

Operational Theories of Improvement in Functional Languages*

(Extended Abstract)

David Sands[†]

Department of Computing, Imperial College

180 Queens Gate, London SW7 2BZ

email: `ds@uk.ac.ic.doc`

Abstract

In this paper we address the technical foundations essential to the aim of providing a semantic basis for the formal treatment of relative efficiency in functional languages. For a general class of “functional” computation systems, we define a family of improvement preorderings which express, in a variety of ways, when one expression is more efficient than another. The main results of this paper build on Howe’s study of equality in lazy computation systems, and are concerned with the question of when a given improvement relation is subject to the usual forms of (in)equational reasoning (so that, for example, we can improve an expression by improving any sub-expression). For a general class of computation systems we establish conditions on the operators of the language which guarantee that an improvement relation is a precongruence. In addition, for a particular higher-order nonstrict functional language, we show that any improvement relation which satisfies a simple monotonicity condition with respect to the rules of the operational semantics has the desired congruence property.

*A version of this paper will appear in the proceedings of the 1991 Glasgow Functional Programming Workshop, to be published in the *Springer Workshops in Computing* series.

[†]This work was partially funded by ESPRIT BRA 3124, *Semantique*.

1 Introduction

The mathematical tractability of functional programs gives us the ability to show that operations on programs preserve meaning, and has led to much interest in formal methods for program construction. However, although an underlying aim of these activities is to derive efficient programs from inefficient specifications, formal techniques for reasoning about efficiency (and how operations on programs affect it) have received little attention.

This work focuses on the problems of providing semantic notions efficiency improvement for functional languages; possible applications include the construction of program logics for reasoning about the relative efficiency of program fragments, the construction or validation of efficiency-improving transformations, and the development of calculi for reasoning about the execution-related properties of programs.

In this paper we describe a class of improvement relations definable over a variety of functional languages, and address the technical foundations essential to the aim of providing a semantic basis for the formal treatment of efficiency.

Informally, the notions of improvement that we have in mind are binary relations, R , between programs such that pRq if q is at least as good as p , both *observationally* (can match any observable evaluation) and *intensionally* (is operationally “better”). We study this concept in the context of lazy programming languages, where we associate the term “lazy” with an evaluation process which terminates as soon as the outermost *constructor* of the result is known—practically all higher-order functional languages have a lazy component, in the form of a lambda-abstraction (or partial application), where “ λ ” plays the role of a lazy constructor.

With the above applications in mind, what we want is a notion of improvement that satisfies the following: it should be a preordering (*i.e.* p is improved by p , and, if p is improved by q , and q is improved by r , then p is improved by r) and it should be substitutive, *i.e.* we can reason about improvement by substitution of “improvement for improvement” (*c.f.* “substitution of equals for equals”, a.k.a. equational reasoning). These requirements are summarised by saying that improvement should be a *precongruence*.

One way of defining such relations would be to make these conditions true by construction. However, in order to verify improvement “laws”, such a definition is rather unhelpful. A more useful form would be a definition for which reasoning about improvement reduces to reasoning about some basic evaluation relation. However, the basic evaluation relation for a lazy language is too “shallow”, in itself, to capture the behaviour of a term in any context; as a generalisation of “applicative bisimulation” [Abr90] we define the desired improvement relation, *improvement simulation*, by analogy with strong (bi)simulation in process algebra [Mil83]. The remaining problem is to verify, for a given language and a particular notion of improvement, that improvement simulation is a precongruence. This problem is the primary focus of this paper.

In the first part of this paper we define an improvement simulation relation for a broad class of “lazy” languages. The main technical result of this section is that

an improvement simulation is a precongruence exactly when a certain *improvement extensionality* condition holds for each of the operators of the language. This result is based on, and generalises, the main result of Howe’s study of equality in lazy computation systems [How89]; as in [How89], the significance of the condition is that in practice it seems relatively simple to check.

In the second part of the paper we study a particular lazy computation system, a variant of the lazy lambda calculus with constants. Using the result of the previous section we show that any improvement simulation induced by a particular computational property is guaranteed to be a precongruence whenever the property satisfies a simple monotonicity requirement with respect to the rules of the operational semantics. The significance of this characterisation is that many computational properties can be given by simple inductive definitions, for which the required monotonicity condition is easily checked. We illustrate these ideas with some examples, and note that the result is applicable to a variety of languages defined by structural operational semantics, and in particular those defined by the *structured computation system* format of [How91].

In conclusion we consider related works, and possible further developments.

2 Improvement in Lazy Computation Systems

2.1 Lazy Computation Systems

A lazy computation system [How89]¹ is a particular syntactic form well-suited to specifying the syntax of strict and non-strict functional languages (although non-determinism can also be handled in this framework), together with an evaluation relation. We begin with the language. A lazy computation language is reminiscent of the syntax of terms in combinatory reduction systems [Klo80] and Martin-Löf’s type theory, although unlike these, application is not a predefined construct (*c.f.* Aczel’s syntax, [Klo80], remark 1.5).

