Improvement Theory and its Applications

David Sands Chalmers University and the University of Göteborg

Abstract

An *improvement theory* is a variant of the standard theories of observational approximation (or equivalence) in which the basic observations made of a functional program's execution include some intensional information about, for example, the program's computational cost. One program is an improvement of another if its execution is more efficient in any program context. In this article we give an overview of our work on the theory and applications of improvement. Applications include reasoning about time properties of functional programs, and proving the correctness of program transformation methods. We also introduce a new application, in the form of some bisimulation-like proof techniques for equivalence, with something of the flavour of Sangiorgi's "bisimulation up-to expansion and context".

1 Introduction

An *improvement theory* is a variant of the standard theories of observational approximation (or equivalence) in which the basic observations made of a program's execution include some intensional information about, for example, the program's computational cost. One program is an improvement of another if its execution is more efficient in any program context.

In this article we survey a number of applications of a particular improvement theory developed for a small untyped functional language.

The initial motivation for considering such relations was to support a simple calculus for reasoning about time behaviour of non-strict functional programs (Sands 1993). From this work it was a natural step to consider a more general class of improvement theories for functional languages, providing contextually robust theories of optimisation (Sands 1991).

However, the real pay-off for this study is not in reasoning about efficiency properties *per se*, but in reasoning about equivalence: *The Improvement Theorem* (Sands 1995b) provides a condition for the total correctness of transformations on recursive programs. Roughly speaking, the Improvement Theorem says that if the local steps of a transformation are contained in a particular improvement theory, then correctness of the transformation follows. This result has furnished:

[0]To appear: Eds. A. Gordon and A Pitts, *Higher-Order Operational Techniques in Semantics*, Cambridge University Press, 1997.

- an equivalence-preserving variant of the classic *Unfold-Fold* transformation in a higher-order functional languages (Sands 1995b), and
- the first correctness proofs for a number of well-known transformation methods(Sands 1996a) (see also (Sands 1995a)).

New work reported in this article is the development of some proof techniques for equivalence based on the improvement theory, with something of the flavour of Sangiorgi's "bisimulation up-to expansion and context".

Overview

In the next section we introduce the syntax and semantics of the language used in the remainder of the article. **Section 3** introduces a simple improvement theory for the language based on observing the number of recursive calls made during evaluation. We introduce proof techniques for establishing improvement, and introduce the *tick algebra* which facilitates the calculation of simple improvement properties. **Section 4** considers the application which originally motivated the study of improvement relations, namely the problem of *time analysis* of functional programs. **Section 5** describes a correctness problem in program transformation. We present The Improvement Theorem, which shows, in principle, how the problem can be solved by using improvement. **Section 6** illustrates this application by considering a particular program transformation method. Finally, **Section 7** introduces and illustrates some proof techniques for the equivalence of programs which can be derived from the Improvement Theorem.

2 Preliminaries

We summarise some of the notation used in specifying the language and its operational semantics. The subject of this study will be an untyped higher-order non-strict functional language with lazy data-constructors.

We assume a flat collection of mutually recursive definitions of constants, each of the form $\mathsf{f} \triangleq e_{\mathsf{f}}$, where e_{f} is a closed expression. Symbols f, g, $\mathsf{h} \ldots$, range over the constants, $f, h, x, y, z \ldots$ over variables and e, e_1, $e_2 \ldots$ over expressions. The syntax of expressions is as follows:

$$\begin{array}{rcll}
e & = & x \mid \mathsf{f} \mid e_1\, e_2 & \text{(Variable; Recursive constant; Application)} \\
 & \mid & \lambda x.e & \text{(Lambda-abstraction)} \\
 & \mid & \mathsf{case}\ e\ \mathsf{of} & \text{(Case expressions)} \\
 & & \quad pat_1 \Rightarrow e_1 \ldots pat_n \Rightarrow e_n & \\
 & \mid & c(\bar{e}) & \text{(Constructor expressions and constants)} \\
 & \mid & p(\bar{e}) & \text{(Strict primitive functions)} \\
 & & & \\
pat & = & c(\bar{x}) & \text{(Patterns)}
\end{array}$$

Somewhat improperly we will refer to the constants f, g, etc., as function names, or function calls. In general they need not be functions however.

We assume that each constructor c and each primitive function p has a fixed arity, and that the constructors include constants (i.e. constructors of arity zero). Constants will be written as c rather than $c()$. The primitives and constructors are not curried – they cannot be written without their full complement of operands. We assume that the primitive functions map constants to constants.

We can assume that the case expressions are defined for any subset of patterns $\{pat_1 \ldots pat_n\}$ such that the constructors of the patterns are distinct. A variable can occur at most once in a given pattern; the number of variables must match the arity of the constructor, and these variables are considered to be bound in the corresponding branch of the case-expression.

A list of zero or more expressions $e_1, \ldots e_n$ will often be denoted $\bar{e}$. Application, as is usual, associates to the left, so $((\cdots(e_0 e_1)\ldots)e_n)$ may be written as $e_0\ e_1 \ldots e_n$, and further abbreviated to $e_0\ \bar{e}$.

The expression written $e\{\bar{x} := \bar{e}'\}$ will denote simultaneous (capture-free) substitution of a sequence of expressions $\bar{e}'$ for free occurrences of a sequence of variables $\bar{x}$, respectively, in the expression e. We will use σ, σ', ϕ etc. to range over substitutions. The term $\text{FV}(e)$ will denote the set of free variables of expression e, and $\bar{\text{FV}}(e)$ will be used to denote a (canonical) list of the free variables of e. Sometimes we will informally write "substitutions" of the form $\{\bar{\mathsf{g}} := \bar{e}\}$ to represent the replacement of occurrences of function symbols $\bar{\mathsf{g}}$ by expressions $\bar{e}$. This is not a proper substitution since the function symbols are not variables. Care must be taken with such substitutions since the notion of equivalence between expressions is not closed under these kind of replacements.

A *context*, ranged over by C, C_1, etc. is an expression with zero or more "holes", $[\,]$, in the place of some subexpressions; $C[e]$ is the expression produced by replacing the holes with expression e. Contrasting with substitution, occurrences of free variables in e may become bound in $C[e]$; if $C[e]$ is closed then we say it is a *closing context* (for e).

We write $e \equiv e'$ to mean that e and e' are identical up to renaming of bound variables. Contexts are identified up to renaming of those bound variables which are not in scope at the positions of the holes.

2.1 Operational Semantics, Approximation and Equivalence

The operational semantics is used to define an evaluation relation $\Downarrow$ (a partial function) between closed expressions and the "values" of computations. The set of values, following the standard terminology (see e.g. (Peyton Jones 1987)), are called *weak head normal forms*. The weak head normal forms, $w, w_1, w_2, \ldots \in \text{WHNF}$ are just the constructor-expressions $c(\bar{e})$, and the *Closures* (lambda expressions), as given by

$$w \;=\; c(\bar{e}) \;\mid\; \lambda x.e$$

The operational semantics is call-by-name, and $\Downarrow$ is defined in terms of a one-step evaluation relation using the notion of a *reduction context* (Felleisen, Friedman, and Kohlbecker 1987). If $e{\Downarrow}w$ for some closed expression e then we say that e *evaluates to* w. We say that e *converges*, and sometimes write $e{\Downarrow}$ if there exists a w such that $e{\Downarrow}w$. Otherwise we say that e *diverges*. We make no finer distinctions between divergent expressions, so that run-time errors and infinite loops are identified.

Reduction contexts, ranged over by $I\!\!R$, are contexts containing a single hole which is used to identify the next expression to be evaluated (reduced).

Definition 1 *A reduction context $I\!\!R$ is given inductively by the following grammar*

$$I\!\!R = [\,] \mid I\!\!R\; e \mid \mathsf{case}\; I\!\!R\; \mathsf{of}\; pat_1 \Rightarrow e_1 \ldots pat_n \Rightarrow e_n \;\mid p(\bar{c}, I\!\!R, \bar{e})$$

The reduction context for primitive functions forces left-to-right evaluation of the arguments. This is just a matter of convenience to make the one-step evaluation relation deterministic.

Now we define the one step reduction relation. We assume that each primitive function p is given meaning by a partial function $[\![p]\!]$ from vectors of constants (according to the arity of p) to the constants (nullary constructors). We do not need to specify the exact set of primitive functions; it will suffice to note that they are strict—all operands must evaluate to constants before the result of an application, if any, can be returned— and are only defined over constants, not over arbitrary weak head normal forms.

Definition 2 *One-step reduction $\mapsto$ is the least relation on closed expressions satisfying the rules given in Figure 1.*

$$\begin{array}{rcll}
I\!\!R[\mathsf{f}] & \mapsto & I\!\!R[e_{\mathsf{f}}] & \textbf{(fun)} \\
 & & (\text{if } \mathsf{f} \text{ is defined by } \mathsf{f} \triangleq e_{\mathsf{f}}) & \\
I\!\!R[(\lambda x.e)\, e'] & \mapsto & I\!\!R[e\{x := e'\}] & (\beta) \\
I\!\!R[\mathsf{case}\; c_i(\bar{e})\; \mathsf{of}\; \ldots c_i(\bar{x}_i) \Rightarrow e_i \ldots] & \mapsto & I\!\!R[e_i\{\bar{x}_i := \bar{e}\}] & \textbf{(case)} \\
I\!\!R[p(\bar{c})] & \mapsto & I\!\!R[c'] & \textbf{(prim)} \\
 & & (\text{if } [\![p]\!]\bar{c} = c') &
\end{array}$$

Figure 1: One-step reduction rules

In each rule of the form $I\!\!R[e] \mapsto I\!\!R[e']$ in Figure 1, the expression e is referred to as a *redex*. The one step evaluation relation is deterministic; this relies on the fact that if $e_1 \mapsto e_2$ then e_1 can be uniquely factored into a reduction context $I\!\!R$ and a redex e' such that $e_1 = I\!\!R[e']$. Let $\mapsto^*$ denote the transitive reflexive closure of $\mapsto$.

Definition 3 *Closed expression* e *converges to weak head normal form* w*,* $e{\Downarrow}w$*, if and only if* $e \mapsto^* w$*.*

Using this notion of convergence we now define the standard notions of operational approximation and equivalence. We use is the standard Morris-style contextual ordering, or *observational approximation* see e.g. (Plotkin 1975).The notion of "observation" we take is just the fact of convergence, as in the lazy lambda calculus (Abramsky 1990). Operational equivalence equates two expressions if and only if in all closing contexts they give rise to the same observation – i.e. either they both converge, or they both diverge.

Definition 4 *1.* e operationally approximates e'*,* $e \sqsubseteq e'$*, if for all contexts* C *such that* $C[e]$*,* $C[e']$ *are closed, if* $C[e]{\Downarrow}$ *then* $C[e']{\Downarrow}$*.*

2. e *is* operationally equivalent *to* e'*,* $e \cong e'$*, if* $e \sqsubseteq e'$ *and* $e' \sqsubseteq e$*.*

Choosing to observe, say, only computations which produce constants would give rise to slightly weaker versions of operational approximation and equivalence - but the above versions would still be *sound* for reasoning about the weaker variants of the relation.

