

Proving the Correctness of Recursion-Based Automatic Program Transformations¹

David Sands²

*Department of Computing Science,
Chalmers University of Technology and Göteborg University,
S-412 96 Göteborg, Sweden; dave@cs.chalmers.se*

This paper shows how the *Improvement Theorem*—a semantic condition for establishing the total correctness of program transformation on higher-order functional programs—has practical value in proving the correctness of automatic techniques. To this end we develop and study a family of automatic program transformations. The root of this family is a well-known and widely studied transformation called *deforestation*; descendants include generalisations to richer input languages (e.g. higher-order functions), and more powerful transformations, including a source-level representation of some of the techniques known from Turchin's *supercompiler*.

1 Introduction

Transformation of recursive programs Source-to-source transformation methods for functional programs, such as *partial evaluation* [15] and *deforestation* [38,3], perform equivalence preserving modifications to the definitions in a given program. These methods fall in to a class which has been called *generative set transformations* [23]: transformations built from a small set of rules which gain their power from their compound and selective application. The classic example of this (informal) class is Burstall and Darlington's unfold-fold method [2]; many automatic transformations of this class can be viewed as specialised instances of unfold-fold rules.

¹ This is a revised and extended version of a paper which appears in the proceedings of TAPSOFT '95 [28]

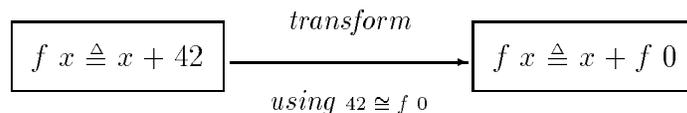
² This work was developed at, and partially funded by the Department of Computer Science, University of Copenhagen (DIKU)

These methods improve the efficiency of programs by performing local optimisations, thus transferring run-time computations to compile-time. In order to compound the effect of these relatively simple local optimisations, it is desirable that such transformations have the ability to introduce recursion. Transformations such as deforestation [38] (a functional form of loop-fusion) and partial evaluation (and analogous transformations on logic programs) have this capability via a process of selectively memoising previously encountered expressions, and introducing recursion according to a “*déjà vu*” principle [15]. See [24] for an overview of transformation strategies which fit this style.

The Problem of Correctness Program transformations performed by a tool such as an optimising compiler should preserve the *extensional* meaning of programs — the properties concerning *what* is computed — in order to be of any practical value. In this case we say that the transformation is *correct*.

One might say that there are two problems with correctness – the first being that it has not been widely recognised as a problem! Because the individual transformation components often represent quite simple operations on programs and are obviously meaning-preserving, confidence in the correctness of such transformation methods or systems is high. The problem with this view, for transformations that can introduce recursion, is that correctness cannot be argued by simply showing that the basic transformation steps are meaning-preserving. Yet this problem (exemplified below) runs contrary to many informal (and some formal) arguments which are used in attempts to justify correctness of particular transformation methods.

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):



This example fits into the framework of the unfold-fold method (first apply the *law* $42 \cong 0 + 42$, and then *fold* $0 + 42$ to get $f\ 0$), and thus illustrates the well-known fact that, in general, unfold-fold transformations preserve only partial correctness. It also serves as a reminder that one cannot argue correctness of a transformation method by simply showing that it can be recast as an unfold-fold transformation. This is an important point because many transformations are cited as being instances of this class.

A Solution, in Principle To obtain total correctness without losing the local, stepwise character of program transformation, it is clear that a stronger condition than extensional equivalence is necessary. In [29] we presented such a condition, *improvement*. The Improvement Theorem states that if the local steps of a transformation are improvements, in a formal sense, then the transformation will be correct, and, *a fortiori*, will yield an “improved” program. The improvement relation is defined in terms of the number of recursive function calls performed during computation: one expression is an improvement over another if in all program contexts it terminates at least as often, but never requires a greater number of function calls in order to do so. The method applies to call-by-name and call-by-value functional languages, including higher-order functions and lazy data structures. In [29] the improvement theorem was used to design a method for restricting the unfold-fold method, such that correctness and improvement are guaranteed. It is also claimed that the improvement theorem has practical value in proving the correctness — without need for further restrictions — of transformation methods suitable for highly optimising compilers.

In this paper we substantiate this claim.

A Solution, in Practice Our aims are firstly to show that the improvement theorem applies to existing transformations. We consider a family of “automatic” program transformations to illustrate the application of the improvement theorem. The root of this family is a well-known and widely studied transformation called *deforestation* [38]. The descendants include various generalisations of the algorithm to handle richer input languages, e.g. including higher-order functions, and more powerful transformations. The more “powerful” methods which are also covered by these generalisations include a source-level representation of Turchin’s *driving*, as found in the *supercompiler* [35].

An essential component of all of these transformations is the ability to create new recursive structures. We provide what we believe to be the first correctness proof for any of this class of transformations, including deforestation, to deal explicitly and correctly with the recursion introduction.

Related Work In this paper we focus on correctness of programs produced by transformation. This does not, for example, include the question of whether transformation algorithms actually terminate and produce a program in all cases. The main technical difficulty in proving correctness is in handling transformations which build recursive programs.

In the study of correctness issues in program transformation of the kind addressed in this paper it is typical to *ignore* the folding or memoisation as-

pects of the algorithms. This often occurs because the correctness issues studied relate to the transformation *algorithm* rather than the correctness of the resulting program. For example, studies of correctness in partial evaluation [11][22][39] ignore the memoisation aspects entirely and deal with the orthogonal issue of the correctness of *binding time analysis*, which controls *where* transformation occurs in a program. Transformations considered by Steckler [34] are quite orthogonal to the ones studied here, since they concern local optimisations which are justified by global data-flow properties of the program in which they are performed. To the author’s knowledge, the only other correctness proofs for automatic transformations of recursive programs which use some form of folding are in the study of related logic-program transformation, e.g. [18] [16]. For an extensive comparison of the improvement theorem with other general techniques for correct transformations, see [30].

1.1 Overview

Section 2 introduces the syntax, operational semantics and definitions of operational approximation and equivalence for a higher-order functional language.

Section 3 provides the definition and some properties of improvement, and the Improvement Theorem is stated. An alternative form of the Improvement Theorem is also given, based on local recursive definitions (**letrec**).

Section 4 applies the improvement theorem to prove correctness of the deforestation transformation in its original formulation, extended to explicitly account for folding using local recursive definitions. The correctness proof illustrates use of the local variant of the Improvement Theorem.