DEFINITION 2.1 *A lazy computation language L is specified by a triple $(\mathbf{O}, \mathbf{K}, \alpha)$, where \mathbf{O} is a set of operators of the language, \mathbf{K} is a subset of \mathbf{O} , the canonical operators, and α is a function $(\mathbf{O} \rightarrow \{(k_1, \dots, k_n) | n, k_i \geq 0\})$ specifying the arity of each operator. \square*

The arity of an operator is given as a list whose length corresponds to the number of *operands*, and whose respective elements correspond to the number of bound variables that will be “attached” to that operand.

¹We adopt the slightly generalised notion as suggested in [How91].

DEFINITION 2.2 Fix an infinite set V of variables x, y, x_1, x_2 etc. The terms, $T(L)$, ranged over by a, b, a_1, a_2 etc. are formed as follows:

- $V \subseteq T(L)$
- if $F \in \mathbf{O}$, $\alpha(F) = (k_1, \dots, k_n)$ and $a_1, \dots, a_n \in T(L)$, then

$$F((\bar{x}_1).a_1, \dots, (\bar{x}_n).a_n) \in T(L)$$

where each \bar{x}_i is a list of k_i variables.

□

For example, in an ML-like language, a list case-expression of the form:

```

case E of nil => A
      (cons x y) => B

```

could be expressed in a lazy computation language, using an operator *case* of arity $(0, 0, 2)$, as a term of the form $case(E, A, (x, y).B)$, where, as in the sequel, we abbreviate operands of the form $(\bar{x}).e$ as simply e . Some other notational conventions used are summarised below.

NOTATION

Variables A list of zero or more variables x_1, \dots, x_n will often be denoted \bar{x} , and similarly for a list of operands. Variable-binding structure in $T(L)$ is given by specifying that in an operand of the form $(\bar{x}).b$, the free occurrences of variables \bar{x} in b are bound. The closed terms, written $T(L)^\circ$, are defined to be the terms with no free variables.

Substitution We use the notation $a\{b_1, \dots, b_n/x_1, \dots, x_n\}$ to mean term a with free occurrences of x_1, \dots, x_n simultaneously replaced by the terms b_1, \dots, b_n . A formal definition of substitution is omitted. Let \equiv denote syntactic equality on terms, which will be taken to include pairs of terms which are the same up to renaming of bound variables (alpha-conversion).

Relations If R is a binary relation on $T(L)^\circ$ then we extend R to $T(L)$ by: $a R b$ if and only if $a\sigma R b\sigma$ for all closing substitutions σ . If we have a relation S on $T(L)$, extend S to operands by:

$$(\bar{x}_1 \dots \bar{x}_n).b S (\bar{x}'_1 \dots \bar{x}'_n).b' \iff b\{z_1, \dots, z_n/\bar{x}_1, \dots, \bar{x}_n\} S b'\{z_1, \dots, z_n/\bar{x}'_1, \dots, \bar{x}'_n\}$$

where z_1, \dots, z_n are not free in b, b' . We will use variables e, e_1, e_2 etc. to range over operands. For commonly-indexed lists of operands \bar{e}, \bar{e}' we will write $\bar{e} R \bar{e}'$ to mean $\forall i. e_i R e'_i$. Finally, let $;$ denote relation composition, so $a R; S c$ if and only if there exists a b such that $a R b$ and $b S c$. □

Evaluation

A lazy computation system is essentially an evaluation relation which relates closed terms (of a particular lazy computation language) to their “values”. The values are the *canonical terms*, which are defined to be any closed term of the form $C(\bar{e})$ where $C \in \mathbf{K}$. The evaluation relation is expressed as a family of relations over which we have a well-founded ordering which encapsulates notion of (induction on) the “size” of a computation.

DEFINITION 2.3 *A lazy computation system is a pair $(L, \{\xrightarrow{w}\}_{w \in W})$ where L is a lazy computation language, $\{\xrightarrow{w}\}$ is a family of binary relations indexed by a well-founded ordering $<_W$, between closed terms and canonical terms. \square*

EXAMPLE 2.4 The lazy lambda calculus [Abr90, Ong88] shares the syntax of the pure untyped lambda calculus, but has an operational semantics given by an evaluation relation $M \Downarrow N$ (“ M converges to principal weak head normal form N ”) defined inductively over closed lambda terms as follows:

$$\bullet \lambda x.M \Downarrow \lambda x.M \quad \bullet \frac{M \Downarrow \lambda x.P \quad P\{N/x\} \Downarrow Q}{MN \Downarrow Q}$$