3 A Theory of Improvement

In this section we introduce a theory of improvement, as a refinement of the theory of operational approximation. Roughly speaking, *improvement* is a refinement of operational approximation which expression e is improved by e' if, in all closing contexts, computation using e' is no less efficient than when using e, measured in terms of the number of function calls (f, g, etc.) made. From the point of view of applications of the theory to program transformation and analysis, the important property of improvement is that it is a contextual congruence—an expression can be improved by improving a sub-expression. For reasoning about improvement a more tractable formulation of the improvement relation is introduced and some proof techniques related to this formulation are used.

Variations on the Definition of Improvement There are a number of variations that we can make in the definition of improvement. We could, for example, additionally count the number of primitive functions called. Such variations might be used to give additional information about transformations. However, the fact that we count the number of recursive function calls in the definition of improvement will be *essential* to the Improvement Theorem presented in the next section; the Theorem does not hold if we use an improvement metric which does not count these function calls.

We begin by defining a variation of the evaluation relation which includes the number of applications of the **(fun)** rule.

Definition 5 *Define* $e \overset{\bullet}{\mapsto} e'$ *if* $e \mapsto e'$ *by application of the* **(fun)***rule; define* $e \overset{\circ}{\mapsto} e'$ *if* $e \mapsto e'$ *by application of any other rule.*

Define the family of binary relations on expressions $\{\mapsto_n\}_{n \geq 0}$ *inductively as follows:*

$$\begin{array}{rll} e \mapsto_0 e' & \textit{if} & e \overset{\circ}{\mapsto}{}^* e' \\ e \mapsto_{k+1} e' & \textit{if} & e \overset{\circ}{\mapsto}{}^* e_1 \overset{\bullet}{\mapsto} e_2 \mapsto_k e' \textit{ for some } e_1, e_2. \end{array}$$

We say that a closed expression e *converges in* n **(fun)***-steps to weak head normal form* w*, written* $e{\Downarrow}^n w$ *if* $e \mapsto_n w$.

The determinacy of the one-step evaluation relation guarantees that if $e{\Downarrow}^n w$ and $e{\Downarrow}^{n'} w'$ then $w \equiv w'$ and moreover $n = n'$. It will be convenient to adopt the following abbreviations:

- $e{\Downarrow}^n \stackrel{\text{def}}{=} \exists w.\, e{\Downarrow}^n w$
- $e{\Downarrow}^{n \leq m} \stackrel{\text{def}}{=} e{\Downarrow}^n \;\&\; n \leq m$
- $e{\Downarrow}^{\leq m} \stackrel{\text{def}}{=} \exists n.\, e{\Downarrow}^{n \leq m}$

Now improvement is defined in a way analogous to observational approximation:

Definition 6 (Improvement) e is improved by e', $e \gtrsim e'$, *if for all contexts* C *such that* $C[e]$, $C[e']$ *are closed, if* $C[e]{\Downarrow}^n$ *then* $C[e']{\Downarrow}^{\leq n}$.

It can be seen from the definition that $\gtrsim$ is a *precongruence* (transitive, reflexive, closed under contexts, i.e. $e \gtrsim e' \Rightarrow C[e] \gtrsim C[e']$) and is a refinement of operational approximation, i.e. $e \gtrsim e' \Rightarrow e \sqsubseteq e'$.

We also define a strong version of improvement which contains (by definition) operational equivalence:

Definition 7 (Strong Improvement, Cost-Equivalence) *The strong improvement relation* $\gtrsim_s$ *is defined by:* $e \gtrsim_s e'$ *if and only if* $e \gtrsim e'$ *and* $e \cong e'$.

The cost equivalence relation, $\lesssim\!\!\gtrsim$*, is defined by:* $e \lesssim\!\!\gtrsim e'$ *if and only if* $e \gtrsim e'$ *and* $e' \gtrsim e$.

If R is a relation, then let R^{-1} denote the inverse of the relation, so that $a \; R \; b \iff b \; R^{-1} \; a$. It is not difficult to see that $\gtrsim_s = (\gtrsim) \cap (\sqsupseteq)$. This fact, and other relationships between the various preorders and equivalence relations we have considered so far, are summarised in the Hasse diagram of Figure 2. In this lattice, the binary meet (greatest lower bound) corresponds to the set-intersection of the relations, and the top element, $\top$, relates any two expressions.

3.1 Proving Improvement

Finding a more tractable characterisation of improvement (than that provided by Def. 6) is desirable in order to establish improvement laws (and the Improvement Theorem itself). The characterisation we use says that two expressions are in the

$$\begin{tikzcd}
 & & \top & & \\
 & \underset{\sim}{\sqsubset} \arrow[ur] & & \underset{\sim}{\sqsubset}^{-1} \arrow[ul] & \\
\underset{\sim}{\rhd} \arrow[ur] & & \cong \arrow[ul] \arrow[ur] & & \underset{\sim}{\rhd}^{-1} \arrow[ul] \\
 & \underset{\sim}{\rhd}_s \arrow[ul] \arrow[ur] & & \underset{\sim}{\rhd}^{-1}_s \arrow[ul] \arrow[ur] & \\
 & & \underset{\sim}{\lhd\!\rhd} \arrow[ul] \arrow[ur] & & \\
 & & \equiv \arrow[u] & &
\end{tikzcd}$$

Figure 2: A $\cap$-semi-sub-lattice of preorders

improvement relation if and only if they are contained in a certain kind of *simulation* relation. This is a form of *context lemma* eg. (Milner 1977; Abramsky 1990; Howe 1989), and the proof of the characterisation uses previous technical results concerning a more general class of improvement relations (Sands 1991). In (Sands 1991), we abstracted over the way improvement is defined in terms of the operational semantics, and following Howe (1989), we gave some conditions which guarantee that the simulation relations are sound for reasoning about improvement.

Definition 8 *A relation $\mathcal{IR}$ on closed expressions is an* improvement simulation *if for all e, e', whenever $e \; \mathcal{IR} \; e'$, if $e{\Downarrow}^n w_1$ then $e'{\Downarrow}^{\leq n} w_2$ for some w_2 such that either:*

1. $w_1 \equiv c(e_1 \ldots e_n)$, $w_2 \equiv c(e'_1 \ldots e'_n)$, *and* $e_i \; \mathcal{IR} \; e'_i$, $(i \in 1 \ldots n)$, or
2. $w_1, w_2 \in$ *Closures, and for all closed* e_0, $(w_1\, e_0) \; \mathcal{IR} \; (w_2\, e_0)$

For a given relation $\mathcal{IR}$ and weak head normal forms w_1 and w_2 we will abbreviate the property "(1) or (2)" in the above by $w_1 \; \mathcal{IR}^{\dagger} \; w_2$.

So, intuitively, if an improvement simulation relates e to e', then if e converges, e' does so at least as efficiently, and yields a "similar" result, whose "components" are related by that improvement simulation.

The key to reasoning about the improvement relation is the fact that $\underset{\sim}{\rhd}$, restricted to closed expressions, is itself an improvement simulation, and is in fact the *largest* improvement simulation. Furthermore, improvement on open expressions can be characterised in terms of improvement on all closed instances. This is summarised in the following.

Let $\underset{\approx}{\triangleright}$ denote the largest improvement simulation. It is easy to verify that this exists, and is given by the union of all simulation relations. Let $\underset{\approx}{\triangleright}^{\circ}$ denote its extension to open terms specified by $e \underset{\approx}{\triangleright}^{\circ} e'$ if and only if $e\sigma \underset{\approx}{\triangleright} e'\sigma$ for all closing substitutions σ.

Lemma 1 (Improvement Context-Lemma) *For all e, e', $e \underset{\sim}{\triangleright} e'$ if and only if $e \underset{\approx}{\triangleright}^{\circ} e'$.*

The most important part of the lemma, that $\underset{\approx}{\triangleright}^{\circ} \subseteq \underset{\sim}{\triangleright}$, can be proved by a straightforward application of Howe's method (Howe 1989), as illustrated in (Sands 1991). We omit the proof here. The proof of the "completeness" part of the lemma, that $e \underset{\sim}{\triangleright} e'$ implies $e \underset{\approx}{\triangleright}^{\circ} e'$, is rather more involved than the proof of the corresponding property for operational approximation. We give an outline of the main ingredients:

- Characterise $\underset{\approx}{\triangleright}$ inductively in the form $\bigcap_{i<\omega} \underset{\approx_i}{\triangleright}$ (where $\underset{\approx_i}{\triangleright}$ is "simulation up to a depth i"). Then $e \underset{\approx}{\not\triangleright} e'$ implies that there exists a smallest $m > 0$ such that $e \underset{\approx_m}{\not\triangleright} e'$.
- It is sufficient to assume e and e' closed, and to show that $e \underset{\approx}{\not\triangleright} e'$ implies $e \underset{\sim}{\not\triangleright} e'$. By induction on the minimal m above, this can be done by building a context which distinguishes e and e'. This in turn depends on the existence of contexts which can magnify any non-zero computational cost by an arbitrary degree.

The lemma provides a basic proof technique:

> to show that $e \underset{\sim}{\triangleright} e'$ it is sufficient to find an improvement-simulation containing each closed instance of the pair.

An alternative presentation of the definition of improvement simulation is in terms of the maximal fixed point of a certain monotonic function on relations. In that case the above proof technique is called *co-induction*. This proof technique is crucial to the proof of the Improvement Theorem which follows in the next section. It can also be useful in proving that specific transformation rules are improvements. Here is an illustrative example; it also turns out to be a useful transformation rule:

Proposition 9 *If the free variables of $I\!R$ are distinct from the variables in $pat_1, \ldots, pat_n$, then*

$$\begin{aligned} & I\!R[\mathsf{case}\ x\ \mathsf{of}\ pat_1 \Rightarrow e_1 \cdots pat_n \Rightarrow e_n\ ] \\ \underset{\sim}{\triangleleft\!\triangleright}\quad & \mathsf{case}\ x\ \mathsf{of}\ pat_1 \Rightarrow I\!R[e_1] \cdots pat_n \Rightarrow I\!R[e_n] \end{aligned}$$

PROOF. We illustrate just the $\underset{\sim}{\triangleright}$-half. The other half is similar. Let R be the relation containing $\equiv$, together with all pairs of closed expressions of the form:

$$\begin{aligned} (\ & I\!R[\mathsf{case}\ e_0\ \mathsf{of}\ c_1(\bar{x}_1) \Rightarrow e_1 \ldots c_n(\bar{x}_n) \Rightarrow e_n\ ], \\ & \mathsf{case}\ e_0\ \mathsf{of}\ c_1(\bar{x}_1) : I\!R[e_1] \ldots c_n(\bar{x}_n) : I\!R[e_n]\ \ ) \end{aligned} \qquad (3.1)$$