Section 5 considers generalisations to the deforestation transformation, in particular to include higher-order functions. The generalisations are guided by a reformulation of the deforestation transformation into a stepwise rule-based approach. The stepwise formulation has two advantages. Firstly, it suggests a “natural” generalisation of the transformation to the higher-order case, and secondly, as investigated in the following section, it provides a much simplified proof of correctness³.

³ This does not consider termination aspects of deforestation *algorithms*, although we expect that the stepwise formulation will also be useful here.

Section 6 builds a framework for proving the correctness of recursion-based program transformations. An abstract transformation algorithm is described, based on successive transformations of some recursive definitions, using a memo-table to record expressions previously transformed, and parameterised by a transformation relation. Using the Improvement Theorem, correctness of any transformation is reduced to a local correctness condition on the transformation relation. We use this framework to establish the correctness of the generalised deforestation transform. The proof is robust with respect to the folding strategy, and is modular with respect to the transformation steps.

Section 7 illustrates the robustness of the proof by considering a number of further generalisations, including the “positive supercompilation” rule from [33].

2 Preliminaries

We summarise some of the notation used in specifying the language and its operational semantics. The subject of this study will be an untyped higher-order non-strict functional language with lazy data-constructors. Our technical results will be specific to this language, but related results can be established for call-by-value languages.

2.1 Language

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. Symbols $\mathbf{f}, \mathbf{g}, \mathbf{h} \dots$, range over function names, $f, h, x, y, z \dots$ over variables and $e, e_1, e_2 \dots$ over expressions. The

syntax of expressions is as follows:

$$\begin{array}{l}
e = x \quad | \quad \mathbf{f} \quad | \quad e_1 e_2 \quad (\text{Variable; Function name; Application}) \\
\quad | \quad \lambda x. e \quad (\text{Lambda-abstraction}) \\
\quad | \quad \mathbf{case} \ e \ \mathbf{of} \quad (\text{Case expressions}) \\
\quad \quad \quad \mathit{pat}_1 : e_1 \ \dots \ \mathit{pat}_n : e_n \\
\quad | \quad c(\bar{e}) \quad (\text{Constructor expressions and constants}) \\
\quad | \quad p(\bar{e}) \quad (\text{Strict primitive functions}) \\
\\
\mathit{pat} = c(\bar{x}) \quad (\text{Patterns})
\end{array}$$

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

We can assume that the case expressions are defined for any subset of patterns $\{\mathit{pat}_1 \dots \mathit{pat}_n\}$ such that the constructors of the patterns are distinct. A variable can occur at most one in a given pattern; the number of variables must match the arity of the constructor, and these variables are considered to be bound in the corresponding branch of the case-expression.

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

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

A *context*, ranged over by C, C_1 , etc. is an expression with zero or more “holes”, $[\]$, in the place of some subexpressions; $C[e]$ is the expression produced by replacing the holes with expression e . Contrasting with substitution,

occurrences of free variables in e may become bound in $C[e]$; if $C[e]$ is closed then we say it is a *closing context* (for e).

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

2.2 Operational Semantics, Approximation and Equivalence

The operational semantics is used to define an evaluation relation \Downarrow (a partial function) between closed expressions and the “values” of computations. The set of values, following the standard terminology (see e.g. [25]), 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 $c(\bar{e})$, and the *Closures*, as given by the following grammar:

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

The operational semantics is call-by-name, and \Downarrow is defined in terms of a one-step evaluation relation using the notion of a *reduction context* [6]. If $e \Downarrow w$ for some closed expression e then we say that e *evaluates to* w . We say that e *converges*, and sometimes write $e \Downarrow$ if there exists a w such that $e \Downarrow w$. Otherwise we say that e *diverges*. We make no finer distinctions between divergent expressions, so that run-time errors and infinite loops are identified.

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

Definition 1 A *reduction context* \mathcal{R} is given inductively by the following grammar

$$\mathcal{R} = [] \quad | \quad \mathcal{R} e \quad | \quad \mathbf{case} \ \mathcal{R} \ \mathbf{of} \ pat_1 : e_1 \dots pat_n : e_n \quad | \quad p(\bar{c}, \mathcal{R}, \bar{e})$$

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

Now we define the one step reduction relation. We assume that each primitive function p is given meaning by a partial function $\llbracket p \rrbracket$ 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 constants before the result of

$$\begin{array}{l}
\mathbb{R}[\mathbf{f} \ e_1 \dots e_{\alpha_{\mathbf{f}}}] \mapsto \mathbb{R}[e_{\mathbf{f}}\{x_1 \dots x_{\alpha_{\mathbf{f}}} := e_1 \dots e_{\alpha_{\mathbf{f}}}\}] \quad \text{(fun)} \\
\text{(if } \mathbf{f} \text{ is defined by } \mathbf{f} \ x_1 \dots x_{\alpha_{\mathbf{f}}} \triangleq e_{\mathbf{f}} \text{)} \\
\mathbb{R}[(\lambda x.e) \ e'] \mapsto \mathbb{R}[e\{x := e'\}] \quad (\beta) \\
\mathbb{R}[\text{case } c_i(\bar{e}) \text{ of } \dots c_i(\bar{x}_i) : e_i \dots] \mapsto \mathbb{R}[e_i\{\bar{x}_i := \bar{e}\}] \quad \text{(case)} \\
\mathbb{R}[p(\bar{c})] \mapsto \mathbb{R}[c'] \quad \text{(prim)} \\
\text{(if } \llbracket p \rrbracket \bar{c} = c' \text{)}
\end{array}$$

Fig. 1. One-step reduction rules

an application, if any, can be returned— and are only defined over constants, not over arbitrary weak head normal forms.

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

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

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

Using this notion of convergence we now define the standard notions of operational approximation and equivalence. The operational approximation we use is the standard Morris-style contextual ordering, or *observational approximation* see e.g. [26]. The notion of “observation” we take is just the fact of convergence, as in the lazy lambda calculus [1]. Operational equivalence equates two expressions if and only if in all closing contexts they give rise to the same observation – i.e. either they both converge, or they both diverge.

Definition 4 (i) e operationally approximates e' , $e \sqsubseteq e'$, if for all contexts C such that $C[e], C[e']$ are closed, if $C[e] \Downarrow$ then $C[e'] \Downarrow$.
(ii) e is operationally equivalent to e' , $e \cong e'$, if $e \sqsubseteq e'$ and $e' \sqsubseteq e$.

