

Total Correctness by Local Improvement in the Transformation of Functional Programs

DAVID SANDS

Chalmers University and the University of Göteborg

The goal of program transformation is to improve efficiency while preserving meaning. One of the best-known transformation techniques is Burstall and Darlington's unfold-fold method. Unfortunately the unfold-fold method itself guarantees neither improvement in efficiency nor total correctness. The correctness problem for unfold-fold is an instance of a strictly more general problem: transformation by locally equivalence-preserving steps does not necessarily preserve (global) equivalence. This article presents a condition for the total correctness of transformations on recursive programs, which, for the first time, deals with higher-order functional languages (both strict and nonstrict) including lazy data structures. The main technical result is an *improvement theorem* which says that if the local transformation steps are guided by certain optimization concerns (a fairly natural condition for a transformation), then correctness of the transformation follows. The improvement theorem makes essential use of a formalized improvement theory; as a rather pleasing corollary it also guarantees that the transformed program is a formal improvement over the original. The theorem has immediate practical consequences: it is a powerful tool for proving the correctness of existing transformation methods for higher-order functional programs, without having to ignore crucial factors such as *memoization* or *folding*, and it yields a simple syntactic method for guiding and constraining the unfold-fold method in the general case so that total correctness (and improvement) is always guaranteed.

Categories and Subject Descriptors: D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.2.4 [Software Engineering]: Program Verification—*correctness proofs*; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Verification

Additional Key Words and Phrases: Correctness, improvement, operational equivalence, program transformation, unfold-fold

1. MOTIVATION

The context of this study is transformations on functional programs. Source-to-source transformation methods for recursive programs, such as *unfold-fold transformation*, *partial evaluation*, and *deforestation* [Burstall and Darlington 1977; Jones et al. 1993; Wadler 1990], proceed by performing a sequence of equivalence-

A preliminary (unpublished) version of this article was circulated in May 1994, under the same title. A short version appears in the proceedings of POPL '95.

This work was performed while the author was employed at the Department of Computer Science, University of Copenhagen (DIKU). The work was partially funded by the DART project (Danish Research Council), and the Department of Computer Science at Copenhagen University. The author is partially supported by ESPRIT BRA "Coordination."

Author's address: Department of Computing Science, Chalmers University of Technology and the University of Göteborg, S-412 96 Göteborg, Sweden; email: dave@cs.chalmers.se.

© ACM. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2) pp 175–234, March 1996.

preserving steps on the definitions in a given program.

The main goal of such methods is to improve the efficiency of programs, but not at the expense of their meaning. Program transformations should preserve the *extensional* meaning of programs in order to be of any practical value. In this case we say that the transformation is *correct*.

1.1 The Problem

Equivalence-Preserving Steps that Do Not Preserve Equivalence. 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. Yet this observation (clarified below) runs contrary to many informal (and some formal) arguments which are used in attempts to justify correctness of particular transformation methods.

The problem arises through transformation steps for which “equivalence” depends critically on (i.e., is *local* to) the function being transformed. Suppose we begin with a function definition $f\ x \triangleq e$, and we transform the right-hand side of the definition according to some equivalence $e \cong e'$, to obtain a revised definition $f\ x \triangleq e'$. If the equivalence $e \cong e'$ is independent of the definition of f (i.e., f is considered to be a free variable in e and e') then we rightly expect that the old and new definitions will be equivalent, since we expect that a reasonable definition of equivalence will be a congruence relation. But for many approaches to the transformation of programs (we might arguably say *most*), the transformation step from e to e' depends critically on the definition of f . A typical example of such a local equivalence would be an *unfold* step which replaces a recursive call to f by the corresponding instance of the body, or its inverse, a *fold* step. As a result, the new definition may not be semantically equivalent to the original. In particular it may introduce new sources of recursion, or change the structure of the recursion, leading to worse termination properties. To take a concrete (but contrived) example to illustrate this point, consider the following transformation (where \triangleq denotes a function definition, and \cong is semantic equivalence with respect to the current definition):

$$\boxed{f\ x \triangleq x + 42} \xrightarrow[\text{using } 42 \cong f\ 0]{\text{transform}} \boxed{f\ x \triangleq x + f\ 0}$$

The problem comes from the fact that the equivalence used in a transformation step (in this example, the replacement of 42 by the call $f\ 0$) is not necessarily an equivalence with respect to the *new* definition.

This article proposes a solution to this problem, in the general setting of “transformation by equivalences,” and studies the application of this solution to a particular transformation framework known as the *unfold-fold method*.

Unfold-Fold Transformations. Unfold-fold transformation [Burstall and Darlington 1977] is a very general transformation framework which employs a collection of simple syntactic equivalences, in particular, unfolding and folding, as mentioned above, and the application of laws about primitive functions. Attempts to mechanize, or at least systematize, unfold-fold transformations address the issues of *what* to transform, and in what order to apply the basic transformation rules. These

decisions form what is known as transformation *strategies* (e.g., Feather [1979] and Pettorossi and Proietti [1993]), and typical examples are *fusion*, which aims to transform nested recursive functions into a single recursion, and *tupling*, which aims to combine the computation of several independent recursive function calls. Regardless of the strategy (which usually dictates what new function definitions to construct), in practice unfold-fold transformations follow a common pattern:

- (1) Within the bodies of some function definitions, *unfold* some recursive function calls, thereby exposing computation, and possible optimization;
- (2) simplify the expressions by application of *laws* about primitive functions (for example, arithmetic laws);
- (3) *fold* instances of the original function definitions, thereby compounding the effects of the above steps so that they take effect in every recursive call.

We will illustrate the potential correctness problems even for this reasonable strategy. Let us imagine a fictitious program transformation system which, together with some heuristics, implements the above general strategy when given some definitions to transform. Let us suppose that the system is guided by the following heuristics:

- Unfold nondeterministically, but limited by some bounds on expression size and number of unfoldings;
- apply only laws which reduce the size of expressions, and do so nondeterministically and exhaustively;
- fold as many instances of the right-hand sides of the original definitions, but subject to the constraint that there should be no more fold steps than there were unfolds.

Now let us suppose that we feed the following definition into the transformation system:

```
f x  $\triangleq$  if x then (f x) else true.
```

The transformation process begins by unfolding function calls:

```
if x then (f x) else true
 $\xrightarrow{\text{unfold}}$  if x then (if x then (f x) else true)
                else true
 $\xrightarrow{\text{unfold}}$  if x then (if x then (if x then (f x) else true)
                else true)
                else true.
```

At this point we suppose that the expression is deemed to be too large to continue unfolding, so a database of laws is consulted in order to simplify this expression. The following law for conditionals can be applied:

$$\text{if } x \text{ then (if } x \text{ then } y \text{ else } z') \text{ else } z \cong \text{if } x \text{ then } y \text{ else } z$$

So applying this law twice we obtain:

$$\xrightarrow{\text{law} \times 2} \text{if } x \text{ then (f } x \text{) else true}$$

Now the transformation moves into the final stage. Recognizing that this expression is the right-hand side of the initial definition, a single fold step is performed:

$$\xrightarrow{\text{fold}} \mathbf{f} x.$$

The transformed definition is thus $\mathbf{f} x \triangleq \mathbf{f} x$ and illustrates the well-known fact that unfold-fold transformations do not, in general, preserve total correctness. It also serves as a counterexample to folk-law in functional programming which says that “more unfolds than folds” is sufficient to guarantee the correctness of the unfold-fold method. Of course, when applying unfold-fold transformations by hand, one is not likely to reproduce the transformation above, since the final definition is so obviously not equivalent to the original. But the following facts remain:

- We can make no guarantees about the correctness of automatic program transformations on the grounds that the basic transformation steps, like unfold-fold, are equivalences.
- Even for those unfold-fold transformation sequences which *are* equivalence preserving, the transformation itself does not serve as a formal (or informal) proof of this fact.

Consequences. The general problem—that transformation by local equivalences is not sound in general—has important consequences:

- (1) Some transformation methods simply do not preserve correctness in general. It is well known that this is the case for unfold-fold transformations.
- (2) Many well-known and widely studied transformation methods have not been proved to be correct, and it seems that the correctness problem has received little attention because of an implicit (and incorrect) assumption that it is sufficient to argue the correctness at the level of the basic steps. In particular we believe this to be the case for forms of partial evaluation of functional programs which involve producing specialized, potentially recursive versions of functions, as well as transformations such as deforestation and supercompilation. For these kinds of transformations, arguments of correctness based on the nature of the local transformation steps are unsatisfactory, because the transformations perform *memoization* which is analogous to using *folding* in the unfold-fold method.¹

The Contribution of this Work. This article presents a solution to the problem, which deals with higher-order functional languages (both strict and nonstrict) including lazy data structures.

The main technical result is the *Improvement Theorem*, which says that if the transformation steps are guided by certain optimization concerns (a fairly natural condition for program transformation), then correctness of the transformation follows.

¹A number of rigorous studies of correctness in partial evaluation [Gomard 1992; Palsberg 1993; Wand 1993] ignore the memoization aspects and deal with the orthogonal issue of the correctness of *binding-time analysis*, which controls *where* transformation occurs in a program. Consel and Khoo [1993] address memoization issues, but in an abstract form which does not deal with the construction of recursive programs.

The above notion of optimization is based on a formal *improvement theory*. An expression e is improved by e' if in all closing contexts C , if computation of $C[e]$ terminates, then so does $C[e']$, but requires no more evaluation steps than $C[e]$. The important property of improvement from the point of view of program transformation is that it is substitutive—an expression can be improved by improving a subexpression. For reasoning about the improvement relation a more tractable formulation and some related proof techniques are developed.

The Improvement Theorem shows that if e is improved by e' (in addition to e being operationally equivalent to e') then a transformation which replaces e by e' is totally correct; in addition this also guarantees that the transformed program is a formal improvement over the original. (Notice that in the above example, replacement of 42 by the equivalent term $f\ 0$ is not an improvement, since the latter requires evaluation of an additional function call. We will provide a more detailed analysis of the second transformation later, where we show how the improvement theory rightly prevents us from amortizing the cost of the last step against the earlier transformation steps).

The significance of the Improvement Theorem is that it finds immediate practical application to the consequences of the problem, namely:

- (1) A simple syntactic method for restricting the general (incorrect) unfold-fold method is provided. The method is based on a single annotation (whose meaning and algebraic properties are given by the improvement theory) which effectively guides and constrains transformations to guarantee correctness and improvement.
- (2) It can be applied to give total correctness proofs for existing automatic transformation methods. A notable example, which we have considered in Sands [1995b], is a higher-order variant of the well-known deforestation method [Wadler 1990]. With a new formulation of the deforestation algorithm (extended to deal with higher-order functions) the proof of correctness, including the crucial *fold-ing* process, becomes simple and modular.

In this article we focus primarily on the first of these areas, the general case of unfold-fold transformations. Although the transformations considered in the second of these areas could also be thought of in terms of the general unfold-fold method, our aims are somewhat different in these two settings. In the second case we are interested in proving the correctness of transformation methods without essentially changing or further constraining the transformation (of course, we can only do this if the transformation turns out to be correct!). In the study of unfold-fold we are dealing with a much more general transformation framework, but our task is quite different: the method is not correct in general, and so we must find a way of restricting or modifying the method to guarantee correctness.

We provide a small illustration of the second application with a correctness proof for a simple mechanizable program transformation, as described in Wadler [1989a], which aims to eliminate instances of the concatenate operation from programs. A more substantial example which includes a higher-order generalization of deforestation is detailed in Sands [1995b].

In the next section we outline the contents of the article, providing sufficient detail to enable us to state the main technical result.

2. OVERVIEW

In this section we outline the structure and contents of the remainder of the article. In particular, we state the main technical result, the Improvement Theorem.

Section 3 deals with preliminaries including the syntax and operational semantics of a simple higher-order functional language. The key points are:

- The introduction of a small untyped functional language, assuming recursive (curried) function definitions of the form $\mathbf{f} \ x_1 \dots x_k \triangleq e$;
- The definition of operational semantics, in terms of a reduction relation on expressions. This determines when a closed expression e evaluates to a value (weak head normal form) w , written $e \Downarrow w$. The operational semantics models the call-by-name strategy.
- From the operational semantics, the fundamental notions of *operational approximation* and operational equivalence are defined; e *observationally approximates* e' , written $e \sqsubseteq e'$, if for all contexts C , if evaluation of $C[e]$ terminates, then so does evaluation of $C[e']$. Expression e is deemed to be operationally equivalent to e' , $e \cong e'$, if they approximate each other.

Section 4 provides a formal definition of a transformation. The definition captures the essence of “transformation by equivalences.” We summarize the main definitions and properties:

- To simplify reasoning, a transformation is viewed as a construction of a set of new functions from a given set: given a function $\mathbf{f} \ \vec{x} \triangleq e$, a *transformation* is an equivalence $e \cong e'$, together with a new definition $\mathbf{f}' \ \vec{x} \triangleq e' \{ \mathbf{f}' / \mathbf{f} \}$.
- We say that a particular transformation of the above form is *correct* if $\mathbf{f} \cong \mathbf{f}'$.
- Not all transformations are correct. They are, however, partially correct, in the sense that $\mathbf{f}' \sqsubseteq \mathbf{f}$.

Section 5 gives the definition and properties of improvement, and the Improvement Theorem is stated; we summarize the main definitions and state the Improvement Theorem (in a simplified form):

- Improvement is defined by strengthening the requirements of operational approximation:

Definition (Improvement). Expression e is *improved by* e' , $e \triangleright e'$, if for all contexts C such that $C[e], C[e']$ are closed, if computation of $C[e]$ terminates using n function-calls then computation of $C[e']$ also terminates, but uses no more than n function-calls.

- The Improvement Theorem is given. In slightly simplified form:

IMPROVEMENT THEOREM. *If $\mathbf{f} \ \vec{x} \triangleq e$ and $e \triangleright e'$ (where the free variables of e' are contained in \vec{x}) then $\mathbf{f} \triangleright \mathbf{g}$, where $\mathbf{g} \ \vec{x} \triangleq e' \{ \mathbf{g} / \mathbf{f} \}$.*

- By combining the Improvement Theorem with the partial-correctness property for transformations (Section 4) we obtain a condition for total correctness: a transformation from $\mathbf{f} \ \vec{x} \triangleq e$ to $\mathbf{g} \ \vec{x} \triangleq e' \{ \mathbf{g} / \mathbf{f} \}$ (i.e., via equivalence $e \cong e'$) is correct if $e \triangleright e'$.

Section 6 investigates the theory of improvement:

- A binary relation R on closed expressions is defined to be an *improvement simulation* if whenever $(e, e') \in R$ then if computation of e terminates, producing w , and using n function calls, then computation of e' also terminates, producing some w' in no more than n function calls. Furthermore, informally speaking, we can say that w and w' are values of the same “kind” (e.g., they both have the same outermost constructor), and their “components” are also related by R .
- The *Improvement Context Lemma* says that e is improved by e' if and only if all closed instances are contained in some improvement simulation.
- A variation of the proof technique associated to the improvement context lemma is introduced and put to use in the proof of the Improvement Theorem.

Section 7 outlines some simple techniques for establishing the correctness of transformations which are complementary to the Improvement Theorem. They center around the fact that transformations are partially correct (cf. Section 7), and hence that if a transformation is “reversible” then correctness follows. A transformation from \mathbf{f} to \mathbf{g} is reversible if there is also a transformation from \mathbf{g} to \mathbf{h} such that \mathbf{f} and \mathbf{h} are syntactically identical up to renaming.

Section 8 provides a simple illustration of an application of the Improvement Theorem to prove the correctness of a small automatic program transformation. The program transformation in question [Wadler 1989a] is aimed at eliminating instances of the concatenate operator (append). With a slightly modified version of the transformation algorithm, correctness reduces to showing that each rewrite rule which defined the transformation is contained in the improvement relation.

The correctness argument for the original formulation illustrates the interaction between proofs using the Improvement Theorem and proofs justified by the more basic methods of Section 7.

Section 9 considers the general unfold-fold method and how to modify it to ensure that it is always correct.

To summarize, we introduce an “improved” unfold-fold method (and an extension of it) which guarantees that the program thus obtained is equivalent to the original. The problem is that fold steps, viewed in isolation, are not improvement steps. The solution we propose follows, in spirit, the idea of “more unfolds than folds,” in that it amortizes the cost of fold steps against the savings made by unfolding steps. This amortization guarantees that the local transformation steps are improvements, and correctness follows from the Improvement Theorem. To make this work (to disallow, for example, the incorrect transformation in the opening section) the unfold steps and fold steps must be related in a certain sense. This is formalized with the help of the improvement theory.

Section 10 presents a detailed survey of related work, focusing on study of correctness of transformations in declarative languages and the relationship to:

- the basic transformations (Section 7),
- the Improvement Theorem (Section 5), and
- the application of these techniques to the unfold-fold transformation (Section 9).

Section 11 concludes by considering variations in the Improvement Theorem and areas for further work.

$\mathbf{f}, \mathbf{g}, \mathbf{h} \dots$	\in	Function Name	
$x, y, z \dots$	\in	Var	
$e, e_1, e_2 \dots$	\in	Expression	$= x$
			\mathbf{f}
			$\lambda x. e$ (Lambda abstraction)
			$e_1 e_2$ (Application)
			$e_1 @ e_2$ (Strict application)
			case e of (Case expressions)
			$c_1(\vec{x}_1) : e_1$
			\vdots
			$c_n(\vec{x}_n) : e_n$
			$c(\vec{e})$ (Constructors)
			$p(\vec{e})$ (Primitive functions)

Fig. 1. Expression syntax.

An appendix contains details of the theory of operational approximation and a least fixed-point theorem which is needed in the technical development.

3. PRELIMINARIES

We summarize some of the notation used in specifying the language and its operational semantics. The subject of this study will be an untyped, higher-order, nonstrict functional language with lazy data constructors. Our technical results will be specific to this language (and its call-by-name operational semantics), but the inclusion of a strict application operator and arbitrary strict primitive functions (which could include constructors and destructors for strict data structures) should be sufficient to convince the reader that the bulk of the theory carries over to call-by-value languages without particular difficulty. Further discussion of evaluation orders can be found in the concluding section of the article.

We assume a flat set of mutually recursive function definitions of the form $\mathbf{f} x_1 \dots x_{\alpha_{\mathbf{f}}} \triangleq e_{\mathbf{f}}$ where $\alpha_{\mathbf{f}}$, the arity of function \mathbf{f} , is greater than or equal to zero. (For an indexed set of functions we will sometimes refer to the arity by index, α_i , rather than function name.) $\mathbf{f}, \mathbf{g}, \mathbf{h} \dots$, range over function names, $x, y, z \dots$ over variables, and $e, e_1, e_2 \dots$ over expressions. The syntactic categories are given in Figure 1.

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 constants include **true**, **false**, and **nil**; the expression $h.t$ will be used as shorthand for **cons**(h, t) (where **cons** is a (binary) constructor).

The primitives and constructors are not curried—they cannot be written without their full complement of operands. We can assume that the case expressions are defined for any finite subset $\{c_1 \dots c_n\}$ of constructors; the number of variables in \vec{x}_i must match the arity of the constructor c_i , and these variables are considered to be bound in e_i .

A list of zero or more expressions e_1, \dots, e_n will often be denoted \vec{e} . Application, as is usual, associates to the left, so $((\dots(e_0 e_1) \dots) e_n)$ will be written as $e_0 e_1 \dots e_n$; sometimes we will further abbreviate this to $e_0 \vec{e}$. The expression written $e\{\vec{e}'/\vec{x}\}$ will denote simultaneous capture-free substitution of a sequence of expressions \vec{e}' for free occurrences of a sequence of variables \vec{x} , respectively, in the expression e . The term $\text{FV}(e)$ will denote the free variables of expression e . Sometimes we will (informally) write substitutions of the form $\{\vec{e}'/\vec{g}\}$ to represent the replacement of occurrences of function symbols \vec{g} by expressions \vec{e}' .²

A *context*, ranged over by C, C_1 , etc., is an expression with zero or more occurrences of a “hole”, $[]$, 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 . A context is called *open* if it contains free variables.

Expressions are identified up to renaming of bound variables. This syntactic equivalence relation is denoted by \equiv . Contexts are identified up to renaming of those bound variables which cannot capture variables placed in their holes.

3.1 Operational Semantics

The purpose of the operational semantics is to define an evaluation relation \Downarrow (a partial function) between closed expressions and the “values” of computations. The operational semantics is a standard call-by-name one. Of the many ways in which such a relation could be defined, we choose a one-step reduction relation on expressions (\mapsto) whose application is governed by *reduction contexts*, in the style of Felleisen et al. [1987]. This choice is not critical to the development, but the notions of one-step reduction, and of reduction contexts, are useful.

The set of values, following the standard terminology (e.g., see Peyton Jones [1987]), are called *weak head normal forms*. The weak head normal forms

$$w, w_1, w_2, \dots \in \text{WHNF}$$

are just the constructor expressions $c(\vec{e})$, and the *Closures*, as given by the following grammar:

$$w = c(\vec{e}) \mid \text{Closures}$$

$$\begin{aligned} \text{Closures} &= \mathbf{f}e_1 \dots e_k \quad (0 \leq k < \alpha_{\mathbf{f}}) \\ &\mid \lambda x.e. \end{aligned}$$

If $e \Downarrow w$ for some closed expression e then we say that e *evaluates to weak head normal form* 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 (cf. Felleisen et al. [1987]), ranged over by \mathcal{R} , are contexts containing a single hole which is used to identify the next expression to be evaluated (reduced).

²This replacement operation must be handled with care, since the various equivalences and pre-orderings will only be closed under proper substitutions.

$\mathcal{R}[\mathbf{f} \ e_1 \dots e_{\alpha_f}] \mapsto \mathcal{R}[e_{\mathbf{f}}\{e_1 \dots e_{\alpha_f}/x_1 \dots x_{\alpha_f}\}]$	(fun)
$\mathcal{R}[(\lambda x.e)e'] \mapsto e\{e'/x\}$	(β)
$\mathcal{R}[w@w'] \mapsto \mathcal{R}[w \ w']$	(sapply)
$\mathcal{R}[\mathbf{case} \ c_i(\vec{e}) \ \mathbf{of} \ c_1(\vec{x}_1) : e_1 \dots c_n(\vec{x}_n) : e_n]$ $\mapsto \mathcal{R}[e_i\{\vec{e}/\vec{x}_i\}] \ (1 \leq i \leq n)$	(case)
$\mathcal{R}[p(\vec{c})] \mapsto \mathcal{R}[c'] \ (\text{if } \llbracket p \rrbracket \vec{c} = c')$	(prim)

Fig. 2. One-step reduction rules.

