation.* PhD thesis, Stanford University, August 1985.

[Wad88] P. Wadler. Strictness analysis aids time analysis. In *15th ACM Symposium on Principals of Programming Languages*, January 1988.

[Weg75] B. Wegbreit. Mechanical program analysis. *C.ACM*, 18:528–539, September 1975.

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

A Proofs

This appendix contains some details of proofs not included in the main text.

Proposition 6.7 For all closed e, e'

- $e \rightarrow_{N} c \iff e \rightarrow_{H} c$
- $e \xrightarrow{t}_{N} v \iff e \xrightarrow{t_1}_{H} h$ and $h \xrightarrow{t_2}_{N} v$ and $t_1 + t_2 = t$ for some h.

PROOF

(i) Follows by routine inductions on the structure of the proofs of $e \rightarrow_{N} c$ and $e \rightarrow_{H} c$.

(ii) ($\Rightarrow$) Induction on the structure of the proof of $e \rightarrow_{N} n$. We give an illustrative case:

$$\boxed{\textbf{Fun} \quad \frac{e_i\{e_1/x_1, \ldots, e_{n_i}/x_{n_i}\} \rightarrow_{N} v}{f_i(e_1, \ldots, e_{n_i}) \rightarrow_{N} v}}$$

Suppose that $f_i(e_1, \ldots, e_{n_i}) \xrightarrow{t}_{N} v$ for some t. Then it follows that

$$e_i\{e_1 \ldots e_{n_i}/x_1 \ldots x_{n_i}\} \xrightarrow{t-1}_{N} v,$$

and so the induction hypothesis gives

$$e_i\{e_1 \ldots e_{n_i}/x_1 \ldots x_{n_i}\} \xrightarrow{t_1}_{H} h \qquad (7)$$

$$h \xrightarrow{t_2}_{N} v \qquad (8)$$

for some head normal form h, and t_1, t_2 such that $t_1 + t_2 = t - 1$. We can conclude from 7 by the semantics for application, $f_i(e_1, \ldots, e_{n_i}) \xrightarrow{1+t_1}_{N} v$ which together with 8 concludes this case.

The other cases follow in a similar manner. Proof in the $\Leftarrow$ direction is again a routine induction on the structure of the inference $e \rightarrow_{H} h$.

□

Lemma 6.8 If $e \preceq e'$ then

(i) if $e \rightarrow_{H} h$ then $e' \rightarrow_{H} h'$ and $\langle e \rangle^{H} = \langle e' \rangle^{H}$

(ii) if $e \rightarrow_{N} v$ then $e' \rightarrow_{N} v$ and $\langle e \rangle^{N} = \langle e' \rangle^{N}$

PROOF

Induction on the structure of value v.

- $v \equiv c$
  Suppose $e \xrightarrow{t}_N c$. Then by Proposition 6.7 $e \xrightarrow{t}_H c$. By definition of cost simulation, $e' \xrightarrow{t}_H c$, and again from the Proposition $e' \xrightarrow{t}_N c$.

- $v \equiv v_1 : v_2$
  Suppose $e \xrightarrow{t}_N v_1 : v_2$. The induction hypothesis gives: for all t' and for all e_a, e_b such that $e_a \preceq e_b$
  $$\begin{array}{lcl} e_a \xrightarrow{t'}_N v_1 & \Rightarrow & e_b \xrightarrow{t'}_N v_1 \\ e_a \xrightarrow{t'}_N v_2 & \Rightarrow & e_b \xrightarrow{t'}_N v_2 \end{array}$$
  From 6.7 we know that, for some e_1, e_2, $e \xrightarrow{t'}_H e_1 : e_2$ and $e_1 : e_2 \xrightarrow{t''}_N v_1 : v_2$ with $t = t' + t''$. This implies that
  $$\begin{array}{l} e_1 \xrightarrow{t_1}_N v_1 \\ e_2 \xrightarrow{t_2}_N v_2 \\ \text{where } t'' = t_1 + t_2 \end{array} \qquad (9)$$
  Since $e \preceq e'$ we know that $e' \rightarrow_H e'_1 : e'_2$ for some e'_1, e'_2 such that $e_1 \preceq e'_1$ and $e_2 \preceq e'_2$. Now we have, by an instance of the induction hypothesis that, $i = 1, 2$
  $$e'_i \xrightarrow{t_i}_N v_i \qquad (10)$$
  from which we have, by rule **Cons** that $e'_1 : e'_2 \xrightarrow{t''}_N v_1 : v_2$. Finally $e' \xrightarrow{t}_N v_1 : v_2$ follows from proposition 6.7 ($\Leftarrow$).