It is sufficient to show that R is an improvement simulation. Suppose $e \; R \; e'$, and suppose further that $e{\Downarrow}^n w$. We need to show that $e'{\Downarrow}^{\leq n} w'$ for some w' such that $w \; R^\dagger \; w'$. If $e \equiv e'$ then this follows easily. Otherwise e and e' have the form of (3.1). Now since $I\!\!R[\mathsf{case}\;[\,]\;\mathsf{of}\;c_1(\bar{x}_1) \Rightarrow e_1 \ldots c_n(\bar{x}_n) \Rightarrow e_n\;]$ is a reduction context, then we must have

$$\begin{array}{ll} & I\!\!R[\mathsf{case}\;e_0\;\mathsf{of}\;c_1(\bar{x}_1) \Rightarrow e_1 \ldots c_n(\bar{x}_n) \Rightarrow e_n\;] \\ \mapsto^k & I\!\!R[\mathsf{case}\;c_i(\bar{e}'')\;\mathsf{of}\;c_1(\bar{x}_1) \Rightarrow e_1 \ldots c_n(\bar{x}_n) \Rightarrow e_n\;] \end{array}$$

for some expression $c_i(\bar{e}'')$, and some $k \leq n$ and since each of these reductions is "in" e_0, we have matching reduction steps

$$\begin{array}{ll} & \mathsf{case}\;e_0\;\mathsf{of}\;c_1(\bar{x}_1) : I\!\!R[e_1] \ldots c_n(\bar{x}_n) : I\!\!R[e_n] \\ \mapsto^k & \mathsf{case}\;c_i(\bar{e}'')\;\mathsf{of}\;c_1(\bar{x}_1) : I\!\!R[e_1] \ldots c_n(\bar{x}_n) : I\!\!R[e_n] \end{array}$$

Now the former derivative reduces in one more step to $I\!\!R[e_i\{\bar{x}_i := \bar{e}''\}]$, whilst the latter reduces to $I\!\!R[e_i]\{\bar{x}_i := \bar{e}''\}$. Since reduction contexts do not bind variables, and since by assumption the free variables of the patterns are disjoint from the free variables of $I\!\!R$, then these are syntactically equivalent, and so we conclude that

$$\mathsf{case}\;e_0\;\mathsf{of}\;c_1(\bar{x}_1) \Rightarrow I\!\!R[e_1] \ldots c_n(\bar{x}_n) \Rightarrow I\!\!R[e_n] \Downarrow^n w.$$

The remaining conditions for improvement simulation (recall Def. 8 and the $\cdot^\dagger$ operator) are trivially satisfied, since $w \equiv^\dagger w$, which implies $w \; R^\dagger \; w$ as required. □

3.2 The Tick Algebra

A particularly useful method for reasoning about improvement is to make use of certain *identity functions*, i.e., functions f such that for all e, $f\;e \cong e$. Sometimes it is useful to consider functions which are the identity for expressions e of a certain type (e.g., lists). The point of such functions is that they take time to do nothing; in other words $f\;e \mathrel{\underset{s}{\gtrsim}} e$. Such functions facilitate the simple calculation of improvements, and are used extensively in applications of the theory. Here we consider the simplest form of identity function, which we call the *tick* function.

$$\surd e \equiv \mathsf{tick}\;e$$

where tick is an identity function, given by the definition

$$\mathsf{tick} \triangleq \lambda x.x$$

The tick function will be our canonical syntactic representation of a single computation step. From the point of view of observational equivalence we can safely regard it as an annotation, since $\surd e \cong e$; but from the point of view of improvement it is more significant. In particular, since the tick is a function call, from the operational semantics it should be clear that

$$e{\Downarrow}^n h \iff \surd e{\Downarrow}^{n+1} h$$

In terms of improvement, observe that ${}^{\surd}e \mathrel{\underset{\sim}{\triangleright}} e$ but $e \mathrel{\underset{\sim}{\not\triangleright}} {}^{\surd}e$ (except if all closed instances of e diverge).

A key property regarding the tick function is that if function f is defined by $\mathsf{f} \triangleq e$, then

$$\mathsf{f} \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} {}^{\surd}e$$

In Figure 3 gives some simple laws for $\surd$, *the tick algebra*, which we state without proof. In the laws, $I\!\!R$ ranges over reduction contexts, possibly containing free variables. There is a related rule for propagation of ticks over contexts. If we define an open context C to be *strict* if for all closing substitutions ϕ, $C[\bot]\phi\Uparrow$, where $\bot$ is any closed expression such that $\bot\Uparrow$. Then we claim that for all expressions e, $C[{}^{\surd}e] \mathrel{\underset{\sim}{\triangleright}}_s {}^{\surd}C[e]$. The improvement is not reversible since the expression in the hole may get duplicated.

We will see see more of the tick functions when we come to consider applications of the theory.

$$\frac{e \overset{\bullet}{\mapsto} e'}{e \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} {}^{\surd}e'} \qquad \frac{e \overset{\circ}{\mapsto}{}^{*} e'}{e \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} e'}$$

$${}^{\surd}e \mathrel{\underset{\sim}{\triangleright}} e \qquad \frac{{}^{\surd}e_1 \mathrel{\underset{\sim}{\triangleright}} {}^{\surd}e_2}{e_1 \mathrel{\underset{\sim}{\triangleright}} e_2}\text{(similarly for } \mathrel{\underset{\sim}{\triangleright}}_s \text{ and } \mathrel{\underset{\sim}{\triangleleft\!\triangleright}}\text{)}$$

$$I\!\!R[{}^{\surd}e] \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} {}^{\surd}I\!\!R[e] \qquad {}^{\surd}p(e_1 \ldots e_n) \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} p(e_1 \ldots {}^{\surd}e_i \ldots e_n)$$

$$\begin{array}{lcl} {}^{\surd}\ \mathsf{case}\ e\ \mathsf{of} & \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} & \mathsf{case}\ e\ \mathsf{of} \\ \quad c_1(\vec{x}_1) \Rightarrow e_1 \ldots c_n(\vec{x}_n) \Rightarrow e_n & & \quad c_1(\vec{x}_1) \Rightarrow {}^{\surd}e_1 \ldots c_n(\vec{x}_n) \Rightarrow {}^{\surd}e_n \end{array}$$

Figure 3: Tick laws

4 Time Analysis using Cost Equivalence

In this section we consider the application of the theory of improvement — and in particular the theory of cost equivalence — to the problem of reasoning about the running-time properties of programs. This section summarises some of the work developed in Sands (1993).

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. In this section we illustrate how improvement can be used as a tool for reasoning about the running time of lazy functional programs.

In Sands (1990) we introduced a simple set of "naïve" *time rules*, derived directly from the operational semantics. These concern equations on $\langle e \rangle$, the "time" to evaluate expression e to (weak) head normal form, and $\langle e \rangle^N$, the time to evaluate e to normal form. One of the principal limitations of the direct operational approach to reasoning 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 motivated the 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. The resulting theory is the cost equivalence relation defined in the previous section.

4.1 Motivation

As a motivating example for developing techniques to support reasoning about running time, 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[1],

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

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.

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

[1]The example is originally due to D. Turner; it appears as an exercise in informal reasoning about lazy evaluation in (Bird and Wadler 1988)[Ch. 6], and in the majority(!) of papers on time analysis of non-strict evaluation.

of quicksort 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. We will return to these examples.

4.2 Reasoning with Time

We outline how cost-equivalence can be used to help reason about the running time of some simple lazy programs.

Time Equations The basic questions we wish to ask are of the form "How many function-calls are required to compute the weak head normal form of expression e". Typically e will not be a simple closed expression, but will contain a meta-variable ranging over some first-order input values (normal forms). As is usual, the kind of answer we would like is a (closed form) function of the size of the input value, representing the exact, or asymptotic, cost. We will not be particularly formal about the treatment of meta-variables, so most of our reasoning is as if we are dealing with (families of) closed expressions.

The basic idea is that we will use the instrumented evaluation $e{\Downarrow}^n w$ as our cost-model. In order to abstract away from the result of a computation it is convenient to introduce phrases of the form $\langle e \rangle$: the time to compute expression e to weak head normal form. In Sands (1993) we used rules along the lines of the following:

$$\langle w \rangle = 0 \qquad \langle e \rangle = \begin{cases} \langle e' \rangle & \text{if } e \overset{\circ}{\mapsto} e' \\ 1 + \langle e' \rangle & \text{if } e \overset{\bullet}{\mapsto} e' \end{cases}$$

This has an obvious shortcoming that it does not model the usual implementation mechanism. namely *call-by-need*, but proves to be adequate for many purposes. An approach to call-by-need is investigated in Sands (1993) also, but we will not consider it here. Here we will mainly work with the cost-equivalence relation, and only employ the time equations when it is notationally convenient to abstract away from the result computed by the "top-level" expression.

Notation For $n \geq 0$, let ${}^{n}\!\surd e$ denote the application of n ticks to expression e.

The following property follows easily from the tick algebra:

Proposition 10 *For all closed expressions e,*

1. *$e{\Downarrow}^n w$ implies $e \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{n}\!\surd w$*
2. *$e \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{n}\!\surd w' \in$ WHNF implies $e{\Downarrow}^n w$ for some w such that $w \mathrel{\underset{\sim}{\lhd\!\rhd}} w'$.*

$$
\begin{array}{lll}
\mathsf{isort} & \triangleq & \lambda xs.\ \mathsf{case}\ xs\ \mathsf{of} \\
 & & \qquad nil \Rightarrow \mathsf{nil} \\
 & & \qquad h : t \Rightarrow \mathsf{insert}\, h\ (\mathsf{isort}\, t) \\
 & & \\
\mathsf{insert} & \triangleq & \lambda x.\lambda ys.\ \mathsf{case}\ ys\ \mathsf{of} \\
 & & \qquad \mathsf{nil} \Rightarrow x : \mathsf{nil} \\
 & & \qquad h : t \Rightarrow \mathsf{if}\ x \leq h\ \mathsf{then}\ x : (h : t) \\
 & & \qquad\qquad\qquad\qquad \mathsf{else}\ h : (\mathsf{insert}\, x\ t)
\end{array}
$$

Figure 4: Insertion Sort

In terms of the time equations, we note that if $e \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{n}\!\surd e'$ then $\langle e \rangle = n + \langle e' \rangle$. The point of the proposition is that we can use $\mathrel{\underset{\sim}{\lhd\!\rhd}}$ as a basis for reasoning about evaluation steps. Moreover, because $\mathrel{\underset{\sim}{\lhd\!\rhd}}$ is a congruence, we can use it to simplify subexpressions.

Insertion Sort Returning to the example given earlier, Figure 4 presents the insertion sort function, this time in the syntax of the language introduced earlier, but retaining the infix ":" list constructor.

The time to compute the head-normal form of insertion sort is relatively simple to determine. We consider computing the head normal form of insertion sort applied to some arbitrary (evaluated) list of integers $v_n : \ldots : v_1 : \mathsf{nil}$ where $n \geq 1$. Let V_0 denote the list nil and, for each $i < n$, let V_{i+1} denote the list $v_{i+1} : V_i$. The following shows that the time needed to compute the first element of insertion sort is always linear in the length of the argument.

Proposition 11 *For all $i > 0$ there exist expression e' and some value $v \in \{v_1, \ldots, v_i\}$ such that*

$$\mathsf{isort}\, V_i \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{(2i+1)}\!\surd v : e'$$

PROOF. By induction on i, and calculating using the tick algebra. In the base case $(i = 1)$ we have $\mathsf{isort}\, V_1 \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{\surd}\mathsf{insert}\, v_1\ (\mathsf{isort}\, V_0) \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{3}\!\surd v_1 : \mathsf{nil}$.