Definition 3.1.1. A reduction context \mathcal{R} is given inductively by the following grammar

$$\begin{aligned} \mathcal{R} = & [] \mid \mathcal{R} \ e \mid \mathcal{R}@e \mid w@\mathcal{R} \\ & \mid \mathbf{case} \ \mathcal{R} \ \mathbf{of} \ c_1(\vec{x}_1) : e_1 \dots c_n(\vec{x}_n) : e_n \\ & \mid p(\vec{c}, \mathcal{R}, \vec{e}). \end{aligned}$$

Now we define the one-step reduction relation on closed expressions. We assume that each primitive function p is given meaning by a partial function $\llbracket p \rrbracket$ 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 weak head normal form before the application of primitive-function can) and are only defined over constants, not over arbitrary weak head normal forms.

Definition 3.1.2. One-step reduction \mapsto is the least relation on closed expressions satisfying the rules given in Figure 2.

In each rule of the form $\mathcal{R}[e] \mapsto \mathcal{R}[e']$ in Figure 2, 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 \mathcal{R} and a redex e' such that $e_1 \equiv \mathcal{R}[e']$. Let \mapsto^+ and \mapsto^* denote respectively the transitive closure and transitive reflexive closure of \mapsto .

Definition 3.1.3. Closed expression e converges to weak head normal form w , $e \Downarrow w$, if and only if $e \mapsto^* w$.

3.2 Approximation and Equivalence

Here we define the operational approximation relation on expressions, \sqsubseteq and its associated equivalence \cong . The ordering we use is the standard Morris-style contextual ordering, or *observational approximation* [Plotkin 1975; Milner 1977]. The notion of “observation” we take is just the fact of convergence, as in the lazy lambda calculus [Abramsky 1990].

Observational 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 3.2.1.

- (1) e *observationally 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 *observationally equivalent* to e' , $e \cong e'$, if $e \sqsubseteq e'$ and $e' \sqsubseteq e$.

Remark. Note that we can make a number of variations in the form of this definition without changing its meaning. For example, if we change the condition “if $C[e] \Downarrow$ then $C[e'] \Downarrow$ ” to the requirement that

$$\text{if } C[e] \Downarrow c \text{ for some constant } c, \text{ then } C[e'] \Downarrow c \quad (*)$$

then the definition is unchanged. Note that this theory of approximation and equivalence is “lazy” in the sense of Abramsky [1990], so that we can “observe” the difference between “—” and “ $\lambda x.$ —.” This is due to the inclusion of a strict version of application into the language, so that the context $(\lambda x.\mathbf{true})@[]$ can distinguish between these expressions. If we omitted this construct and did not allow closures to be observable (as in $(*)$), then we would obtain a strictly weaker definition of approximation. However, the present theory would still be *sound* for reasoning about approximation and equivalence.

If our aim was to study operational approximation and equivalence, then it would be sufficient to assume that all functions are of arity zero. This is possible because any function definition of the form $\mathbf{f} \ x_1 \cdots x_n \triangleq e$ can be represented, up to equivalence, by a function definition of the form $\mathbf{f} \triangleq \lambda x_1 \dots \lambda x_n.e$. We take advantage of this fact in the proof of the least fixed-point property stated below. However, there are intensional differences between these definitions which become more significant when we develop and apply the theory of improvement. These differences motivate us to include both curried function definitions *and* lambda terms in the language.

3.3 The Theory of Operational Approximation

It is essential to have some characterization of observational approximation to facilitate reasoning about approximation and equivalence. Other than by defining a denotational semantics, the principal technique for functional languages is to establish some form of *context lemma*³ [Abramsky 1990; Bloom 1988; Gordon 1995; Howe 1989; Milner 1977]. We have made use of a characterization in terms of an “applicative (bi)simulation” relation [Abramsky 1990]. This provides a useful proof technique for observational approximation, and we use it to prove a least fixed-point theorem for recursively defined functions. The development is given in the appendix. Here we just state the fixed-point theorem that can be established with the help of these proof techniques.

PROPOSITION 3.3.1 (LEAST PRE-FIXED POINT). *Let $\vec{e} \equiv e_1, \dots, e_n$ be a list of expressions and $\vec{x} \equiv x_1, \dots, x_n$ be a list of variables such that $\text{FV}(\vec{e}) \subseteq \{x_1 \dots x_n\}$.*

³We use the term “Context Lemma” in a broad sense, to encompass both definitions directly extending Milner’s, and bisimulation-like characterizations of operational approximation and equivalence (e.g., Gordon [1995]).

The inequations $e_i \sqsubseteq x_i$, $i = 1 \dots n$, have a solution for $\vec{x} = \vec{g}$, where the functions \vec{g} are defined by

$$\mathbf{g}_i \triangleq e_i\{\vec{g}/\vec{x}\}, \quad i = 1 \dots n.$$

Moreover, for any other solution, $\vec{x} = \vec{e}'$, we have that $\mathbf{g}_i \sqsubseteq e'_i$, $i = 1 \dots n$.

4. DEFINING TRANSFORMATION

In this section we give a definition of transformation. The essence of the definition is that a transformation consists of equivalence-preserving modifications to the bodies of a set of definitions. This is sufficiently general to describe unfold-fold transformations and many variants (and hence too liberal to ensure correctness in general).

4.1 Transformation as Definition Construction

For the purposes of the formal definitions and results, 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 $\mathbf{f} x \triangleq e$, to a new version $\mathbf{f} x \triangleq e'$, will be formally represented by the derivation of a new function $\mathbf{g} x \triangleq e'\{\mathbf{g}/\mathbf{f}\}$, rather than by modification of the definition of \mathbf{f} . Correctness of the transformation now corresponds to validity of the statement: $\mathbf{f} \cong \mathbf{g}$.

The reason why we adopt this representation of a transformation is because observational approximation and equivalence should, strictly speaking, be parameterized by the intended set of function definitions. Such explicit parameterization is unwieldy, but we are able to avoid it by using a certain open-endedness property of the language, viz., that extending the language with new function symbols (i.e., new definitions) *conservatively extends* operational approximation and equivalence. We can similarly “garbage collect” unreachable functions without affecting equivalences which do not pertain to those functions.

To make this statement more precise, let us momentarily parameterize the language and associated relations by a set of definitions. Let F, G range over sets of function definitions, and let $L(F)$ denote the language of expressions (and contexts) whose function symbols are defined in F . Let \sqsubseteq_F denote operational approximation defined with respect to evaluation using definitions F (and defined with respect to evaluation in all $L(F)$ -contexts). So far we have implicitly assumed some fixed set of definitions F and have assumed that the right-hand sides of these definitions are expressions in $L(F)$. Now suppose we wish to extend the definitions in F with some disjoint functions G (whose bodies are in $L(F \cup G)$). The key property is summarized in the following proposition:

PROPOSITION 4.1.1. *For all $e_1, e_2 \in L(F)$, $e_1 \sqsubseteq_F e_2$ if and only if $e_1 \sqsubseteq_{F \cup G} e_2$.*

PROOF SKETCH. First note that if $e \in L(F)$ then the statement $e \Downarrow h$ is not dependent on the definitions in G . Hence the if direction of the proof is immediate. For the only-if direction we need to show that $e_1 \sqsubseteq_F e_2$ implies $e_1 \sqsubseteq_{F \cup G} e_2$. We show the contrapositive. Assume that $e_1 \not\sqsubseteq_{F \cup G} e_2$, i.e., there exists a context $C \in L(F \cup G)$ such that $C[e_1]$ converges and $C[e_2]$ does not, or vice-versa. Assume it is the first of these possibilities (the other case is similar). Now we argue that since

the language has lambda abstractions, we can construct equivalent representations of the functions in G using explicit fixed-point combinators, and hence we can find a context $C' \in L(F)$ such that $C'[e] \cong_{F \cup G} C'[e]$ for all e . Now by definition of operational equivalence we must have that $C'[e_1]$ converges and that $C'[e_2]$ does not converge. Hence $e_1 \not\cong_F e_2$, and we are done.⁴ \square

In conclusion, the implication of these properties is that we will be able to keep the set of definitions *implicit* in the theory of operational approximation and equivalence, by making sure that we never *change* the definition of a given function; transformation just adds new functions which extend the language.

Transformation. Sometimes it is also appropriate to consider weaker transformations, in which the transformation steps can increase the definedness of terms. Examples are unrestricted *unfolding* in a strict language, or optimizations for strict versions of data structures such as $\mathbf{head}(\mathbf{cons}(e_1, e_2)) \sqsubseteq e_1$. Transformations using inequalities in this direction will be called *weak* transformations. Such transformations are common in strict languages, e.g., in Turchin's *supercompiler* [Turchin 1986].

The following definition of transformation (weak transformation) will enable us to formulate the correctness problem and state some relatively standard partial correctness results.

Definition 4.1.2 (Transformation). A transformation (weak transformation, respectively) of a set of function definitions,

$$\{\mathbf{f}_i x_1 \dots x_{\alpha_i} \triangleq e_i\}_{i \in I}$$

is given by a set of expressions

$$\{e'_i\}_{i \in I}, \quad \text{FV}(e'_i) \subseteq \{x_1 \dots x_{\alpha_i}\}$$

such that $e_i \cong e'_i$ (respectively, $e_i \sqsubseteq e'_i$), together with a set of new functions (the transformed program)

$$\{\mathbf{g}_i x_1 \dots x_{\alpha_i} \triangleq e'_i\{\vec{\mathbf{g}}/\vec{\mathbf{f}}\}\}_{i \in I}.$$

We say that there is a transformation (respectively, weak transformation) from \mathbf{f}_i to \mathbf{g}_i , and it will be assumed that the function names \mathbf{g}_i are fresh.

The form of the above definition of a transformation emphasises the transformational derivation as the two-phase operation of a local transformation (of the bodies of the functions) followed by definition construction. We reason about transformations which introduce new auxiliary functions by considering that all auxiliaries are introduced at the beginning of the transformation (cf. *virtual transformation sequences* [Tamaki and Sato 1984]).

A higher-order variant of the classic Unfold-Fold transformation [Burstall and Darlington 1977] can easily be recast as a transformation according to the above

⁴The proof is rather specific to particulars of the language we are using. An alternative approach, achieving the same ends, would be to build this open-endedness property into the definition of operational approximation itself. This is not difficult, but would require us to delve more deeply into the theory of operational approximation and equivalence.

definition. Burstall and Darlington’s original definition was for a first-order functional language where functions are defined by pattern matching, and a call-by-name evaluation is assumed. The example is considered in detail in Section 9, where conditions guaranteeing total correctness are studied.

Definition 4.1.3 (Correctness). A transformation (weak transformation) from \mathbf{f}_i to \mathbf{g}_i is correct (weakly correct) if $\mathbf{f}_i \cong \mathbf{g}_i$ ($\mathbf{f}_i \sqsubseteq \mathbf{g}_i$).

An alternative way of expressing a transformation would be to say that there is a transformation from $\mathbf{f}\vec{x} \triangleq e_{\mathbf{f}}$ to new definition $\mathbf{g}\vec{x} \triangleq e_{\mathbf{g}}$ if and only if $e_{\mathbf{f}} \cong e_{\mathbf{g}}\{\mathbf{f}/\mathbf{g}\}$.

4.2 Completeness

The definition of transformation is “complete” with respect to equivalence of definitions (cf. *second-order replacement* [Kott 1980]), since if $\mathbf{f}\vec{x} \triangleq e$ is equivalent to some new function⁵ $\mathbf{g}\vec{x} \triangleq e'$, it follows that $e \cong e'$, and hence that $e \cong e'\{\mathbf{f}/\mathbf{g}\}$; this latter equivalence defines a transformation from \mathbf{f} to \mathbf{g} .

This shows that unfold-fold transformations are only a special case (of transformation), since the unfold-fold method is known to be incomplete in this sense (see Boudol and Kott [1983], Kott [1980], and Zhu [1994]). Of course, the problem we address in this article is that both the general transformation and the special case of unfold-fold are not *sound*.

4.3 Partial Correctness

The examples in the introduction show that not all transformations are correct. We conclude this section with a partial-correctness result which generalizes the well-known partial correctness result for unfold-fold transformations: if \mathbf{f} is transformed to \mathbf{g} then it is easily verified that \mathbf{f} “satisfies” \mathbf{g} ’s defining equation. So \mathbf{f} is a fixed-point of \mathbf{g} ’s definition. But \mathbf{g} is the *least* fixed-point, and hence $\mathbf{g} \sqsubseteq \mathbf{f}$.

PROPOSITION 4.3.1. *Suppose functions $\{\mathbf{f}_i x_1 \dots x_{\alpha_i} \triangleq e_{\mathbf{f}_i}\}_{i \in I}$ are transformed to*

$$\{\mathbf{g}_i x_1 \dots x_{\alpha_i} \triangleq e_{\mathbf{g}_i}\}_{i \in I},$$

then for all $i \in I$, $\mathbf{g}_i \sqsubseteq \mathbf{f}_i$.

PROOF. We give the case of a single function of arity zero. The generalization to a set of functions of arbitrary arity is straightforward. Given $\mathbf{f} \triangleq e_{\mathbf{f}}$, suppose there is a transformation to $\mathbf{g} \triangleq e_{\mathbf{g}}$. From the definition of transformation, this implies that we must have $e_{\mathbf{f}} \cong e_{\mathbf{g}}\{\mathbf{f}/\mathbf{g}\}$. Since $\mathbf{f} \cong e_{\mathbf{f}}$ we have that $\mathbf{f} \cong e_{\mathbf{g}}\{\mathbf{f}/\mathbf{g}\}$. Now we can apply Proposition 3.3.1 (we have shown that \mathbf{f} is a fixed point of \mathbf{g} ’s defining equation), and we conclude that $\mathbf{g} \sqsubseteq \mathbf{f}$. \square

A version of this result appears in Courcelle [1979] where it is formulated for recursive program schemes (first-order functional programs), but in a more general form.⁶

⁵ \mathbf{g} being “new” means that \mathbf{f} does not depend on \mathbf{g} .

⁶Specifically: the set of all fixed points of the defining equation of the original definition are also fixed points of the derived definition.

4.4 Weak Transformations

In the case of weak transformation, in general we cannot guarantee that the original and transformed programs will be related by operational approximation.

PROPOSITION 4.4.1. *In general, weak transformations are not even partially correct.*

PROOF. Let $\mathbf{f} x \triangleq \mathbf{if} x \mathbf{then} 1 \mathbf{else} (\mathbf{f} \mathbf{false})$. Then since $\mathbf{f} \mathbf{true} \cong 1$ and $\mathbf{f} \mathbf{false} \sqsubseteq 2$ then

$$\mathbf{if} x \mathbf{then} 1 \mathbf{else} (\mathbf{f} \mathbf{false}) \sqsubseteq \mathbf{if} x \mathbf{then} (\mathbf{f} \mathbf{true}) \mathbf{else} 2.$$

So there is a weak transformation from \mathbf{f} to \mathbf{g} where $\mathbf{g} x \triangleq \mathbf{if} x \mathbf{then} (\mathbf{g} \mathbf{true}) \mathbf{else} 2$, but \mathbf{f} and \mathbf{g} are incomparable. \square

We conjecture that the result of a weak transformation will be *consistent* with the original definition, in the sense that if there is a context in which use of either the original or of the transformed functions both result in some weak head normal forms, then the respective weak head normal forms will either both be closures, or both will be terms with the same outermost constructor.

5. CORRECTNESS BY IMPROVEMENT

This section presents the main technical result of the article. It says, roughly speaking, that a transformation from $\mathbf{f} x \triangleq e$, via equivalence $e \cong e'$ to $\mathbf{g} x \triangleq e' \{\mathbf{g}/\mathbf{f}\}$, is totally correct if e' is an *improvement* over e . The improvement relation is expressed in terms of the number of nonprimitive function calls; e is improved by e' (written $e \succcurlyeq e'$) if in all closing contexts $C[e]$ requires evaluation of no fewer nonprimitive functions than $C[e']$.

5.1 Improvement

An intuition behind the use of improvement to obtain total correctness (at least for the case when the program computes a constant) is that improvement $e \succcurlyeq e'$ represents some “progress” toward convergence, and in the transformation to $\mathbf{g} x \triangleq e' \{\mathbf{g}/\mathbf{f}\}$, this progress is enjoyed on every recursive call.

The main problem with formulating an appropriate notion of improvement is that the language has a nondiscrete data domain (i.e., lazy data and higher-order functions) as the results of programs. For this reason, as for operational approximation and equivalence, it is natural to define this as a contextual (pre)congruence.

Definition 5.1.1. Closed expression e converges in n -steps to weak head normal form w , $e \Downarrow_n w$, if $e \Downarrow w$, and this computation requires n reductions of nonprimitive functions using rule (*fun*).

Notation. A *reduction* of a nonprimitive function corresponds to replacing a call instance $\mathbf{f} e_1 \dots e_{\alpha \mathbf{f}}$ with the corresponding instance of the body of \mathbf{f} — namely, $e_{\mathbf{f}} \{e_1 \dots e_{\alpha \mathbf{f}}/x_1 \dots x_{\alpha \mathbf{f}}\}$ — within some reduction context \mathbb{R} , i.e.,

$$\mathbb{R}[\mathbf{f} e_1 \dots e_{\alpha \mathbf{f}}] \mapsto \mathbb{R}[e_{\mathbf{f}} \{e_1 \dots e_{\alpha \mathbf{f}}/x_1 \dots x_{\alpha \mathbf{f}}\}].$$

If $e \mapsto e'$ according to this rule then we write $e \dot{\mapsto} e'$. If $e \mapsto e'$ by any other rule then we write $e \overset{\circ}{\mapsto} e'$.

Notice that we choose not to count beta reduction steps. This increases the flexibility of the Improvement Theorem, since we have the option of representing certain “administrative” computations using lambda expressions, without interfering with the application of the theorem. An example application illustrating this point can be found in Sands [1995b]. If our primary interest was in performance-improvement, rather than just in correctness, then it would be wise also to count beta steps. The theory developed in this article can be easily adapted to the case where beta reductions are also counted.

It will be convenient to adopt the following abbreviations:

$$\begin{aligned} e \Downarrow_n &\stackrel{\text{def}}{=} \exists w. e \Downarrow_n w \\ e \Downarrow_{n \leq m} &\stackrel{\text{def}}{=} e \Downarrow_n \ \& \ n \leq m \\ e \Downarrow_{\leq m} &\stackrel{\text{def}}{=} \exists n. e \Downarrow_{n \leq m}. \end{aligned}$$

Now improvement is defined in an analogous way to observational approximation, but taking into account the number of function calls:

Definition 5.1.2 (Improvement). Expression e is improved by e' , $e \succsim e'$, if for all contexts C such that $C[e], C[e']$ are closed, if $C[e] \Downarrow_n$ then $C[e'] \Downarrow_{\leq n}$.

Some example properties of the improvement relation are given in Section 6.

5.2 The Improvement Theorem

We are now able to state the main theorem, the proof of which is outlined in the appendix.

THEOREM 5.2.1. *Given a set of function definitions*

$$\{\mathbf{f}_i x_1 \dots x_{\alpha_i} \triangleq e_i\}_{i \in I}$$

and a set of expressions

$$\{e'_i\}_{i \in I} \text{ where } \text{FV}(e'_i) \subseteq \{x_1 \dots x_{\alpha_i}\}$$

if $e_i \succsim e'_i$ then $\mathbf{f}_i \succsim \mathbf{g}_i$ where

$$\{\mathbf{g}_i x_1 \dots x_{\alpha_i} \triangleq e'_i\{\vec{\mathbf{g}}/\vec{\mathbf{f}}\}\}_{i \in I}.$$

Since improvement implies observational approximation, as a direct consequence of the theorem we have a condition for correctness and weak correctness:

COROLLARY 5.2.2. *Given a transformation (respectively, weak transformation) from $\{\mathbf{f}_i x_1 \dots x_{\alpha_i} \triangleq e_i\}_{i \in I}$ to $\{\mathbf{g}_i x_1 \dots x_{\alpha_i} \triangleq e'_i\{\vec{\mathbf{g}}/\vec{\mathbf{f}}\}\}_{i \in I}$, if $e_i \succsim e'_i$ then the transformation is correct (weakly correct), and moreover, $\mathbf{f}_i \succsim \mathbf{g}_i$.*

Cost Equivalence. Finally we note a small variation of the Improvement Theorem in the case where the transformation step exactly preserves the number of function calls. First we give a definition:

Definition 5.2.3. Let \triangleq denote *cost equivalence*, the equivalence relation associated to improvement, and given by

$$e_1 \triangleq e_2 \stackrel{\text{def}}{=} e_1 \succsim e_2 \ \& \ e_2 \succsim e_1.$$

As a consequence of this definition, if $e_1 \approx e_2$ then $C[e] \Downarrow_n \iff C[e_2] \Downarrow_n$ for all contexts C . A cost equivalence satisfying this property was first introduced in Sands [1990] for the purpose of reasoning about evaluation cost in a call-by-name functional language, and extensively studied in Sands [1993]. The following variation of the Improvement Theorem is useful in that context as well as the current one:

PROPOSITION 5.2.4. *Given a set of function definitions $\{\mathbf{f}_i x_1 \dots x_{\alpha_i} \triangleq e_i\}_{i \in I}$ and a set of expressions $\{e'_i\}_{i \in I}$ where $\text{FV}(e'_i) \subseteq \{x_1 \dots x_{\alpha_i}\}$, if $e_i \approx e'_i$ then $\mathbf{f}_i \approx \mathbf{g}_i$ where*

$$\left\{ \mathbf{g}_i x_1 \dots x_{\alpha_i} \triangleq e'_i \left\{ \frac{\bar{\mathbf{g}}}{\bar{\mathbf{f}}} \right\} \right\}_{i \in I}.$$

It may appear, at first glance, that this follows easily by a double application of the Improvement Theorem; in fact we have not been able to prove it in this manner. Fortunately, it can be proved by a very minor variation on the proof of the Improvement Theorem (given in the next section) and is therefore omitted.