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

3 Improvement

In this section we outline the main technical result from [29], which says that if transformation steps are guided by certain natural optimisation concerns, then correctness of the transformation follows.

There are two main differences from the results in [29]. First, we add lambda abstractions to our programming language. This is not a major addition from the point of view of the expressive power of the language, since we already had higher-order functions in the guise of curried functions (partial applications). The difference arises because we treat beta-reduction differently from function-call reduction when we define the notion of improvement. This, in turn, extends the power of the improvement theorem, since we can choose between the different representations of higher-order expressions according to our needs. It also means that we can pick out different sub-languages to suit our particular needs.

Secondly, we introduce a “local” version of the main theorem which is applicable to expression-level recursion using a simple “letrec” term.

Summary We summarise the two main concepts introduced in this section:

- (i) We introduce a formal *improvement-theory*. Roughly speaking, *improvement* is a refinement of operational approximation, which says that an expression e is improved by e' if, in all closing contexts, computation using e' is no less efficient than when using e , measured in terms of the number of function calls made. From the point of view of program transformation, the important property of improvement is that it is substitutive—an expression can be improved by improving a sub-expression. For reasoning about improvement a more tractable formulation of the improvement relation is introduced and some proof techniques related to this formulation are used.
- (ii) The *improvement theorem* says that if e is improved by e' , in addition to e being operationally equivalent to e' , then a transformation which replaces e by e' (potentially introducing recursion) is totally correct; in addition this guarantees that the transformed program is a formal improvement over the original. (Notice that in the example in the introduction, replacement of 42 by the equivalent term $f\ 0$ is not an improvement since the latter requires evaluation of an additional function call).

The Role of Improvement We should once again stress that the purpose of using the improvement relation in the above theorem is that it guarantees that the transformation yields an operationally equivalent term. If we do not

impose the condition that e is improved by e' , but just rely on the fact that they are equivalent, then the transformation will not, in general, be correct. The fact that the theorem also guarantees that the transformed program is an improvement over the original is an added bonus. It can also allow us to apply the theorem iteratively. It also gives us an indication of the limits of the method. Transformations which do not improve a program cannot be justified using the improvement theorem alone. However, in combination with some other more basic methods for establishing correctness, the Improvement Theorem can still be effective. We refer the reader to [30] for examples of other more basic methods and how they can be used together with the Improvement Theorem.

Variations on the Definition of Improvement There are a number of variations that we can make in the definition of improvement. We could, for example, additionally count the number of primitive functions called. Such variations might be used to give additional information about transformations. (See [27] for further examples). However, the fact that we count the number of recursive function calls in the definition of improvement is *essential* to the Improvement Theorem; the Theorem does not hold if we use an improvement metric which does not count these function calls.

3.1 The Theory of Improvement

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

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

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

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

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

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

$$\bullet \quad e \Downarrow^n \stackrel{\text{def}}{=} \exists w. e \Downarrow^n w \quad \bullet \quad e \Downarrow^{n \leq m} \stackrel{\text{def}}{=} e \Downarrow^n \ \& \ n \leq m \quad \bullet \quad e \Downarrow^{\leq m} \stackrel{\text{def}}{=} \exists n. e \Downarrow^n \leq m$$

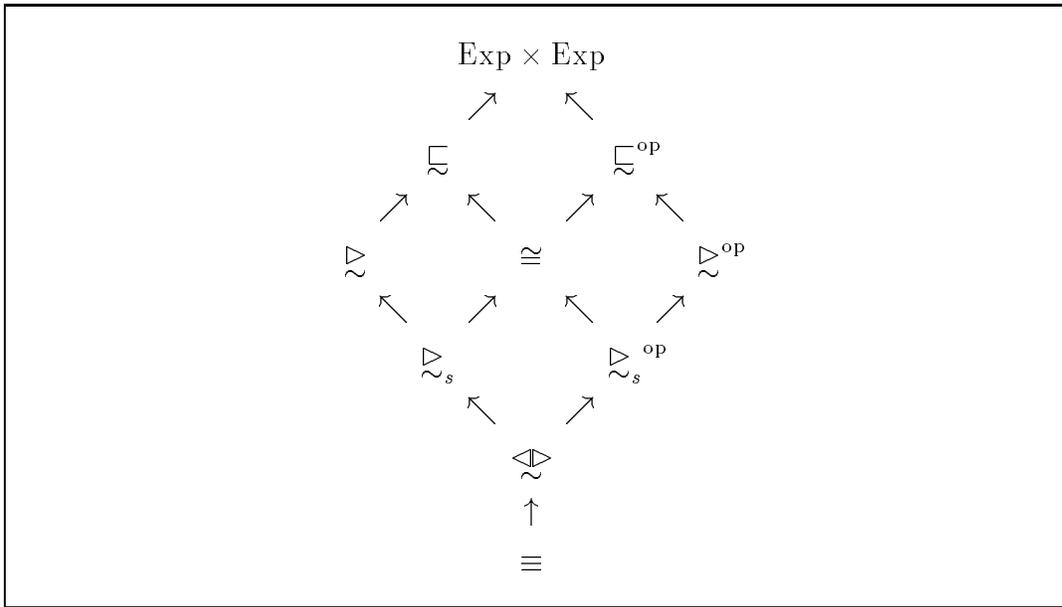


Fig. 2. A \sqcap -semi-sub-lattice of preorders

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

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

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

We also add a strong version of improvement which implies (by definition) operational equivalence:

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

The cost equivalence relation, \triangleleft , is defined by: $e \triangleleft e'$ if and only if $e \triangleright e'$ and $e' \triangleright e$.

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

3.2 The Improvement Theorem

We are now able to state the Improvement Theorem. For the purposes of the formal statement, 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 represented by the derivation of a new function $\mathbf{g} x \triangleq e'\{\mathbf{f} := \mathbf{g}\}$. In this way we do not need to explicitly parameterise operational equivalence and improvement by the intended set of function definitions.

In the following (Theorem 8 – Proposition 11) let $\{\mathbf{f}_i\}_{i \in I}$ be a set of functions indexed by some set I , given by some definitions:

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

Let $\{e'_i\}_{i \in I}$ be a set of expressions such that for each $i \in I$, $\text{FV}(e'_i) \subseteq \{x_1 \dots x_{\alpha_i}\}$.