In the induction case $(i = k + 1)$ we calculate:

$$
\begin{array}{lll}
\mathsf{isort}\, V_{k+1} & \mathrel{\underset{\sim}{\lhd\!\rhd}} & {}^{\surd}\mathsf{insert}\, v_{k+1}\ (\mathsf{isort}\, V_k) \\
 & \mathrel{\underset{\sim}{\lhd\!\rhd}} & {}^{\surd}\mathsf{insert}\, v_{k+1}\ {}^{(2k+1)}\!\surd(v : e') \qquad\qquad\qquad \text{(Hypothesis)} \\
 & & \textit{for some } e' \textit{ and some } v' \in \{v_1, \ldots v_k\} \\
 & \mathrel{\underset{\sim}{\lhd\!\rhd}} & {}^{2}\!\surd\mathsf{case}\ {}^{(2k+1)}\!\surd(v : e')\ \mathsf{of} \ldots \\
 & \mathrel{\underset{\sim}{\lhd\!\rhd}} & {}^{(2(k+1)+1)}\!\surd\mathsf{if}\ v_{k+1} \leq v\ \mathsf{then}\ v_{k+1} : v : e' \\
 & & \qquad\qquad\qquad\quad \mathsf{else}\ v : (\mathsf{insert}\, v_{k+1}\, e') \\
 & \mathrel{\underset{\sim}{\lhd\!\rhd}} & \left\{ \begin{array}{ll} {}^{(2(k+1)+1)}\!\surd v_{k+1} : v : e' & \textit{if } v_{k+1} \leq v \\ {}^{(2(k+1)+1)}\!\surd v : (\mathsf{insert}\, v_{k+1}\, e') & \textit{otherwise} \end{array} \right.
\end{array}
$$

```
qs       ≜  λxs. case xs of
                   nil ⇒ nil
                   h : t ⇒ qs(below h t)++(h : qs(above h t))

below    ≜  λx.λys. case ys of
                      nil ⇒ nil
                      h : t ⇒ if h ≤ x then h : below x t
                                       else below x t
xs++ys   ≜   case xs of
               nil ⇒ ys
               h : t ⇒ h : (t++ys)
```

Figure 5: Functional Quicksort

□

As a direct corollary we have $\langle \mathsf{isort}\, V_i \rangle = 2i + 1$.

Quicksort Now we consider a more involved example. The equations in Figure 5 define a simple functional version of quicksort (qs) using auxiliary functions below and above, and append written here as an infix function ++. Primitive functions for integer comparison have also been written infix to aid readability. The definition for above has been omitted, but is like that of below with the comparison ">" in place of "≤".

The aim will be fairly modest: to 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:

$$\langle \mathsf{qs}\, e \rangle = 1 + k + \begin{cases} 0 & \textit{if}\; e \underset{\sim}{\lhd\!\rhd} {}^{k}\!\surd\mathsf{nil} \\ \left\langle \begin{array}{l} \mathsf{qs}(\mathsf{below}\, h\, t) \\ {+\!\!+}(y : \mathsf{qs}(\mathsf{above}\, h\, t)) \end{array} \right\rangle & \textit{if}\; e \underset{\sim}{\lhd\!\rhd} {}^{k}\!\surd(h : t) \end{cases}$$

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

$$\langle \mathsf{qs}\, e \rangle = 1 + k + \begin{cases} 0 & \textit{if}\; e \underset{\sim}{\lhd\!\rhd} {}^{k}\!\surd\mathsf{nil} \\ 1 + \langle \mathsf{qs}(\mathsf{below}\, y\, z) \rangle & \textit{if}\; e \underset{\sim}{\lhd\!\rhd} {}^{k}\!\surd(h : t) \end{cases} \tag{4.1}$$

Proceeding to the particular problem, it is not too surprising that we will use non-increasing lists v to show that $\langle \mathsf{qs}\, v \rangle = \Omega(n^2)$. 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 set of non-increasing lists $\{A_i\}_{i\geq 0}$ by induction on i: let A_0 denote the list nil, and, for each $k > 0$ let A_{k+1} denote the list $v_{k+1} : A_k$.

The goal is now to show that $\langle \mathsf{qs}(A_n)\rangle$ is quadratic in n. It is easy to see, instantiating (4.1) that

$$\langle \mathsf{qs}(A_{k+1})\rangle = 2 + \langle \mathsf{qs}(\mathsf{below}\ v_{k+1}\ A_k)\rangle$$

but continuing with this simple style of reasoning we quickly see the limitations of the "naive" operational approach to reasoning. Unlike the insertion-sort example where basic operational reasoning is sufficient (see (Sands 1993), where cost equivalence is not used for the isort example), the successive calls to qs become increasingly complex. The key to showing that $\langle \mathsf{qs}\ A_n\rangle$ is quadratic in n is the identification of a cost equivalence which allows us to simplify (a generalised version of) the call to below. To do this we will need another form of identity function.

Identity Functions on Lists We introduce another identity function, this time on the domain of list-valued expressions. Let T be the identity function on lists given by

$$\begin{aligned}\mathsf{T} \triangleq \lambda x.\ &\mathsf{case}\ x\ \mathsf{of}\\ &\quad \mathsf{nil} \Rightarrow \mathsf{nil}\\ &\quad h : t \Rightarrow (h : \mathsf{T}t)\end{aligned}$$

As with the tick function, let T^n denote the n-fold composition of T, where $\mathsf{T}^0 e$ is just e. It follows from this definition that

$$\begin{aligned}\mathsf{T}^n\ \mathsf{nil} &\ \underset{\sim}{\lhd\!\rhd}\ {}^{n}\!\surd\mathsf{nil}\\ \mathsf{T}^n\ (h : t) &\ \underset{\sim}{\lhd\!\rhd}\ {}^{n}\!\surd(h : \mathsf{T}^n\ t)\end{aligned}$$

So T^n is an identity function for lists which increases the cost of producing each constructor in the list's structure by n ticks. We use T to characterise a key property of a certain call to below:

Proposition 12 *For all $a \geq 0$, and i, j such that $0 \leq j \leq i$,*

$$\mathsf{below}(v_i, \mathsf{T}^a A_j)\ \underset{\sim}{\lhd\!\rhd}\ \mathsf{T}^{a+1} A_j.$$

PROOF. By induction on j, using the tick algebra (it can also be proved straightforwardly by coinduction).

$$\begin{array}{lll}
\textit{Base: } (j=0) \quad \mathsf{below}(v_i, \mathsf{T}^a A_0) & \mathrel{\underset{\sim}{\lhd\!\rhd}} & \surd\mathsf{case}\ \mathsf{T}^a A_0\ \mathsf{of} \ldots) \\
& \mathrel{\underset{\sim}{\lhd\!\rhd}} & \surd(\mathsf{case}\ {}^{n}\surd\mathsf{nil}\ \mathsf{of}\ \ldots)) \\
& \mathrel{\underset{\sim}{\lhd\!\rhd}} & {}^{a+1}\surd(\mathsf{nil}) \\
& \mathrel{\underset{\sim}{\lhd\!\rhd}} & \mathsf{T}^{a+1} A_0
\end{array}$$

$$\begin{array}{lll}
\textit{Induction: } (j=k{+}1) \quad \mathsf{below}(v_i, \mathsf{T}^a A_{k+1}) & & \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & \surd(\mathsf{case}\ \mathsf{T}^a A_{k+1}\ \mathsf{of} \ldots) & \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & {}^{a+1}\surd(\mathsf{case}\ v_{k+1} : \mathsf{T}^a A_k\ \mathsf{of} \ldots) & \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & {}^{a+1}\surd(\mathsf{if}\ v_{k+1} \le v_i\ \mathsf{then}\ v_{k+1} : \mathsf{below}(v_i, \mathsf{T}^a A_k) & \\
& \qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{below}(v_i, \mathsf{T}^a A_k)) & \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & {}^{a+1}\surd(v_{k+1} : \mathsf{below}(v_i, \mathsf{T}^a A_k)) & \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & {}^{a+1}\surd(v_{k+1} : \mathsf{T}^{a+1} A_k) & \text{(Hypothesis)} \\
\mathrel{\underset{\sim}{\lhd\!\rhd}} & \mathsf{T}^{a+1} A_{k+1} &
\end{array}$$

□

Now we can make use of the proposition in the analysis of quicksort. We consider the more general case of $\langle \mathsf{qs}\,\mathsf{T}^a A_j \rangle$. Considering the cases when $j = 0$ and $j = k+1$, and instantiating the general time equation gives:

$$\begin{aligned}
\langle \mathsf{qs}(\mathsf{T}^a A_0) \rangle &= 1 + a \\[1em]
\langle \mathsf{qs}(\mathsf{T}^a A_{k+1}) \rangle &= 2 + a + \langle \mathsf{qs}(\mathsf{below}\, v_{k+1}\, \mathsf{T}^a A_k) \rangle \\
&= 2 + a + \langle \mathsf{qs}(\mathsf{T}^{a+1} A_k) \rangle
\end{aligned}$$

Thus we have derived a recurrence equation which is easily solved; a simple induction is sufficient to check that

$$\langle \mathsf{qs}\,\mathsf{T}^a A_n \rangle = \frac{n(n+5)}{2} + a(n+1) + 1.$$

Finally, since the A_n are just $\mathsf{T}^0 A_n$, we have that the time to compute the weak head normal form of $\mathsf{qs}\ A_n$ is quadratic in n:

$$\langle \mathsf{qs}\, A_n \rangle = \frac{n(n+5)}{2} + 1.$$

Further Work Under a call-by-need computation model some computations are shared, so we would expect that, for example, $(\lambda x.x + x)\,\surd 0 \mathrel{\underset{\sim}{\lhd\!\rhd}} \surd 0$ where $+$ is a primitive function. Instead, we get $(\lambda x.x + x)\,\surd 0 \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{2}\surd 0$ which can make cost equivalence unreliable for reasoning about non-linear functions. As we mentioned earlier, the problems of reasoning about call-by-need are addressed in Sands (1993)— but not via a call-by-need theory of cost equivalence. Defining call-by-need improvement is not problematic. The problem is to find an appropriate context lemma, and this is a topic we plan to address in future work. The applications of the Improvement Theorem introduced in the next section depend less critically on the intentional qualities of improvement theorem. For these applications the call-by-name origin of our improvement theory is often advantageous, since for example unrestricted beta-reduction is an improvement for call-by-name. We do not yet know whether the development of the next section will go through for a call-by-need theory.

5 The Improvement Theorem

In this section we motivate and introduce the Improvement Theorem. The Improvement Theorem employs the improvement relation to prove the correctness of program transformations. The interesting point of this application is that the goal is not to prove that program transformations improve programs *per se*, but simply to prove that they produce operationally equivalent programs.

5.1 The Correctness Problem

The goal of program transformation is to improve efficiency while preserving meaning. Source-to-source transformation methods such as *unfold-fold transformation*, *partial evaluation* (specialisation) and *deforestation* (Burstall and Darlington 1977; Jones, Gomard, and Sestoft 1993; Wadler 1990), are some well-known examples. These kind of transformations are characterised by the fact that

- they utilise a small number of relatively simple transformation steps, and
- in order to compound the effect of these relatively simple local optimisations, the transformations have the ability to introduce new recursive calls.