6. THE THEORY OF IMPROVEMENT

This section introduces a proof technique for establishing improvements, which arises from an alternative characterization of the improvement relation. Most importantly, the proof technique is essential in our proof of the Improvement Theorem, which is given at the end of the section. Readers more interested in the applications may safely skip this section after observing the properties of improvement listed in Proposition 6.1.3, which presents a collection of important properties from the perspective of transformation (such as transitivity and congruence) and other useful properties of improvement (like the fact that it contains one-step reduction).

6.1 A Context-Lemma for Improvement

To facilitate reasoning about the improvement relation it is essential to obtain a more tractable characterization (than that provided by Definition 5.1.2)—in particular one which does not involve a quantification over all contexts. This is also the case for operational approximation, and the following characterization of the improvement relation is mirrored in our treatment of operational approximation, as detailed in the appendix.

It turns out that \approx is an instance of the improvement theories previously investigated by the author [Sands 1991] (but quite independently of the transformation problem addressed in this article). This earlier study provides some crucial technical background for “improvement” orderings in functional languages. Most importantly, it provides a definition of *improvement simulation* as a generalization of Abramsky’s *applicative bisimulation* and provides some results directly extending those of Howe [1989], which help to characterize when these relations are contextual congruences. We used these results to show that improvement can be expressed as an improvement simulation.

In what follows we describe this characterization and outline the associated proof technique (for improvement) which it provides.

Definition 6.1.1. 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) $w_1 \equiv c(e_1 \dots e_n)$, $w_2 \equiv c(e'_1 \dots e'_n)$, and $e_i \mathcal{IR} e'_i$, ($i \in 1 \dots n$) or
- (2) $w_1 \in \text{Closures}$, $w_2 \in \text{Closures}$, and for all closed e_0 , $(w_1 e_0) \mathcal{IR} (w_2 e_0)$.

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 (and in particular, the key to proving the correctness of the Improvement Theorem) is the fact that \succsim , restricted to closed expressions, is itself an improvement simulation (and is in fact the *maximal* improvement simulation). Furthermore, improvement on open expressions can be characterized in terms of improvement on all closed instances. This is summarized in the following:

LEMMA 6.1.2 (IMPROVEMENT CONTEXT LEMMA). *For all $e, e', e \succsim e'$ if and only if there exists an improvement simulation \mathcal{IR} such that for all closing substitutions σ , $e\sigma \mathcal{IR} e'\sigma$.*

PROOF. The main part of the proof (the if direction) follows from Sands [1991, Theorem 2.14], after recasting the language into the appropriate syntactic form. The only-if direction follows from the fact that the language has sufficiently many “destructors” — although the details are somewhat tedious (see, for example, the corresponding proof about a similar relation in Sands [1993, Theorem B.2]). \square

The lemma provides a basic proof technique, sometimes called *coinduction*:

—to show that $e \succsim e'$ it is sufficient to find an improvement simulation containing each closed instance of the pair.

Here are some example improvement laws which follow either directly from its definition or from the improvement context lemma:

PROPOSITION 6.1.3.

- (1) $e \succsim e' \Rightarrow e \sqsubseteq e'$ (improvement implies approximation)
- (2) \succsim on closed expressions is an improvement simulation
- (3) $e \succsim e' \Rightarrow C[e] \succsim C[e']$ (congruence)
- (4) $e \equiv e' \Rightarrow e \succsim e'$ (reflexivity)
- (5) $e \succsim e' \ \& \ e' \succsim e'' \Rightarrow e \succsim e''$ (transitivity)
- (6) $e \mapsto e' \Rightarrow e \succsim e'$ (improvement contains reduction)
- (7) $e \Downarrow w \Rightarrow e \succsim w$ (improvement contains convergence)
- (8) $\mathbf{f} \ x \succsim e$ if $\mathbf{f} \ x \triangleq e$ (unfolding is an improvement)
- (9) $e \not\succsim \mathbf{f} \ x$ if $\mathbf{f} \ x \triangleq e$ and $\exists e'. \mathbf{f} \ e' \Downarrow$ (folding is not an improvement)
- (10) $\mathcal{R}[\text{case } x \text{ of } \begin{array}{l} c_1(\vec{y}_1) : e_1 \\ \dots \\ c_n(\vec{y}_n) : e_n \end{array}] \overset{\Delta}{\succsim} \text{case } x \text{ of } \begin{array}{l} c_1(\vec{y}_1) : \mathcal{R}[e_1] \\ \dots \\ c_n(\vec{y}_n) : \mathcal{R}[e_n] \end{array}]$ (\mathcal{R} -distribute)

PROOF. (1) follows immediately from the definition that \succsim is a strengthening of the conditions for operational approximation.

For (2), observe that the union of all improvement simulations is an improvement simulation, and hence from the improvement context lemma, that improvement (restricted to closed expressions) is precisely this union.⁷

(3)–(5) follow easily from the definition of improvement.

In (6) we show that the reflexive closure (on closed expressions) of \mapsto (i.e., $\mapsto \cup \equiv$) is an improvement simulation. Since \equiv is an improvement simulation then it is sufficient to check the case when $e_1 \mapsto e_2$. Suppose, then, that $e_1 \Downarrow_n w$. Clearly from Definition 5.1.1 we have $e_2 \Downarrow_{\leq n} w$, and the remaining conditions for improvement simulation are easily satisfied; and hence $(\mapsto \cup \equiv)$ is an improvement simulation.

Part (7) follows from the preceding property, together with transitivity and reflexivity.

Part (8) says that unfolding is an improvement and is a simple instance of (6).

For (9), suppose $\mathbf{f} \ e' \Downarrow_n$ for some e' . Then since $\mathbf{f} \ e' \mapsto e\{e'/x\}$, we have $e\{e'/x\} \Downarrow_{n-1}$ and hence $e \not\Downarrow \mathbf{f} \ x$.

In (10) we show just the \succsim direction—the other direction is similar. We show that the reflexive closure of the relation containing all pairs of closed terms of the form

$$\begin{aligned} & (\mathcal{R}[\mathbf{case} \ e_0 \ \mathbf{of} \ c_1(\vec{x}_1) : e_1 \dots c_n(\vec{x}_n) : e_n], \\ & \quad \mathbf{case} \ e_0 \ \mathbf{of} \ c_1(\vec{x}_1) : \mathcal{R}[e_1] \dots c_n(\vec{x}_n) : \mathcal{R}[e_n]) \end{aligned}$$

is an improvement simulation. The reflexive part (\equiv) is straightforward. Assume that $\mathcal{R}[\mathbf{case} \ e_0 \ \mathbf{of} \ c_1(\vec{x}_1) : e_1 \dots c_n(\vec{x}_n) : e_n] \Downarrow_n w$. Now since

$$\mathcal{R}[\mathbf{case} \ [] \ \mathbf{of} \ c_1(\vec{x}_1) : e_1 \dots c_n(\vec{x}_n) : e_n]$$

is a reduction context, then it should be clear that we must have

$$\begin{aligned} & \mathcal{R}[\mathbf{case} \ e_0 \ \mathbf{of} \ c_1(\vec{x}_1) : e_1 \dots c_n(\vec{x}_n) : e_n] \\ & \quad \mapsto^* \mathcal{R}[\mathbf{case} \ c_i(\vec{e}') \ \mathbf{of} \ c_1(\vec{x}_1) : e_1 \dots c_n(\vec{x}_n) : e_n] \end{aligned}$$

and since each of these reductions is “in” e_0 , we have matching reduction steps

$$\begin{aligned} & \mathbf{case} \ e_0 \ \mathbf{of} \ c_1(\vec{x}_1) : \mathcal{R}[e_1] \dots c_n(\vec{x}_n) : \mathcal{R}[e_n] \\ & \quad \mapsto^* \mathbf{case} \ c_i(\vec{e}') \ \mathbf{of} \ c_1(\vec{x}_1) : \mathcal{R}[e_1] \dots c_n(\vec{x}_n) : \mathcal{R}[e_n] . \end{aligned}$$

Now the former derivative reduces in one more step to $\mathcal{R}[e_i\{\vec{e}'/\vec{x}_i\}]$, while the latter reduces to $\mathcal{R}[e_i\{e'/\vec{x}_i\}]$. Since reduction contexts do not bind variables, these are syntactically equivalent, and so we conclude that

$$\mathbf{case} \ e_0 \ \mathbf{of} \ \mathcal{R}[c_1(\vec{x}_1) : e_1] \dots \mathcal{R}[c_n(\vec{x}_n) : e_n] \Downarrow_n w$$

and the remaining conditions for improvement simulation are trivially satisfied, since we have reflexivity. \square

⁷Alternatively, one can equivalently define the improvement simulations to be the pre-fixed-points of a certain monotonic functional, in the standard way (e.g., see Milner [1989] and Sands [1991]), and using basic set-theoretic fixed-point theory improvement is then characterized as the maximal fixed point (and hence a pre-fixed-point).

6.2 Improvement “Up To”

To prove the Improvement Theorem we first introduce a useful variant of the improvement simulation proof technique, which is well known from CCS as “(bi)simulation up to” [Milner 1989]. First we give a more convenient definition of improvement simulation in terms of the following operation:

Definition 6.2.1. Given a relation R on closed expressions, we define R^\dagger to be the least relation on weak head normal forms such that

- $c(e_1 \dots e_n) R^\dagger c(e'_1 \dots e'_n)$ if $e_i R e'_i$, $i \in \{1 \dots n\}$ and
- $e R^\dagger e'$ if $e, e' \in \text{Closures}$, and for all closed e'' , $(e e'') R (e' e'')$.

So the definition of an improvement simulation can now be written more compactly as:

— A relation \mathcal{IR} on closed expressions is an *improvement simulation* if for all e_1, e_2 , whenever $e_1 \mathcal{IR} e_2$, if $e_1 \Downarrow_n w_1$ then $e_2 \Downarrow_{\leq n} w_2$ for some w_2 such that $w_1 \mathcal{IR}^\dagger w_2$.

If S and T are relations then $S;T$ denotes the relational composition given by the following:

$$a (S;T) b \text{ if and only if } a S a' \text{ and } a' T b \text{ for some } a'.$$

The following properties about $_^\dagger$ will be useful:

LEMMA 6.2.2. *For all weak head normal forms w, w' ,*

- (1) $w \succ^\dagger w' \iff w \succ w'$ and
- (2) $w P^\dagger; Q^\dagger w' \Rightarrow w (P; Q)^\dagger w'$.

The proof is straightforward from the definitions and is omitted.

The up-to proof technique relaxes the condition that the “components” of matching weak head normal forms of related terms must be exactly related by the simulation relation in question—but they should be related up to improvement:

Definition 6.2.3. A relation \mathcal{IR} on closed expressions is an *improvement simulation up to improvement* (abbreviated to an *i-simulation*) if for all e_1, e_2 , whenever $e_1 \mathcal{IR} e_2$, if $e_1 \Downarrow_n w_1$ then $e_2 \Downarrow_{\leq n} w_2$ for some w_2 such that $w_1 \succ; R^\dagger; \succ w_2$.

PROPOSITION 6.2.4. *To prove $e_1 \succ e_2$ it is sufficient to find an i-simulation which contains each closed instance of the pair.*

PROOF. It is enough to show that if R is an i-simulation then $R \subseteq \succ$. First we show that $\succ; R; \succ$ is a simulation. Suppose that $e_1 \succ; R; \succ e_2$ and that $e_1 \Downarrow_m w_1$. Then using the fact that \succ is a simulation, we have the following, for some $e'_1, e'_2, w_2, w'_1, w_2$:

$$\begin{array}{ccccc} e_1 & \succ & e'_1 & R & e'_2 & \succ & e_2 \\ \Downarrow_m & & \Downarrow_{m' \leq m} & & \Downarrow_{n \leq m'} & & \Downarrow_{n' \leq n} \\ w_1 & \succ^\dagger & w'_1 & \succ; R^\dagger; \succ & w'_2 & \succ^\dagger & w_2 \end{array}$$

From Lemma 6.2.2, and transitivity of \succ , it follows that $w_1 (\succ; R; \succ)^\dagger w_2$ in the bottom line of the diagram, and thus $\succ; R; \succ$ is a simulation. This shows that $(\succ; R; \succ) \subseteq \succ$. Since improvement is reflexive, it follows that $R \subseteq \succ$. \square

6.3 On Curried Function Definitions

Consider two similar function definitions of the form $\mathbf{f}_1 x_1 \cdots x_n \triangleq e$ and $\mathbf{f}_2 \triangleq \lambda x_1 \dots \lambda x_n. e$. We will use certain similarities between such functions in order to simplify proofs involving the improvement relation. However, we will also point out certain differences which explain why we study a language which permits both forms of definition.

Firstly, note that the expressions \mathbf{f}_1 and \mathbf{f}_2 are not cost equivalent. The reason is that \mathbf{f}_1 is a weak head normal form, so it evaluates in zero steps to itself. The expression \mathbf{f}_2 evaluates in one step to the weak head normal form $\lambda x_1 \dots \lambda x_n. e$.

One might feel that the distinction between \mathbf{f}_1 and \mathbf{f}_2 is unnecessarily pedantic. However, if we were to weaken the theory of improvement adding the equation $\mathbf{f}_1 \triangleq \mathbf{f}_2$, then the ‘‘Improvement Theorem’’ would become unsound.⁸

Eliminating Functions of Arity > 0. It is possible to express \mathbf{f}_1 in terms of \mathbf{f}_2 as follows:

$$\mathbf{f}_1 \triangleq \lambda x_1 \dots \lambda x_n. \mathbf{f}_2 x_1 \cdots x_n.$$

This is possible because the extra beta reductions involved in applying the expression on the right are not counted in the definition of improvement. More generally, any reasoning involving functions of arity > 0 (like \mathbf{f}_1) can be reduced to reasoning about functions of arity zero (like \mathbf{f}_2) by using the following:

PROPOSITION 6.3.1. *For any function definition $\mathbf{f} x_1 \dots x_n \triangleq e$, we have $\mathbf{f} \triangleq \lambda x_1 \dots \lambda x_n. \mathbf{g} x_1 \dots x_n$ where $\mathbf{g} \triangleq \lambda x_1 \dots \lambda x_n. e \{ \lambda x_1 \dots \lambda x_n. \mathbf{g} x_1 \cdots x_n / \mathbf{f} \}$.*

The proposition extends to sets of mutually recursive functions in the obvious way. The proof proceeds by showing that the symmetric closure of

$$R = \{ e' \{ \mathbf{f} / y \}, e' \{ \lambda x_1 \dots \lambda x_n. \mathbf{g} / y \} \mid \text{FV}(e') \subseteq \{ y \} \}$$

is an i-simulation. We omit the details.

The upshot is that in any proof about expressions involving the improvement relation we can assume, without loss of generality, that all function definitions involved have arity zero. We will take advantage of this in the proofs which follow. In particular, this means that we can assume that all closures are lambda expressions.

Using Functions of Arity > 0. There are good reasons not to abandon definitions of nonzero arity altogether. One reason is that they simplify the presentation of transformation systems and rules, in particular those which are based on first-order languages. More importantly, the definition of a transformation is sensitive to the arity of the functions involved. Transformation of a function of the form $\mathbf{f}_1 x_1 \cdots x_n \triangleq e$ must, by definition, derive a function of the same arity. For a definition of the form $\mathbf{f}_2 \triangleq \lambda x_1 \dots \lambda x_n. e$, a transformation is not obliged to leave the abstractions $\lambda x_1 \dots \lambda x_n$ intact. It turns out that having control over the arity of a definition can be a useful restriction on a transformation. We use this restriction in a crucial way in a transformation method described in Section 9.6.

⁸Proof hint: this can be shown by taking $\mathbf{f}_1 x \triangleq \mathbf{f}_1$ and $\mathbf{f}_2 \triangleq \lambda x. \mathbf{f}_1$, and showing that the extension of improvement to include $\mathbf{f}_1 \triangleq \mathbf{f}_2$ together with the Improvement Theorem allows us to conclude that $\mathbf{f}_1 \triangleq \mathbf{g}_1$ where $\mathbf{g}_1 \triangleq \mathbf{f}_1$.

6.4 Proof of the Improvement Theorem

The preceding proof technique is useful for establishing concrete improvement laws. Here we use it to establish the main theorem. We make a slight generalization of the theorem. In the sequel let $\vec{\mathbf{f}} \equiv \mathbf{f}_1 \dots \mathbf{f}_n$ and $\vec{\mathbf{g}} \equiv \mathbf{g}_1 \dots \mathbf{g}_n$ be defined as follows:

$$\left. \begin{array}{l} \mathbf{f}_i \triangleq e_i\{\vec{\mathbf{f}}/\vec{y}\} \\ \mathbf{g}_i \triangleq e'_i\{\vec{\mathbf{g}}/\vec{y}\} \end{array} \right\} i \in \{1 \dots n\}$$

where $\text{FV}(e_i, e'_i) \subseteq \vec{y}$, and $e_i\{\vec{\mathbf{f}}/\vec{y}\} \succeq e'_i\{\vec{\mathbf{f}}/\vec{y}\}$. Following Proposition 6.3.1, the assumption that all functions have arity zero is without loss of generality.

We will show that $\vec{\mathbf{f}} \succeq \vec{\mathbf{g}}$. The theorem is then a simple corollary, taking the case when e_i and e'_i contain no occurrences of \mathbf{f} or \mathbf{g} .

We prove this by constructing a relation that includes the set $\{(\mathbf{f}_i, \mathbf{g}_i) \mid i \in \{1 \dots n\}\}$ and which we show to be an i-simulation.

Definition 6.4.1. Define relation \mathcal{IS} on closed expressions by

$$\begin{aligned} \mathcal{IS} &\stackrel{\text{def}}{=} \{(e\phi, e\gamma) \mid \text{FV}(e) \subseteq \vec{y}\} \text{ where} \\ \phi &\stackrel{\text{def}}{=} \{\vec{\mathbf{f}}/\vec{y}\} \\ \gamma &\stackrel{\text{def}}{=} \{\vec{\mathbf{g}}/\vec{y}\}. \end{aligned}$$

To show that \mathcal{IS} is an improvement simulation up-to improvement we will need some technical results connecting \mathcal{IS} and the one-step reduction relation \mapsto .

Recall that improvement is measured in terms of the number of steps of the function application rule, and that if $e \mapsto e'$ according to this rule then we write $e \dot{\mapsto} e'$; if $e \mapsto e'$ by any other rule (the redexes are disjoint, so only one rule can apply) then we write $e \overset{\circ}{\mapsto} e'$. In what follows let $\overset{\circ}{\mapsto}^*$ denote the transitive reflexive closure of $\overset{\circ}{\mapsto}$. The following key proposition details the interactions between the one-step reduction and the relation \mathcal{IS} .

PROPOSITION 6.4.2.

- (1) If $e_1 \mathcal{IS} e_2$ then $e_1 \in \text{WHNF}$ if and only if $e_2 \in \text{WHNF}$.
- (2) If $e_1 \mathcal{IS} e_2$ and $e_1 \overset{\circ}{\mapsto} e'_1$ then $e_2 \overset{\circ}{\mapsto} e'_2$ and $e'_1 \mathcal{IS} e'_2$.
- (3) If $e_1 \mathcal{IS} e_2$ and $e_1 \dot{\mapsto} e'_1$ then $e_2 \dot{\mapsto} e'_2$ and $e'_1 (\succeq; \mathcal{IS}) e'_2$.

PROOF. In what follows, suppose that $e_1 \mathcal{IS} e_2$. Then we have an expression e_0 containing at most free variables \vec{y} such that $e_1 \equiv e_0\phi$ and $e_2 \equiv e_0\gamma$.

- (1) Suppose that $e_1 \in \text{WHNF}$ (the case for e_2 is symmetric). Then there are two possible cases for the structure of e_0 :
 - (a) $e_0 \equiv c(\vec{e})$ and hence $e_2 \equiv c(\vec{e}\gamma) \in \text{WHNF}$, or
 - (b) $e_0 \equiv \lambda x.e$ and hence $e_2 \equiv (\lambda x.e)\gamma \in \text{WHNF}$.
- (2) Suppose that $e_0\{\vec{\mathbf{f}}/\vec{y}\} \overset{\circ}{\mapsto} e'_1$. Then we can write e_0 as $C[e]$ for some open context C and expression e such that $C\phi$ is a reduction context and $e\{\vec{\mathbf{f}}/\vec{y}\}$ is a redex. (Note that $C\phi$ is well defined since C cannot capture variables.) Using part (1) of the proposition, it is easy to see that $C\gamma$ is also a reduction context. So now we show, by considering cases according to e , that $e\gamma$ is also a redex, and that the reducts of $e\phi$ and $e\gamma$ are also related by \mathcal{IS} , from which it easily follows

that e'_1 and e'_2 are also \mathcal{IS} -related.

Case $e \equiv \text{case } c_i(\vec{e}) \text{ of } c_1(\vec{x}_1) : e'_1 \dots c_n(\vec{x}_n) : e'_n$. Clearly $e\gamma$ is also a redex, so the reducts of $e\phi$, $e\gamma$ are, respectively, $e'_i\phi\{\vec{e}/\vec{x}\}$ and $e'_i\gamma\{\vec{e}/\vec{x}\}$. We can assume, without loss of generality, that $\vec{y} \cap \vec{x} = \emptyset$, so these redexes are $e'_i\{\vec{e}/\vec{x}\}\phi$ and $e'_i\{\vec{e}/\vec{x}\}\gamma$.

The case where $e \equiv (\lambda x.e')e''$ is similar to the above. The remaining cases where $e \equiv p(\vec{c})$ and $e \equiv e'@e''$ are straightforward.

- (3) Suppose $e_0\phi \dot{\mapsto} e'_1$. Then as above we argue that we can write e_0 as $C[e]$ for some open context C and expression e such that $C\{\vec{f}/\vec{y}\}$ is a reduction context and $e\{\vec{f}/\vec{y}\}$ is a (fun)-redex. There are two possible cases for the structure of expression e :

Case $e \equiv \mathbf{h}$. The details are straightforward since $e\phi = \mathbf{h}$ and $e\gamma = \mathbf{h}$, and we use the fact that $\equiv \subseteq \mathcal{IS} \subseteq (\triangleright; \mathcal{IS})$.