We can recast this as a lazy computation system as follows: writing $\lambda x.M$ as $\lambda((x).M)$, and MN as $@(M, N)$, define $\mathbf{O} = \{\lambda, @\}$, $\mathbf{K} = \{\lambda\}$ with $\alpha(\lambda) = (1)$ and $\alpha(@) = (0, 0)$. Now take the set W to be the set of proofs built with the above rules, with the well-founded ordering being the “subproof” relation. Then define $M \xrightarrow{w} N$ whenever w is a proof of $M \Downarrow N$. \square

In the remainder of the paper it will be convenient to define the following abbreviations on evaluation judgements:

- $a \longrightarrow b \iff \exists w. a \xrightarrow{w} b$,
- $a \Downarrow^w \iff \exists b, w. a \xrightarrow{w} b$,
- $a \uparrow \iff \neg(a \Downarrow^w)$.

2.2 Improvement Simulation

In our consideration of lazy computation systems, the choice of set W will be of particular importance, since it will not only be used to formalise a general notion of induction on the “size” of a computation (via the well-founded ordering, $<_W$), but also to capture information pertaining to the *computational* properties of evaluation. In what follows we will consider additional structure on the set W provided by a *computation preorder*, \succeq . Intuitively, $w \succeq w'$ means that some computational property associated with w is improved by the computation property associated with w' . So, for example, W might be the set of possible computation traces (e.g. sequences of abstract machine states), then we could define $w' <_W w$ if w' is a strict sub-trace of w , and $w \succeq w'$ if w' contains fewer evaluations of arithmetic operators.

Roughly speaking, the basic notion of “improvement” between closed terms is the largest relation that satisfies the following:

if A is “improved by” B , then whenever A can be evaluated to some canonical term, then B can be evaluated in an “improved fashion” (\succeq) to a term with a matching outermost canonical operator, and “improved” operands.

We formalise this notion of improvement with the following definitions:

DEFINITION 2.5 For any given improvement preorder, \succeq , define the monotone function $\llbracket _ \rrbracket_{\succeq} \in T(L)^\circ \times T(L)^\circ \rightarrow T(L)^\circ \times T(L)^\circ$ as follows. For any binary relation R on closed terms, let $\llbracket R \rrbracket_{\succeq}$ be the least relation on closed terms such that $a_1 \llbracket R \rrbracket_{\succeq} a_2$ if

whenever $a_1 \xrightarrow{w_1} C(\bar{e}_1)$ then there exist $w_2, C(\bar{e}_2)$ such that $a_2 \xrightarrow{w_2} C(\bar{e}_2)$, with $\bar{e}_1 R \bar{e}_2$ and $w_1 \succeq w_2$

□

DEFINITION 2.6

A relation R is an improvement simulation (with respect to \succeq) if $R \subseteq \llbracket R \rrbracket_{\succeq}$ □

DEFINITION 2.7 Let \triangleleft_{\succeq} denote the largest improvement simulation. □

Since $\llbracket _ \rrbracket_{\succeq}$ is a monotone function on the powerset lattice of all binary relations on $T(L)^\circ$ then by the Knaster-Tarski fixed point theorem it has a greatest fixed point (given by $\bigcup \{R \mid R \subseteq \llbracket R \rrbracket_{\succeq}\}$, i.e. the union of all improvement simulations), so the above definition is well defined with \triangleleft_{\succeq} equal to this greatest fixed point. We will refer to this relation as *the* improvement simulation. The importance of this maximal fixed point definition is that it comes with a useful proof technique, sometimes referred to in the context of process algebra as “Park Induction” (in reference to [Par80]), or in a more general setting as “co-induction” ([MT90]):

To show that $R \subseteq \triangleleft_{\succeq}$, for some binary relation R on $T(L)^\circ$, it is sufficient to show that R is an improvement simulation.

So, in order to show that two closed terms are related by \triangleleft_{\succeq} it is sufficient to exhibit any improvement simulation which relates them.

In what follows we will routinely omit the \succeq -subscripts, unless we are specifically considering different computation preorderings over the same lazy computation system.

PROPOSITION 2.8 If R and S are binary relations on $T(L)^\circ$ then $\llbracket R \rrbracket; \llbracket S \rrbracket \subseteq \llbracket R; S \rrbracket$.

PROOF Suppose $a \llbracket R \rrbracket; \llbracket S \rrbracket c$. If $a \uparrow$ then for all binary relations P on $T(L)^\circ$, and all $d \in T(L)^\circ$, $a \llbracket P \rrbracket d$, so in particular $a \llbracket R; S \rrbracket c$. Otherwise, if $a \xrightarrow{w_1} C(\bar{e}_1)$ then $\exists b \in T(L)^\circ$. $b \xrightarrow{w_2} C(\bar{e}_2)$, with $w_1 \succeq w_2$ and $\bar{e}_1 R \bar{e}_2$, and such that $c \xrightarrow{w_3} C(\bar{e}_3)$ with $w_2 \succeq w_3$ and $\bar{e}_2 S \bar{e}_3$. Transitivity of \succeq gives $w_1 \succeq w_3$, so $a \llbracket R; S \rrbracket c$ as required. □

It follows from this result that if R and S are improvement simulations then so is $R; S$.