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

$$\{\mathbf{g}_i x_1 \dots x_{\alpha_i} \triangleq e'_i\{\bar{\mathbf{f}} := \bar{\mathbf{g}}\}\}_{i \in I}$$

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

Theorem 8 (Partial Correctness) *If $e_i \cong e'_i$ for all $i \in I$, then $\mathbf{g}_i \sqsubseteq \mathbf{f}_i$, $i \in I$.*

This is the “standard” partial correctness result (see eg. [17][5]) associated with e.g. unfold-fold transformations. It follows easily from a least fixed-point theorem for \sqsubseteq (the full details for this language can be found in [30]) since the $\bar{\mathbf{f}}$ are easily shown to be fixed points of the defining equations for functions $\bar{\mathbf{g}}$.

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

Theorem 9 (The Improvement Theorem [29]) *If we have $e_i \succsim e'_i$ for all $i \in I$, then $\mathbf{f}_i \succsim \mathbf{g}_i$, $i \in I$.*

The proof of the Theorem, given in detail in [30], makes use of the alternative characterisation of the improvement relation given later.

Putting the two theorems together, we get:

Corollary 10 *If we have $e_i \succsim_s e'_i$ for all $i \in I$, then $\mathbf{f}_i \succsim_s \mathbf{g}_i$, $i \in I$.*

Informally, this implies that:

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

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

Proposition 11 *If $e_i \triangleleft_s e'_i$ for all $i \in I$, then $\mathbf{f}_i \triangleleft_s \mathbf{g}_i$, $i \in I$.*

3.3 Proving Improvement

Finding a more tractable characterisation of improvement (than that provided by Def. 6) is essential in establishing improvement laws (and in the proof of the Improvement Theorem itself). The characterisation we use says that two expressions are in the improvement relation if and only if they are contained in a certain kind of *simulation* relation. This is a form of *context lemma* eg. [1,14], and the proof of the characterisation uses previous technical results concerning a more general class of improvement relations [27].

Definition 12 *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:*

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

For a given relation \mathcal{IR} and weak head normal forms w_1 and w_2 we will abbreviate the property “(i) or (ii)” in the above by $w_1 \mathcal{IR}^\dagger w_2$.

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

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

Lemma 13 (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 , $e\sigma \mathcal{IR} e'\sigma$.*

The lemma provides a basic proof technique:

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

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

Proposition 14

$$\begin{aligned} & \mathcal{IR}[\mathbf{case } x \mathbf{ of } pat_1 : e_1 \cdots pat_n : e_n] \\ \simeq & \mathbf{case } x \mathbf{ of } pat_1 : \mathcal{IR}[e_1] \cdots pat_n : \mathcal{IR}[e_n] \end{aligned}$$

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

$$\begin{aligned} & (\mathcal{IR}[\mathbf{case } e_0 \mathbf{ of } c_1(\bar{x}_1) : e_1 \dots c_n(\bar{x}_n) : e_n], \\ & \mathbf{case } e_0 \mathbf{ of } c_1(\bar{x}_1) : \mathcal{IR}[e_1] \dots c_n(\bar{x}_n) : \mathcal{IR}[e_n]) \end{aligned} \tag{1}$$

It is sufficient to show that R is an improvement simulation. Suppose $e R e'$, and suppose further that $e \Downarrow^n w$. We need to show that $e' \Downarrow^{\leq n} w'$ for some w' such that $w R^\dagger w'$. If $e \equiv e'$ then this follows easily. Otherwise e and e' have the form of (1). Now since $\mathcal{IR}[\mathbf{case } [] \mathbf{ of } c_1(\bar{x}_1) : e_1 \dots c_n(\bar{x}_n) : e_n]$ is a reduction context, then we must have

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

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

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

Now the former derivative reduces in one more step to $\mathbb{R}[e_i\{\bar{x}_i := \bar{e}''\}]$, whilst the latter reduces to $\mathbb{R}[e_i]\{\bar{x}_i := \bar{e}''\}$. Since reduction contexts do not bind variables, and since \mathbb{R} must be closed, these are syntactically equivalent, and so we conclude that

$$\text{case } e_0 \text{ of } \mathbb{R}[c_1(\bar{x}_1) : e_1] \dots \mathbb{R}[c_n(\bar{x}_n) : e_n] \Downarrow^n w.$$

The remaining conditions for improvement simulation (recall Def. 12 and the \cdot^\dagger operator) are trivially satisfied, since $w \equiv^\dagger w$, which implies $w R^\dagger w$ as required. \square

We can also use the context lemma to build some simpler tools for proving improvement properties. The following will be adequate for many of the local transformation steps described in the remainder of the paper:

Proposition 15 *If $e_1 \mapsto_m e'_1$ and $e_2 \mapsto_n e'_2$ then*

- (i) $e'_1 \cong e'_2$ if and only if $e_1 \cong e_2$
- (ii) $m \geq n$ and $e'_1 \triangleright e'_2$ implies $e_1 \triangleright e_2$
- (iii) $m \geq n$ and $e'_1 \triangleright_s e'_2$ implies $e_1 \triangleright_s e_2$
- (iv) $m = n$ implies $(e'_1 \triangleleft e'_2 \iff e_1 \triangleleft e_2)$

PROOF. Assume that $e_1 \mapsto_m e'_1$ and $e_2 \mapsto_n e'_2$.

- (i) Follows from the fact that \mapsto is contained in \cong , which we state without proof.
- (ii) Suppose that $m \geq n$ and $e'_1 \triangleright e'_2$. By the context lemma (Lemma 13), it is sufficient to find an improvement simulation containing all closed instances of (e_1, e_2) . Let $R = \triangleright \cup \{(e_1\theta, e_2\theta) \mid e_1\theta, e_2\theta \text{ are closed}\}$; we will show that R is an improvement simulation. Suppose $(e, e') \in R$. Assume that $e \Downarrow^k w_1$. To show that R is an improvement simulation we are required to prove that $e \Downarrow^{\leq k} w_2$ for some w_2 such that $w_1 R^\dagger w_2$. Now by the definition of R , either $e \triangleright e'$, or (e, e') is equal to $(e_1\theta, e_2\theta)$ for some closing substitution θ . In the first case we are done, since \triangleright is itself an improvement simulation. In the second case, since $e_1 \mapsto_m e'_1$ we know that $e'_1 \Downarrow^{k-m} w_1$. Now since $e'_1 \triangleright e'_2$ we have $e'_2 \Downarrow^{\leq k-m} w_2$ for some w_2 such that $w_1 \triangleright^\dagger w_2$. But the fact that $e_2 \mapsto_n e'_2$ implies that $e'_2 \Downarrow^{n'} w_2$ where $n' \leq k - m + n \leq k$. Finally, since $(\triangleright) \subseteq R$ we have that $w_1 R^\dagger w_2$ as required.
- (iii) Follows by combining (i) and (ii).
- (iv) Follows from (ii) and symmetry. \square

\square

3.4 An Improvement Theorem for Local Recursion

In this section we introduce a form of the improvement theorem which deals with local expression-level recursion, expressed with a fixed point combinator or with a simple “letrec” definition. This will be useful for reasoning about transformations which introduce recursion through such a mechanism; the next section provides an example of its application.