Case $e \equiv y_j$, for some $j \in 1 \dots n$. In this case $e\phi \equiv \mathbf{f}_j$, which reduces to $e_j\phi$. Now since $\mathbf{g}_j \triangleq e'_j\gamma$, $e\gamma$ similarly reduces to $e'_j\gamma$. So $e_1 \dot{\mapsto} C[e_j]\phi$ (since C cannot bind any of the \vec{y}), and similarly $e_2 \dot{\mapsto} C[e'_j]\gamma$. Now by definition of the \vec{f} and \vec{g} , and the substitutivity/congruence properties of \triangleright , we have $C[e_j]\phi \triangleright C[e'_j]\phi$, and by definition of \mathcal{IS} that $C[e'_j]\phi \mathcal{IS} C[e'_j]\gamma$, and so we conclude $C[e_j]\phi (\triangleright; \mathcal{IS}) C[e'_j]\gamma$ as required. \square

Now we can furnish a proof of Theorem 5.2.1:

PROOF (THEOREM 5.2.1). Assume $e_1 \mathcal{IS} e_2$ and $e_1 \Downarrow_n w_1$. We show by complete induction on n that $e_2 \Downarrow_{\leq n} w_2$ for some w_2 such that $w_1 (\triangleright; \mathcal{IS}) w_2$. By simple properties of \triangleright and \mathcal{IS} we can see that this is sufficient to show that \mathcal{IS} is an i-simulation up-to, and hence that $\mathbf{f}_i \triangleright \mathbf{g}_i$.

Let $\phi = \{\vec{f}/\vec{y}\}$ and $\gamma = \{\vec{g}/\vec{y}\}$. We have, by definition of \mathcal{IS} , an expression e such that $e\phi \equiv e_1$ and $e\gamma \equiv e_2$.

Base ($n = 0$). Then $e\phi \dot{\mapsto}^* w_1$. By part (2) of Proposition 6.4.2, $e\gamma \dot{\mapsto}^* e'$ for some e' such that $w_1 \mathcal{IS} e'$. By part (1) of Proposition 6.4.2 it follows that $e' \in \text{WHNF}$ and hence $e' \equiv w_2$, and we are done.

Induction ($n \geq 1$). Since $e\phi \Downarrow_{n \geq 1}$ then for some e_0, e_1 , $e\phi \dot{\mapsto}^* e_0$, $e_0 \dot{\mapsto} e_1$ and $e_1 \Downarrow_{n-1} w_1$. We summarize the main argument with the diagram in Figure 3. We complete the squares (A)-(D) from top left to bottom right:

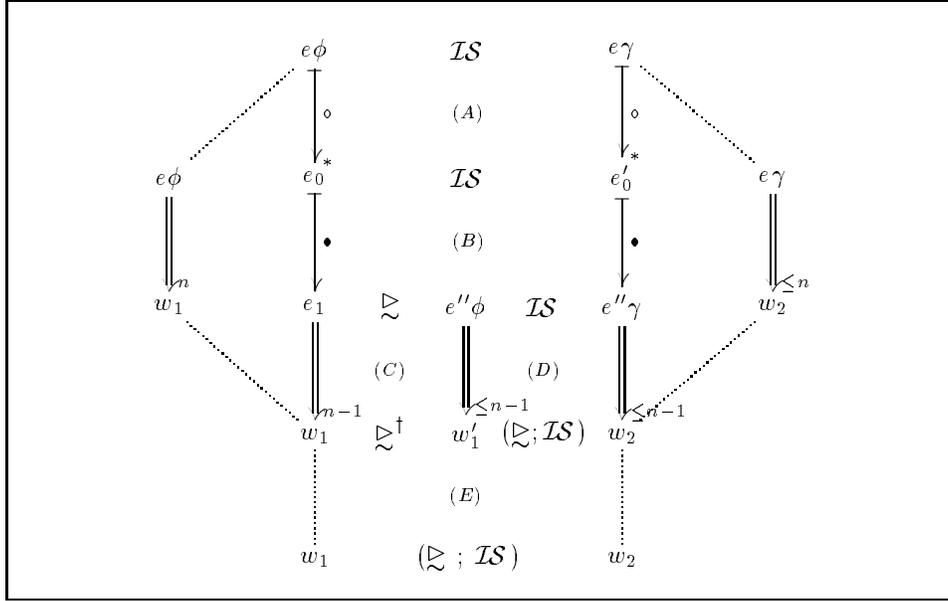


Fig. 3. Induction argument for the Improvement Theorem.

- (A) by part (2) of Proposition 6.4.2;
- (B) by part (3) of Proposition 6.4.2;
- (C) since improvement is an improvement simulation;
- (D) by the induction hypothesis, and finally
- (E) the bottom line follows from the preceding line by Lemma 6.2.2 and transitivity of \lesssim . \square

7. BASIC CORRECTNESS RESULTS

Before we present some applications of the improvement theorem, it is useful to note some basic techniques for establishing the correctness of transformations which complement the improvement-based method. These basic techniques are corollaries of the partial correctness property (Proposition 4.3.1) and follow from the fact that “reversible” transformations are correct.

7.1 Reversible Transformations

A simple corollary of the partial correctness property is that if we have a transformation from \vec{f} to \vec{g} and a transformation from \vec{g} to \vec{h} such that $\vec{h} \sqsubset \vec{f}$ then $\vec{f} \cong \vec{g}$ (since $\vec{f} \sqsubset \vec{g} \sqsubset \vec{h} \sqsubset \vec{f}$). In essence this is just McCarthy’s recursion induction principal [McCarthy 1967]. A simple but practical instance of this scheme is the following *reversible transformation*: a transformation from \vec{f} to \vec{g} is reversible if there exists a transformation from \vec{g} to \vec{h} such that the definitions for \vec{f} and \vec{h} are syntactically equivalent up to variable renaming. In practice reversible transformations have limited power, but are useful for proving simple transformations that do not introduce new recursive structure and can be useful to complement other correct transformation methods.

Abstraction. A common form of transformation, which is easily justified by appealing to reversibility, is *abstraction*. The abstraction transformation lifts some instances of subexpressions from the right-hand sides of a set of definitions and replaces them with function calls for some new functions.

The abstraction process can be used in conjunction with a call-by-need implementation to avoid repeated evaluation of subexpressions. A well-known example is Hughes' supercombinator abstraction [Hughes 1982]. Another form of abstraction which is common in program transformation is *syntactic generalization* in which an expression e is replaced by a function call $\mathbf{g} e_1 \dots e_n$, where \mathbf{g} is a new function defined by $\mathbf{g} x_1 \dots x_n \triangleq e'$, such that $e \equiv e'\{e_1 \dots e_n/x_1 \dots x_n\}$.

General statements about abstractions and their correctness are notationally rather complex. In practice we have found it is easier to appeal to a reversibility argument on a case-by-case basis than to instantiate very general statements about abstraction.

Independent Transformations. Finally, we consider a class of reversible transformation which we call *independent transformations*. These are a class of correct transformations whose correctness follows from the fact that the transformation step does not depend on the functions being transformed. At the expression level this transformation (and its correctness) are easily understood. Stated in terms of explicit lambda expressions, if $e \cong e'$, then by congruence $\mathbf{fix} \lambda z. e \cong \mathbf{fix} \lambda z. e'$, where \mathbf{fix} is a fixed-point combinator. Stated in terms of recursive definitions, this corresponds to the transformation from $\mathbf{f} x \triangleq e\{\mathbf{f}/z\}$ to $\mathbf{g} x \triangleq e'\{\mathbf{g}/z\}$ (where we assume that e, e' are independent of either \mathbf{f} or \mathbf{g}).

The generalization of this transformation to a set of mutually recursive functions, and proof that the resulting transformation is reversible (and hence correct), is left as an easy exercise.

There are reversible transformations which are not improvements and therefore cannot be justified by the improvement theorem. Because of this, these more basic methods for arguing correctness are complementary to the application of the improvement theorem. The idea is that even though a transformation from \mathbf{f} to \mathbf{g} may not be provable by the improvement theorem, it may be possible to justify it by splitting the transformation into two or more stages, and justifying some stages by a reversibility argument, and some by appealing to the improvement theorem. We will see an example of this at the end of the next section.

8. APPLICATION TO THE VERIFICATION OF TRANSFORMATION METHODS

The Improvement Theorem has immediate application to program transformation methods. In Sands [1995b] it is used to provide a total correctness proof for an automatic transformation based on a higher-order variant of the deforestation method [Wadler 1990].

In this section we illustrate the utility of the Improvement Theorem with a smaller example: 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 [1989a]

(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 (written here as an infix function $++$) of the form $\mathbf{f} \ e_1 \dots e_n \ ++ \ e'$, by finding a function \mathbf{f}^+ which satisfies

$$\mathbf{f}^+ \ x_1 \dots x_n \ y \cong (\mathbf{f} \ x_1 \dots x_n) \ ++ \ y.$$

Definition 8.1 (“Concatenate Vanishes”). The transformation has two phases: *initialization*, which introduces an initial definition for \mathbf{f}^+ , and *transformation*, which applies a set of rewrites to the right-hand sides of all definitions.

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

$$\mathbf{f}^+ \ x_1 \dots x_n \ y \triangleq e \ ++ \ y.$$

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

- (1) $\mathbf{nil} \ ++ \ x \rightarrow x$
- (2) $(x.y) \ ++ \ z \rightarrow x.(y \ ++ \ z)$
- (3) $(x \ ++ \ y) \ ++ \ z \rightarrow x \ ++ \ (y \ ++ \ z)$
- (4) $(\mathbf{case} \ x \ \mathbf{of} \ \dots \rightarrow \mathbf{case} \ x \ \mathbf{of} \ \dots)$

$c_1(\vec{y}_1) : e_1$	$c_1(\vec{y}_1) : (e_1 \ ++ \ z)$
\dots	\dots
$c_n(\vec{y}_n) : e_n \ ++ \ z$	$c_n(\vec{y}_n) : (e_n \ ++ \ z)$
- (5) $(\mathbf{f} \ x_1 \dots x_n) \ ++ \ y \rightarrow \mathbf{f}^+ \ x_1 \dots x_n \ y$
- (6) $(\mathbf{f}^+ \ x_1 \dots x_n \ y) \ ++ \ z \rightarrow \mathbf{f}^+ \ x_1 \dots x_n \ (y \ ++ \ z)$

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

Henceforth, let \rightarrow be the rewrite relation generated by the above rules (i.e., the compatible closure) and \rightarrow^+ 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 8.2. Here is a simple example to illustrate the effect of the transformation: \mathbf{itrav} computes the *inorder* traversal of a binary tree. Trees are assumed to be built from a nullary \mathbf{leaf} constructor, and a ternary \mathbf{node} , comprising a left subtree, a node-element, and a right subtree.

$$\mathbf{itrav} \ t \triangleq \mathbf{case} \ t \ \mathbf{of}$$

$\mathbf{leaf} : \mathbf{nil}$
$\mathbf{node}(l, n, r) : (\mathbf{itrav} \ l) \ ++ \ (n.\mathbf{itrav} \ r).$

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

$$\text{itrav}^+ t y \triangleq (\text{case } t \text{ of} \\ \quad \text{leaf} : \text{nil} \\ \quad \text{node}(l, n, r) : (\text{itrav } l) ++ (n.\text{itrav } r) \\) ++ y$$

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

$$\begin{aligned} & \text{case } t \text{ of leaf} : \text{nil} \\ & \quad \text{node}(l, n, r) : (\text{itrav } l) ++ (n.\text{itrav } r) \\ \rightarrow & \text{case } t \text{ of leaf} : \text{nil} \\ & \quad \text{node}(l, n, r) : \text{itrav}^+ l (n.\text{itrav } r) \end{aligned}$$

$$\begin{aligned} & (\text{case } t \text{ of leaf} : \text{nil} \\ & \quad \text{node}(l, n, r) : (\text{itrav } l) ++ (n.\text{itrav } r)) ++ y \\ \rightarrow & \text{case } t \text{ of leaf} : \text{nil} ++ y \\ & \quad \text{node}(l, n, r) : ((\text{itrav } l) ++ (n.\text{itrav } r)) ++ y \\ \rightarrow^+ & \text{case } t \text{ of leaf} : y \\ & \quad \text{node}(l, n, r) : \text{itrav}^+ l (n.\text{itrav}^+ r y) \end{aligned}$$

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

$$\begin{aligned} \text{itrav } t & \triangleq \text{case } t \text{ of leaf} : \text{nil} \\ & \quad \text{node}(l, n, r) : \text{itrav}^+ l (n.\text{itrav } r) \\ \\ \text{itrav}^+ t y & \triangleq \text{case } t \text{ of leaf} : y \\ & \quad \text{node}(l, n, r) : \text{itrav}^+ l (n.\text{itrav}^+ r y) \end{aligned}$$

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.

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 Proposition 4.3.1) 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 improvement relation.

PROPOSITION 8.3. *The transformations rules (1)–(6) are 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 similarly proved from the operational semantics by showing that its reflexive closure is an improvement simulation. Rule (4) can be proved with the help of Proposition 6.1.3, part (10), 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:

$$\begin{aligned} (\mathbf{f}^+ x_1 \dots x_n y) ++ z &\stackrel{\triangleright}{\sim} ((\mathbf{f} x_1 \dots x_n) ++ y) ++ z \quad (\text{since } (v) \subset (\triangleright)) \\ &\stackrel{\triangleright}{\sim} (\mathbf{f} x_1 \dots x_n) ++ (y ++ z) \quad (\text{by } (iii)) \\ &\stackrel{\triangleright}{\sim} \mathbf{f}^+ x_1 \dots x_n (y ++ z) \quad (\text{by } (v)) \end{aligned}$$

□

Then we get the following, directly from the Improvement Theorem:

PROPOSITION 8.4. *The transformation is correct, and the resulting functions are improvements over the originals.*

8.1 A Variation

In the introduction to this section we mentioned a small difference to the transformation as described by Wadler [1989a]. This is a small “performance bug” highlighted by the use of the Improvement Theorem. Wadler includes a rule which replaces the original definition of the function (\mathbf{f}) by

$$\mathbf{f} x_1 \dots x_n \triangleq \mathbf{f}^+ x_1 \dots x_n \mathbf{nil}. \quad (1)$$

In order to apply the Improvement Theorem it was necessary to omit this rule. This is because the rule can lead to less efficient programs. More specifically, there are many examples where the number of computation steps needed to compute the first k elements of the expression $\mathbf{f} e_1 \dots e_n$ will be increased by $\mathcal{O}(k)$ function calls. However, in cases where the concatenate operator is successfully eliminated, the overhead of using (1) reduces to $\mathcal{O}(1)$. What is more, this version is more efficient in terms of code size. In our modification, one could say that we have solved the “performance bug” at the expense of a “code size” bug.

In the remainder of this section we show how the correctness of this original formulation of the transformation can be justified. We make use of the Improvement Theorem by factoring the transformation into two steps and justifying these independently. The first step is not an improvement, but can be easily justified by elementary means, appealing to reversibility (cf. Section 7). The second step (the core of the transformation) *is* an improvement and can be justified by the Improvement Theorem in the manner of the transformation considered earlier.

We describe the original variation of the transformation as follows.

Initialization. As before, for some function $\mathbf{f} x_1 \dots x_n \triangleq e$ define a *new* function

$$\mathbf{f}^+ x_1 \dots x_n y \triangleq e ++ y.$$

Transformation Step (1). Replace definition $(\mathbf{f} x_1 \dots x_n \triangleq e)$ by

$$\mathbf{f} x_1 \dots x_n \triangleq \mathbf{f}^+ x_1 \dots x_n \mathbf{nil}.$$

Transformation Step (2). As before, apply the rewrite rules (1)–(6) to the right-hand sides of all definitions.

PROPOSITION 8.1.1. *The original variation of the transformation is correct.*

PROOF. We prove the correctness of the two steps separately, and the result follows by transitivity.

Step (1). Given the following well-known equivalence $x \dashv\vdash \mathbf{nil} \cong x$, it is easy to establish that the transformation step is reversible (see Section 7) and hence correct.

Step (2). Since the first step is correct, we can still argue that all the rewrite rules are equivalences, and so, just as for the earlier transformation, it is sufficient to show that the rules are contained in the improvement relation. Correctness then follows from the Improvement Theorem (but note that the improvement property established by the improvement theorem is with respect to the functions obtained after Step (1), and not the original functions). The difference is that we must verify this improvement property with respect to definition (1). The only rule which needs to be checked is (5), since the others are independent of definition of \mathbf{f} . But this is easily verified, and we leave it as a simple exercise. \square

9. APPLICATION TO UNFOLD-FOLD TRANSFORMATIONS

In this section we consider a different kind of problem: unfold-fold transformation. The problem is different from the task of verifying specific transformation methods (such as the one in the previous section) because, in general, unfold-fold is not correct. Our task is to design, with the help of the Improvement Theorem, a simple method for constraining the transformation process such that correctness is ensured.

In summary of this section, we will introduce an “improved” unfold-fold method (and an extension of it) which guarantees that the program thus obtained is equivalent to the original. The method follows the spirit of the idea of “more unfolds than folds” in that it amortizes the cost of fold steps (which introduce new function calls) against the savings made by unfolding steps. This amortization guarantees that the local transformation steps are improvements, and hence that correctness follows from the Improvement Theorem. To make this work (to disallow, for example, the incorrect transformation in the introduction) the unfold steps and fold steps must be related in a certain sense. This is formalized with the help of the improvement theory.

9.1 The Unfold-Fold Transformation

Burstable and Darlington’s original definition was for a first-order functional language where functions are defined by pattern matching, and a call-by-name evaluation is assumed [Burstable and Darlington 1977]. The following six rules were introduced for transforming recursion equations:

- (1) *Definition* introduces a new recursion equation whose left-hand side is not an instance of the left-hand side of a previous definition.
- (2) *Instantiation* introduces a substitution instance of an existing equation.

- (3) *Unfolding* replaces an instance of a function call by the appropriate instance of the body of the function.
- (4) *Folding* replaces an instance of a body of a function by a corresponding call.
- (5) *Abstraction* replaces occurrences of some expressions by a variable bound by a *where* clause.
- (6) *Laws* rewrite according to laws about primitive functions.

There is an important point which is not made clear from this standard definition (but which is essential in practice): the folding step—which replaces an instance of the body of a function definition by the corresponding instance of the call to the function—is able to make use of an *earlier version* of the function. In other words, one can replace an instance of the body of an older version of a function with the corresponding call. Without this ability it would not be possible to obtain nontrivial recursive programs. In the original formulation of the transformation from Burstall and Darlington [1977] one is also allowed to *unfold* using any earlier definition, but in practice this is less critical.

Unfold-fold transformation carries over to higher-order functional languages essentially unchanged. To suit our particular language but without significant loss of generality we simplify the rules we will consider and the style in which they are applied.

- The definition rule above may be applied freely, since it just introduces new functions (and conservatively extends the theories of operational approximation and improvement).
- Instantiation will not be used explicitly; the role of instantiation will be played by certain distribution laws for case expressions. In any case, unrestricted instantiation is problematic because it is not even locally equivalence preserving, since it can force premature evaluation of expressions (a problem noted in Bird [1984], and addressed in some detail in Runciman et al. [1989]) and is better suited to a typed language in which one can ensure that the set of instances is exhaustive.
- Abstraction will not be used explicitly, since this can easily be represented by laws using lambda abstractions to represent the binding (which has the advantage of enabling us to deal implicitly with possible problems with bound variables which do not arise in Burstall and Darlington’s language).

9.2 Simplified Unfold-Fold Transformation

In our setting, an unfold-fold transformation will consist of adding new definitions and (repeatedly) transforming the right-hand sides of some set of definitions $\{\mathbf{f}_i \vec{x}_i \triangleq e_i\}_{i \in I}$, according to unfolding, folding, and rewriting laws, to obtain a set of expressions $\{e'_i\}_{i \in I}$. Using these expressions a set of new function definitions is constructed, namely

$$\{\mathbf{f}'_i \vec{x}_i \triangleq e'_i\{\vec{\mathbf{f}}'/\vec{\mathbf{f}}\}\}_{i \in I}.$$

Henceforth, the terms *unfolding* and *folding* refer to local transformations on expressions and not to operations which in themselves give rise to new definitions.

Given definitions $\mathbf{f}_i \vec{x}_i \triangleq e_i$, the unfold and fold rewrites are just instances of the congruences $\mathbf{f}_i \vec{x}_i \cong e_i$ (although this law does not hold in a call-by-value language

under arbitrary substitutions, so for strict languages further restrictions must be applied). We take the laws to be operational equivalences by definition. So we see that an unfold-fold transformation step is now a transformation according to Definition 4.1.2, and we obtain the standard partial correctness result that the new program may be less defined than the original (see Kott [1978; 1985] and Courcelle [1979]).

9.3 Correctness by Improvement in Unfold-Fold

If we restrict the application of the laws so that they are improvement steps (as they usually are in practice), then since the unfolding step is an improvement, transformation steps not involving folds are totally correct, and the results are improvements. This follows directly from Corollary 5.2.2. (In fact it is not too difficult to show that any transformation without folding is correct, provided that the laws are not dependent on the definitions of the recursive functions [Courcelle 1986; Zhu 1994].

The problem is that no fold step, viewed in isolation, is an improvement.⁹ The key to guaranteeing correctness is to ensure that we pay for each fold step at some point, thus maintaining overall improvement.

9.4 The Tick Algebra

In order to obtain correctness through the Improvement Theorem, it is sufficient to verify that the net effect of the local transformation steps is an improvement. Intuitively, unfolding “speeds up” an expression, and folding “slows it down”; so if the transformation sequence contains some folding steps, it must also contain some corresponding unfolding steps. A key point which must be formalized is that the unfolding steps and folding steps must be suitably related for the net result to be an improvement.