Transformations such as deforestation (Wadler 1990) (a functional form of loop-fusion) and program specialisation (and analogous transformations on logic programs) are able to introduce new recursive structures via a process of selectively memoising previously encountered expressions, and introducing recursion according to a "*déjà vu*" principle (Jones, Gomard, and Sestoft 1993). In the classic unfold-fold transformation, it is the *fold* step which introduces recursion. See Pettorossi and Proietti (1995) for an overview of transformation strategies which fit this style.

The problem is that for many transformation methods which deal with recursive programs (including those methods mentioned above), correctness cannot be argued by simply showing that the basic transformation steps are meaning preserving.[2] Yet this problem (exemplified below) runs contrary to many informal—and some "formal"—arguments which are used in attempts to justify correctness of particular transformation methods. This is the problem for which the Improvement Theorem was designed to address.

[2]One might say that there are two problems with correctness – the other problem is that it has not been widely recognised as a problem!

To take a simple example to illustrate the problem, consider the following "transformation by equivalence-preserving steps". Start with the function repeat which produces the "infinite" list of its argument:

$$\mathsf{repeat}\, x \triangleq x : (\mathsf{repeat}\, x)$$

Suppose the function tail computes the tail of a list. The following property can be easily deduced: $\mathsf{repeat}\, x \cong \mathsf{tail}(\mathsf{repeat}\, x)$. Now suppose that we use this "local equivalence" to transform the body of the function to obtain a new version of the function:

$$\mathsf{repeat}'\, x \triangleq x : (\mathsf{tail}(\mathsf{repeat}'\, x))$$

The problem is that this function is not equivalent to the original, since it can never produce more than first element in the list.

One might be tempted to suggest that this is "just a problem of name-capture", and to conclude that the problem can be solved by

- not allowing transformations on the body which depend on the function being transformed; in the example above, this means that if we transform the body, repeat must be treated as a free variable, or

- making the new function non-recursive, so that e.g., we would obtain:

$$\mathsf{repeat}'\, x \triangleq x : (\mathsf{tail}(\mathsf{repeat}\, x))$$

Unfortunately these "solutions", while preventing us from performing incorrect transformations, also prevent us from performing any interesting correct ones!

5.2 A Solution: Local Improvement

To obtain total correctness without losing the local, stepwise character of program transformation, it is clear that a stronger condition than extensional equivalence for the local transformation steps is needed. In Sands (1995b) we presented such a condition, namely *improvement*.

In the remainder of this section we outline the main technical result from Sands (1995b), which says that if transformation steps are guided by certain natural optimisation concerns, then correctness of the transformation follows. We also present "local" version of the Improvement Theorem which is stated at expression-level recursion using a simple "letrec" construct.

More precisely, the *Improvement Theorem* says that if e is improved by e', in addition to e being operationally equivalent to e', then a transformation which replaces e by e' (potentially introducing recursion) is totally correct; in addition this guarantees that the transformed program is a formal improvement over the original.

Notice that in the problematic example above, replacement of $\mathsf{repeat}\, x$ by the equivalent term $\mathsf{tail}(\mathsf{repeat}\, x)$ is not an improvement since the latter requires evaluation of an additional call to repeat in order to reach weak head normal form.

The fact that the theorem, in addition to establishing correctness, also guarantees that the transformed program is an improvement over the original is an added bonus. It can also allow us to apply the theorem iteratively. It also gives us an indication of the limits of the method. Transformations which do not improve a program cannot be justified using the Improvement Theorem alone. However, in combination with some other more basic methods for establishing correctness, the Improvement Theorem can still be effective. We refer the reader to Sands (1996b) for examples of other more basic methods and how they can be used together with the Improvement Theorem.

For the purposes of the formal statement of the Improvement Theorem, transformation is viewed as the introduction of some *new* functions from a given set of definitions, so the transformation from a program consisting of a single function $\mathsf{f} \triangleq e$ to a new version $\mathsf{f} \triangleq e'$ will be represented by the derivation of a new function $\mathsf{g} \triangleq e'\{\mathsf{f} := \mathsf{g}\}$. In this way we do not need to explicitly parameterise operational equivalence and improvement by the intended set of function definitions.

In the following (Theorem 1 – Proposition 14) let $\{\mathsf{f}_i\}_{i \in I}$ be a set of functions indexed by some set I, given by some definitions:

$$\{\mathsf{f}_i \triangleq e_i\}_{i \in I}$$

Let $\{e_i'\}_{i \in I}$ be a set of expressions. The following results relate to the transformation of the functions f_i using the expressions e_i': let $\{\mathsf{g}_i\}_{i \in I}$ be a set of new functions (i.e. the definitions of the f_i do not depend upon them) given by definitions

$$\{\mathsf{g}_i \triangleq e_i'\{\bar{\mathsf{f}} := \bar{\mathsf{g}}\}\}_{i \in I}$$

We begin with the standard partial correctness property associated with "transformation by equivalence":

Theorem 1 (Partial Correctness) *If* $e_i \cong e_i'$ *for all* $i \in I$, *then* $\mathsf{g}_i \precsim \mathsf{f}_i$, $i \in I$.

This is the "standard" partial correctness result (see eg. (Kott 1978)(Courcelle 1979)) associated with e.g. unfold-fold transformations. It follows easily from a least fixed-point theorem for $\precsim$ (the full details for this language can be found in Sands (1996b)) since the $\bar{\mathsf{f}}$ are easily shown to be fixed points of the defining equations for functions $\bar{\mathsf{g}}$.

Partial correctness is clearly not adequate for transformations, since it allows the resulting programs to loop in cases where the original program terminated. We obtain a guarantee of total correctness by combining the partial correctness result with the following:

Theorem 2 (The Improvement Theorem (Sands 1995b)) *If we have* $e_i \trianglerighteq e_i'$ *for all* $i \in I$, *then* $\mathsf{f}_i \trianglerighteq \mathsf{g}_i$, $i \in I$.

The proof of the Theorem, given in detail in (Sands 1996b), makes use of the alternative characterisation of the improvement relation given later.

Putting the two theorems together, we get:

Corollary 13 *If we have* $e_i \mathrel{\underset{\sim}{\rhd}}_s e'_i$ *for all* $i \in I$*, then* $\mathsf{f}_i \mathrel{\underset{\sim}{\rhd}}_s \mathsf{g}_i$, $i \in I$*.*

Informally, this implies that:

> if a program transformation proceeds by repeatedly applying some set of transformation rules to a program, providing that the basic rules of the transformation are equivalence-preserving, and also contained in the improvement relation (with respect to the original definitions), then the resulting transformation will be correct. Moreover, the resulting program will be an improvement over the original.

There is also a third variation, a "cost-equivalence" theorem, which is also useful:

Proposition 14 *If* $e_i \mathrel{\underset{\sim}{\lhd\!\rhd}} e'_i$ *for all* $i \in I$*, then* $\mathsf{f}_i \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{g}_i$, $i \in I$*.*

5.3 An Improvement Theorem for Local Recursion

In this section we introduce a form of the improvement theorem which deals with local expression-level recursion, expressed with a fixed point combinator or with a simple "letrec" definition.

Definition 15 *Let* fix *be a recursion combinator defined by* $\mathsf{fix} \triangleq \lambda h.h(\mathsf{fix}\, h)$*. Now define a letrec expression* $\mathsf{letrec}\ h = e\ \mathsf{in}\ e'$ *as a syntactic abbreviation for the term* $(\lambda h.e')(\mathsf{fix}\, \lambda h.e)$*.*

Using these definitions we can present an expression-level version of the Improvement Theorem, analogous to Corollary 13:

Theorem 3 (Sands 1996a) *If* $\lambda h.e_0$ *and* $\lambda h.e_1$ *are closed expressions, then if*

$$\mathsf{letrec}\ h = e_0\ \mathsf{in}\ e_0 \mathrel{\underset{\sim}{\rhd}}_s \mathsf{letrec}\ h = e_0\ \mathsf{in}\ e_1$$

then for all expressions e

$$\mathsf{letrec}\ h = e_0\ \mathsf{in}\ e \mathrel{\underset{\sim}{\rhd}}_s \mathsf{letrec}\ h = e_1\ \mathsf{in}\ e$$

There is a more primitive variant of this theorem expressed in terms of fix which nicely illustrates the reciprocity between the least fixed-point property and the Improvement Theorem[3]

Theorem 4

$$\begin{array}{lccc}
(i) & e_0(\mathsf{fix}\, e_0) \mathrel{\underset{\sim}{\sqsupset}} e_1(\mathsf{fix}\, e_0) & \Longrightarrow & \mathsf{fix}\, e_0 \mathrel{\underset{\sim}{\sqsupset}} \mathsf{fix}\, e_1 \\
(ii) & e_0(\mathsf{fix}\, e_0) \mathrel{\underset{\sim}{\rhd}} e_1(\mathsf{fix}\, e_0) & \Longrightarrow & \mathsf{fix}\, e_0 \mathrel{\underset{\sim}{\rhd}} \mathsf{fix}\, e_1
\end{array}$$

[3]This variant was suggested by Søren Lassen (Aarhus).

PROOF. Parts (i) and (ii) can be established easily from Theorem 1 and Theorem 2 respectively. Here we just sketch how part (ii) can be derived, since part (i) is standard. Assume that $e_0(\mathsf{fix}\, e_0) \underset{\sim}{\trianglerighteq} e_1(\mathsf{fix}\, e_0)$. Define $\mathsf{g} \triangleq e_0(\mathsf{fix}\, e_0)$; it follows from this definition that $\mathsf{g} \underset{\sim}{\lhd\!\rhd} {}^{\surd}e_0(\mathsf{fix}\, e_0) \underset{\sim}{\lhd\!\rhd} \mathsf{fix}\, e_0$. From the initial assumption we can use this to show that $e_0(\mathsf{fix}\, e_0) \underset{\sim}{\trianglerighteq} e_1\, \mathsf{g}$. By Theorem 2 we have $\mathsf{g} \underset{\sim}{\trianglerighteq} \mathsf{h}$ where $\mathsf{h} \triangleq e_1\, \mathsf{h}$. But $\mathsf{h} \underset{\sim}{\lhd\!\rhd} \mathsf{fix}\, e_1$, and hence we can conclude $\mathsf{fix}\, e_0 \underset{\sim}{\trianglerighteq} \mathsf{fix}\, e_1$ are required.

□

6 Example Application to the Correctness of Program Transformations

In this section we illustrate the application of the Improvement Theorem to the verification of the correctness of a small program transformation. The example is taken from Sands (1996b), and concerns a transformation described in Wadler (1989). More extensive examples are found in Sands (1996a), where the Improvement Theorem is used to provide a total correctness proof for an automatic transformation based on a higher-order variant of the deforestation method (Wadler 1990).

The main application studied in Sands (1996b) is the classic unfold-fold method. The problem is different from the task of verifying specific transformation methods (such as the one in this section) because, in general, unfold-fold transformations are not correct. Task is to design, with the help of the Improvement Theorem, a simple syntactic method for constraining the transformation process such that correctness is ensured.