Let **fix** be a recursion combinator defined by

$$\mathbf{fix} f \triangleq f(\mathbf{fix} f)$$

The following property relates recursion expressed using **fix**, and recursive definitions:

Proposition 16 *For all expressions e , if $\lambda f.e$ is closed then $\mathbf{fix} \lambda f.e \triangleleft \mathbf{g}$ where \mathbf{g} is a new function defined by $\mathbf{g} \triangleq e\{f := \mathbf{g}\}$.*

PROOF. Define $\mathbf{g}^- \triangleq e\{f := \mathbf{fix} \lambda f.e\}$. Now since $\mathbf{g}^- \xrightarrow{\bullet} e\{f := \mathbf{fix} \lambda f.e\}$ and $\mathbf{fix} \lambda f.e \xrightarrow{\circ} e\{f := \mathbf{fix} \lambda f.e\}$ it follows by Prop. 15(iv) that $\mathbf{g}^- \triangleleft \mathbf{fix} \lambda f.e$. Since cost equivalence is a congruence relation, we have that $e\{f := \mathbf{fix} \lambda f.e\} \triangleleft e\{f := \mathbf{g}^-\}$, and so by Proposition 11, we have a cost-equivalent transformation from \mathbf{g}^- to \mathbf{g} , and hence $\mathbf{g} \triangleleft \mathbf{g}^- \triangleleft \mathbf{fix} \lambda f.e$ \square

Now we can define a **letrec** expression by “translation” using **fix**.

Definition 17 $\mathbf{letrec} h \bar{x} = e \mathbf{in} e' \stackrel{\text{def}}{=} (\lambda h.e')(\mathbf{fix} \lambda h.\lambda \bar{x}.e)$

The following properties are consequences of the definition, which are easily proven:

Proposition 18

- (i) $\mathbf{letrec} h \bar{x} = e \mathbf{in} e' \triangleleft e'\{h := (\mathbf{fix} \lambda h.\lambda \bar{x}.e)\}$
- (ii) $\mathbf{letrec} h \bar{x} = e \mathbf{in} h \triangleleft \mathbf{fix} \lambda h.\lambda \bar{x}.e$
- (iii) $\mathbf{letrec} h \bar{x} = e \mathbf{in} e' \triangleleft e'\{h := \mathbf{letrec} h \bar{x} = e \mathbf{in} h\}$
- (iv) $(\mathbf{letrec} h \bar{x} = e \mathbf{in} e')e'' \triangleleft \mathbf{letrec} h \bar{x} = e \mathbf{in} e'e''$ if $h \notin \text{FV}(e'')$

Now we give a local improvement theorem analogous to Corollary 10:

Theorem 19 *If variables h and \bar{x} include all the free variables of both e_0 and e_1 , then if*

$$\mathbf{letrec} h \bar{x} = e_0 \mathbf{in} e_0 \triangleright_s \mathbf{letrec} h \bar{x} = e_0 \mathbf{in} e_1$$

then for all expressions e

$$\mathbf{letrec} \ h \ \bar{x} = e_0 \ \mathbf{in} \ e \ \succsim_s \ \mathbf{letrec} \ h \ \bar{x} = e_1 \ \mathbf{in} \ e$$

PROOF. Using Prop. 18(iii) we can see that (from the premise of the theorem) it is sufficient to prove $\mathbf{letrec} \ h \ \bar{x} = e_0 \ \mathbf{in} \ h \ \succsim_s \ \mathbf{letrec} \ h \ \bar{x} = e_1 \ \mathbf{in} \ h$; by definition of \mathbf{letrec} we can see that this is equivalent to showing that $\mathbf{fix} \ \lambda h. \lambda \bar{x}. e_0 \ \succsim_s \ \mathbf{fix} \ \lambda h. \lambda \bar{x}. e_1$.

Define a new function $\mathbf{g} \triangleq \lambda \bar{x}. e_0 \{h := \mathbf{g}\}$. By Prop. 16, $\mathbf{g} \triangleleft \mathbf{fix} \ \lambda h. \lambda \bar{x}. e_0$. Now we use this, and the properties listed in Prop. 18 to “transform” the body of \mathbf{g} :

$$\begin{aligned} \lambda \bar{x}. e_0 \{h := \mathbf{g}\} &\triangleleft \lambda \bar{x}. e_0 \{h := \mathbf{fix} \ \lambda h. \lambda \bar{x}. e_0\} \\ &\triangleleft \lambda \bar{x}. \mathbf{letrec} \ h \ \bar{x} = e_0 \ \mathbf{in} \ e_0 \\ &\succsim_s \lambda \bar{x}. \mathbf{letrec} \ h \ \bar{x} = e_0 \ \mathbf{in} \ e_1 \\ &\triangleleft \lambda \bar{x}. e_1 \{h := \mathbf{fix} \ \lambda h. \lambda \bar{x}. e_0\} \\ &\triangleleft \lambda \bar{x}. e_1 \{h := \mathbf{g}\} \end{aligned}$$

So by Prop. 10, $\mathbf{g} \succsim_s \mathbf{g}'$ where $\mathbf{g}' \triangleq e' \{h := \mathbf{g}'\}$. Hence by Prop. 16, $\mathbf{fix} \ \lambda h. \lambda \bar{x}. e_0 \triangleleft \mathbf{g} \succsim_s \mathbf{g}' \triangleleft \mathbf{fix} \ \lambda h. \lambda \bar{x}. e_1$. \square

4 Deforestation

In this section we recall a well-known example of a recursion-based program transformation, namely deforestation, and show how the local version of the Improvement Theorem can be used to furnish a correctness proof.

Deforestation [38] is a transformation developed for first-order lazy functional programs, which aims to eliminate the construction of intermediate data structures (eg. trees, hence the name). The aim of the transformation is the fusion of code which produces some data structure with the code which consumes it. The general aims of the transformation are well known in the transformation literature as a form of loop fusion; deforestation is an attempt to make this transformation fully mechanisable.