□

Theorem (completeness)7.4
For all closed expressions e, if $e \xrightarrow{m}_H h$ then $\vdash_K e =_{\langle\rangle} \mathsf{I}^m(h)$.
PROOF Routine induction on the structure of the proof of $e \xrightarrow{m}_H h$. We argue by cases according to the last rule applied. We sketch a couple of cases only.

$\boxed{\textbf{Fun}}$

Assume that $e \equiv f(e_1 \ldots e_n)$, where f is defined by $f(y_1 \ldots y_n) = e_f$. Then the last inference in the proof of $e \xrightarrow{m}_H h$ has the form:

$$\frac{e_f\{e_1/y_1 \cdots e_n/y_n\} \rightarrow_H h}{f(e_1, \ldots, e_n) \rightarrow_H h}$$

It follows that $e_f\{e_1/y_1 \cdots e_n/y_n\} \xrightarrow{m-1}_H h$, and so by the induction hypothesis,

$$\vdash_K e_f\{e_1/y_1 \cdots e_n/y_n\} =_{\langle\rangle} \mathsf{I}^{n-1}(h).$$

By congruence it follows that

$$\vdash_K \mathsf{I}(e_f\{e_1\sigma/y_1 \cdots e_n\sigma/y_n\}) =_{\langle\rangle} \mathsf{I}^n(h),$$

and finally by (**Fun.**l) and transitivity we have $\vdash_K f(e_1 \ldots e_n) =_{\langle\rangle} \mathsf{l}^n(h)$.

$\boxed{\textbf{Case}}$

We consider only one of the two possible last rules in the evaluation of the case expression: assume the last inference has the form

$$\frac{e_1 \to_H e_h : e_t \quad e_3\{e_h/x, e_t/xs\} \to_H h}{\left(\begin{array}{rcl} \mathsf{case}\ e_1\ \mathsf{of} & & \\ \mathsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array}\right) \to_H h}$$

It follows that $e_1 \overset{m_1}{\to}_H e_h : e_t$ and $e_3\{e_h/x, e_t/xs\} \overset{m_2}{\to}_H h$ for some m_1, m_2 satisfying $m_1 + m_2 = m$. The induction hypothesis then gives

$$\begin{array}{rl} & \vdash_K \quad e_1 =_{\langle\rangle} \mathsf{l}^{m_1}(e_h : e_t) \\[1em] \Rightarrow & \vdash_K \left(\begin{array}{rcl} \mathsf{case}\ e_1\ \mathsf{of} & & \\ \mathsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array}\right) =_{\langle\rangle} \left(\begin{array}{rcl} \mathsf{case}\ \mathsf{l}^{m_1}(e_h : e_t)\ \mathsf{of} & & \\ \mathsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array}\right) \\[1em] \Rightarrow & \vdash_K \left(\begin{array}{rcl} \mathsf{case}\ e_1\ \mathsf{of} & & \\ \mathsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array}\right) =_{\langle\rangle} \mathsf{l}^{m_1} \left(\begin{array}{rcl} \mathsf{case}\ e_h : e_t\ \mathsf{of} & & \\ \mathsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array}\right) \\[1em] \Rightarrow & \vdash_K \left(\begin{array}{rcl} \mathsf{case}\ e_1\ \mathsf{of} & & \\ \mathsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array}\right) =_{\langle\rangle} \mathsf{l}^{m_1}(e_3\{e_h/x, e_t/xs\}) \end{array}$$

The induction hypothesis applied to the second sub-proof gives

$$\vdash_K e_3\{e_h/x, e_t/xs\} =_{\langle\rangle} \mathsf{l}^{m_2}(h)$$

and so by congruence (substituting the right hand side for the left into the previous equation) we conclude that

$$\vdash_K \left(\begin{array}{rcl} \mathsf{case}\ e_1\ \mathsf{of} & & \\ \mathsf{nil} & \Rightarrow & e_2 \\ x : xs & \Rightarrow & e_3 \end{array}\right) =_{\langle\rangle} \mathsf{l}^{m_1}(\mathsf{l}^{m_2}(h)),$$

and the desired result follows from the fact that $\mathsf{l}^{m_1}(\mathsf{l}^{m_2}(h)) \equiv \mathsf{l}^m(h)$. □