PROPOSITION 2.9 \triangleleft is a preorder.

PROOF It is easily established that syntactic equivalence (\equiv) on closed terms is an improvement simulation², and hence that \triangleleft is reflexive. Transitivity follows from the fact that since \triangleleft is an improvement simulation, so is $\triangleleft; \triangleleft$. \square

2.3 Improvement Extensionality

We want to establish conditions which guarantee that \triangleleft is a precongruence, i.e. that $F(\bar{e}) \triangleleft F(\bar{e}')$ whenever $\bar{e} \triangleleft \bar{e}'$. A simple inductive argument on any term a establishes that this is equivalent to the requirement that \triangleleft be closed under substitution, i.e. $a\{\bar{e}/\bar{x}\} \triangleleft a\{\bar{e}'/\bar{x}\}$ whenever $\bar{e} \triangleleft \bar{e}'$.

For a particular language, one method for giving a direct proof that the (bi)simulation is substitutive is to form an intermediate relation which is substitutive by construction, and to show that this is equal to the (bi)simulation relation in question.

The improvement-extensionality condition which we develop is expressed in terms of conditions on just such an intermediate relation. Here we adopt the intermediate relation given in [How89]³, albeit defined relative to a different maximal simulation relation:

DEFINITION 2.10 (Howe) Define $a \triangleleft^* b$ inductively on the syntax of a :

$$\begin{aligned} x \triangleleft^* b & \text{ if } x \triangleleft b \\ F(\bar{e}) \triangleleft^* b & \text{ if } \exists F(\bar{e}'). \bar{e} \triangleleft^* \bar{e}' \text{ and } F(\bar{e}') \triangleleft b \end{aligned}$$

\square

The definition of \triangleleft^* is best understood in terms of the properties that it satisfies, viz.,

²We assume that the evaluation relation respects syntactic equivalence.

³Technical remark: in the case where the language contains expressions with bound variables the exact choice of such an intermediate relation seems crucial to such a direct proof of congruence. One possibility is to take the minimal substitution-closure of the simulation relation (as in [San91] for a specific improvement simulation); for languages with more complex forms of abstraction (the paradigmatic example being lambda abstraction), a more elaborate relation seems necessary, such as the transitive closure of this substitution-closure (as in [Tho90]). In this general setting, the form of Howe's intermediate relation can be shown somewhere between these definitions (using proposition 2.11), although in the cases where simulation is closed under substitution, these definitions all coincide with the simulation relation itself.

PROPOSITION 2.11 \sqsubseteq^* is the smallest binary relation on $T(L)$ such that:

- (i) $\sqsubseteq \subseteq \sqsubseteq^*$, i.e., $a \sqsubseteq b \Rightarrow a \sqsubseteq^* b$
- (ii) $F(\bar{e}) \sqsubseteq^* F(\bar{e}')$ if $\bar{e} \sqsubseteq^* \bar{e}'$
- (iii) \sqsubseteq^* ; $\sqsubseteq = \sqsubseteq^*$, i.e., $a \sqsubseteq^* b$ and $b \sqsubseteq c$ imply $a \sqsubseteq^* c$

PROOF Properties (i)–(iii) follow easily from the definition 2.10, together with the fact that \sqsubseteq is reflexive. Binary relations on $T(L)$ which satisfy (ii) and (iii) form a sub-lattice of the powerset lattice of all binary relations on $T(L)$, so a smallest relation satisfying (i)–(iii) exists (and is given by the intersection of all relations above \sqsubseteq in this sub-lattice), and so it remains to show that \sqsubseteq^* is minimal. Suppose that \sqsubseteq^- satisfies (i)–(iii), and that $\sqsubseteq \subseteq \sqsubseteq^- \subseteq \sqsubseteq^*$. Now suppose that $a \sqsubseteq^* b$. We prove by induction on a that $a \sqsubseteq^- b$ and hence that $\sqsubseteq^- = \sqsubseteq^*$.

($a \equiv x$): By definition, $x \sqsubseteq^* b$ implies $x \sqsubseteq b$, and so by (i), $x \sqsubseteq^- b$.