In this section we will restrict our attention to a small subset of the language introduced in Section 2. This subset corresponds to the core language introduced in [38]. Let meta-variables e, e' etc. range over the first-order subset of the language built from constructor-expressions, case expressions, and function applications of the form $\mathbf{f} \ e_1 \cdots e_{\alpha f}$, where the body of $\mathbf{f} \ e_1 \cdots e_{\alpha f}$ is

$$\begin{aligned}
(1) \quad T[x] &= x \\
(2) \quad T[c(e_1 \dots e_k)] &= c(T[e_1] \dots T[e_k]) \\
(3) \quad T[\mathbf{f} \bar{e}] &= T[e_f \{\bar{x} := \bar{e}\}] \\
(4) \quad T[\mathbf{case} \ x \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n] \\
&= \mathbf{case} \ x \ \mathbf{of} \ pat'_1 : T[e'_1] \dots pat'_n : T[e'_n] \\
(5) \quad T[\mathbf{case} \ c(\bar{e}) \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n] \\
&= T[e'_i \{\bar{x} := \bar{e}\}] \quad \text{if } pat'_i \equiv c(\bar{x}) \\
(6) \quad T[\mathbf{case} \ (\mathbf{f} \bar{e}) \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n] \\
&= T[\mathbf{case} \ (e_f \{\bar{x} := \bar{e}\}) \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n] \\
(7) \quad T[\mathbf{case} \ (\mathbf{case} \ e_0 \ \mathbf{of} \ pat_1 : e_1 \dots pat_m : e_m) \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n] \\
&= T[\mathbf{case} \ e_0 \ \mathbf{of} \\
&\quad pat_1 : (\mathbf{case} \ e_1 \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n) \\
&\quad \dots \\
&\quad pat_m : (\mathbf{case} \ e_m \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n)]
\end{aligned}$$

Fig. 3. Original transformation rules for Deforestation.

also a first-order expression of this form. We will sometimes abbreviate such a function application as $\mathbf{f} \bar{e}$, and assume that the function \mathbf{f} is defined by $\mathbf{f} \bar{x} \triangleq e_f$.

The heart of the deforestation algorithm is the set of seven rules reproduced in Figure 3.

The transformation rules resemble a recursively defined interpreter. The problem with the bare rules, operationally speaking, is that they will “loop” on most open terms containing calls to recursive functions. Consider (a small example from [38]) the application of the transformation rules to the term

$\mathbf{flip}(\mathbf{flip} x)$ where \mathbf{flip} is defined by

$$\begin{aligned} \mathbf{flip} x &\triangleq \mathbf{case} x \mathbf{ of} \\ &\quad \mathbf{Leaf} z : \mathbf{Leaf} z \\ &\quad \mathbf{Branch}(l, r) : \mathbf{Branch}(\mathbf{flip} r, \mathbf{flip} l) \end{aligned}$$

By application of the rules we can see that

$$\begin{aligned} T[\mathbf{flip}(\mathbf{flip} x)] \\ &= \mathbf{case} x \mathbf{ of} \\ &\quad \mathbf{Leaf} z : \mathbf{Leaf} z \\ &\quad \mathbf{Branch}(l, r) : \mathbf{Branch}(T[\mathbf{flip}(\mathbf{flip} l)], T[\mathbf{flip}(\mathbf{flip} r)]) \end{aligned} \tag{2}$$

Clearly the transformation algorithm will not terminate, since renamings of the initial term are encountered in the branches. To enable well-formed (ie. finite) programs to be produced, the crucial step in the algorithm is to add *folding*. The idea of folding is that when T is applied to a “previously encountered” expression, the transformation is shortcut and recursion is introduced. In order to introduce recursion, new function definitions need to be constructed, and the terms encountered by the transformation must be recorded, or *memoised*. Wadler notes [38]:

When should new definitions be introduced? Any infinite sequence of steps must contain applications of rules (3) and (6), the unfold rules. Therefore, it is sufficient to take as right-hand sides, each term of the form $T[\dots]$ encountered just before applying rules (3) or (6). Keep a list of such terms. Whenever a term is encountered for a second time, create the appropriate function definition and replace each instance of the term by a corresponding call to the function.

In the above example, this results in the following function:

$$\begin{aligned} T[\mathbf{flip}(\mathbf{flip} x)] &= \mathbf{f}_0 x, \quad \text{where} \\ \mathbf{f}_0 x &\triangleq \mathbf{case} x \mathbf{ of} \\ &\quad \mathbf{Leaf} z : \mathbf{Leaf} z \\ &\quad \mathbf{Branch}(l, r) : \mathbf{Branch}(\mathbf{f}_0 l, \mathbf{f}_0 r) \end{aligned}$$

A more declarative view of the process of folding is obtained by viewing T as a mapping from terms to *term-trees* (i.e. possibly infinite expressions). Folding then corresponds to transforming the infinite tree produced by T into a finite graph, by collapsing equivalent nodes. For deforestation, nodes

are considered equivalent if they are equivalence up to variable renaming (so-called “identical folding” in [9]). See for example Ferguson and Wadler’s account of folding in deforestation [7], and related descriptions of folding for “process trees” in [9].

Wadler originally argued that the expression level transformation is obviously correct (since it essentially uses just unfolding, and simplifications which eliminate constructors).

But the property that the local steps are equivalence-preserving, whilst necessary, does not in itself imply the correctness of the resulting programs, because it does not address the memoisation process used to introduce recursion.⁴ What remains to be achieved is to show that the resulting programs are equivalent to the originals – and in particular in the presence of folding. Since folding is so crucial to the deforestation algorithm, and is at the heart of the problem of proving correctness, we will present a modification of the transformation rules which makes folding explicit.

4.1 *Explicit Memoisation and Folding*

Some earlier explicit accounts of folding [7,19] have taken the declarative view mentioned above.⁵ Firstly, the infinite expression-tree produced by $T[\]$ is annotated with expressions; if e annotates some expression-tree t , then t (ignoring annotations which might occur therein) was obtained by applying $T[\]$ to e . Folding is then implemented by walking down the expression tree and introducing recursion whenever an annotation occurs twice on the same branch. At the first occurrence a recursive definition is set up, and at the subsequent occurrences recursive calls are made.