6.1 Concatenate Vanishes

We consider a simple mechanizable transformation which aims to eliminate calls to the concatenate (or *append*) function. The effects of the transformation are well known, such as the transformation of a naive quadratic-time reverse function into a linear-time equivalent.

The systematic definition of the transformation used here is due to Wadler (Wadler 1989) (with one small modification). Wadler's formulation of this well-known transformation is completely mechanizable, and the transformation "algorithm" always terminates. Unlike many other mechanizable transformations (such as deforestation and partial evaluation), it can improve the asymptotic complexity of some programs.

The basic idea is to eliminate an occurrence of concatenate (defined in Fig. 5) of the form $\mathsf{f}\, e_1 \ldots e_n \mathbin{+\!\!+} e'$, by finding a function f^+ which satisfies

$$\mathsf{f}^+\, x_1 \ldots x_n\, y \cong (\mathsf{f}\, x_1 \ldots x_n) \mathbin{+\!\!+} y.$$

Definition 1 ("Concatenate Vanishes") The transformation has two phases: *initialization*, which introduces an initial definition for f^+, and *transformation*, which applies a set of rewrites to the right-hand sides of all definitions.

Initialization For some function $\mathsf{f}\ x_1 \ldots x_n \triangleq e$, for which there is an occurrence of a term $(\mathsf{f}\ e_1 \ldots e_n) +\!\!+ e'$ in the program, define a *new* function

$$\mathsf{f}^+ x_1 \ldots x_n\ y \triangleq e +\!\!+ y.$$

Transformation Apply the following rewrite rules, in any order, to all the right-hand sides of the definitions in the program:

$$\begin{array}{lrcl}
(1) & \mathsf{nil} +\!\!+ x & \to & x \\
(2) & (x : y) +\!\!+ z & \to & x : (y +\!\!+ z) \\
(3) & (x +\!\!+ y) +\!\!+ z & \to & x +\!\!+ (y +\!\!+ z) \\
(4) & (\ \mathsf{case}\ x\ \mathsf{of} & \to & \mathsf{case}\ x\ \mathsf{of} \\
 & \quad c_1(\vec{y}_1) \Rightarrow e_1 & & \quad c_1(\vec{y}_1) \Rightarrow (e_1 +\!\!+ z) \\
 & \quad \ldots & & \quad \ldots \\
 & \quad c_n(\vec{y}_n) \Rightarrow e_n) +\!\!+ z & & \quad c_n(\vec{y}_n) \Rightarrow (e_n +\!\!+ z) \\
(5) & (\mathsf{f}\ x_1 \ldots x_n) +\!\!+ y & \to & \mathsf{f}^+\ x_1 \ldots x_n\ y \\
(6) & (\mathsf{f}^+\ x_1 \ldots x_n\ y) +\!\!+ z & \to & \mathsf{f}^+\ x_1 \ldots x_n\ (y +\!\!+ z)
\end{array}$$

In rule (4) (strictly speaking it is a rule *schema*, since we assume an instance for each vector of expressions $e_1 \ldots e_n$) it is assumed that z is distinct from the pattern variables $\vec{y}_i$.

Henceforth, let $\to$ be the rewrite relation generated by the above rules (i.e., the compatible closure) and $\to^+$ be its transitive closure.

It should be clear that the rewrites can only be applied a finite number of times, so the transformation always terminates—and the rewrite system is Church-Rosser (although this property is not needed for the correctness proof).

Example 16 *The following example illustrates the effect of the transformation:* itrav *computes the* inorder *traversal of a binary tree. Trees are assumed to be built from a nullary* leaf *constructor, and a ternary* node*, comprising a left subtree, a node-element, and a right subtree.*

$$\begin{array}{l}
\mathsf{itrav}\ t \triangleq\ \ \mathsf{case}\ t\ \mathsf{of} \\
\qquad\qquad \mathsf{leaf} \Rightarrow \mathsf{nil} \\
\qquad\qquad \mathsf{node}(l, n, r) \Rightarrow (\mathsf{itrav}\ l) +\!\!+ (n : \mathsf{itrav}\ r).
\end{array}$$

The second branch of the case expression is a candidate for the transformation, so we define:

$$\begin{array}{l}
\mathsf{itrav}^+\ t\ y \triangleq (\mathsf{case}\ t\ \mathsf{of} \\
\qquad\qquad \mathsf{leaf} \Rightarrow \mathsf{nil} \\
\qquad\qquad \mathsf{node}(l, n, r) \Rightarrow (\mathsf{itrav}\ l) +\!\!+ (n : \mathsf{itrav}\ r) \\
\qquad\quad) +\!\!+ y
\end{array}$$

Now we transform the right-hand sides of these two definitions, respectively:

$$\begin{array}{l}\mathsf{case}\ t\ \mathsf{of}\ \ \mathsf{leaf} \Rightarrow \mathsf{nil}\\ \quad \mathsf{node}(l,n,r) \Rightarrow (\mathsf{itrav}\ l) +\!\!+ (n : \mathsf{itrav}\ r)\\ \rightarrow \begin{array}{l}\mathsf{case}\ t\ \mathsf{of}\ \ \mathsf{leaf} \Rightarrow \mathsf{nil}\\ \quad \mathsf{node}(l,n,r) \Rightarrow \mathsf{itrav}^{+}\ l\ (n : \mathsf{itrav}\ r)\end{array}\end{array}$$

$$\begin{array}{ll}\multicolumn{2}{l}{(\mathsf{case}\ t\ \mathsf{of}\ \ \mathsf{leaf} \Rightarrow \mathsf{nil}}\\ \multicolumn{2}{l}{\quad \mathsf{node}(l,n,r) \Rightarrow (\mathsf{itrav}\ l) +\!\!+ (n : \mathsf{itrav}\ r)) +\!\!+ y}\\ \rightarrow & \mathsf{case}\ t\ \mathsf{of}\ \ \mathsf{leaf} \Rightarrow \mathsf{nil} +\!\!+ y\\ & \quad \mathsf{node}(l,n,r) \Rightarrow ((\mathsf{itrav}\ l) +\!\!+ (n : \mathsf{itrav}\ r)) +\!\!+ y\\ \rightarrow^{+} & \mathsf{case}\ t\ \mathsf{of}\ \ \mathsf{leaf} \Rightarrow y\\ & \quad \mathsf{node}(l,n,r) \Rightarrow \mathsf{itrav}^{+}\ l\ (n : \mathsf{itrav}^{+}\ r\ y)\end{array}$$

The resulting expressions are taken as the right-hand sides of new versions of itrav *and* itrav^{+} *respectively (where we elide the renaming):*

$$\begin{array}{lll}\mathsf{itrav}\ t & \triangleq & \mathsf{case}\ t\ \mathsf{of}\ \ \mathsf{leaf} \Rightarrow \mathsf{nil}\\ & & \quad \mathsf{node}(l,n,r) \Rightarrow \mathsf{itrav}^{+}\ l\ (n : \mathsf{itrav}\ r)\\ \\ \mathsf{itrav}^{+}\ t\ y & \triangleq & \mathsf{case}\ t\ \mathsf{of}\ \ \mathsf{leaf} \Rightarrow y\\ & & \quad \mathsf{node}(l,n,r) \Rightarrow \mathsf{itrav}^{+}\ l\ (n : \mathsf{itrav}^{+}\ r\ y)\end{array}$$

The running time of the original version is quadratic (worst case) in the size of the tree, while the new version is linear (when the entire result is computed).

The following correctness result for *any* transformation using this method shows that the new version must be an *improvement* over the original, which implies that the new version never leads to more function calls, regardless of the context in which it is used.

6.2 Correctness

It is intuitively clear that each rewrite of the transformation is an equivalence; the first two rules comprise the definition of concatenate; the third is the well-known associativity law; the fourth is a consequence of distribution law for case expressions; and the last two follow easily from the preceding rules and the initial definitions. This is sufficient (by Theorem 1) to show that the new versions of functions are less in the operational order than the originals, but does not guarantee equivalence. In particular note that rule (5) gives the transformation the ability to introduce recursion into the definition of the new auxiliary functions. To prove total correctness we apply the Improvement Theorem; it is sufficient to verify that the transformation rewrites are all contained in the strong improvement relation.

Proposition 17 *The transformations rules* (1)–(6) *are strong improvements.*

PROOF. [Outline] Using the context lemma for improvement it is sufficient to consider only closed instances of the rewrites. Rules (1) and (2) are essentially just unfoldings of the standard definition of concatenate and thus are improvements. Rule (3) can be proved from the operational semantics by showing that its reflexive closure is an improvement simulation (it is also proved using a new proof technique in the next section). Rule (4) can be proved with the help of Proposition 9, observing that the context $[\;]{+\!\!+}z$ unfolds to a reduction context. Rule (5) follows directly from the definition of f^{+} provided by the initialization, since after two reduction steps on each side of the laws the left and right-hand sides are identical. Furthermore, this law is a "cost equivalence" — it is also an improvement in the other direction, and so for (6) we have that:

$$\begin{array}{lcll}
(\mathsf{f}^{+}\, x_1 \ldots x_n\, y){+\!\!+}z & \underset{\sim}{\lhd\!\rhd} & ((\mathsf{f}\, x_1 \ldots x_n){+\!\!+}y){+\!\!+}z & (\text{since } (5) \subset (\underset{\sim}{\lhd\!\rhd})) \\
 & \underset{\sim}{\rhd}_s & (\mathsf{f}\, x_1 \ldots x_n){+\!\!+}(y{+\!\!+}z) & (\text{by } (3)) \\
 & \underset{\sim}{\rhd}_s & \mathsf{f}^{+}\, x_1 \ldots x_n\, (y{+\!\!+}z) & (\text{by } (5))
\end{array}$$

□

Then we get the following from the Improvement Theorem (Corollary 13).

Proposition 18 *The transformation is correct, and the resulting functions are improvements over the originals.*

7 Proof Techniques Based on Improvement

In this section we consider a new application: the construction of proof techniques for $\cong$ which make use of the improvement relation.

7.1 Equivalence by Transformation

A method for correctly transforming programs provides us with a natural method for proving the equivalence of expressions. For example, let e and e' be two arbitrary expressions, and let the sequence of variables $\vec{x}$ contain all their free variables. Then e and e' can be proved equivalent by showing that the new (nonrecursive) functions f and f' defined by

$$\begin{array}{lcl}
\mathsf{f} & \triangleq & \lambda\vec{x}.e \\
\mathsf{f}' & \triangleq & \lambda\vec{x}.e'
\end{array}$$

can be transformed into a common third function.

This approach to equivalence proofs was suggested by Kott (1982). Kott used a variant of the unfold-fold transformation method [4] to transform functions f and f' into functions g and g' respectively, such that g and g' are syntactically identical up to renaming. Kott called this "the Mc Carthy method" after McCarthy's recursion induction principal (McCarthy 1967).

In this section we present a proof technique for equivalence which is derived from this transformational viewpoint, using the improvement theorem. However, our method will abstract away from the transformational origins of the approach. The method will not explicitly construct functions like f and f' above—except in the correctness proof of the method itself.