We introduce a mechanism to enable the required net improvement condition to be maintained stepwise through the transformation—rather than enforce it through a post hoc justification of each transformation. We will extend the unfold-fold process with a single “annotation”, \surd , “tick.” Although we will think of the tick as an annotation, formally it is a function

$$\surd_e \equiv \mathbf{tick} e$$

where \mathbf{tick} is an identity function, given by the definition

$$\mathbf{tick} x \triangleq x.$$

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

$$e \Downarrow_n h \iff \surd_e \Downarrow_{n+1} h.$$

⁹Except for the trivial case when the function introduced by the fold step is everywhere undefined.

In terms of improvement, observe that $\surd_e \succsim e$ but $e \not\prec \surd_e$ (except if all closed instances of e diverge).

The technique for ensuring correctness of unfold-fold will be to use a tick to pay for a fold step, paid for by a “nearby” unfold. With respect to a definition $\mathbf{f}x \triangleq e$, the unfolding and folding steps correspond to instances of the law $\mathbf{f}x \cong e$. In terms of improvement, we have the following laws (which follow easily from the improvement context lemma) relating a function and its body:

PROPOSITION 9.4.1. *If function \mathbf{f} is defined by $\mathbf{f} \vec{x} \triangleq e$, then*

$$\begin{aligned} \mathbf{f} \vec{x} &\succsim \surd_e \\ \surd_e &\succsim \mathbf{f} \vec{x} \end{aligned}$$

PROOF. Easy by construction of appropriate improvement simulations, since $\surd_e \dot{\mapsto} e$ and $\mathbf{f} \vec{x} \dot{\mapsto} e$. \square

The idea of our “improved” unfold-fold transformation will be to use (substitution instances of) these improvements, in place of unfolding and folding, respectively. Since unfolding speeds up a computation by one step, the first improvement “saves” that computation step in the form of an application of the tick function. We can understand an instance of the second improvement as follows: folding introduces one extra function call, but we can make this an improvement step providing we eliminate a tick function call in the same step.

The remaining problem is to determine when unfold steps and fold steps are “suitably related.” In order to perform a fold we must be able to move a tick—generated by some earlier unfold step—to the fold site. This is achieved by the *tick algebra*. The *tick algebra* is a collection of improvement laws for propagating ticks around an expression while preserving or increasing improvement. Figure 4 gives some basic laws for \surd which will augment the transformation rules. We also need a distribution law for nested case expressions (Proposition 6.1.3, part (10)). The basic reduction steps of the operational semantics are also included in the improvement relation, and are therefore useful. In the laws, \mathbb{R} ranges over reduction contexts, possibly containing free variables.

The laws are straightforward to prove using the improvement context lemma. But it is not intended (or expected) that one should (need to) prove tick-laws “on-the-fly.” The method is intended to be viewed as completely syntactic—given a “reasonable” set of tick laws.

9.5 “Improved” Unfold-Fold Transformations

We show by two small examples how the tick algebra can be used to maintain improvement throughout the steps of a transformation, thereby guaranteeing total correctness and improvement. This will be followed by the formal definition of improved unfold-fold transformation and its proof of correctness.

Example 9.5.1. In Figure 5 we give a standard unfold-fold example, but now locally maintaining the improvement relation at each step of the transformation by introducing ticks at unfold steps, and propagating these, via the tick laws, to the fold site. The example ensures, by construction, that the derived program `sumsq'` is an improvement over the original (in addition to being equivalent). Note that

$$\begin{array}{c}
\bullet \sqrt{e} \succsim e \qquad \bullet \frac{\sqrt{e_1} \succsim \sqrt{e_2}}{e_1 \succsim e_2} \qquad \bullet R[\sqrt{e}] \simeq \sqrt{R[e]} \\
\bullet \frac{\mathbf{f} \vec{x} \triangleq e}{\mathbf{f} \vec{x} \simeq \sqrt{e}} \qquad \bullet \sqrt{p(e_1 \dots e_n)} \simeq p(e_1 \dots \sqrt{e_i} \dots e_n) \\
\bullet \sqrt{\text{case } e \text{ of}} \qquad \simeq \qquad \text{case } e \text{ of} \\
\quad c_1(\vec{x}_1) : e_1 \dots c_n(\vec{x}_n) : e_n \qquad c_1(\vec{x}_1) : \sqrt{e_1} \dots c_n(\vec{x}_n) : \sqrt{e_n}
\end{array}$$

Fig. 4. Tick laws.

in this particular case, since the improvement steps above are all represented by cost equivalences, we get a good picture of the *degree* of improvement: namely, one function call is saved for each call to `sumsq`'s, plus one more when the argument is nil.

Example 9.5.2. To take a smaller (but less standard) example, consider the usual Y -combinator of the lambda calculus (in head normal form) given by

$$\lambda h.h((\lambda x.h(xx))\lambda x.h(xx)).$$

A direct translation of this into recursion equations would be the expression Y , where

$$\begin{aligned}
Yh &\triangleq h(Dh(Dh)) \\
Dhx &\triangleq h(xx)
\end{aligned}$$

Now we transform the definition of Y to obtain a direct recursive version, by unfolding the call to D then immediately folding against Y :

$$\begin{aligned}
Yh &\triangleq h(Dh(Dh)) \\
&\simeq h(\sqrt{h(Dh(Dh))}) \quad (\text{unfold } D) \\
&\simeq h(Yh) \quad (\text{fold with } Y)
\end{aligned}$$

thus obtaining a new definition $Y'h \triangleq h(Y'h)$. The correctness of this method shows that Y is equivalent to Y' , and $Y \succsim Y'$. (In fact since the steps of the transformation are all cost equivalences, Proposition 5.2.4 gives us that $Y \simeq Y'$.)

Example 9.5.3. We take the unfold-fold transformation given in the introduction and illustrate why we cannot justify it in the “improved” unfold-fold method. Recall the following definition from the introduction which with the usual interpretation of the conditional function returns `true` when the argument is `false` and is undefined otherwise:

$$\mathbf{f} x \triangleq \mathbf{if} x \text{ then } (\mathbf{f} x) \text{ else } \mathbf{true}.$$

Now we attempt to simulate the unfold-fold transformation from the introduction, using only improvement steps. We make use of the following improvement law, which we state without proof:

$$\mathbf{if} x \text{ then } (\mathbf{if} x \text{ then } y \text{ else } z') \text{ else } z \succsim \mathbf{if} x \text{ then } y \text{ else } z.$$

Consider the following definitions:

$$\begin{aligned} \text{sum } xs &\triangleq \text{ case } xs \text{ of} \\ &\quad \text{nil} : 0 \\ &\quad y.ys : y + \text{sum } ys \\ \text{sq } x &\triangleq x * x \\ \text{map } f \text{ } xs &\triangleq \text{ case } xs \text{ of} \\ &\quad \text{nil} : \text{nil} \\ &\quad y.ys : (f y). \text{map } f \text{ } ys \end{aligned}$$

We will transform the following function:

$$\text{sumsq } xs \triangleq \text{sum}(\text{map } \text{sq } xs).$$

Transforming the right-hand side we obtain:

$$\begin{aligned} &\text{sum}(\text{map } \text{sq } xs) \\ &\rightsquigarrow \text{sum} \sqrt{\left(\begin{array}{l} \text{case } xs \text{ of} \\ \text{nil} : \text{nil} \\ y.ys : (\text{sq } y). \text{map } \text{sq } ys \end{array} \right)} \quad (\text{unf. map}) \\ &\rightsquigarrow \sqrt{\text{case} \sqrt{\left(\begin{array}{l} \text{case } xs \text{ of} \\ \text{nil} : \text{nil} \\ y.ys : (\text{sq } y). \text{map } \text{sq } ys \end{array} \right)}} \text{ of } (\text{unf. sum}) \\ &\quad \text{nil} : 0 \\ &\quad x.xs : x + \text{sum } xs \\ &\rightsquigarrow \sqrt{\text{case } xs \text{ of}} \quad (\sqrt{\text{case-laws}}) \\ &\quad \text{nil} : 0 \\ &\quad y.ys : (\text{sq } y) + \text{sum}(\text{map } \text{sq } ys) \\ &\rightsquigarrow \sqrt{\text{case } xs \text{ of}} \quad (\sqrt{\text{-laws}}) \\ &\quad \text{nil} : \sqrt{0} \\ &\quad y.ys : (\text{sq } y) + \sqrt{\text{sum}(\text{map } \text{sq } ys)} \\ &\rightsquigarrow \sqrt{\text{case } xs \text{ of}} \quad (\text{fold } \text{sumsq}) \\ &\quad \text{nil} : \sqrt{0} \\ &\quad y.ys : (\text{sq } y) + \text{sumsq } ys \end{aligned}$$

After eliminating the ticks we construct the new definition:

$$\begin{aligned} \text{sumsq}' xs &\triangleq \text{ case } xs \text{ of} \\ &\quad \text{nil} : 0 \\ &\quad y.ys : (\text{sq } y) + \text{sumsq}' ys \end{aligned}$$

Fig. 5. Improved `sumsq` transformation.

We can transform the body of the definition as follows:

$$\begin{aligned}
& \mathbf{if } x \mathbf{ then } (\mathbf{f } x) \mathbf{ else true} \\
& \quad \sphericalangle \mathbf{if } x \mathbf{ then } \sphericalangle (\mathbf{if } x \mathbf{ then } (\mathbf{f } x) \mathbf{ else true}) \mathbf{ else true} \quad (\text{unfold}) \\
& \quad \sphericalangle \mathbf{if } x \mathbf{ then } \sphericalangle (\mathbf{if } x \mathbf{ then } \sphericalangle (\mathbf{if } x \mathbf{ then } (\mathbf{f } x) \mathbf{ else true})) \quad (\text{unfold}) \\
& \quad \quad \quad \mathbf{else true} \\
& \quad \quad \quad \mathbf{else true} \\
& \quad \sphericalangle \mathbf{if } x \mathbf{ then } \sphericalangle (\mathbf{f } x) \mathbf{ else true} \quad (\sphericalangle - \text{distribution and law (twice)})
\end{aligned}$$

We have expressed many of the transformation steps as cost equivalences, to indicate that we have not discarded any potentially useful ticks. Now the final step of the incorrect transformation was to fold the expression $\mathbf{if } x \mathbf{ then } \mathbf{f } x \mathbf{ else true}$. To do this in the transformation above, we would need to be able to propagate a tick from the first branch of the conditional to the outside. This is not possible with the tick algebra we have considered, or *any* (sound) tick algebra for that matter, since

$$\mathbf{if } x \mathbf{ then } \sphericalangle (\mathbf{f } x) \mathbf{ else true} \not\approx \sphericalangle \mathbf{if } x \mathbf{ then } \mathbf{f } x \mathbf{ else true} .$$

The general technique of the “improved” unfold-fold method should now be clear from the examples. The following gives the formal definition and its correctness.

Definition 9.5.4. A pair of expressions (e_1, e_2) is defined to be an *improvement law* if

- (1) neither expression contains recursive function names other than the tick function, and
- (2) $e_1 \cong e_2$, and
- (3) $e_1 \succ e_2$.

Definition 9.5.5. Improved Unfold-Fold Transformation Given a set of function definitions $\mathbf{f}_i \vec{x}_i \triangleq e_i$, $i \in \{1 \dots m\}$ an improved unfold-fold transformation is any transformation which begins with the sequence of expressions (the bodies of the functions) e_1, \dots, e_m and which constructs a new sequence of expressions e'_1, \dots, e'_m , by some finite iteration of applications of the following rules to any subexpressions of the sequence:

- (1) (improved unfolding) replace an instance of a function call $\mathbf{f}_i \vec{x}_i$, with the corresponding instance of the expression $\sphericalangle e_i$;
- (2) (improved folding) replace an instance of the expression $\sphericalangle e_j$, for some $j \in \{1 \dots m\}$, with the corresponding instance of the call $\mathbf{f}_j \vec{x}_j$;
- (3) (improvement laws) replace an instance of the left-hand side of an improvement law by the corresponding instance of the right-hand side.

Finally, construct the new functions $\mathbf{f}'_i \vec{x}_i \triangleq e'_i \{\vec{\mathbf{f}}'/\vec{\mathbf{f}}\}$, $i \in \{1 \dots m\}$.

THEOREM 9.5.6. *Any improved unfold-fold transformation is correct and yields new functions which are improvements over the respective originals.*

PROOF. Each replacement operation preserves equivalence and is an improvement: for (3) this follows by definition; for improved unfolding and improved folding, this follows from the cost equivalence $\mathbf{f } \vec{x} \sphericalangle \sphericalangle \sphericalangle e$ given in Proposition 9.4.1.

Since improvement and operational equivalence are (pre)congruences, it follows that $e_i \cong e'_i$ and $e_i \succsim e'_i$, $i \in \{1 \dots m\}$. Finally, by Corollary 5.2.2 and Theorem 5.2.1, respectively, we have that the transformation is correct ($\mathbf{f}_i \cong \mathbf{f}'_i$) and yields functions which are improvements over the originals ($\mathbf{f}_i \succsim \mathbf{f}'_i$). \square

Notice that we leave the set of improvement laws open. In fact we can safely allow replacements using any operational equivalence which is also an improvement (i.e., those which depend on the functions), but this goes beyond what is usually considered to be unfold-fold transformations; so we restrict ourselves to proper “laws,” in the sense that they are equivalences independent of the user-defined functions. (In Section 9.6 we consider an extension of this method for which this condition is a necessary one.)

The tick laws are crucial in the examples. The following proposition covers these and other useful improvement laws:

PROPOSITION 9.5.7. *The following pairs of expressions are instances of improvement laws:*

- (1) (e, e') and (e', e) , whenever $e \overset{\circ}{\mapsto} e'$ (i.e., $e \mapsto e'$ via any reduction rule other than (fun));
- (2) $(\surd x, x)$;
- (3) $(\mathbb{R}[\surd x], \surd \mathbb{R}[x])$ and the symmetric pair $(\surd \mathbb{R}[x], \mathbb{R}[\surd x])$;
- (4) $(\surd p(x_1 \dots x_n), p(x_1 \dots \surd x_i \dots x_n))$ and the symmetric pair;
- (5) (e, e') whenever e and e' satisfy the following conditions
 - (a) $e \cong e'$, and
 - (b) e and e' are built from only primitive function calls, constants, and variables; and
 - (c) each variable in e' occurs at most once.
- (6) $\left(\begin{array}{c} \text{case } x \text{ of} \\ c_1(\vec{x}_1) : e_1 \dots c_n(\vec{x}_n) : e_n \end{array}, \begin{array}{c} \text{case } x \text{ of} \\ c_1(\vec{x}_1) : \surd e_1 \dots c_n(\vec{x}_n) : \surd e_n \end{array} \right)$
and the symmetric pair;
- (7) $\left(\begin{array}{c} \mathbb{R}[\text{case } x \text{ of} \\ c_1(\vec{x}_1) : e_1 \dots c_n(\vec{x}_n) : e_n \end{array}, \begin{array}{c} \text{case } x \text{ of} \\ c_1(\vec{x}_1) : \mathbb{R}[e_1] \dots c_n(\vec{x}_n) : \mathbb{R}[e_n] \end{array} \right)$
and the symmetric pair.

PROOF. For each pair it is necessary to show that they are expressible as instances of operational equivalences and improvements not involving recursive functions. Proving operational equivalence and improvement for the pairs (with the exception of (5) is straightforward, so we omit the details.

With regard to (3) and (7) we need to check that they can be expressed as instances not involving function symbols. This follows (via a simple structural induction) from the fact that any (open) reduction context \mathbb{R} can be expressed as $\mathbb{R}'\sigma$ for some reduction context \mathbb{R}' not containing function names, and some substitution σ .

For (5) we need to prove $e \succsim e'$. We give a sketch. Assume that there is some instance of e that converges (otherwise we are finished). Consider any closed instance of the pair $(e\sigma, e'\sigma)$. Suppose $e\sigma$ converges in k steps. Since e is built from only primitive functions, variables, and constants, it follows that $e\sigma$ must evaluate

to a constant. It follows by operational equivalence that $e'\sigma$ evaluates to the same constant. It remains to show that $e'\sigma$ converges in less than or equal to k steps.

First, note that e and e' are strict in all their variables, i.e., if $e\sigma \Downarrow$ then it follows that $x\sigma \Downarrow$ for all x in the free variables of e , and similarly for e' . Now we argue that the free variables of e and e' must be the same, since if they were not, then we could find a substitution τ which made one expression converge and the other diverge, contradicting the assumption that $e \cong e'$. Now suppose $e'\sigma$ converges in m steps. Since e and e' are built from only primitive functions (which are strict, and reduce in zero steps), constants, and variables, and since each variable in e' occurs only once it can be established by a straightforward induction on the structure of e' that $m = \sum \{n \mid y \in \text{FV}(e'), y\sigma \Downarrow_n\}$ and since the free variables of e' also occur in e at least once, we conclude by a similar argument that $k \geq m$. \square

9.6 An Extension to Improved Unfold-Fold

There is a commonly occurring problem with the improved unfold-fold method as outlined above. The problem relates to folding in a “lazy” context. Any expression to be folded must be ticked, but ticks cannot propagate across lazy contexts, e.g., $x.[\]$, so if the lazy context is produced by an unfolding step, the tick generated by the unfold cannot be used to justify the fold step.

The practical solution to this problem is presented in terms of an extension to the improved unfold-fold method. The solution has the additional advantage that, in practice, it can greatly simplify the amount of tick bookkeeping necessary.

The Problem

Example 9.6.1. Consider the following definitions where the conditional is just shorthand for the corresponding boolean case expression, and the period “.” is infix `cons`:

```

filter  $p$   $xs$   $\triangleq$  case  $xs$  of
    nil : nil
     $y.y$  : if  $p$   $y$  then  $y.$ filter  $p$   $ys$ 
           else filter  $p$   $ys$ 
iterate  $f$   $x$   $\triangleq$   $x.$ (iterate  $f$  ( $f$   $x$ ))

```

Suppose we wish to transform the expression:

```
filter  $p$  (iterate  $f$   $x$ ).
```

So, for example, `filter even (iterate (add1) 1)` is the “infinite” stream of even integers greater than one. We begin unfold-fold transformation by introducing a definition:

```
fit  $p$   $f$   $x$   $\triangleq$  filter  $p$  (iterate  $f$   $x$ ).
```

Now we attempt to transform the right-hand side by improved unfold-fold steps (we use cost equivalence steps to show that we have not discarded any potentially useful ticks):

$$\begin{array}{ll}
\mathbf{filter}p(\mathbf{iterate}f\ x) & \\
\rightsquigarrow \mathbf{filter}p\ \surd(x.(\mathbf{iterate}f\ (f\ x))) & \text{unfold, tick-elim.} \\
\rightsquigarrow \surd\mathbf{if}\ p\ x\ \mathbf{then}\ x.\mathbf{filter}p(\mathbf{iterate}f\ (f\ x)) & \text{unfold, laws} \\
\quad \mathbf{else}\ \mathbf{filter}p(\mathbf{iterate}f\ (f\ x)) & \\
\rightsquigarrow \surd\mathbf{if}\ p\ x\ \mathbf{then}\ \surd x.\mathbf{filter}p(\mathbf{iterate}f\ (f\ x)) & \text{tick-laws} \\
\quad \mathbf{else}\ \surd\mathbf{filter}p(\mathbf{iterate}f\ (f\ x)) & \\
\rightsquigarrow \surd\mathbf{if}\ p\ x\ \mathbf{then}\ \surd x.\mathbf{filter}p(\mathbf{iterate}f\ (f\ x)) & \text{fold} \\
\quad \mathbf{else}\ \mathbf{fit}\ p\ f\ (f\ x) &
\end{array}$$

At this point we want to fold the other instance of $\mathbf{filter}p(\mathbf{iterate}f\ (f\ x))$ in the first branch of the conditional, but the tick cannot propagate over the constructor to enable this fold, even though the result of this fold is correct.

Extended Improved Unfold-Fold. The practical solution to this problem comes in the form of an extension to the improved unfold-fold method. The extension will allow more folds. The basic intuition of the solution is as follows. Suppose we are performing transformation steps on the body of some function \mathbf{g} . Further, suppose there is an instance $e_{\mathbf{f}}$ of the body of some function \mathbf{f} , which we wish to replace by the corresponding call to \mathbf{f} (in the previous example, the functions \mathbf{f} and \mathbf{g} were the same function, \mathbf{fit}). In the improved unfold-fold transformation method, one attempts to propagate ticks (generated from earlier unfold steps) to the expression $e_{\mathbf{f}}$ in order to perform an improved fold. This is sound because the extra function call \mathbf{f} thus introduced is balanced by the elimination of the tick function call.

The extended method described in the remainder of this section provides an alternative means to justify the fold step: instead of paying for the fold step at the site of the fold operation (the subexpression $e_{\mathbf{f}}$ which occurs in the body of \mathbf{g}), it can be paid for at the outermost level of the body of the function \mathbf{f} . In other words, if we can propagate a tick from the body of the function \mathbf{f} to the outermost level of the expression, then if we eliminate this tick we have guaranteed to speed up the function \mathbf{f} by at least one step, thus making it safe to fold any instances of \mathbf{f} . This idea of paying for a fold step by speeding up the *callee*, rather than caller, is subject to one further restriction: the initial definition of the function \mathbf{f} must be nonrecursive, in the sense that it does not occur in the right-hand side of any definition (including \mathbf{f}). In practice this condition is not too restrictive because it covers the common case where \mathbf{f} is a newly introduced definition (a “eureka”), like \mathbf{fit} in the example.

Definition 9.6.2 (Extended Improved Unfold-Fold Transformation). Assume we wish to transform a set of function definitions $\mathbf{f}_i\ \vec{x}_i \triangleq e_i$, $i \in \{1 \dots m\}$. An extended improved unfold fold transformation is any transformation performed as follows. Begin with the sequence of expressions e_1, \dots, e_m and construct a new sequence of expressions e'_1, \dots, e'_m , by some finite iteration of applications of the following rules. In addition, during the transformation each function definition is either *marked* or *unmarked*. Initially all definitions are unmarked. In the following rules, j ranges over $\{1 \dots m\}$:

- (1) (improved unfolding) replace an occurrence of an instance of a function call, $\mathbf{f}_j\ \vec{x}_j$, with the corresponding instance of the expression $\surd e_j$, providing the definition of \mathbf{f}_j is not marked;

- (2) (improved folding) replace an occurrence of an instance of the expression $\surd e_j$, with the corresponding instance of the call $\mathbf{f}_j \vec{x}_j$;
- (3) (improvement laws) replace an occurrence of an instance of the left-hand side of an improvement law by the corresponding instance of the right-hand side;
- (4) (marked folding) replace an occurrence of an instance of the expression e_j , for some j with the corresponding instance of the call $\mathbf{f}_j \vec{x}_j$, providing that \mathbf{f}_j is marked.
- (5) (marking) if the j th expression of the sequence is $\surd e$, for some e , and definition \mathbf{f}_j is unmarked and nonrecursive, then replace the j th expression by e , and mark \mathbf{f}_j .