In our scheme, we merge these two phases to yield a simpler account. The combination of these phases has the advantages that it does not need to introduce infinite-terms (cf. [7]), and it is not dependent on lazy-evaluation within the meta-language defining $T[\]$ (cf. [19]). To represent the construction of recursive programs we make use of the *letrec* construct (Def. 17) in the output syntax.

The basic idea is that the transformation $T[\]$ is given a parameter ρ , which contains a record of the terms encountered so far. Only terms of the form of

⁴ There are some other approaches to fusion – which sometimes also go under the name “deforestation”, e.g. [8] but which do not encounter this problem since they do not operate directly on recursive definitions.

⁵ It should be noted that folding is introduced in [7] for the purpose of a termination proof for the algorithm (applied to a certain class of terms); folding is introduced in [19] in a discussion of implementation issues.

rules (3) and (6) are recorded, since these are the only rules that can lead to a nonterminating transformation. This extra parameter is the memo-list, and will be modelled by a substitution. The domain of the substitution is the set of local function names which are in scope, and the range is the set of expressions which have been seen before. If at some point in the transformation we have that $\rho(f) = \lambda x_1. \dots \lambda x_n. e$, where e is a first-order expression, it means that the expression e has been “seen before”, and that the action of the transformer, roughly speaking, was to introduce a call to a new function $f x_1. \dots x_n$.

The transformation rules with explicit folding are given in Figure 4. The parameter ρ is written as a subscript to avoid later confusion with the application of a substitution.

We have combined rules (3) and (6) into a single rule by abstracting over the context in which the function call occurs; E ranges over single-hole contexts of the following form:

$$E ::= [] \mid \mathbf{case} [] \mathbf{of} \textit{pat}_1 : e_1 \dots \textit{pat}_n : e_n .$$

Suppose that e is in the right syntactic form to apply rule (3)/(6). Now if $\rho(h) \equiv \lambda \bar{y}. e$ for some h then we know that a renaming of expression e has been encountered before, and that we can just make a call to h with arguments \bar{y} . Otherwise we construct a local recursive definition, and add a binding to ρ , thus recording the expression encountered and the name of this local definition. Introducing local definitions gives the remainder of the transformation the opportunity to introduce recursive calls to this new function whenever a renaming of the expression occurs in the transformation of the body of the new definition. Of course, if recursion is *not* subsequently introduced, then the letrec is redundant. Such redundant letrecs can easily be eliminated by a simple local transformation (namely unfolding).

There are a couple of extra conditions relating to name-clashes which must be satisfied by renaming the bound variables in a term before applying a rule.

- In rule (4) we assume that the variables $Domain(\rho)$ are distinct from the free variables in $\textit{pat}'_1 \dots \textit{pat}'_n$, so that the variables in the patterns do not capture the variables which are introduced by the folding process.
- In rule (7) we assume that $FV(\textit{pat}_1 \dots \textit{pat}_n)$ is disjoint from $FV(e'_1 \dots e'_n)$ and $Domain(\rho)$.

The deforestation algorithm applied to some open expression e is now com-

$$(1) T[[x]]_\rho = x$$

$$(2) T[[c(e_1 \dots e_k)]]_\rho = c(T[[e_1]]_\rho \dots T[[e_k]]_\rho)$$

$$(3)/(6) T[[E[\mathbf{f} \bar{e}]]]_\rho = h \bar{y}, \quad \text{if } \exists h. \rho(h) \equiv \lambda \bar{y}. E[\mathbf{f} \bar{e}]$$

$$= \mathbf{letrec} \ h \bar{y} = T[[e']]_{\rho'} \ \mathbf{in} \ h \bar{y}, \ \text{otherwise,}$$

where $h \notin (\bar{y} \cup \text{Domain}(\rho))$
and $e' = E[e_{\mathbf{f}}\{\bar{x} := \bar{e}\}]$
and $\rho' = \rho \cup \{h := \lambda \bar{y}. E[\mathbf{f} \bar{e}]\}$

where $\bar{y} = \bar{\text{FV}}(E[\mathbf{f} \bar{e}])$

$$(4) T[[\mathbf{case} \ x \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n \]]_\rho$$

$$= \mathbf{case} \ x \ \mathbf{of} \ pat'_1 : T[[e'_1]]_\rho \dots pat'_n : T[[e'_n]]_\rho$$

$$(5) T[[\mathbf{case} \ c(\bar{e}) \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n \]]_\rho$$

$$= T[[e'_i\{\bar{x} := \bar{e}\}]]_\rho \quad \text{if } pat'_i \equiv c(\bar{x})$$

$$(7) T[[\mathbf{case} \ (\mathbf{case} \ e_0 \ \mathbf{of} \ pat_1 : e_1 \dots pat_m : e_m) \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n \]]_\rho$$

$$= T[[\mathbf{case} \ e_0 \ \mathbf{of}$$

$$\quad pat_1 : (\mathbf{case} \ e_1 \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n)$$

$$\quad \dots$$

$$\quad pat_1 : (\mathbf{case} \ e_m \ \mathbf{of} \ pat'_1 : e'_1 \dots pat'_n : e'_n) \]]_\rho$$

Fig. 4. Deforestation with explicit folding

pletely described by $T[[e]]_\epsilon$, where ϵ is the empty environment.

Example 20

$$T[[\mathbf{flip}(\mathbf{flip} \ x)]]_\epsilon =$$

$$\mathbf{letrec} \ h_0 \ x = (\mathbf{letrec} \ h_1 \ x = \mathbf{case} \ x \ \mathbf{of}$$

$$\quad \mathbf{Leaf} \ z : \mathbf{Leaf} \ z$$

$$\quad \mathbf{Branch}(l, r) : \mathbf{Branch}(h_0 \ l, h_0 \ r) \ \mathbf{in} \ h_1 \ x)$$

$$\mathbf{in} \ h_0 \ x$$

There is one redundant letrec, which can be eliminated by unfolding the call to $h_1 x$, giving:

$$\begin{aligned} & \mathbf{letrec} \ h_0 x = \mathbf{case} \ x \ \mathbf{of} \\ & \qquad \qquad \qquad \mathbf{Leaf} \ z : \mathbf{Leaf} \ z \\ & \qquad \qquad \qquad \mathbf{Branch}(l r) : \mathbf{Branch}(h_0 l, h_0 r) \\ & \mathbf{in} \ h_0 x \end{aligned}$$

4.2 Correctness

There are many interpretations of the term “correctness”, which sometimes include efficiency properties, and termination properties of algorithms. Here we are only interested in proving that the transformation, whenever it terminates, gives an equivalent expression. Henceforth this is what we mean by the term “correctness”.