Theorem (Precongruence II) If $\tilde{e} \sqsubseteq \tilde{e}'$ for some commonly indexed families of closed expressions $\tilde{e}, \tilde{e}'$, then for all expressions e containing at most variables $\tilde{x}$

$$e\{\tilde{e}/\tilde{x}\} \sqsubseteq e\{\tilde{e}'/\tilde{x}\}$$

PROOF Define the relation

$$\mathcal{I}= \{(e\{\tilde{e}/\tilde{x}\}, e\{\tilde{e}'/\tilde{x}\}) : e \textit{ contains at most variables } \tilde{x}, \tilde{e} \sqsubseteq \tilde{e}'\}$$

It is sufficient to show that $\mathcal{I} \subseteq \mathcal{A}(\mathcal{I})$. Abbreviate substitutions $\{\tilde{e}/\tilde{x}\}$ and $\{\tilde{e}'/\tilde{x}\}$ by σ and σ' respectively. Assume that $e\sigma \xrightarrow{t}_H h$. It will be sufficient to prove that $e\sigma' \xrightarrow{t}_H h'$ for some h' such that $h(\mathcal{R}^{:} \cup \mathcal{R}^{\lambda})h'$. We establish this by induction on the structure of the proof of $e\sigma \rightarrow_H h$, and by cases according to the structure of expression e. The cases for all expressions other than lambda abstraction and application are identical to lemma 6.10. The remaining cases are:

$\boxed{e \equiv \lambda y.b}$ Assume that y is not in the domain of σ (since if it is, we can just rename). By the axiom for lambda expressions, together with its time rule, we have $e\sigma \equiv \lambda y.(b\sigma) \xrightarrow{0}_H \lambda y.(b\sigma)$ and $e\sigma' \equiv \lambda y.(b\sigma') \xrightarrow{0}_H \lambda y.(b\sigma')$. It remains to show that $\lambda y.(b\sigma)\ \mathcal{I}^{\lambda}\ \lambda y.(b\sigma')$. This follows easily from the fact that for all closed a,

$$(\lambda y.(b\sigma))a \equiv ((\lambda y.b)a)\sigma\ \mathcal{I}\ ((\lambda y.b)a)\sigma' \equiv (\lambda y.(b\sigma'))a.$$

$\boxed{e \equiv e_1 e_2}$. By the rule for application it follows that $e_1\sigma \xrightarrow{t_1}_H \lambda y.b$ and that $b\{e_2\sigma/y\} \xrightarrow{t_2}_H h$ for some t_1, t_2 such that $t = 1+t_1+t_2$. From the former judgement the inductive hypothesis allows us to conclude that $e_1\sigma' \xrightarrow{t_1}_H \lambda y.b'$ for some b' such that $\lambda y.b\ \mathcal{I}^{\lambda}\ \lambda y.b'$.

So, for each closed expression a, we have substitutions δ, δ' (with a nonempty common domain, and $\sqsubseteq$-related range) such that $(\lambda y.b)a \equiv d\delta$ and $(\lambda y.b')a \equiv d\delta'$ for some expression d containing at most the variables in the domain of δ. Since a is closed, we can without loss of generality assume that the structure of d satisfies one of the following three cases:

(i) $d \equiv x$, and hence $(\lambda y.b)a \sqsubseteq (\lambda y.b')a$, or

(ii) $d \equiv xa$ and hence $\lambda y.b \sqsubseteq \lambda y.b'$, or

(iii) $d \equiv (\lambda y.c)a$, and hence $(\lambda y.b)\ \mathcal{I}\ (\lambda y.b')$

Now consider the judgement $b\{e_2\sigma/y\} \xrightarrow{t_2}_H h$. By the rule for application this is equivalent to the judgement $(\lambda y.b)e_2\sigma \xrightarrow{1+t_2}_H h$. By the case analysis above, taking $a \equiv e_2\sigma$, either

(i) $(\lambda y.b)a \sqsubseteq (\lambda y.b')a$, and so by the definition of $\sqsubseteq$, $(\lambda y.b)e_2\sigma \xrightarrow{1+t_2}_H h'$ with $h(\mathcal{R}^{:} \cup \mathcal{R}^{\lambda})h'$, and hence $b\{e_2\sigma/y\} \xrightarrow{t_2}_H h'$, *or*

(ii) $(\lambda y.b) \sqsubseteq (\lambda y.b')$ and so by definition $(\lambda y.b)a \sqsubseteq (\lambda y.b')a$ and we argue as above, *or*