Finally, construct the new functions $\mathbf{f}'_i \vec{x}_i \triangleq e'_i\{\vec{\mathbf{f}}'/\vec{\mathbf{f}}\}$, $i \in \{1 \dots m\}$.

THEOREM 9.6.3. *Any extended improved unfold-fold transformation is correct and yields new functions which are improvements over the respective originals.*

PROOF. Partial correctness is immediate, since the transformation steps are all equivalences. It is sufficient to show that the new functions are improvements over the originals.

Consider a transformation beginning with functions $\mathbf{f}_i \vec{x}_i \triangleq e_i$, $i \in \{1, \dots, m\}$ and producing functions

$$\mathbf{f}'_i \vec{x}_i \triangleq e'_i\{\vec{\mathbf{f}}'/\vec{\mathbf{f}}\}, \quad i \in \{1, \dots, m\}.$$

Let k be the number of transformation steps performed, starting at the expressions e_i , and ending with expressions e'_i . Let expressions e_{i1}, \dots, e_{ik} ($i \in \{1, \dots, m\}$) denote the intermediate expressions after each step of the transformation, where $e_{ik} \equiv e'_i$.

We consider the case where only one function becomes marked during the transformation. The general case is a straightforward (but notationally complex) generalization. Assume that only the function \mathbf{f}_1 becomes marked during the transformation, and assume that \mathbf{f}_1 becomes marked at step j , $1 \leq j \leq k$.

Define the following functions:

$$\mathbf{f}''_i \vec{x}_i \triangleq e_{ij}\{\mathbf{f}''_1/\mathbf{f}_1\} \quad i \in \{1 \dots m\}.$$

Claim (1). $\mathbf{f}_i \succcurlyeq \mathbf{f}''_i$.

Claim (2). $\mathbf{f}''_i \succcurlyeq \mathbf{f}'_i$.

The proof of the theorem then follows by transitivity. It remains to prove the claims.

PROOF OF CLAIM (1). Since e_{ij} is obtained from e_i without using the marked folding rule, the functions \mathbf{f}''_i are derivable via an improved unfold-fold transformation. Hence the claim follows from Theorem 9.6.3. \square

Before we move to the proof of Claim (2), we need a couple of technical lemmas giving us some properties of the intermediate functions \mathbf{f}''_i .

LEMMA 9.6.4. $e_1 \succcurlyeq \mathbf{f}''_1 \vec{x}_1$.

PROOF. Recall that $\mathbf{f}_1'' \bar{x}_1 \triangleq e_{1j}$. Since e_{1j} is the expression obtained after rule (5), and since all the preceding transformation steps are all improvements (since they are all steps (1)–(3)), we must have that $e_1 \succeq \surd_{e_{1j}}$. By Claim (1), it follows that $\surd_{e_{1j}} \succeq \surd_{e_{1j}} \{\mathbf{f}_1''/\mathbf{f}_1\}$, and by Proposition 6.1.3 we have that $\surd_{e_{1j}} \{\mathbf{f}_1''/\mathbf{f}_1\} \triangleleft \mathbf{f}_1'' \bar{x}_1$. Putting these together we conclude that $e_1 \succeq \mathbf{f}_1'' \bar{x}_1$ as required. \square

LEMMA 9.6.5. *Let e be any expression not involving functions \mathbf{f}_i'' . Let e' be any expression obtained from e by an application of one of the rules (1)–(4) of Definition 9.6.2, under the assumption that the definition of \mathbf{f}_1 is marked (and nonrecursive). Then it follows that $e\{\mathbf{f}_1''/\mathbf{f}_1\} \succeq e'\{\mathbf{f}_1''/\mathbf{f}_1\}$.*

PROOF. We argue by cases according to rules. We will also adopt the following convention: if θ denotes some substitution, then θ' will denote the substitution obtained by applying the replacement $\{\mathbf{f}_1''/\mathbf{f}_1\}$ to the range of θ .

- (1) Since e does not contain any instances of functions $\bar{\mathbf{f}}''$, and since \mathbf{f}_1 is marked, in this case a subexpression $(\mathbf{f}_i \bar{x}_i)\theta$ is replaced by $e_i\theta$ for some $i > 1$. It follows that a corresponding subexpression $(\mathbf{f}_i \bar{x}_i)\theta\{\mathbf{f}_1''/\mathbf{f}_1\}$ occurs in $e\{\mathbf{f}_1''/\mathbf{f}_1\}$, so by a similar replacement we can obtain $e'\{\mathbf{f}_1''/\mathbf{f}_1\}$. Since unfolding is an improvement, we have that $e\{\mathbf{f}_1''/\mathbf{f}_1\} \succeq e'\{\mathbf{f}_1''/\mathbf{f}_1\}$.
- (2) Consider the case when we replace an occurrence of $\surd_{e_1}\theta$ by $(\mathbf{f}_1 \bar{x}_1)\theta$ (for the other cases we can argue as above). There is a corresponding occurrence of $\surd_{e_1}\theta\{\mathbf{f}_1''/\mathbf{f}_1\}$. Now since e_1 by assumption does not contain \mathbf{f}_1 , we have that $\surd_{e_1}\theta\{\mathbf{f}_1''/\mathbf{f}_1\} \equiv \surd_{e_1}\theta'$, and since $\surd_{e_1}\theta' \succeq (\mathbf{f}_1 \bar{x}_1)\theta'$ we can conclude that $e\{\mathbf{f}_1''/\mathbf{f}_1\} \succeq e'\{\mathbf{f}_1''/\mathbf{f}_1\}$.
- (3) Since improvement laws do not depend on function definitions (and are improvements by definition), the law is also applicable in $e\{\mathbf{f}_1''/\mathbf{f}_1\}$, and the result is an improvement.
- (4) Replaces an occurrence of $e_1\theta$ by $(\mathbf{f}_1 \bar{x}_1)\theta$. Since e_1 does not contain \mathbf{f}_1 we have that:

$$\begin{aligned} e_1\theta\{\mathbf{f}_1''/\mathbf{f}_1\} &\equiv e_1\theta' && \text{(since } \mathbf{f}_1 \text{ nonrecursive)} \\ &\succeq (\mathbf{f}_1'' \bar{x}_1)\theta' && \text{(Lemma 9.6.4)} \\ &\equiv (\mathbf{f}_1'' \bar{x}_1)\theta\{\mathbf{f}_1''/\mathbf{f}_1\} \\ &\equiv (\mathbf{f}_1 \bar{x}_1)\theta\{\mathbf{f}_1''/\mathbf{f}_1\} \end{aligned}$$

Thus we can conclude that $e_1\theta\{\mathbf{f}_1''/\mathbf{f}_1\} \succeq (\mathbf{f}_1 \bar{x}_1)\theta\{\mathbf{f}_1''/\mathbf{f}_1\}$ and hence that $e\{\mathbf{f}_1''/\mathbf{f}_1\} \succeq e'\{\mathbf{f}_1''/\mathbf{f}_1\}$ as required. \square

PROOF OF CLAIM (2). Using the Improvement Theorem, it is sufficient to show that

$$e_{ij}\{\mathbf{f}_1''/\mathbf{f}_1\} \succeq e_{ik}\{\bar{\mathbf{f}}''/\bar{\mathbf{f}}\} \quad (2)$$

since this ensures that the functions \mathbf{f}_i' can be obtained from the \mathbf{f}_i'' in the manner of the Improvement Theorem. To establish (2) it is enough to show that

$$e_{ij}\{\mathbf{f}_1''/\mathbf{f}_1\} \succeq e_{ik}\{\mathbf{f}_1''/\mathbf{f}_1\}, \quad (3)$$

since by Claim (1) we have that $e_{ik}\{\mathbf{f}'_1/\mathbf{f}_1\} \succeq e_{ik}\{\bar{\mathbf{f}}''/\bar{\mathbf{f}}\}$. But (3) is readily established by inductive application of Lemma 9.6.5, (noting that improvement is transitive and reflexive), and this completes the proof. \square

Example. Returning to the problematic example, we transform the function `fit` using the extended method. We begin with expression `filterp(iterate f x)`. We will use “ \rightarrow ” to denote an application of one or more transformation rules¹⁰ and derive

$$\begin{aligned} & \text{filterp}(\text{iterate } f \ x) \\ & \rightarrow \text{filterp}(x.(\text{iterate } f \ x)) && \text{(unfold, tick-elim)} \\ & \rightarrow \checkmark \text{if } p \ x \ \text{then } x.\text{filterp}(\text{iterate } f \ (f \ x)) && \text{(unfold)} \\ & \quad \text{else } \text{filterp}(\text{iterate } f \ (f \ x)) \\ & \rightarrow \text{if } p \ x \ \text{then } x.\text{filterp}(\text{iterate } f \ (f \ x)) && \text{(mark fit)} \\ & \quad \text{else } \text{filterp}(\text{iterate } f \ (f \ x)) \\ & \rightarrow \text{if } p \ x \ \text{then } x.\text{fit } p \ f \ (f \ x) && \text{(marked fold, twice)} \\ & \quad \text{else } \text{fit } p \ f \ (f \ x) \end{aligned}$$

Now we conclude with the construction of the new function

$$\text{fit}' \ p \ f \ x \triangleq \text{if } p \ x \ \text{then } x.\text{fit}' \ p \ f \ (f \ x) \\ \quad \text{else } \text{fit}' \ p \ f \ (f \ x)$$

Notice that the amount of tick manipulation is very minimal. The reader is invited to rework the example from Figure 5 using the extended method, making use of marked folding instead of improved folding, and compare the degree of tick “bookkeeping” required.

As a final remark on the extended method, we note that it is sensitive to the arity of the initial definitions. If we had started the above transformation with a similar definition

$$\text{fit}'' \ p \ f \triangleq \lambda x.\text{filterp}(\text{iterate } f \ x)$$

then we would not have succeeded, because we cannot propagate a tick up to the top level to enable the marking step. It is possible to come up with a version of the transformation rules which would circumvent this problem, but the rules become much more complex, since they involve a much more complex definition of marked folding.

9.7 Further Extensions

In conclusion of this section we note some ideas for further extensions to the unfold-fold transformations described here.

Improvement Lemmas. One reasonable question is whether the transformation for eliminating concatenate described in the previous section can be justified using the improved unfold-fold method (or its extension). The answer is: almost. Each of the transformation rules (1)-(6) of Definition 8.1 can be justified as sequences of improved unfold-fold steps, with the exception of rule (3) (the associativity property

¹⁰In the earlier examples we simply used the improvement relation. Since marked folding is not a local improvement it does not make sense to use it here.

of concatenation) and rule (6) (which depends on rule (3)). As we mentioned earlier, we could simply add (3) as an “improvement law,” although in general we would probably need to add conditions that say that functions involved in such extended improvement laws are not themselves subject to further transformations.

A more interesting point is how we might establish this lemma using the improved unfold-fold method itself. The idea of using unfold-fold to prove lemmas comes from Kott [1980]. Proietti and Pettorossi [1994] describe how it can be done in such a way that it can be safely integrated into correct unfold-fold transformations. Here we outline how it can be achieved within the framework of improved unfold-fold transformations.

We can achieve safe “unfold-fold lemmas” using the improved unfold-fold method as follows. Suppose we wish to replace an occurrence of expression e_1 by e_2 during an improved unfold-fold transformation. The replacement will not violate the correctness of the transformation providing that $e_1 \cong e_2$, and $e_1 \succsim e_2$. We can establish these properties using the improved unfold-fold method as follows. First define $\mathbf{g}_1 \vec{x} \triangleq e_1$ and $\mathbf{g}_2 \vec{x} \triangleq e_2$, where \vec{x} includes the free variables of e_1 and e_2 . It is necessary and sufficient to prove that $\mathbf{g}_1 \cong \mathbf{g}_2$ and $\mathbf{g}_1 \succsim \mathbf{g}_2$.

We can proceed by transforming \mathbf{g}_1 and \mathbf{g}_2 , by improved unfold-fold transformation, to obtain new functions \mathbf{g}'_1 and \mathbf{g}'_2 . If we do this in such a way that \mathbf{g}'_1 and \mathbf{g}'_2 are syntactically equivalent, then we have proved that $\mathbf{g}_1 \cong \mathbf{g}_2$, and hence that $e_1 \cong e_2$. To establish that $e_1 \succsim e_2$ it is sufficient to add the condition that the improvement laws used to obtain \mathbf{g}'_2 are cost equivalences (so for example we are not allowed to perform tick elimination). By this means (using Proposition 5.2.4) we have guaranteed that $\mathbf{g}'_2 \triangleleft \mathbf{g}_2$. Thus we have $\mathbf{g}_1 \succsim \mathbf{g}'_1 \triangleleft \mathbf{g}'_2 \triangleleft \mathbf{g}_2$ and hence that $e_1 \succsim e_2$.

To take the example of the associativity property of concatenate, take

$$\begin{aligned} \mathbf{g}_1 x y z &\triangleq (x ++ y) ++ z \\ \mathbf{g}_2 x y z &\triangleq x ++ (y ++ z) \end{aligned}$$

and routinely derive

$$\begin{aligned} \mathbf{g}'_1 x y z &\triangleq \sqrt{\text{case } x \text{ of}} \\ &\quad \text{nil} : y ++ z \\ &\quad h.t : h.(\mathbf{g}'_1 t y z) \\ \mathbf{g}'_2 x y z &\triangleq \sqrt{\text{case } x \text{ of}} \\ &\quad \text{nil} : y ++ z \\ &\quad h.t : h.(\mathbf{g}'_2 t y z) \end{aligned}$$

where \mathbf{g}'_2 is obtained by a single improved-unfold step, and so is cost equivalent to \mathbf{g}_2 . Thus we have proved that $(x ++ y) ++ z \cong x ++ (y ++ z)$ and $(x ++ y) ++ z \succsim x ++ (y ++ z)$.

Folding Using Previous Definitions. Derived functions are improvements on the originals, so we can allow folding against functions obtained from previous transformation steps. For example, if an improved unfold-fold transformation from \mathbf{f} derives a function \mathbf{f}' then $\mathbf{f} \succsim \mathbf{f}'$. So a subsequent transformation can include a more general form of fold, in which an instance of the body of \mathbf{f} (suitably “ticked”) can be replaced by a call to \mathbf{f}' . Integration of this idea with the extended method is a topic for further work.

Negative Ticks. Transformations in which folds steps occur before any unfold steps can be allowed by “borrowing” ticks and “paying them back” later. This corresponds to use of the law $\surd e_1 \gtrsim \surd e_2 \Rightarrow e_1 \gtrsim e_2$ in order to establish the improvement property. It might be possible to incorporate this idea into a stepwise transformation (which uses only axioms, congruence properties, and transitivity) by introducing *negative* ticks. At the time of writing we have not found examples which motivate a deeper investigation of this idea.

10. RELATED WORK

Although the techniques developed in this article may find applications to the verification of schematic transformations (e.g., see Huet and Lang [1978]), our main focus has been on transformations of the “generative set” nature,¹¹ where a small set of rules are repeatedly applied in a variety of ways—the archetypical example being the unfold-fold method. These styles of program transformation, including partial evaluation, have been active research topics in functional programming for the last decade and more, and the problem of correctness has received surprisingly little attention, particularly in contrast to the situation in logic programming.

In this section we present a detailed survey of related work in the study of correctness of transformations in declarative languages and the relationship to:

- the basic transformations (Section 7),
- the Improvement Theorem (Section 5), and
- the application of these techniques to the unfold-fold transformation (Section 9).

10.1 The Basic Transformations

Our basic definition of a transformation as a replacement of an expression by an equivalent one within a recursive definition, and the fact that this operation is partially, but not totally, correct (Proposition 4.3.1), occurs in a number of guises, as do a number of the “basic” transformations of Section 7.

Partial Correctness. In the setting of recursive program schemes (which for the present purposes we can consider to be first-order nonstrict functional programs), Kott [1978] (and Courcelle [1979]) studied transformations consisting of unfolding, folding, and laws about primitives; one of the basic correctness results in these works is a partial-correctness result for the transformation (Theorem 1 of Kott [1978] and Theorem 5.20 of Courcelle [1979]), which in turn depends on a least fixed-point property of the recursive schemes. A direct proof of partial correctness of transformation under general equivalence (rather than the more restrictive equivalence under transformation by unfold-fold plus laws) is given by Scherlis for a first-order strict functional language Scherlis [1980, Program Substitution Theorem, 2.7]. With respect to unfold-fold transformations in their full (but arguably unnecessary) generality—where one is allowed to unfold as well as fold against earlier versions of a given function—partial correctness follows from the main results of Zhu [1994].

¹¹For a classification of transformation approaches see Partsch and Steinbruggen [1983].

Unique Fixed Points for Total Correctness. Courcelle [1979] studies conditions under which a system of recursive equations has a unique solution (modulo some equational theory of the primitive functions). One of the applications of such conditions is the total-correctness problem in unfold-fold transformations, but the idea is equally applicable to the more general definition of transformation used here. The basic idea is as follows. Just as in Proposition 4.3.1, the partial-correctness result follows from the observation that, when \mathbf{f} is transformed to \mathbf{g} , \mathbf{f} is a fixed point of \mathbf{g} 's defining equation, and thus $\mathbf{g} \sqsubseteq \mathbf{f}$. Now if we can show that \mathbf{g} 's definition has a *unique* fixed point, then it follows that \mathbf{f} and \mathbf{g} are equivalent. Courcelle's conditions to guarantee uniqueness of fixed points are expressed in terms of properties of the intended algebraic laws of the primitive functions rather than the program itself. The conditions are somewhat technical, but also very restrictive—e.g., in the presence of nonlinear laws—and so have limited practical application to unfold-fold transformations.

A related approach to correctness appears in the synthesis method of Manna and Waldinger [1979]. Their method (independently proposed at about the same time as Burstall and Darlington's unfold-fold transformation) uses the same basic steps as the unfold-fold method; but in their system, a derivation of a program g from f must come together with a termination proof — i.e., a proof that g terminates for all inputs. Given the partial-correctness property, this is sufficient to guarantee total correctness because it shows that the function computed by g is maximal in its domain. The applicability of the method depends on f being total: the method is applied to a language with a discrete data domain (i.e., no lazy data structures or higher-order functions).

Reversible Transformation. A simple application of the partial-correctness property ensures that a *reversible* transformation is totally correct.

Based on this observation, in a later work by Courcelle [1986] a much simpler and more syntactic condition for total correctness of unfold-fold-like transformations is presented. The method restricts the application of the unfold-fold steps so as to guarantee that the transformation is reversible. The restriction is very simple: partition the set of functions \vec{f} to be transformed into two sets $\vec{f}^{\#}$ and $\vec{f}^{\bar{\#}}$; transform only the functions $\vec{f}^{\bar{\#}}$, but only allowing unfold and fold steps which use functions in $\vec{f}^{\#}$. It is easy to see that this method guarantees reversibility, but in practice this transformation is too restrictive to justify nontrivial transformations, since it never allows the introduction of direct recursion.

The use of reversibility conditions also occurs in the study of unfold-fold transformations of logic programs. Maher [1987] and Gardner and Shepherdson [1991] define a folding operation for logic programs which only allows folding against a clause *in the current program*. This guarantees that the folding step is reversible (via an unfolding), and certain total-correctness results can then be established. Pettorossi and Proietti [1993] discuss reversible folding and the principle of using reversibility to obtain correctness for *goal replacement*, which can be thought of as the analogy of the definition of transformation in this article. As Pettorossi and Proietti (and others) note, the reversibility restrictions seriously limit the power of the transformation, and this also appears to be the case for functional languages.

10.2 Replacement Transformations

The Expressive Power of Unfold-Fold Transformations. Kott [1980] notes that the unfold-fold transformation method (in his formulation) is incomplete: there are equations which cannot be synthesized from each other. This result¹² is a consequence of the fact that in any unfold-fold there is a certain linear relationship (“parallel-outermost linear” in the terminology of Boudol and Kott [1983]) between the respective approximants (in the Kleene-chain) to the least fixed points of the original and transformed programs, but that there exist equivalent programs whose approximants are related in a “nonlinear” fashion.

Kott’s observation seems to have been independently rediscovered by Zhu [1994], where the result is strengthened to some extent by use of a more general definition of unfold-fold transformation and clarified by application to a number of concrete examples which show that the incompleteness has some practical significance. Related results applicable to pure unfolding and folding transformations are derived by Amtoft [1993] using so-called multilevel transition systems. Incompleteness of the unfold-fold method motivated Kott to define a (complete) generalization called *second-order replacement* (see also Kott [1985]). The basic definition is of transformation via replacement of equivalent expressions (in the context of recursive definitions); the definition of a transformation used in this article is the direct analogy of this definition for a higher-order language. Completeness of the second-order replacement method (with respect to the corresponding definition of equivalence), as with our transformation, is obvious. Kott suggests an application of second-order replacement to proving equivalence of programs by using a variant of the above reversibility condition to ensure that the replacement is totally correct.

Replacement in Logic Programming Languages. The analogy of a replacement transformation in logic programs would be to allow the replacement of a sequence of goals by a “logically equivalent” sequence. Just as for our definition of a transformation, total correctness is not guaranteed because the logical equivalence does not necessarily hold in the new program. Relative to unfold-fold transformations, replacement transformations have not been widely studied in the logic programming community.