7.2 Strong Improvement Up To Context

We introduce some notation to simplify the presentation. Let $\rhd$ denote a refinement of strong improvement given by

$$e \rhd e' \iff e \underset{\sim_s}{\rhd} {}^{\surd}e'.$$

Definition 19 *If R is a binary relation on expressions, let R^{C} denote the* closure under substitution and context *of R, given by*

$$\{(C[e_1\sigma_1, \ldots, e_n\sigma_n], C[e_1'\sigma_1, \ldots, e_n'\sigma_n] \mid e_i \; R \; e_i', i \in 1 \ldots n\},$$

where C denotes an arbitrary n-holed context, and the σ_i are arbitrary substitutions.

Definition 20 (bisimulation up to improvement and context) *A binary relation on expressions R is a* bisimulation up to improvement and context *if*

whenever $e_1 \; R \; e_2$, then there exists expressions e_1' and e_2' such that

$$e_1 \rhd e_1', \; e_2 \rhd e_2', \quad \textit{and } e_1' R^{\mathsf{C}} e_2'.$$

Theorem 5 *If R is a bisimulation up to improvement and context, then $R \subseteq \cong$.*

PROOF. Let $R = \{(e_i, e_i')\}_{i \in I}$. Then by definition of improvement context simulation, for each $i \in I$ there exist contexts C_i, $i \in I$ such that $e_i \rhd C_i[\vec{e}]$ and $e_i' \rhd C_i[\vec{e}\,']$ where where the respective pairs of expressions from $\vec{e}$ and $\vec{e}\,'$ are substitution instances of pairs in R. To simplify the exposition, assume that the C_i

[4]Unfortunately, in (Kott 1982) the proposed method of ensuring that the unfold-fold transformations are correct is unsound. Kott states (Proposition 1, *loc. cit.*) that unfold-fold transformations for a first-order call-by-value language are correct whenever the number of folds does not exceed the number of unfolds. A counterexample can be found in Sands (1996b)(Section 1). Kott's oversight is that the language in question contains a non-strict operator—the conditional. Without this operator the Proposition is sound, but is not useful.

have only one distinct type of hole. The generalisation of the proof to the case of "polyadic" contexts is completely straightforward, but notationally cumbersome.

So for each $(e_i, e'_i) \in R$ we have a substitution σ_i such that $e_i \rhd C_i[e_j\sigma_i]$ and $e'_i \rhd C_i[e'_j\sigma_i]$ for some $j \in I$. Thus we define the following set of functions:

$$\left\{ \begin{array}{lll|l} \mathsf{f}_i & \triangleq & \lambda\vec{x}_i.C_i[e_j\sigma_i], & e_i \rhd C_i[e_j\sigma_i],\ \ \vec{x}_i = \mathrm{FV}(e_i\ C_i[e_j\sigma_i]\ e'_i\ C_i[e'_j\sigma_i]) \\ \mathsf{g}_i & \triangleq & \lambda\vec{x}_i.C_i[e'_j\sigma_i] & e'_i \rhd C_i[e'_j\sigma_i] \end{array} \right\}$$

Since $\rhd \subseteq \cong$, we have that $e_i \cong C_i[e_j\sigma_i]$ and $e'_i \cong C_i[e'_j\sigma_i]$. Hence it is sufficient to prove that $\mathsf{f}_i \cong \mathsf{g}_i$, $i \in I$. We will do this by showing that the f_i and the g_i can be correctly transformed, respectively, into functions f'_i and g'_i such that f'_i and g'_i are syntactically identical.

From the above definitions is easy to see that, for each $k \in I$

$$\begin{array}{lll} e_k & \underset{\sim}{\rhd}_s & {}^{\surd}C_k[e_l\sigma_k] \ \text{ for some } l \in I \\ & \underset{\sim}{\lhd\!\rhd} & \mathsf{f}_k\,\vec{x}_k \end{array}$$

Thus we have $C_i[e_j\sigma_i] \underset{\sim}{\rhd}_s C_i[(\mathsf{f}_j\,\vec{x}_j)\sigma_i]$. Using this fact together with The Improvement Theorem (viz. Corollary 13) we have that $\mathsf{f}_i \underset{\sim}{\rhd}_s \mathsf{f}'_i$ where

$$\mathsf{f}'_i \triangleq \lambda\vec{x}_i.C_i[(\mathsf{f}'_j\,\vec{x}_j)\sigma_i].$$

Following the same lines we can show that $\mathsf{g}_i \underset{\sim}{\rhd}_s \mathsf{g}'_i$ where

$$\mathsf{g}'_i \triangleq \lambda\vec{x}_i.C_i[(\mathsf{g}'_j\,\vec{x}_j)\sigma_i].$$

Since these function definitions are identical modulo naming, clearly we have $\mathsf{f}_i \underset{\sim}{\rhd}_s \mathsf{f}'_i \underset{\sim}{\lhd\!\rhd} \mathsf{g}'_i$ and $\mathsf{g}_i \underset{\sim}{\rhd}_s \mathsf{g}'_i$. Since operational equivalence is contained in strict improvement, this concludes the proof. □

Taking the special case of bisimulation up to improvement and contextwhich contains just a single pair of expressions (and where the context is unary), we obtain:

Corollary 21 *Expressions e and e' are operationally equivalent if there exists some context C, and some substitution σ such that*

$$e \rhd C[e\sigma] \quad \textit{and} \quad e' \rhd C[e'\sigma]$$

We mention this special case because it turns out to be sufficient for many examples. The following variation of the above Theorem enables us to prove strong improvement properties:

Definition 22 (strong improvement up to improvement and context) *A binary relation on expressions R is a* strong improvement up to improvement and context *if whenever $e_1\ R\ e_2$, then there exists expressions e'_1 and e'_2 such that*

$$e_1 \rhd e'_1, \quad e_2 \underset{\sim}{\lhd\!\rhd} {}^{\surd}e'_2, \quad \textit{and } e'_1 R^{\mathrm{C}} e'_2.$$

Proposition 23 *If R is a strong improvement up to improvement and context then $R \subseteq \underset{\sim}{\rhd}_s$.*

PROOF. (Sketch) As in the proof of Theorem 5, except that we obtain $e'_i \underset{\sim}{\lhd\!\rhd} \mathsf{g}_i \underset{\sim}{\lhd\!\rhd} \mathsf{g}'_i$, by making use of Proposition 14. □

There are many other obvious variations of these proof methods along similar lines; for example for proving weaker properties like operational approximation ($\underset{\sim}{\sqsubset}$) and improvement ($\underset{\sim}{\rhd}$).

7.3 Examples

Here we consider a few simple examples to illustrate the application of the proof techniques given above. Corollary 21 will in fact be sufficient for current purposes. We will routinely omit simple calculations where they only employ the rules of the tick algebra.

Definition 2 *A context C is a* pseudo reduction context *if it is a single-holed context which does not capture variables, and which satisfies the following properties:*

1. $C[{}^{\surd}e] \underset{\sim}{\lhd\!\rhd} {}^{\surd}C[e]$
2. $C[\mathsf{case}\ e_0\ \mathsf{of}\ pat_1 \Rightarrow e_1 \ldots pat_n \Rightarrow e_n\ ] \underset{\sim}{\lhd\!\rhd}$
   $\mathsf{case}\ e_0\ \mathsf{of}\ pat_1 \Rightarrow C[e_1] \ldots pat_n \Rightarrow C[e_n]$, $\quad \mathrm{FV}(C)$ *distinct from* $\mathrm{FV}(pat_i)$

In what follows, we state that certain contexts are pseudo reduction contexts by way of a proof hint. To establish this property in all cases below requires nothing more than the tick algebra, together with Proposition 9 (the propagation-rule for reduction contexts).

Associativity of Append We can prove the well-known associativity property of append by simply showing that $R = \{((x{+\!\!+}y){+\!\!+}z, x{+\!\!+}(y{+\!\!+}z))\}$ is a bisimulation up to improvement and context. In fact, in Section 6 we used a stronger property of the pair above, namely that it is contained in strong improvement. This is shown by proving that R is a strong improvement up to improvement and context. Contexts of the form $[\ ]{+\!\!+}e$ are pseudo reduction context, and using this fact the following calculations are routine:

$$\begin{array}{lll} (x{+\!\!+}y){+\!\!+}z & \underset{\sim}{\rhd}_s & {}^{\surd}\,\mathsf{case}\ x\ \mathsf{of} \\ & & \quad \mathsf{nil} \Rightarrow y{+\!\!+}z \\ & & \quad h : t \Rightarrow h : (\boxed{(t{+\!\!+}y){+\!\!+}z}) \end{array}$$

$$\begin{array}{lll} x{+\!\!+}(y{+\!\!+}z) & \underset{\sim}{\lhd\!\rhd} & {}^{\surd}\,\mathsf{case}\ x\ \mathsf{of} \\ & & \quad \mathsf{nil} \Rightarrow y{+\!\!+}z \\ & & \quad h : t \Rightarrow h : (\boxed{t{+\!\!+}(y{+\!\!+}z)}) \end{array}$$

The subexpressions are boxed to highlight the common context. The two boxed subexpressions are just renamings of the respective expressions on the left-hand sides. This is sufficient to show that R is a strong improvement up to improvement and context, and hence that $(x{+\!\!+}y){+\!\!+}z \mathrel{\underset{\sim_s}{\rhd}} x{+\!\!+}(y{+\!\!+}z)$.

The example can, of course, be proved by considering all closed instances, and using coinduction. The points to note are that the proof works directly on open expressions, and that the "commmon context" (the expression "outside" the boxes above) does indeed capture variables.

A Filter Example Continuing with standard examples, consider map and filter defined by:

$$\begin{array}{ll}
\mathsf{filter} \triangleq & \mathsf{map} \triangleq \lambda m.\lambda xs.\ \mathsf{case}\ xs\ \mathsf{of} \\
\quad \lambda p.\lambda xs.\ \mathsf{case}\ xs\ \mathsf{of} & \qquad \mathsf{nil} \Rightarrow \mathsf{nil} \\
\qquad\qquad \mathsf{nil} \Rightarrow \mathsf{nil} & \qquad y : ys \Rightarrow (f\ y) : \mathsf{map}\ f\ ys \\
\qquad\qquad y : ys \Rightarrow & \\
\qquad\qquad \mathsf{if}\ p\,y\ \mathsf{then}\ y : \mathsf{filter}\ p\ ys & \\
\qquad\qquad\qquad \mathsf{else}\ \mathsf{filter}\ p\ ys &
\end{array}$$

Now we wish to prove that

$$e = \mathsf{map}\,f(\mathsf{filter}\,(p \circ f)\,xs) \cong (\mathsf{filter}\,p\,(\mathsf{map}\,f\,xs) = e'$$

where $(p \circ f)$ is just shorthand for $\lambda a.(p\,(f\ a))$, (and where the conditional expression in the definition of filter represents the obvious case-expression on booleans). We can do this by finding a context C, and a substitution σ such that $e \rhd C[e\sigma]$ and $e' \rhd C[e'\sigma]$. Observing that filter z [ ] and map f [ ] are pseudo reduction contexts, two simple calculations are sufficient to derive a suitable C and σ, namely:

$$\sigma = \{xs := ys\} \text{ and } C = \begin{array}{l} \mathsf{case}\ xs\ \mathsf{of} \\ \quad \mathsf{nil} \Rightarrow \mathsf{nil} \\ \quad y : ys \Rightarrow \mathsf{if}\ p\,(f\ y)\ \mathsf{then}\ f\,y : [\ ] \\ \qquad\qquad\qquad\qquad\quad \mathsf{else}\ [\ ]. \end{array}$$