(iii) $(\lambda y.b)\ \mathcal{I}\ (\lambda y.b')$ with $(\lambda y.b)a \equiv d\delta$ and $(\lambda y.b')a \equiv d\delta'$. Assume, without loss of generality, that the variables in σ, δ and $\{y\}$ are all distinct. Massaging substitutions (details omitted), it is easy to see that

$$b\{e_2\sigma/y\} \equiv d\delta\{e_2\sigma/y\} \equiv d\{e_2/y\}\delta\sigma\ \mathcal{I}\ d\{e_2/y\}\delta'\sigma' \equiv d\delta\{e_2\sigma/y\} \equiv b'\{e_2\sigma/y\}.$$

Since this shows that $b\{e_2\sigma/y\} \mathcal{I} b'\{e_2\sigma/y\}$, by the induction hypothesis in this case we can also conclude that $b'\{e_2\sigma/y\} \xrightarrow{t_2}_H h'$ for some h' such that $h(\mathcal{R}^{:} \cup \mathcal{R}^{\lambda})h'$.

Using rule for application we conclude that $e\sigma' \xrightarrow{t}_H h'$ for some h' such that $h(\mathcal{R}^{:} \cup \mathcal{R}^{\lambda})h'$, as required. □

B The Largest Cost Precongruence

In this appendix we give the details of the technical development outlined in Section 6.4, which characterises when cost simulation is the largest cost congruence.

Definition **B.1**

$$e_1 \leq_c e_2 \iff \forall C.C[e_1]\ \mathcal{F}(\top)\ C[e_2]$$

□

Theorem **B.2** $\preceq = \leq_c$

Since $\preceq$ is the maximal fixed point of $\mathcal{F}$, to prove the "difficult" half of this equality it is sufficient to show that $\leq_c \subseteq \mathcal{F}(\leq_c)$. However, we have been unable to prove this property by "co-induction" (although the proof for pure operational simulation is straightforward—see *e.g.* [How89]). Instead, we will use an alternative definition of cost simulation as the limit of a descending sequence of relations beginning with $\top$:

Proposition **B.3**

$$\preceq = \bigcap_{n\in\omega} \mathcal{F}^n(\top)$$

where $\mathcal{F}^0$ *is the identity function, and* $\mathcal{F}^{n+1} = \mathcal{F}^n \circ \mathcal{F}$

proof It is sufficient to show that $\mathcal{F}$ is *anticontinuous* (see *e.g.* [CC79]). That is, for every decreasing sequence $R_1 \supseteq R_2 \supseteq \cdots R_n \supseteq \cdots$,

$$\bigcap_n \mathcal{F}(R_n) = \mathcal{F}(\bigcap_n R_n)$$

The $\supseteq$ half follows directly from monotonicity. Now suppose that $e(\bigcap_n \mathcal{F}(R_n))e'$. The only non-trivial case is when $e \xrightarrow{t}_H e_1 : e_2 :$ for each n, we have that $e\mathcal{F}(R_n)e'$, so from the definition of $\mathcal{F}$, we have that $e' \xrightarrow{t}_H e_1' : e_2'$ with $e_i(R_n)e_i'$, and hence that $e_i(\bigcap_n(R_n))e_i'$, giving $e(\mathcal{F}(\bigcap_n R_n))e'$ as required. □

Now we sketch the proof of theorem B.2.

PROOF $\preceq \subseteq \leq_c$ follows from the fact that $\preceq$ is a precongruence, and that $\mathcal{F}$ is monotone. It remains to show that that $\preceq \supseteq \leq_c$. We show the contrapositive: that for all expressions e_1, e_2,

$$e_1 \not\preceq e_2 \Rightarrow e_1 \not\leq_c e_2.$$

For each $n \geq 0$, define $\preceq_n = \mathcal{F}^n(\top)$. Since the $\preceq_n$ form a decreasing chain, by propostion B.3 it follows that there exists a smallest $k > 0$ (and hence a largest $\preceq_k$) such that $e_1 \not\preceq_k e_2$. Call such a $\preceq_k$ the *maximum distinguisher* for (e_1, e_2). Negating the definition of $\leq_c$, we see that $e_1 \not\leq_c e_2$ if there exists a context C such that $C[e_1] \not\preceq_1 C[e_2]$. Call such a context C a *distinguishing context* for (e_1, e_2).

We prove by induction on k that for all e_1, e_2, if $\preceq_k$ the maximum distinguisher for (e_1, e_2) then there exists a distinguishing context C_{e_1,e_2} for (e_1, e_2).

Base: $\preceq_1$ Since $e_1 \not\preceq_1 e_2$, it immediately follows that the simple context [ ] distinguishes (e_1, e_2).