A more restricted notion of replacement, together with conditions ensuring total correctness, was introduced by Tamaki and Sato [1984] as an extension to the unfold-fold method. Tamaki and Sato’s replacement condition guarantees that the logical equivalence in question does not depend on the clause which is transformed — hence the goal replacement step is reversible. This reversibility is sufficient to yield total correctness (with respect to a least Herbrand model semantics) (see also Pettorossi and Proietti [1993]). Gardner and Shepherdson [1991] study similar goal replacement conditions (correcting an error in the formulation of Tamaki and Sato’s condition) for stronger semantics, and Maher [1987] defines replacement conditions which also effectively mean that the equivalence in question does not depend on the clause which is transformed.

Tamaki and Sato’s main technical result concerning goal replacement is a condition that guarantees total correctness when it is used in conjunction with a cor-

¹²The main technical results are developed in Boudol and Kott [1983].

rectness preserving unfold-fold transformation approach (discussed below). The condition in question is derived from the total correctness proof of their unfold-fold strategy: it relates to the improvement condition of our main theorem in the sense that it requires reduction in the size of the proof trees of (respectively) the clauses and their intended replacements.

A related goal replacement condition is considered by Proietti and Pettorossi for computed answer substitutions [Proietti and Pettorossi 1994]. In this work the partial correctness of goal replacement itself is established by using (correct) unfold-fold transformation (following a strategy suggested by Kott [1980]). Progress properties of the unfold-fold proof which relate to the number of unfolding versus the number of folding steps are then used to establish total correctness. In conclusion of the previous section we showed how this idea could be justified in conjunction with the improved unfold-fold method. However, it seems that we can go further than Proietti and Pettorossi and allow this replacement operation to be used “recursively” within the sub-unfold-fold proofs.

Bossi, Cocco, and Etalle: Replacement Using Semantic Delay. Starting with the more general form of replacement, Bossi, Cocco, and Etalle [Bossi et al. 1992b] study conditions guaranteeing total correctness with respect to both Fitting’s and Kunen’s semantics. The main condition relates two quantities: the *dependency degree* and the *semantic delay*. Suppose we wish to replace some conjunction of literals C by D in the body of a clause cl in some program P . The first requirement is that C and D are semantically equal in the given program (note that Tamaki and Sato’s condition —and in effect many of the other conditions for replacement in the literature— requires that they are equal in $P - cl$). Now if D is independent of cl the transformation is totally correct (since no new loops can be introduced). Otherwise there is some dependency between D and cl . They define the *dependency degree*, roughly, to be the shortest path from a literal in D to the clause cl . The *semantic delay* of D with respect to C is a semantic measure (based on the minimal ordinal number of iterations of the least fixed-point operator required to prove the truth or falsehood of each closed instance of the literals) of how much “slower” D is than C . Their main theorem says that if the dependency degree of D on cl is not less than the semantic delay (in the program P) of D with respect to C , then the replacement is correct. The intuition is that under the conditions of the theorem “there is no room to introduce a loop.”

There is a strong analogy between the notion of improvement and that of semantic delay. In particular, if the semantic delay is zero then in some sense D is an improvement over C , and the replacement is correct (irrespective of the dependency degree). However, in practice the theorem is not used in this way: the use of the dependency degree part seems essential. Bossi et al. show how the theorem can be used to justify operations such as folding, fattening, and thinning (see also Bossi et al. [1992a] which considers the application to folding for the computed answer substitution semantics). It may be possible to simplify the application of the techniques to Tamaki and Sato-style unfold-fold transformations by allowing semantic delay to be negative and by adopting an analogy of the tick function to perform the necessary accounting. Bossi et al. [1992a] argue that their theorem allows some folding steps to be justified without appealing to the “transformation

history” (the unfold-fold steps taken so far). Our justification of folding steps *necessarily* depends on the transformation history, but this is not a negative point since we use the history to justify in a simple syntactic way the necessary semantic conditions; as a corollary we guarantee a desirable improvement property of the resulting program.

Pursuing the analogy in the other direction, can we make use of some form of dependency degree? The main obstacles seem to be the nondiscrete nature of data in our language that leads to more distinctions in termination behavior than just “looping” and “nonlooping.” It is possible that we can achieve the same effect by developing a “weighted” version of improvement in which we can assign different weights to each function. Amtoft [1993] suggests that his framework can account for the dependency degree idea using a similar idea of weights.

10.3 Unfold-Fold-Specific Correctness Conditions

There are a number of approaches to total correctness which are specific to the unfold-fold transformations. In the setting of first-order functional languages we consider the work of Kott [1978; 1985], who was the first to study total correctness of unfold-fold transformation, and the work of Yongqiang, Ruzhan, and Xiaorong [1987]. We also mention the related transformation method of Scherlis [1981] for which a total correctness result has been established. Finally we will consider some key methods in the extensive literature on unfold-fold transformations in logic programming languages and their relation to the method described in Section 9.

Kott: A Theoretical Study of Unfold-Fold. Of the very few studies of correctness in unfold-fold transformations of functional programs, Kott’s work [Kott 1978; 1985] is probably the most well known.

We believe that Kott’s results on the subject have not been well understood. References to Kott’s work within the functional programming community usually produce a slogan to the effect of “an unfold-fold transformation is correct if there are at least as many unfolds as there are folds”—and indeed Kott’s work has some theorems which are conditional on this “at least as many unfolds as there are folds” property. This unqualified slogan is certainly not an accurate picture of Kott’s technical results; nevertheless this basic idea can provide a reasonable intuition for many of the methods for total correctness in logic programming—as noted by Pettorossi and Proietti [1993], who add the intuition that this condition ensures that “going backward in the computation (as folding does) does not prevail over going forward in the computation (as unfolding does).” Amtoft [1992; 1993] takes this intuition as a starting point. With suitable generalizations (e.g., noting that “some unfoldings are more important than others”) he is able to construct a framework which can explain many of the total correctness conditions used in the logic programming literature.

In fact, Kott’s results are restrictive, primarily, we believe, because they do not satisfactorily take into account *where* the respective folds and unfolds occur,¹³ other than by applying some very general restrictions to the transformations and by

¹³In our setting this is managed by the tick algebra. For example if a tick from an unfolding cannot be propagated (using the full theory of improvement) to an intended folding site then the unfold and fold steps are *unrelated* with respect to termination properties.

weakening the correctness condition.

We will primarily discuss results from the later work [Kott 1985] which is closely related to, but less restrictive than, the results in Kott [1978].

The language studied is a first-order nonstrict functional language consisting of a set of mutually recursive equations. The semantic setting is that of algebraic semantics, so the results are parameterized on an interpretation of the meaning of the primitive functions (which could include lazy and strict data constructors, as well as conditional expressions.)

A restricted form of unfold-fold transformation is studied, in which the body of a single function (among a set of functions) is rewritten by a sequence of unfolding steps, a sequence of applications of laws about the primitives, and finally a sequence of folding steps (strictly in that order). The expression thus obtained forms the body of a new version of the function.

The results are split into two cases according to the very last folding step performed. The cases do not seem to be exhaustive, since the term folded in the last folding step is assumed to contain (the results of) all previous folding steps.

The interesting case is when the last fold step introduces direct recursion, and we explain the main results with the help of the example of an incorrect transformation given in the introduction, and Example 9.5.3. The transformation begins with

$$\mathbf{f} \ x \triangleq \mathbf{if} \ x \ \mathbf{then} \ (\mathbf{f} \ x) \ \mathbf{else} \ \mathbf{true}$$

and proceeds by two unfold steps, two applications of the law:

$$\mathbf{if} \ x \ \mathbf{then} \ (\mathbf{if} \ x \ \mathbf{then} \ y \ \mathbf{else} \ z') \ \mathbf{else} \ z \cong \mathbf{if} \ x \ \mathbf{then} \ y \ \mathbf{else} \ z$$

and concludes with a fold step, yielding the new version of the function: $\mathbf{f} \ x \triangleq \mathbf{f} \ x$.

The transformation is within the scope of Kott's definition.¹⁴ Notice again that there are more unfolds than folds, but that the transformation yields a definition which is not equivalent (in the intended interpretation) to the original.

Kott's results (Theorem (4.2)1 and Proposition (4.2)1 of Kott [1985, p.426]) state that the original and transformed versions are equivalent in all interpretations for which an *additional* law holds. An algorithm is given for producing the required extra law, which has the form $e \cong -$, where e is an expression not containing any recursive function symbols, but possibly containing $-$ (representing a looping term). The extra law is dependent on the transformation. The expression e is constructed from the transformation sequence by combining the expressions immediately before the sequence of laws are applied, with those immediately after. In this case the algorithm produces the law

$$\mathbf{if} \ x \ \mathbf{then} \ - \ \mathbf{else} \ \mathbf{true} \cong -$$

which clearly does not hold in the *intended* interpretation.

In comparison with the tick approach (Example 9.5.3), the unfolding steps and the applications of the law in the above example are allowed (since they are all improvement steps), but the fold step cannot be justified because the ticks generated by the unfolding steps cannot be propagated to the outer level (to "pay" for the fold) because the conditional *is not strict* in its second argument.

¹⁴The law is not left linear and so would not be allowed under the additional restrictions of the earlier paper [Kott 1978], but a similar example can be constructed using only left-linear laws.

Kott defines the additional law constructed in the theorem to be *bad* if the constructed expression ϵ in the law does not contain $-$. Such a law is “bad” because it is unlikely to be true for the intended interpretation and therefore tells us nothing about the actual validity of the unfold-fold transformation in question.

Kott [1985] has a proposition (Proposition (4.2) 3) which says that the constructed law will be bad if and only if there are more folds than unfolds. However, this is a syntactic definition of “bad,” and it does not catch all degenerate cases; there are correct transformations for which the number of unfolds is not less than the number of folds, but for which the extra “law” is false in the intended interpretation. As a (contrived) example, begin with the original definition of the function \mathbf{f} in the above example, and obtain a syntactically equivalent “new” version by applying one unfold and then one fold step. Even though the transformation is trivially correct, Kott’s theorem still only guarantees correctness under the condition that the law `if x then $-$ else true` \cong `true` is satisfied.

A consequence of the above proposition about “bad” transformations is the following (Theorem (4.2)2): if the interpretation of all the basic functions is strict, then whenever the number of unfoldings is greater or equal to the number of foldings, the transformation is correct. The intuition for this result is that the “extra” law in this case will already be a consequence of the strict interpretation. Kott claims that this theorem “is of practical interest if we think about programs written in LISP.” Unfortunately the practical interest of the theorem is severely limited by the fact that for a first-order pure subset of LISP (even if we consider a nonstrict semantics) it is necessary to have at least one nonstrict primitive function, typically “if-then-else,” to perform interesting computations.

In comparison, in addition to the fact that we can handle higher-order functions and much less restricted transformations, our approach to obtaining unfold-fold transformations is technically simpler, and more practical, since it can guide the transformation rather than being a post hoc verification.

Yongqiang et al.’s Approach. Yongqiang et al. [1987] propose a method for the correctness of unfold-fold transformations on FP programs (a first-order strict functional language with only “functional” expressions, i.e., no (object) variables—see Backus [1978]). The method is in some sense a dual to the approach suggested by the Improvement Theorem, since it views the computations by the primitive functions as the significant ones and the recursive function call steps as being “free.”

The conditions are quite technical (and we will not attempt to reproduce them here), but one particular reductive measure function is given — together with a reductive set of laws — for which it is claimed that there are many practical transformations.

The method is described in terms of a transformation of a single recursive function, and in the formal definition of an unfold-fold transformation it is assumed that each transformation step derives a new program from the previous one. Clearly this is not as intended, since one must be able to refer back to earlier versions in order to perform useful fold steps. This point seems to be easy to fix (although proofs of the results are not included in the article), but there are some remaining questions when one combines this with the possibility of mutually recursive functions, since, for example, if one is also allowed to unfold using earlier definitions then the

method breaks.

It is not clear (to this author) how the method can be extended to handle lazy data structures, but it is claimed that it can handle some higher-order functions.

Scherlis: Expression Procedures. Considering more specific transformation methods, of particular note is Scherlis' transformation method based on *expression procedures* [Scherlis 1980]. This method is less general than (i.e., can be simulated by) the unfold-fold method, but has the distinction that it preserves total correctness without need for any global constraints. The method is proved correct (for a strict first-order language) using a notion of *progressiveness*, based on reduction orderings (related ideas are used in Reddy [1989] for the synthesis of Noetherian rewrite rules from equational specifications).

In order to describe his transformation method, Scherlis extends programs in a strict first-order functional language with *expression procedures*. Expression procedures are pairs of equivalent expressions that are more general in form than the left and right-hand sides of a function definition, since the left-hand side of an expression procedure is allowed to be a complex term. The transformation system consists of just four rules (ignoring definition deletion) which successively modify a (generalized) program (and do not need to refer back into the transformation history). Briefly, the four rules are (ignoring the side conditions on application relating, in particular, to the intended strict semantics):

- (1) Composition: introduces a new expression procedure of the form $C[(\mathbf{f} x)\sigma] \triangleq C[(e\sigma)]$ from an existing definition $\mathbf{f} x \triangleq e$ and a strict context C .
- (2) Abstraction: the usual notion of simultaneous abstraction extended to the right-hand sides of expression procedures.
- (3) Application: unfolding, generalized to include replacing an instance of the left-hand side of an expression procedure with the corresponding right-hand side.
- (4) Laws: applying laws about primitives to the right-hand sides of definitions.

Notice that there is no folding rule: recursion is typically introduced by first abstracting the body of an expression procedure and then using the application rule to introduce recursion in the abstracted version.

Correctness. The proof of correctness of the expression procedure approach shows that the transformations on the bodies of functions are equivalence preserving. In turn this fact depends on the expression procedures being consistent with the program (i.e., equivalences). These properties essentially give the usual partial correctness result. The final step is to show that the expression procedures are *progressive*—i.e., they guarantee termination properties. Interpreted operationally, this means that one can freely use expression procedures as computation rules without introducing new nonterminating computations.

Progressiveness and Improvement. Our initial aim was to extend Scherlis' correctness result to a higher-order (and nonstrict) language using the improvement theorem. Improvement can be used to characterize the key progressiveness property of expression procedures, but we are unable to prove correctness using the Improvement Theorem because the abstraction rule (which for this transformation is crucial) arbitrarily introduces an additional function call.

In a separate study [Sands 1995a] we prove the correctness of the expression-procedure approach for a lazy higher-order language using a version of improvement, but without using the corresponding Improvement Theorem. The key to the proof is to characterize the progressiveness property of each expression procedure $e_1 \triangleq e_2$ by the requirement that

$$e_1 \succsim \sqrt{e_2}.$$

It should be easy to see that the composition rule defined above guarantees this property for new expression procedures. The problem is to show that this property is maintained by applications of the other transformation rules. To show this, a *family* of improvement relations is defined. Improvement is parameterized by a *weighting* which assigns a positive integer “cost” to each function definition. The improvement theory used here corresponds to the instance where the weight of each function is one. The core of the proof is to show that after each transformation step there exists some weighting which establishes the above property. For this, an additional syntactic restriction on abstraction is added.

Unfold-Fold Transformations of Logic Programs. In contrast to the research on functional languages, the literature on correctness issues in unfold-fold transformations is extensive (e.g., see Tamaki and Sato [1984], Seki [1993; 1991], Amtoft [1992], Kawamura and Kanamori [1990], Kanamori and Fujita [1986], Sato [1990], and Proietti and Pettorossi [1991]) — for a brief survey see Pettorossi and Proietti [1993]. The basis of many of these results is the original paper by Tamaki and Sato [1984], where a particular formulation of the unfold-fold method is introduced. For the most part, the proliferation of results on this topic within the logic programming community is due to the wide spectrum of semantics that can reasonably be used in defining the correctness problem, e.g., least Herbrand model, computed-answer substitutions, finite failures, etc. We have touched upon many of the techniques from logic programming in the preceding discussion — for example, where they rely on certain general ideas such as “partial correctness + reversibility = total correctness” or where the techniques use a more general form of transformation such as replacement (which is often able to simulate unfold-fold transformations).

In this section we consider the key ideas in the original definition (and proof of total correctness) of an unfold-fold transformation method due to Tamaki and Sato [1984]. The relationship to the use of the tick annotations in Section 9 will also be discussed in the light of a direct generalization of Tamaki and Sato’s method proposed by Kanamori and Fujita [1986].

In logic programming, Tamaki and Sato’s formulation of unfold-fold transformation is often taken to be *the* definition. It was the first method proposed for logic languages which preserved total correctness (with respect to the least Herbrand model) without need for a post hoc proof of correctness.

The basic part of Tamaki and Sato’s transformation system consists of applying three rules: definition, unfolding, and folding. These steps are the natural analogues of the corresponding steps in a functional language (we will not go into details here), but their application is restricted in a number of ways.

The unfold and fold steps are defined with respect to particular definitions in the *transformation sequence*. The transformation sequence consists of a sequence of programs P_i . Each unfold or fold step defines a new program in the sequence

in which the transformed clause is replaced with the modified one. In the initial program, P_0 , we identify a subset of clauses D which are nonrecursive (i.e., their bodies contain only clauses defined in $P_0 \setminus D$). Unfolding at step $i + 1$ uses only the clauses defined in P_i . Folding at step $i + 1$ operates on a clause in P_i but folds against a predicate as defined in D . Note that this description is of the *virtual transformation sequence* in which all definitions (“eureka”) are collected at the beginning.

To obtain totally correct transformations Tamaki and Sato add some further restrictions to the folding steps based on the transformation history. These restrictions are that the above form of folding can occur in a clause if either

- (1) it is one of the original clauses in $P_0 \setminus D$ or
- (2) it is a clause which is the result of applying at least one unfolding.

This method is implemented in the transformation by a process of *marking* foldable clauses during the transformation process.

Marks vs. Ticks. The informal connection with the method of “ticks” of Section 9 is clear: to “pay” for a fold step which is not independently justifiable by reversibility (case (1) above) we need to justify it with an earlier fold step. The tick algebra is necessary in our setting because we need to ensure that the unfold and fold steps are related. Some analogy of the tick algebra is not necessary in Tamaki and Sato’s language because the simpler syntactic structure of programs ensures that any unfolding in a clause will be “related” to any possible folding. Another way of saying this is that ticks anywhere in the body of a program can be propagated to the top level. So the *marks* of Tamaki and Sato informally correspond to “one or more ticks in the body.”¹⁵ With this insight we anticipate Kanamori and Fujita’s generalization of Tamaki and Sato’s method.

From Marks to Counters. Kanamori and Fujita’s [1986] refinement of Tamaki and Sato’s transformation replaces the basic “foldable” marks by a counter (a natural number) attached to each clause. Unfolding a clause C with counter γ in a clause D will add γ to D ’s counter. Folding (in a clause with a non-zero counter) decrements the counter of the clause. Given the above analogy with the tick approach, the counter corresponds to the number of ticks in the body of a function. The details of the proof of the counter approach (a direct generalization of Tamaki and Sato’s proof) reveal analogies with the use of ticks to represent degrees of improvement. A refinement of Tamaki and Sato’s goal replacement conditions is provided by the use of counters. This refinement can be explained by analogy with improvement-based transformation with the following observation: during an unfold-fold transformation a law of the form $e \gtrsim \sqrt{\dots \sqrt{e'}}$ is more useful than its consequence $e \gtrsim e'$, since with the former transformation potentially more folding steps can be justified.

¹⁵We have adopted the notion of “marking” in the extended improved unfold-fold method (defined in Section 9.6). Although we have taken the terminology of certain clauses being “marked” from Tamaki and Sato, it should be noted that the conditions for marking in Tamaki and Sato’s method are more analogous to improved folding. Our condition for marked folding requires that the function introduced by the fold step is marked. Tamaki and Sato’s “marked folding” requires that the clause in which the fold occurs is marked.

11. CONCLUSIONS AND FURTHER WORK

We have presented an *improvement theorem* which says that if the local transformation steps are improvements, then correctness of the transformation follows. The significance of the theorem has been demonstrated with applications to existing transformation methods (which lacked rigorous correctness proofs) (see also Sands [1995b]) and to the long-standing problem of correct unfold-fold transformation in higher-order functional languages. The fact that the improvement condition is a (pre-)congruence relation means that the local “stepwise” character of transformation methods remains, both in correctness proofs which employ the improvement theorem (cf. Section 8) and in the method proposed to constrain unfold-fold transformations (Section 9).

11.1 Variations

Our development has been with respect to a particular language. There are a number of variations in language and the theory of improvement which could be considered. Here we outline the consequences of some orthogonal variations and some areas for further work.

Types. The addition of types to the language does not significantly affect the development of the improvement theorem. Typed versions of improvement can be developed along the lines of typed theories of equivalence (e.g., see Gordon [1995]). From the point of view of the improvement theorem, types play a minor role, and for this reason we have stuck to an untyped language. However, a statically typed version of improvement is useful since some (local) equivalences depend critically on types. For example, “instantiation” makes sense in a typed language, since if e is of list type, then

$$e \stackrel{\sim}{\sim} \text{case } e \text{ of} \\ \quad \text{nil} : \text{nil} \\ \quad h.t : h.t$$

Other important examples of local improvements which hold in polymorphically typed languages include the parametricity theorems which are derived from polymorphic types (e.g., see Wadler [1989b]).

Call-by-Value. The theory of improvement for a call-by-value (CBV) version of the language is straightforward to develop. The theory of improvement can also be developed along similar lines, and the improvement theorem goes through for call-by-value without any significant changes. The more significant differences come in the application of the theorem. Firstly CBV-improvement is not closed under arbitrary substitution, but only under substitution of values, so that $e \succ_{cbv} e'$ does not in general imply that $e\sigma \succ_{cbv} e'\sigma$. A related point is that arbitrary unfolding or beta reduction is not a CBV-improvement, since unfolding can duplicate expressions which would otherwise have been evaluated only once. This means that unfold-fold transformations must use linearity properties to justify unfoldings or must use the rule:

$$\mathbf{f} e_0 \succ_{cbv} \text{let } x = e_0 \text{ in } e, \quad \text{if } \mathbf{f} \text{ is defined by } \mathbf{f} x \triangleq e.$$

Clearly one needs further CBV-improvement laws for let expressions. It appears, then, to require more effort to verify the correctness of call-by-value transformations. On the positive side, the additional reward of such correctness proofs is that the improvement theory for call-by-value is a more realistic than that for call-by-name (since actual implementations of call-by-name languages use a call-by-*need* evaluation mechanism).