Equivalence of Fixed Point Combinators The previous examples can also be proved using more standard variations of "applicative bisimulation" (e.g., see Gordon (1995)). The following example proves a property for which the usual bisimulation-like techniques are rather ineffective. The problem is to prove the equivalence of two different fixed point combinators. Define the following:

$$\begin{array}{rcl}
\mathsf{Y} & \triangleq & \lambda h.h(\mathsf{D}\,h\,(\mathsf{D}\,h)) \\
\mathsf{D} & \triangleq & \lambda h.\lambda x.h(x\,x)
\end{array}$$

$$\text{and as before, } \mathsf{fix} \triangleq \lambda h.h(\mathsf{fix}\,h)$$

We can prove that $\mathsf{Y} \cong \mathsf{fix}$ by constructing a bisimulation up to improvement and contextwhich contains the above pair of expressions. As in the previous examples, a relation which contains *only* this pair is sufficient. We begin with Y:

$$\begin{aligned}\mathsf{Y} \;&\underset{\sim}{\lhd\!\rhd}\; {}^{\surd}\lambda h.h\,(\mathsf{D}\,h\,(\mathsf{D}\,h))\\ &\underset{\sim}{\lhd\!\rhd}\; {}^{\surd}\lambda h.h\,{}^{\surd}(h\,(\mathsf{D}\,h\,(\mathsf{D}\,h)))\\ &\underset{\sim}{\lhd\!\rhd}\; {}^{\surd}\lambda h.h\,(\boxed{\mathsf{Y}}\,h)\end{aligned}$$

From the definition of fix we have immediately that $\mathsf{fix} \underset{\sim}{\lhd\!\rhd} {}^{\surd}\lambda h.h(\boxed{\mathsf{fix}}\,h)$, and we are done. By two applications of Proposition 23 we also have that $\mathsf{Y} \underset{\sim}{\lhd\!\rhd} \mathsf{fix}$. Note that if we had started with $\mathsf{Y}' \triangleq \lambda h.\mathsf{D}\,h\,(\mathsf{D}\,h)$ we would not have been able to proceed as above; but in this case $\mathsf{Y}' \underset{\sim}{\rhd}_s \mathsf{Y}$ follows by just unfolding D once.

7.4 Related Proof Techniques

We have used the terminology of "bisimulation up to" from CCS (see e.g., (Milner 1989)). The closest proof technique to that presented here is Sangiorgi's "weak bisimulation up-to context and up to expansion" which we discuss below.

Functional Proof Methods In the setting of functional languages Gordon (1995) has considered a number of variations on the basic Abramsky-style applicative bisimulation, including some "small step" versions. The method of bisimulation up to improvement and contextintroduced here is sufficiently strong to prove all the examples from (Gordon 1995) which hold in an untyped language[5], and many of the proofs become simpler, and are certainly more calculational in style. Although Gordon's methods are "complete" in theory, in practice one must be able to write down *representations* of the bisimulations in question; it does not seem possible to write down an appropriate bisimulation to prove $\mathsf{fix} \cong \mathsf{Y}$ without judicious quantification over contexts.

Pitts (Pitts 1995) has considered a variation of "bisimulation up to context" where two expressions are bisimilar up to context if they both evaluate to weak head normal forms with the same outermost constructor (or they both diverge), and the respective terms under the constructor can be obtained by substituting bisimilar terms into a common subexpression. The weakness of Pitt's method is that the "context" in question cannot capture variables.

Process Calculus Proof Methods The proof methods described in this section were derived from the improvement theorem. However, there turns out to be a very closely related proof technique developed by Sangiorgi for the pi-calculus. This relationship has influenced the terminology we have chosen.

In Sands (1991) we noted the similarity between the definition of improvement, and the efficiency preorder for CCS investigated by Arun-Kumar and Hennessy

[5]The operational equivalence in this paper is different from Gordon's because his language is statically typed, and because we observe termination of functions. The only example equivalence from (Gordon 1995) which does not hold here is the "Take Lemma".

(Arun-Kumar and Hennessy 1991). The efficiency preorder is based on the number of internal (silent) actions performed by a process, and expressed as a refinement of weak bisimulation. As with improvement, the application of the efficiency preorder to the problem of reasoning about "ordinary" equivalence (in the case of CCS, weak bisimulation) seems to have come later; in (Sangiorgi and Milner 1992) the efficiency preorder, also dubbed *expansion*, is used to provide a "bisimulation up to" proof technique for weak bisimulation.

In the setting of the pi-calculus, Sangiorgi (1995, 1994) has used a refinement of these earlier proof techniques to great effect in studying the relationships between various calculi. The proof technique in question is called "bisimulation up to context and up to $\gtrsim$", where $\gtrsim$ is the expansion relation. We will not go into the details of this proof technique here, but it contains similar ingredients to our bisimulation up to improvement and context. There are also some important differences. We use the $\rhd$ relation as an abstract alternative to using the one-step evaluation relation. This enables us to handle open expressions, but the proof technique fails to be complete (for example, it cannot prove that any pair of weak head normal forms are equivalent).

We summarise some informal correspondences between the notions in process calculus and the relations defined in this article in Table 1. Our attempts to complete this picture find a more exact correspondence between the proof techniques relating to bisimulation up to improvement and contexthave so far been unsuccessful.

Proces Calc.	Functional
silent transition ($\xrightarrow{\tau}$)	$\overset{\circ}{\mapsto}^{*}\overset{\bullet}{\mapsto}\overset{\circ}{\mapsto}^{*}$
strong bisimulation	cost equivalence $\underset{\sim}{\lhd\!\rhd}$
weak bisimulation	operational equivalence ($\cong$) (see Gordon (1995))
expansion ($\gtrsim$)	strong improvement ($\underset{\sim}{\rhd}_{s}$)
($\xrightarrow{\tau}\gtrsim$)	strict improvement ($\rhd$)

Table 1: Informal relationships to Notions in Process Algebra

Another notion in concurrency which appears to have some connection to the Improvement Theorem, but which we have not yet investigated, is the metric-space semantics in the context of timed systems (e.g., Reed and Roscoe (1986)).

Acknowledgements Alan Jeffrey and Søren Lassen provided feedback on an earlier draft and made a number of valuable suggestions. The connections between proof techniques using the Improvement Theorem and Sangiorgi's work which have been sketched in this article where discovered in collaboration with Davide Sangiorgi. Thanks to Davide for numerous interesting discussions on the subject.

References

Abramsky, S. (1990). The lazy lambda calculus. In D. Turner (Ed.), *Research Topics in Functional Programming*, pp. 65–116. Reading, Mass.: Addison-Wesley.

Arun-Kumar, S. and M. Hennessy (1991). An efficiency preorder for processes. In *TACS*. LNCS 526.

Bird, R. and P. Wadler (1988). *Introduction to Functional Programming*. Prentice Hall.

Burstall, R. and J. Darlington (1977). A transformation system for developing recursive programs. *J. ACM 24*(1), 44–67.

Courcelle, B. (1979). Infinite trees in normal form and recursive equations having a unique solution. *Mathematical Systems Theory 13*(1), 131–180.

Felleisen, M., D. Friedman, and E. Kohlbecker (1987). A syntactic theory of sequential control. *Theoretical Computer Science 52*(1), 205–237.

Gordon, A. D. (1995). Bisimilarity as a theory of functional programming. Technical Report BRICS NS-95-3, BRICS, Aarhus University, Denmark. Preliminary version in MFPS'95.

Howe, D. J. (1989). Equality in lazy computation systems. In *The 4th Annual Symposium on Logic in Computer Science*, New York, pp. 198–203. IEEE.

Jones, N. D., C. Gomard, and P. Sestoft (1993). *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, N.J.: Prentice-Hall.

Kott, L. (1978). About transformation system: A theoretical study. In B. Robinet (Ed.), *Program Transformations*, Paris, pp. 232–247. Dunod.

Kott, L. (1982). The Mc Carthy's recursion induction principle: "oldy" but "goody". *Calcolo 19*(1), 59–69.

McCarthy, J. (1967). *A Basis for a Mathematical Theory of Computation*. Amsterdam: North-Holland.

Milner, R. (1977). Fully abstract models of the typed lambda-calculus. *Theoretical Computer Science 4*(1), 1–22.

Milner, R. (1989). *Communication and Concurrency*. Englewwood Cliffs, N.J.: Prentice-Hall.

Pettorossi, A. and M. Proietti (1995). Rules and strategies for transforming functional and logic programs. To appear ACM Computing Surveys. (Preliminary version in *Formal Program Development*, LNCS 755, Springer-Verlag).

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages*. Prentice-Hall International Ltd. London.

Pitts, A. M. (1995, March). An extension of Howe's "$(-)^*$" construction to yeild simulation-up-to-context results. Unpublished Manuscript, Cambridge.

Plotkin, G. D. (1975). Call-by-name, Call-by-value and the λ-calculus. *Theoretical Computer Science 1*(1), 125–159.

Reed, G. and A. Roscoe (1986). Metric space models for real-time concurrency. In *ICALP'86*, Volume 298 of *LNCS*. Springer-Verlag.

Sands, D. (1990). *Calculi for Time Analysis of Functional Programs*. Ph. D. thesis, Dept. of Computing, Imperial College, Univ. of London, London.

Sands, D. (1991). Operational theories of improvement in functional languages (extended abstract). In *Proceedings of the 4th Glasgow Workshop on Functional Programming* (Skye, Scotland), pp. 298–311. Springer-Verlag.

Sands, D. (1993). A naïve time analysis and its theory of cost equivalence. TOPPS Rep. D-173, DIKU. Also in *Logic and Comput., 5*, 4, pp 495-541, 1995.

Sands, D. (1995a). Higher-order expression procedures. In *Proceeding of the ACM SIGPLAN Syposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'95*, New York, pp. 190–201. ACM.

Sands, D. (1995b). Total correctness by local improvement in program transformation. In *POPL '95*. ACM Press. Extended version in (Sands 1996b).

Sands, D. (1996a). Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science A*(167), xxx–xxx. To appear. Preliminary version in TAPSOFT'95, LNCS 915.

Sands, D. (1996b). Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 18*(2), 175–234.

Sangiorgi, D. (1994). Locality and non-interleaving semantics in calculi for mobile processes. Technical report, LFCS, University of Edinburgh, Edinburgh, U.K.

Sangiorgi, D. (1995). Lazy functions and mobile processes. Rapport de recherche 2515, INRIA Sophia Antipolis.

Sangiorgi, D. and R. Milner (1992). The problem of "weak bisimulation up to". In *Proceedings of CONCUR '92*, Number 789 in Lecture Notes in Computer Science. Springer-Verlag.

Wadler, P. (1989). The concatenate vanishes. Univ. of Glasgow, Glasgow, Scotland. Preliminary version circulated on the fp mailing list, 1987.

Wadler, P. (1990). Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science 73*(1), 231–248. Preliminary version in ESOP 88, Lecture Notes in Computer Science, vol. 300.