($a \equiv F(\bar{e})$): By definition there exists $F(\bar{e}')$ such that $\bar{e} \sqsubseteq^* \bar{e}'$ and $F(\bar{e}') \sqsubseteq b$. The induction hypothesis gives $\bar{e} \sqsubseteq^- \bar{e}'$, so by property (ii) $F(\bar{e}) \sqsubseteq^- F(\bar{e}')$. We conclude by property (iii) that $F(\bar{e}) \sqsubseteq^- b$ as required. □

One particularly useful consequence of the definition of \sqsubseteq^* is the following:

PROPOSITION 2.12 (Howe) If $a \sqsubseteq^* a'$ and $b \sqsubseteq^* b'$ then $b\{a/x\} \sqsubseteq^* b'\{a'/x\}$.

PROOF (By induction on b) □

To prove that \sqsubseteq is a precongruence it is necessary and sufficient to prove that $F(\bar{e}) \sqsubseteq F(\bar{e}')$ if $\bar{e} \sqsubseteq \bar{e}'$, since this amounts to proving that $\sqsubseteq = \sqsubseteq^*$. Therefore to show that \sqsubseteq is substitutive it is necessary and sufficient to show that $\sqsubseteq^* \subseteq \sqsubseteq$. Since \sqsubseteq is the maximal fixed point of $\llbracket \cdot \rrbracket$, it is sufficient to prove that \sqsubseteq^* is an improvement simulation, i.e. that $\sqsubseteq^* \subseteq \llbracket \sqsubseteq^* \rrbracket$.

Following [How89] we give a characterisation of when \sqsubseteq^* is a precongruence, in terms of a condition on the operators of the language (analogous to Howe's extensionality condition).

DEFINITION 2.13 An operator F is improvement extensional if for any closed terms $F(\bar{e}_1)$, $F(\bar{e}_2)$, whenever

- (i) $\bar{e}_1 \sqsubseteq^* \bar{e}_2$,
- (ii) $F(\bar{e}_1) \downarrow^{w_1}$, and
- (iii) $\forall a_1, a_2 \in T(L)^\circ$, if $a_1 \downarrow^w$ with $w <_W w_1$, then $a_1 \sqsubseteq^* a_2 \Rightarrow a_1 \llbracket \sqsubseteq^* \rrbracket a_2$

then $F(\bar{e}_1) \llbracket \sqsubseteq^* \rrbracket F(\bar{e}_2)$. □

Now if we say that a lazy computation system is improvement extensional whenever all of its operators are, we have the following result:

THEOREM 2.14 *A lazy computation system is improvement extensional if and only if \triangleleft is a precongruence.*

PROOF

(\Leftarrow) If \triangleleft is a precongruence then from proposition 2.11 it is easily shown that $\triangleleft = \triangleleft^*$. Improvement-extensionality then follows from the definition of \triangleleft .

(\Rightarrow) It is sufficient to show that $\triangleleft^* \subseteq \langle\langle \triangleleft^* \rangle\rangle$, i.e. that

$$\forall F(\bar{e}), b \in T(L)^\circ, F(\bar{e}) \triangleleft^* b \Rightarrow F(\bar{e}) \langle\langle \triangleleft^* \rangle\rangle b.$$

In the case where $F(\bar{e}) \uparrow$ then $F(\bar{e}) \langle\langle \triangleleft^* \rangle\rangle b$ immediately follows. Otherwise, suppose $F(\bar{e}) \downarrow^{w_1}$, then we prove by induction on w_1 that $F(\bar{e}) \langle\langle \triangleleft^* \rangle\rangle b$. By the definition of \triangleleft^* we have a term $F(\bar{e}')$ such that $\bar{e} \triangleleft^* \bar{e}'$ (and hence that $F(\bar{e}) \triangleleft^* F(\bar{e}')$), and $F(\bar{e}') \triangleleft b$. By proposition 2.12 we can assume that $F(\bar{e}')$ is closed. The induction hypothesis coincides with condition (iii) in the definition of improvement extensionality, so we conclude that $F(\bar{e}) \langle\langle \triangleleft^* \rangle\rangle F(\bar{e}')$. Then we have that $F(\bar{e}) \langle\langle \triangleleft^* \rangle\rangle \triangleleft b$. Now since

$$\begin{aligned} \langle\langle \triangleleft^* \rangle\rangle; \triangleleft &= \langle\langle \triangleleft^* \rangle\rangle; \langle\langle \triangleleft \rangle\rangle \\ &\subseteq \langle\langle \triangleleft^*; \triangleleft \rangle\rangle && \text{(Prop. 2.8)} \\ &= \langle\langle \triangleleft^* \rangle\rangle && \text{(Prop. 2.11)} \end{aligned}$$

we conclude that $F(\bar{e}) \langle\langle \triangleleft^* \rangle\rangle b$ as required. □

3 Monotone Improvement in the Lazy Lambda Calculus

Unlike the standard notions of operational approximation and equivalence, it is likely that we may wish to consider a variety of improvement relations over a fixed lazy computation system. In particular we are interested in various notions of improvement over variants of the lazy lambda calculus, which include constants and lazy constructors, as a prototypical non-strict functional language.