Call-by-Need. A problem left open in Sands [1991] is: is there a tractable *call-by-need* theory of improvement? A “reasonable” call-by-need theory is a prerequisite to answering the following natural question:

Does the improvement theorem hold for call-by-need?

At the time of writing we do not know the answer to this question. A first step is to find a characterization of the call-by-need improvement theory. A recently proposed term-based call-by-need theory [Ariola et al. 1995] might prove an interesting basis for this investigation. Interestingly, many of the call-by-*need* reduction rules are anticipated by the above call-by-*value* theory of improvement. As it stands, the call-by-value, call-by-name, and call-by-need improvement relations are incomparable.

We might also look for a contribution from improvement theory to the study of call-by-need calculi, since improvement defined in terms of a call-by-need reduction relation might provide a more appropriate semantic criterion for the “correctness” of call-by-need calculi (since, as shown by Ariola et al. [1995], the criterion of observational congruence does not distinguish call-by-name and call-by-need).

Alternative Improvement Relations. There are many possible variations on the theory of improvement which stem from the many possible ways of counting the “cost” of evaluation. The characterization of improvement in terms of a simulation relation (Definition 6.1.1) and the corresponding context lemma can be proved once and for all for a large class of improvement relations (see Sands [1991] for further details). It appears that the Improvement Theorem also holds for many of these variations. In particular, it is easy to verify that the Improvement Theorem holds for versions of improvement which also count other operations, other than just function calls (e.g., β -reductions). However, most of these variations do not add significantly to the transformations that we are able to prove.

A version of improvement which deserves some further investigation is the *weighted improvement* from Sands [1995a], which assigns a different (positive) weight to each function definition. If the Improvement Theorem holds for weighted improvement (for any weight) then it should be possible to verify the transformation described in Sands [1995a]. We leave this topic as another avenue for further work.

APPENDIX

The Theory of Operational Approximation

In this appendix we fill in some details of the theory of operational approximation, leading to a proof of a least (post-)fixed-point theorem. The development largely mirrors that of the theory of improvement from Section 6.

The key to studying operational approximation and equivalence is to find tractable characterizations of these relations; the characterization which we use says that two expressions are related by observational approximation if there is a certain kind of

simulation relation which relates them.¹⁶ An expression is simulated by another if (1) whenever the first evaluates to a weak head normal form, so does the second and (2) the weak head normal forms are of the same kind (same constructor, or both are closures) whose “components” are themselves in the simulation relation. To simplify the presentation and use of simulation we define the following:

Definition A.1. If R is a relation on closed expressions then let R^\dagger denote the least relation on weak head normal forms such that $e_0 R^\dagger e_1$ if

- $e_0 \equiv c(\vec{e}), e_1 \equiv c(\vec{e}')$ and $\vec{e} R \vec{e}'$ and
- $e_0, e_1 \in \text{Closures}$, and if for all closed $e, (e_0 e) R (e_1 e)$.

Definition A.2. A relation R on closed expressions is a *simulation* if for all e_1, e_2 , whenever $e_1 R e_2$, if $e_1 \Downarrow w_1$ then $e_2 \Downarrow w_2$ for some w_2 such that $w_1 R^\dagger w_2$.

There is a largest simulation relation, given by the union of all simulation relations. The following important characterization says that this maximal simulation coincides with observational approximation on closed expressions:

THEOREM A.3. *For all e, e' we have $e \sqsubseteq e'$ if and only if there exists a simulation R such that for all closing substitutions $\sigma, e\sigma R e'\sigma$.*

PROOF. Using the now standard techniques of Howe [1989] (see also Gordon [1995]), and recasting the language in the form of Howe’s *lazy computation systems*, the proof follows easily. \square

Proof Techniques for Operational Approximation. Now we have our basic proof technique for demonstrating operational approximations, a so-called *coinduction* principle (after Milner and Tofte [1991]):

- to show that $e \sqsubseteq e'$ it is sufficient to find a simulation relation containing each closed instance of the pair.

The above properties ensure that the existence of such a simulation relation is a necessary and sufficient condition for $e \sqsubseteq e'$. In fact, there is a useful refinement of this proof technique which is well known from (bi)simulation in CCS — that of *bisimulation up-to* [Milner 1989]:

Definition A.4. A relation R on closed expressions is a *simulation up to observational approximation* if for all e_1, e_2 , whenever $e_1 R e_2$, if $e_1 \Downarrow w_1$ then $e_2 \Downarrow w_2$ for some w_2 such that $w_1 \sqsubseteq; R^\dagger; \sqsubseteq w_2$.

PROPOSITION A.5. *If R is a simulation up to observational approximation then $R \subseteq \sqsubseteq$*

PROOF. A straightforward adaptation of the up-to technique for improvement given in Section 6, Prop. 6.2.4. \square

¹⁶Given that our operational semantics is in terms of reduction contexts, it would be more standard to characterize observational approximation in terms of reduction contexts. We stick with a “simulation” characterization because (1) we take advantage of some metatheory for simulation relations [Howe 1989; Sands 1991], (2) we will make use of certain proof techniques associated with (bi)simulation relations, and (3) because it gives a more reasonable treatment of constructor expressions

To summarize:

—to prove that $e \sqsubseteq e'$ it is sufficient to find a simulation up to observational approximation which contains all closed instances of the pair.

A Least Post-Fixed-Point Theorem

Now we turn to one last property of the language which we will need. This concerns recursive definitions and takes the form of a least fixed-point property. If we view a recursive definition (or, in general, a mutually recursive set of definitions) as an equation in \mathbf{f} (the defined function), then the actual expression \mathbf{f} is a minimal solution (with respect to operational approximation) of this equation. In the proofs that follow we will assume that all functions have arity zero, so that closures are just lambda abstractions. This is without loss of generality since a function of arity > 0 can easily be represented, up to operational equivalence, by a function of arity zero.

PROPOSITION 3.3.1 (LEAST PRE-FIXED POINT). Let $\vec{e} \equiv e_1, \dots, e_n$ be a list of expressions and $\vec{x} \equiv x_1, \dots, x_n$ be a list of variables such that $\text{FV}(\vec{e}) \subseteq \{x_1 \dots x_n\}$.

The inequations $e_i \sqsubseteq x_i$, $i = 1 \dots n$, have a solution for $\vec{x} = \vec{g}$, where the functions \vec{g} are defined by

$$\mathbf{g}_i \triangleq e_i\{\vec{g}/\vec{x}\}, \quad i = 1 \dots n.$$

Moreover, for any other solution, $\vec{x} = \vec{e}'$, we have that $\mathbf{g}_i \sqsubseteq e'_i$, $i = 1 \dots n$.

PROOF. Assume the conditions of the proposition. First we need to show that $\mathbf{g}_i \sqsubseteq e_i\{\vec{g}/\vec{x}\}$, but this is straightforward to argue since the left-hand side reduces to the right-hand side in one reduction step. Now suppose that there is another solution \vec{e}' . We need to show that $\mathbf{g}_i \sqsubseteq e'_i$, $i = 1 \dots n$. It is sufficient to show that $e_i\{\vec{g}/\vec{x}\} \sqsubseteq e_i\{\vec{e}'/\vec{x}\}$. Define the following relation:

$$\begin{aligned} R &\stackrel{\text{def}}{=} \{(e_0\gamma, e_0\epsilon) \mid \text{FV}(e_0) \subseteq \vec{x}\} \\ \text{where } \gamma &\stackrel{\text{def}}{=} \{\vec{g}/\vec{x}\} \\ \epsilon &\stackrel{\text{def}}{=} \{\vec{e}'/\vec{x}\} \end{aligned}$$

It is now sufficient to show that R is a simulation up to observational approximation. Now we take an arbitrary pair of R -related terms, which we can write as $e\gamma$ and $e\epsilon$, where $\gamma = \{\vec{g}/\vec{x}\}$ and $\epsilon = \{\vec{e}'/\vec{x}\}$ for some e , \vec{x} . Suppose that $e\gamma \Downarrow w$. We show, by induction on the length of this reduction (and by cases according to the syntactic form of e), that $e\epsilon \Downarrow w'$ for some w' such that $w (\sqsubseteq; R; \sqsubseteq)^\dagger w'$.

Base. $e\gamma \in \text{WHNF}$. There are two subcases according to the structure of e : either e is a constructor expression or a lambda abstraction. Note that e cannot be a variable x_i , since \mathbf{g}_i is not a weak head normal form. We will consider the case where $e \equiv \lambda y.e''$. The other case is easier. Assume without loss of generality that the variable y is distinct from the variables \vec{x} . Now $e\gamma \equiv \lambda y.e''\gamma \Downarrow \lambda y.e''\gamma$ and similarly $e\epsilon \equiv \lambda y.e''\epsilon$. From the definition of bisimulation up-to, it remains to show that, for all closed expressions e''' ,

$$(\lambda y.e''\gamma)e''' (\sqsubseteq; R; \sqsubseteq) (\lambda y.e''\epsilon)e'''.$$

It is sufficient to show that $(\lambda y.e''\gamma)e''' R (\lambda y.e''\epsilon)e'''$, but this is immediate since e''' is closed, and so we have that $(\lambda y.e''\gamma)e''' \equiv ((\lambda y.e'')e''')\gamma$ and $(\lambda y.e''\epsilon)e''' \equiv ((\lambda y.e'')e''')\epsilon$.

Induction. Assume $e\gamma \mapsto e'$ and $e' \Downarrow w$. We can write e uniquely as $C[e'']$, where C does not capture variables; $C\gamma$ is a reduction context; and $e''\gamma$ is a redex. We proceed by cases according to the reduction rule applied. We will only consider the more interesting case, when the redex is a function application (the remaining cases are then straightforward). There are two possible subcases for the structure of e :

- (1) $e \equiv C[\mathbf{f}]$ for some function \mathbf{f} given by $\mathbf{f} \triangleq e_{\mathbf{f}}$ and
- (2) $e \equiv C[x_i]$ for some $i \in \{1 \dots n\}$.

In case (1) $e\gamma \mapsto C[e_{\mathbf{f}}]\gamma$. Now we have that $e\epsilon \cong C[e_{\mathbf{f}}]\epsilon$. The induction hypothesis gives that $C[e_{\mathbf{f}}]\epsilon \Downarrow w''$ for some w'' such that $w (\underline{\sqsubset}; R; \underline{\sqsubset})^\dagger w''$. By simulation, $e\epsilon \Downarrow w'$ for some w' such that $w'' \cong^\dagger w'$, and $w (\underline{\sqsubset}; R; \underline{\sqsubset})^\dagger w'$ then follows from properties of \cdot^\dagger and $\underline{\sqsubset}$. In case (2) $e\gamma \equiv C[x_i]\gamma \mapsto C\gamma[e_i\{\vec{g}/\vec{x}\}]$. We summarize the remainder of the proof in the following diagram, which is filled from left to right and top to bottom (the first square by the induction hypothesis, the second by the assumption that \vec{e}' is also a solution to the inequations, and by simulation:

$$\begin{array}{ccccccc}
C[x_i]\gamma \mapsto C\gamma[e_i\{\vec{g}/\vec{x}\}] & \equiv & C[e_i]\gamma & R & C[e_i\{\vec{e}'/\vec{x}\}]\epsilon & \underline{\sqsubset} & C\epsilon[e'_i] \equiv C[x_i]\epsilon \equiv e\epsilon \\
\Downarrow & & \Downarrow & & \Downarrow & & \Downarrow \\
w & & w (\underline{\sqsubset}; R; \underline{\sqsubset})^\dagger w'' & & \underline{\sqsubset}^\dagger w' & & w'
\end{array}$$

Finally, in the bottom line, $w (\underline{\sqsubset}; R; \underline{\sqsubset})^\dagger w'$ follows easily from properties of \cdot^\dagger and $\underline{\sqsubset}$. \square

ACKNOWLEDGMENTS

The author gratefully acknowledges Torben Amtoft for discussions on the problem early in the development of this work and his help in obtaining some of the references. Thanks to the ‘‘Topps’’ group at DIKU, and in particular to Robert Gluck, Morten Heine Sørensen, John Hatcliff, Neil Jones, Kristian Nielsen, Bob Paige, and Tom Reps for numerous discussions on the subject of program transformation and for comments on earlier drafts. Philip Wadler and the anonymous referees gave valuable comments and suggestions which have helped to improve both the presentation and the technical content. Thanks to L. P. Zong and Mian Tuo for help in obtaining reference information.

REFERENCES

- ABRAMSKY, S. 1990. The lazy lambda calculus. In *Research Topics in Functional Programming*, D. Turner, Ed. Addison-Wesley, Reading, Mass., 65–116.
- AMTOFT, T. 1992. Unfold/fold transformations preserving termination properties. In *PLILP '92*. Lecture Notes in Computer Science, vol. 631. Springer-Verlag, Berlin, 187–201.
- AMTOFT, T. 1993. Sharing of computations. Ph.D. thesis, DAIMI, Aarhus Univ., Aarhus, Denmark.
- ARIOLA, Z., FELLEISEN, M., MARAIST, J., ODERSKY, M., AND WADLER, P. 1995. The call-by-need lambda calculus. In *The 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM Press, New York.

- BACKUS, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM* 21, 8 (Aug.), 613–641.
- BIRD, R. 1984. Using circular programs to eliminate multiple traversals of data. *Acta Informatica* 21, 1, 239–250.
- BLOOM, B. 1988. Can LCF be topped? Flat lattice models of typed lambda calculus. In *The 3rd Annual Symposium on Logic in Computer Science*. IEEE, New York.
- BOSSI, A., COCCO, N., AND ETALLE, S. 1992a. On safe folding. In *PLILP '92*. Lecture Notes in Computer Science, vol. 631. Springer-Verlag, Berlin, 172–186.
- BOSSI, A., COCCO, N., AND ETALLE, S. 1992b. Transforming normal programs by replacement. In *The 3rd Workshop on Meta-Programming in Logic, META '92*. Lecture Notes in Computer Science, vol. 649. Springer-Verlag, Berlin, 265–279.
- BOUDOL, G. AND KOTT, L. 1983. Recursion induction principle revisited. *Theor. Comput. Sci.* 22, 1, 135–173.
- BURSTALL, R. AND DARLINGTON, J. 1977. A transformation system for developing recursive programs. *J. ACM* 24, 1 (Jan.), 44–67.
- CONSEL, C. AND KHOO, S. 1993. On-line and off-line partial evaluation: Semantic specification and correctness proofs. Tech. Rep., Yale Univ., New Haven, Conn. Apr.
- COURCELLE, B. 1979. Infinite trees in normal form and recursive equations having a unique solution. *Math. Syst. Theor.* 13, 1, 131–180.
- COURCELLE, B. 1986. Equivalences and transformations of regular systems—applications to recursive program schemes and grammars. *Theor. Comput. Sci.* 42, 1, 1–122.
- FEATHER, M. 1979. A system for assisting program transformations. Ph.D. thesis, Univ. of Edinburgh, Edinburgh.
- FELLEISEN, M., FRIEDMAN, D., AND KOHLBECKER, E. 1987. A syntactic theory of sequential control. *Theor. Comput. Sci.* 52, 1, 205–237.
- GARDNER, P. AND SHEPHERDSON, J. 1991. Unfold/fold transformations of logic programs. In *Computational Logic: Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin, Eds. MIT Press, Cambridge, Mass.
- GOMARD, C. 1992. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Trans. Program. Lang. Syst.* 14, 2, 147–172.
- GORDON, A. D. 1995. Bisimilarity as a theory of functional programming. In *Proceedings of the 11th Conference on Mathematical Foundations of Programming Semantics, MFPS'95*. Electronic Notes in Computer Science, vol. 1. Elsevier Science, Amsterdam.
- HOWE, D. J. 1989. Equality in lazy computation systems. In *The 4th Annual Symposium on Logic in Computer Science*. IEEE, New York, 198–203.
- HUET, G. AND LANG, B. 1978. Proving and applying program transformations expressed with second order patterns. *Acta Inf.* 11, 1 (Jan.), 31–55.
- HUGHES, R. 1982. Super-combinators: A new implementation method for applicative languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Languages*. ACM, New York, 1–10.
- JONES, N. D., GOMARD, C., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, N.J.
- KANAMORI, T. AND FUJITA, H. 1986. Unfold/fold transformation of logic programs with counters. Tech. Rep. ICOT TR-179, ICOT Research Center, Tokyo.
- KAWAMURA, T. AND KANAMORI, T. 1990. Preservation of stronger equivalence in unfold/fold logic program transformation. *Theor. Comput. Sci.* 1, 73, 139–154.
- KOTT, L. 1978. About transformation system: A theoretical study. In *Program Transformations*, B. Robinet, Ed. Dunod, Paris, 232–247.
- KOTT, L. 1980. A system for proving equivalences of recursive programs. In *The 5th Conference on Automated Deduction*, W. Bibel and R. Kowalski, Eds. Lecture Notes in Computer Science, vol. 87. Springer-Verlag, Berlin, 63–69.
- KOTT, L. 1985. Unfold/fold transformations. In *Algebraic Methods in Semantics*, M. Nivat and J. Reynolds, Eds. Cambridge University Press, Cambridge, Chapter 12, 412–433.

- MAHER, M. 1987. Correctness of a logic program transformation system. Tech. Rep., IBM T. J. Watson Research Center, Yorktown Heights, N.Y. Revised 1989.
- MANNA, Z. AND WALDINGER, R. 1979. Synthesis: Dreams \Rightarrow programs. *ACM Trans. Program. Lang. Syst.* 5, 4.
- MCCARTHY, J. 1967. *A Basis for a Mathematical Theory of Computation*. North-Holland, Amsterdam.
- MILNER, R. 1977. Fully abstract models of the typed λ -calculus. *Theor. Comput. Sci.* 4, 1.
- MILNER, R. 1989. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, N.J.
- MILNER, R. AND TOFTE, M. 1991. Co-induction in relational semantics. *Theor. Comput. Sci.* 87, 1, 209–220.
- PALSBERG, J. 1993. Correctness of binding time analysis. *J. Funct. Program.* 3, 3, 347–364.
- PARTSCH, P. AND STEINBRUGGEN, R. 1983. Program transformation systems. *ACM Comput. Surv.* 15, 1, 199–236.
- PETTOROSSO, A. AND PROIETTI, M. 1993. Transformation of logic programs: Foundations and techniques. Tech. Rep. R 369, CNR Istituto di Analisi dei Sistemi ed Informatica, Rome.
- PEYTON JONES, S. L. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall International Ltd., London.
- PLOTKIN, G. D. 1975. Call-by-name, call-by-value and the λ -calculus. *Theor. Comput. Sci.* 1, 1, 125–159.
- PROIETTI, M. AND PETTOROSSO, A. 1991. Semantics preserving transformation rules for Prolog. In Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '91. *SIGPLAN Not.* 26, 9 (Sept.).
- PROIETTI, M. AND PETTOROSSO, A. 1994. Total correctness of a goal replacement rule based on the unfold-fold proof method. CNR Istituto di Analisi dei Sistemi ed Informatica, Rome.
- REDDY, U. 1989. Rewriting techniques for program synthesis. In *Rewriting Techniques and Applications*. Lecture Notes in Computer Science, vol. 355. Springer-Verlag, Berlin, 388–403.
- RUNCIMAN, C., FIRTH, M., AND JAGGER, N. 1989. Transformation in a non-strict language: An approach to instantiation. In *Functional Programming, Glasgow 1989: Proceedings of the 1st Glasgow Workshop on Functional Programming*. Springer-Verlag, Berlin.
- 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). Springer-Verlag, Berlin, 298–311.
- 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, 495–541.
- SANDS, D. 1995a. Higher-order expression procedures. In *Proceeding of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'95*. ACM, New York, 190–201.
- SANDS, D. 1995b. Proving the correctness of recursion-based automatic program transformations. In *The International Joint Conference on Theory and Practice of Software Development (TAPSOFT/FASE '95)*. Lecture Notes in Computer Science, vol. 915. Springer-Verlag, Berlin. Extended version to appear in *Theor. Comput. Sci.*
- SATO, T. 1990. An equivalence preserving first order unfold/fold transformation system. In *The 2nd International Conference on Algebraic and Logic Programming*. Lecture Notes in Computer Science, vol. 462. Springer-Verlag, Berlin, 175–188.
- SCHERLIS, W. 1980. Expression procedures and program derivation. Ph.D. thesis, Stanford Rep. STAN-CS-80-818, Dept. of Computer Science, Stanford Univ., Stanford, Calif.
- SCHERLIS, W. L. 1981. Program improvement by internal specialisation. In *The 8th Symposium on Principals of Programming Languages*. ACM, New York.
- SEKI, H. 1991. Unfold/fold transformation of stratified programs. *Theor. Comput. Sci.* 86, 1, 107–139.
- SEKI, H. 1993. Unfold/fold transformation of general logic programs for the well-founded semantics. *J. Logic Program.* 16, 1, 5–23.

- TAMAKI, H. AND SATO, T. 1984. Unfold/fold transformation of logic programs. In *The 2nd International Logic Programming Conference*, S. Tarnlund, Ed. MIT Press, Cambridge, Mass., 127–138.
- TURCHIN, V. F. 1986. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.* 8, 3 (July), 292–325.
- WADLER, P. 1989a. The concatenate vanishes. Univ. of Glasgow, Glasgow, Scotland. Preliminary version circulated on the fp mailing list, 1987.
- WADLER, P. 1989b. Theorems for free! In *Functional Programming Languages and Computer Architecture, FPCA '89 Conference Proceedings*. ACM, New York, 347–359.
- WADLER, P. 1990. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.* 73, 1, 231–248. Preliminary version in ESOP 88, Lecture Notes in Computer Science, vol. 300.
- WAND, M. 1993. Specifying the correctness of binding time analysis. *J. Funct. Program.* 3, 3, 365–387.
- YONQUIANG, S., RUZHAN, L., AND XIAORONG, H. 1987. Termination preserving problem in the transformation of applicative programs. *J. Comput. Sci. Tech.* 2, 3, 191–201.
- ZHU, H. 1994. How powerful are folding/unfolding transformations? *J. Funct. Program.* 4, 1 (Jan.), 89–112.

Received January 1995; revised October 1995; accepted December 1995