Induction: $\preceq_{k+1}$ Since $\preceq_{k+1}$ is maximum for (e_1, e_2), it follows that $e_1 \xrightarrow{t}_H u_1$ and $e_2 \xrightarrow{t}_H u_2$. Since $k + 1 > 1$, it follows, again by maximality, that $u_1 = p_1 : q_1$ and $u_2 = p_2 : q_2$ for some p_1, q_1, p_2, q_2. Now $\preceq_k$ must be a maximum distingisher for either (p_1, p_2) or (q_1, q_2), since if it were not, $\preceq_{k+1}$ would not be maximal for (e_1, e_2). Suppose that it is maximal for (p_1, p_2) (the other case is similar). By induction, there is a distingishing context C_{p_1,p_2}, so we can easily construct a context

$$\begin{array}{rcl} \multicolumn{3}{l}{\mathsf{case}\ [\ ]\ \mathsf{of}} \\ \mathsf{nil} & \Rightarrow & \mathsf{nil} \\ x : xs & \Rightarrow & C_{p_1,p_2}[x] \end{array}$$

which distinguishes (e_1, e_2). □

DEFINITION **B.4** *Pure cost congruence, $\leq_{pc}$, is defined to be the least relation on closed expressions such that $e_1 \leq_{pc} e_2$ if and only if for all contexts C, whenever $C[e_1] \xrightarrow{t}_H u_1$ then there exists u_2 such that $C[e_2] \xrightarrow{t}_H u_2$.*

□

We will say that a context C *cost distinguishes* a pair of expressions e, e' if $C[e] \not\leq_{pc} C[e']$.

DEFINITION **B.5** *We say that the language satisfies the* **CD** *condition if for all constants c, if $e \rightarrow_H c$ and $e \leq_{pc} e'$ then $e \rightarrow_H c$.*

□

PROPOSITION **B.6** $\leq_c = \leq_{pc} \iff$ *the* **CD** *condition is satisfied.*

PROOF (⇒) Straightforward from the definition of $\leq_c$. (⇐) Clearly $e_1 \leq_c e_2 \Rightarrow e_1 \leq_{pc} e_2$, so it is sufficient to show that $e_1 \not\leq_c e_2 \Rightarrow e_1 \not\leq_{pc} e_2$. Supposing $e_1 \not\leq_c e_2$, then there exists a distinguishing context C for (e_1, e_2). Now suppose that C is *not* sufficient to *cost* distinguish (e_1, e_2). In this case we must have $C[e_1] \overset{t}{\rightarrow}_H u_1$ and $C[e_2] \overset{t}{\rightarrow}_H u_2$ with either

(i) $u_1 = c, u_2 \neq c$, or

(ii) $u_1 = u : u', u_2 = c$ for some constant c.

In the first case $C[e_1] \not\leq_{pc} C[e_2]$ follows from the **CD** condition, and since $\leq_{pc}$ is a precongruence, we must have $e_1 \not\leq_{pc} e_2$. In the second case a context of the form case [ ] of ... can be used to cost distinguish $(C[e_1], C[e_2])$, and hence $e_1 \not\leq_{pc} e_2$ by precongruence. □

Remark The **CD** condition is not particularly strong since it is satisfied if a binary equality test over constants is included as one of the primitive functions; for each constant c, consider the context if $[\,] = c$ then nil else *fail*, where *fail* is any expression lacking a head normal form. On the other hand, the **CD** condition fails if we have two distinct constants in the language, call them *error1* and *error2*, over which all primitive functions are undefined (*i.e.* $\textbf{apply}(p, \ldots, error_i, \ldots)$ is undefined). So we could never distinguish between *error1* and *error2* on the basis of cost alone, but because they are distinct they would not be cost equivalent.

Now we have that cost equivalence is the largest congruence with respect to the time rules for head-normal-form (modulo the partial correctness of the rules, which is reflected by convergence conditions).

COROLLARY **B.7** *If the* **CD** *condition is satisfied, then*

$$e =_{\langle\rangle} e' \iff (\forall C.\ (C[e_1]\downarrow_H \& C[e_2]\downarrow_H) \Rightarrow \langle C[e_1]\rangle^H = \langle C[e_2]\rangle^H)$$

Lemma 6.7, together with the fact that $=_{\langle\rangle}$ is a congruence allows us to extend the above corollary to include evaluation to normal-form, and the $\langle\ \rangle^N$ rules. In fact it is also possible to show that we can completely replace "H" by "N" in the corollary, but this is left as an exercise.