For such languages we have found that the improvement extensionality condition holds, and is straightforward to check, for a number of simple notions of improvement. However it is somewhat tedious to re-establish improvement extensionality for each minor variant of the computation ordering⁴. In this section we give a

⁴Notwithstanding certain simple closure conditions for precongruent improvement simulations, i.e. that if \triangleleft_I and \triangleleft_J are precongruences, then so are the improvement simulations \triangleleft_{I-1} and $\triangleleft_{I \cap J}$

structured operational semantics for a lazy lambda calculus (which defines a lazy computation system), and define a simple monotonicity condition on computation preorderings. Using the results of the previous section it is possible to show that any improvement simulation induced by a monotone computation order is guaranteed to be a precongruence.

Although we present this result in the context of a specific computation system, we believe that it is widely applicable; as evidence, we note that it applies to the class of *structured computation systems* defined by Howe [How91].

To illustrate the usefulness of this characterisation, we give some examples of monotone computation orders.

3.1 A Lazy Lambda Calculus with Constants

We take the lazy computation language defined in example 2.4, and add a lazy list-constructor, **cons** (a canonical operator of arity $(0,0)$), together with a list case expression built with a noncanonical operator, **case** (of arity $(0,0,2)$). We add some constants, $c \in \text{Const}$, (i.e. canonical operators of arity $()$) including the empty list, **nil**, and some strict primitive functions, $p \in \text{Prim}$, over these constants. For convenience we write $\lambda((x).a)$ as $\lambda x.a$, application as $a b$, constants $c()$ as simply c , and operands of the form $() . b$ as just b .

A lazy computation system for this language is then specified by first giving an operational semantics: figure 1 defines rule schemas which allow us to conclude evaluation judgements of the form $a \Downarrow b$ (“closed term a converges to principal weak head normal form b ”). We will treat rules c and p are shorthand for a set of rules,

$$\begin{array}{c}
 \frac{}{\lambda x.b \Downarrow \lambda x.b} \lambda \quad \frac{}{\text{cons}(e_1, e_2) \Downarrow \text{cons}(e_1, e_2)} \text{cons} \quad \frac{}{c \Downarrow c} c \\
 \\
 \frac{e_1 \Downarrow \lambda x.b \quad b\{e_2/x\} \Downarrow a}{e_1 e_2 \Downarrow a} \textcircled{\text{C}} \\
 \\
 \frac{e_1 \Downarrow \text{nil} \quad e_2 \Downarrow a}{\text{case}(e_1, e_2, (x, y).e_3) \Downarrow a} \text{case1} \quad \frac{e_1 \Downarrow \text{cons}(e_h, e_t) \quad e_3\{e_h, e_t/x, y\} \Downarrow a}{\text{case}(e_1, e_2, (x, y).e_3) \Downarrow a} \text{case2} \\
 \\
 \frac{e_1 \Downarrow c_1 \cdots e_n \Downarrow c_n \quad (\llbracket p \rrbracket(c_1, \dots, c_n) = a)}{p(e_1, \dots, e_n) \Downarrow a} p
 \end{array}$$

Figure 1: Operational Semantics

one for each $c \in \text{Const}$ and $p \in \text{Prim}$; informally we assume that each primitive function p of n arguments is given meaning by some partial map $\llbracket p \rrbracket$, from a tuple of n constants to a constant. Now we define the lazy computation system as in example 2.4: let W be the set of closed proofs (derivations) in the above system, i.e. proofs of judgements of the form $a \Downarrow b$, where a and b are closed terms. Define

the family $\{\xrightarrow{\Delta}\}_{\Delta \in W}$ as $a \xrightarrow{\Delta} b$ whenever Δ is a proof of the judgement $a \Downarrow b$. The well-founded ordering $<_W$ is taken to be the immediate-subproof relation.

3.2 Monotone Computation Preorders

DEFINITION 3.1 *A computation preorder \succeq is rule-monotone if for all judgements S, S' , and all $\Delta_1, \dots, \Delta_n, \Delta'_1, \dots, \Delta'_n \in W$, if*

$$\Phi = \frac{\Delta_1, \dots, \Delta_n}{S} \mathbf{r} \quad \Phi' = \frac{\Delta'_1, \dots, \Delta'_n}{S'} \mathbf{r}$$

are proofs (by rule \mathbf{r}), and if for each i , $\Delta_i \succeq \Delta'_i$, then $\Phi \succeq \Phi'$. \square

THEOREM 3.2 *If \succeq is rule-monotone, then \triangleleft_{\succeq} is a precongruence.*

PROOF Assume that \succeq is rule-monotone, then by theorem 2.14 it is necessary and sufficient to show that each operator is improvement extensional. Details not included for lack of space, but fairly straightforward: the case for $\textcircled{\ast}$ requires prop. 2.12, and the lack of restrictions on the exact set primitive functions relies on the fact that they are partial maps between constants, not arbitrary terms. \square

It turns out that this result generalises to a large class of lazy computation systems, namely those definable by the so-called *structured computation systems* of [How91]. A structured computation system is a lazy computation system defined by SOS-style rules which obey certain simple syntactic restrictions (basically, a requirement that meta-variables in the antecedents of a rule do not form cyclic dependencies, and that the expression schemes should be sufficiently “simple”). Howe shows that any structured computation system is extensional. Analogously, a simple adaptation of this proof yields the result that a structured computation system is improvement-extensional with respect to any monotone computation preorder. The details of this result will be reported elsewhere.

Deriving Computation Orders from Property Maps

In an abstract setting, the actual choice computation preorder, \succeq , is somewhat arbitrary. However, in choosing W to be the entire derivation tree of an evaluation judgement, and by identifying computational properties with properties of the proof tree, we can obtain definitions of various computation preorders which are, to some extent⁵, correct with respect to the operational semantics.

For example, suppose the constants of the above language include the integers, and we have integer multiplication as a primitive function; then one notion of improvement, could be based on the number of multiplications performed, viz. the number of instances of the rule p for which $p \equiv \text{multiply}$ in respective proofs.

⁵In general, this assumes that we can identify the *computation process* and the *proof tree*. For some languages (unlike this one and most other functional languages we have considered) the proof-construction process (*i.e.* interpretation) may involve backtracking, in which case this computational information would not be present in the final proof.

Following this approach, it is convenient to define various computation orders by providing a map from proofs to a set of “computational properties”. The set of computational properties will represent some aspect of resource-use, and therefore will have some natural ordering. We require that this should be a preorder, although in practice, for a sufficiently abstract notion of property we would expect a partial order.

If we call a function from W to any preorder (P, \sqsubseteq) a *property map*, then any property map $f \in (W \rightarrow (P, \sqsubseteq))$ induces an improvement preordering on W , \succeq_f by:

$$\Delta_1 \succeq_f \Delta_2 \iff f(\Delta_1) \sqsupseteq f(\Delta_2).$$

By a small abuse of notation, will denote the corresponding improvement simulation by \triangleleft_f . A natural way of defining a property map is by induction on proofs, by cases according to the last rule applied. For such definitions, the rule-monotonicity property is usually straightforward to check.

3.3 Examples

In the remainder of this section we give some examples of rule-monotone improvement orderings, defined in terms of various property maps.

- (i) The property map, $W \rightarrow \{*\}$ gives (trivially) a rule-monotone improvement ordering, and the corresponding notion of improvement simulation is essentially just the usual “applicative simulation” [Abr90].
- (ii) Define the property map **@time** $\in (W \rightarrow (\mathbf{N}, \leq))$, as

$$\mathbf{@time}(\Delta) = \text{the number of instances of the @-rule in } \Delta.$$

This gives an improvement simulation simple measure of sequential time complexity.

- (iii) The property map **callset** $\in (W \rightarrow (\mathcal{P}(\text{Prim}), \subseteq))$ gives the set of primitive functions called in a given proof (ordered by subset inclusion).
- (iv) Define the property map **depth** $\in (W \rightarrow (\mathbf{N}, \leq))$ inductively as follows:

$$\mathbf{depth} \left(\frac{\Delta_1, \dots, \Delta_n}{S} \mathbf{r} \right) = \begin{cases} 1 + \text{Max}(\mathbf{depth}(\Delta_1), \dots, \mathbf{depth}(\Delta_n)) & \text{if } \mathbf{r} \in \text{Prim} \\ 1 + \mathbf{depth}(\Delta_1) + \dots + \mathbf{depth}(\Delta_n) & \text{otherwise.} \end{cases}$$

A proof that $\succeq_{\mathbf{depth}}$ is rule-monotone is left as an exercise for the interested reader. The depth property gives an improvement simulation, $\triangleleft_{\mathbf{depth}}$, which describes improvement of parallelism for a simple parallel implementation of the semantics in which the arguments to primitive functions are evaluated in parallel, and computation is sequential otherwise. A theory of improvement for this relation may contain laws such as

$$\begin{aligned} & \mathbf{case}(x, p(x_1, \dots, x_n), (y_1, y_2).p(x_1, \dots, x_k, e, x_{k+2}, \dots, x_n)) \\ & \triangleleft_{\mathbf{depth}} p(x_1, \dots, x_k, \mathbf{case}(x, x_{k+1}, (y_1, y_2).e), x_{k+2}, \dots, x_n) \end{aligned}$$

which can be verified by the Park induction principal, *i.e.* by constructing a relation which contains every ground instance, and by showing that this relation is an improvement simulation (definition 2.6)

4 Conclusions

4.1 Related Work

The class improvement preorders developed here generalises *cost equivalence* and *program refinement* developed in the author's previous work [San90a, San91] (the above cost equivalence can be generated as the derived the equivalence relations $\triangleleft \cap \triangleright$). The motivation for the development of cost equivalence was to provide substitutive equivalences with respect to a simple calculus for time analysis—for the relationship to other work on time analysis of programs in lazy languages see [San90a, San91]. From the development of cost equivalence a program refinement relation arose naturally. This is a simple instance of the improvement simulation in a language which can easily be shown to be improvement extensional.

In [Tal85] a class preorderings called *comparison relations* are considered for a side-effect free lisp derivative. The class of comparison relations is sufficiently general to express relations analogous to the kinds of improvement relation considered here, and indeed it is suggested that certain comparison relations could be developed to provide soundness and improvement proofs for program transformation laws. Only the maximal comparison relations (essentially, the standard operational approximation and equivalence relations) are considered in detail, and the results are specific to a particular language and model of computation.

As mentioned previously, section 2 is inspired by, and is a generalisation⁶ of [How89], although by contrast the motivation in Howe's case is in connection with the study of open-endedness and type-free reasoning in type theories. Astesiano *et al* [AGR88] consider a highly parameterised definition of bisimulation which also could have been adapted for our purpose, although as far as we know there is no general study of congruence for such relations.

The form of simulation we introduce is related to Thomsen's extended bisimulation in CCS induced by a preorder on *actions* [Tho87], particularly if we consider a notion of improvement bisimulation as $\triangleleft_I \cap \triangleright_{I^{-1}}$. More recently, Arun-Kumar and Hennessy [AKH91] have considered a specific efficiency preorder for CCS processes based on the number of internal (silent) actions performed by a process, and expressed as a refinement of weak bisimulation. They prove that it is preserved by all CCS contexts except summation, and develop a proof system for finite processes.

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 a notion

⁶If we take \succeq to be the whole of $W \times W$ then the improvement extensionality condition can be shown to correspond exactly to the extensionality condition of [How89].

computation for which program equivalence also captures equivalence of resource-requirements (and hence corresponds with the kind of cost equivalences generated by $\langle[-]\rangle$). 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). This is analogous to the approach in [San90a, San91] 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. 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. In addition, Gurr gives an account of a semantic formalisation of non-exact complexity, which although outside the scope of our study, begins naturally by introducing a partial ordering on resources.

4.2 Further Work

The main limitation of this work is in its treatment of nonstrict functional languages. In the example of the lazy lambda calculus the operational semantics describes a *call-by-name* evaluation mechanism, when most actual implementations of lazy evaluation use *call-by-need*. In the case of standard notions of operational equivalence and approximation this is not a problem, but in the case of improvement simulations, this becomes a major shortcoming, since we cannot safely derive appropriate improvement orderings from the model as we did in section 3, since the model no longer reflects many operational properties accurately. It remains to be investigated whether lazy computation systems with “correct” notions of improvement can be derived from appropriate call-by-need operational semantics.

As an alternative patch consider the following: suppose we take a call-by-name notion of improvement simulation (which is a precongruence), and by some indirect means identify some subset of improvement simulation which is valid in a call-by-need model. (What we have in mind is the use of some notion of (sub)linearity or single-threadedness to identify when call-by-need and call-by-name are essentially the same.) Then we conjecture, for many simple notions of improvement (such as the examples in the previous section) that we can safely (w.r.t. call-by-need improvement) close-up this subset under substitution into any context, not just the linear ones. It is likely that notions such as strictness may, as in [San90b], have an important role to play in such an approach.

Acknowledgements Thanks to numerous members of the *Semantique* project for constructive feedback on the material presented here, and to the referees for their helpful comments on an earlier draft.

References

- [Abr90] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.
- [AGR88] E. Astesiano, A. Giovini, and G. Reggio. Generalized bisimulation in relational specifications. In *STACS*. LNCS 294, 1988.
- [AKH91] S. Arun-Kumar and M. Hennessy. An efficiency preorder for processes. In *TACS*. LNCS 526, 1991.
- [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).
- [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.
- [Klo80] J.W. Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. Mathematischen Centrum, 413 Kruislaan, Amsterdam, 1980.
- [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.
- [MT90] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 1990. (to appear).
- [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.
- [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.

- [San91] D. Sands. Time analysis, cost equivalence and program refinement. In *Proceedings of the Eleventh Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 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.
- [Tho87] B. Thomsen. An extended bisimulation induced by a preorder on actions. Master's thesis, Aalborg University, Institute of Electronic Systems, 1987.
- [Tho90] B. Thomsen. *Calculi for Higher Order Communicating Systems*. PhD thesis, Imperial College, 1990.