A need for improvement The correctness of the deforestation algorithm has been asserted many times in the literature. In this section we use the improvement theorem to furnish a proof of correctness. Moreover, we claim that this is the first such proof which both explicitly, and correctly, includes the folding process.

The correctness requirement of the transformation is easily stated: we need to show that for all expressions e , if $T[[e]]_\epsilon$ is defined then $T[[e]]_\epsilon \cong e$.

The obvious proof strategy is to use induction on the size of the transformation $T[[e]]_\epsilon$. Clearly we need a more general theorem in order to apply the induction hypothesis in the crucial case (3)/(6) where the environment is extended. So in general we need to prove a property about a transformation of the form $T[[e]]_\rho$. The term $T[[e]]_\rho$ possibly contains some free variables in the domain of ρ , which will have been introduced whenever a term recorded in ρ is encountered. The following is a first attempt at a more suitable generalisation:

For all expressions e , and environments ρ , if the range of ρ contains only closed expressions, its domain is disjoint from the free variables of e , and if $T[[e]]_\rho$ is well-defined, then $e \cong (T[[e]]_\rho)\rho$.

One could now attempt the proof by induction on the size of the transformation $T[[e]]_\rho$. However, this property, although true, is not sufficiently strong to complete the induction. The problem is that preservation of equivalence is not a sufficiently strong property to justify the recursion introduction – specifically in the case when rule (3)/(6) is applied, in the sub-case where the

expression has not been “seen before”. The solution is to prove a stronger property, namely that $T\llbracket e \rrbracket_\rho$ is also an improvement over e , and to apply the improvement theorem in this crucial case.

Theorem 21 *Let e be a closed expression, and ρ an environment such that*

- (i) *the range of ρ contains only closed expressions, and*
- (ii) *$\text{FV}(e) \cap \text{Domain}(\rho) = \emptyset$.*
- (iii) *$T\llbracket e \rrbracket_\rho$ is well-defined (i.e. the transformation terminates).*

Then $e \succ_s (T\llbracket e \rrbracket_\rho)\rho$.

We will need the following technical lemma, which states a certain trade-off between reduction-steps: informally, it says that if e_0 reduces using one **(fun)**-step to e_1 , then an instance of e_0 “costs the same” as an indirect instance of e_1 , where by “indirection” we mean an occurrence of a function call bound to e_1 by a letrec.

Lemma 22 *For all expressions e and substitutions θ such that $h \notin \text{Domain}(\theta)$, if $e_0 \mapsto_1 e_1$ then*

$$\mathbf{letrec} \ h \bar{y} = e_1 \ \mathbf{in} \ e\{z := e_0\theta\} \triangleleft \mathbf{letrec} \ h \bar{y} = e_1 \ \mathbf{in} \ e\{z := h(\bar{y})\theta\}$$

PROOF. Straightforward from the definition of letrec, by application of Proposition 15(iv) by showing that both expressions have a common \mapsto_1 derivative. \square

Proof of Theorem 21

We will focus only on the case where rule (3)/(6) is applied. The remaining cases are straightforward and require just simple arguments about equivalence and improvement.

In what follows, let $e_0 = E[\mathbf{f} \bar{e}]$, where \mathbf{f} is defined by $\mathbf{f} \bar{x} \triangleq e_{\mathbf{f}}$, $e_1 = E[e_{\mathbf{f}}\{\bar{x} := \bar{e}\}]$, and $\bar{y} = \text{FV}(e_0)$.

Under the conditions (i)–(iii) of the theorem, we are required to show that $e_0 \succ_s (T\llbracket e_0 \rrbracket_\rho)\rho$. According to rule (3)/(6), there are two sub-cases. We consider each case in turn:

- (i) Suppose $\exists h. \rho(h) \equiv \lambda \bar{y}. e_0$, and hence that $T\llbracket e_0 \rrbracket_\rho = h \bar{y}$.

The conditions of the proposition ensure that $\bar{y} \cap \text{Domain}(\rho) = \emptyset$, and so we have that $(T\llbracket e_0 \rrbracket_\rho)\rho = (h \bar{y})\rho = (\lambda \bar{y}. e_0)\bar{y}$. But $(\lambda \bar{y}. e_0)\bar{y}$ is cost equivalent to

e_0 , and since cost-equivalence implies strong improvement we can conclude that $e_0 \succ_s (T[[e_0]]_\rho)\rho$.

(ii) Otherwise we have that $T[[e_0]]_\rho = (\mathbf{letrec} \ h \bar{y} = T[[e_1]]_{\rho'} \ \mathbf{in} \ h \bar{y})\rho$ where $\rho' = \rho \cup \{h := \lambda \bar{y}. e_0\}$ and $h \notin (\bar{y} \cup \text{Domain}(\rho))$. We need to show that

$$e_0 \succ_s (\mathbf{letrec} \ h \bar{y} = T[[e_1]]_{\rho'} \ \mathbf{in} \ h \bar{y})\rho \quad (3)$$

Since $h, \bar{y} \notin \text{Domain}(\rho)$ we have that $(\mathbf{letrec} \ h \bar{y} = T[[e_1]]_{\rho'} \ \mathbf{in} \ h \bar{y})\rho \equiv \mathbf{letrec} \ h \bar{y} = (T[[e_1]]_{\rho'})\rho \ \mathbf{in} \ h \bar{y}$. Since E is a reduction context, it follows that $e_0 \dot{\mapsto} e_1$. By Lemma 22 we have that $\mathbf{letrec} \ h \bar{y} = e_1 \ \mathbf{in} \ e_0 \triangleleft \mathbf{letrec} \ h \bar{y} = e_1 \ \mathbf{in} \ h \bar{y}$. Since $h \notin \text{FV}(e_0)$ then this simplifies to $e_0 \triangleleft \mathbf{letrec} \ h \bar{y} = e_1 \ \mathbf{in} \ h \bar{y}$. Hence it is necessary and sufficient to prove that

$$\mathbf{letrec} \ h \bar{y} = e_1 \ \mathbf{in} \ h \bar{y} \succ_s \mathbf{letrec} \ h \bar{y} = (T[[e_1]]_{\rho'})\rho \ \mathbf{in} \ h \bar{y}.$$

The key step⁶ is that by the letrec-form of Improvement Theorem (19) it is then sufficient to show that

$$\mathbf{letrec} \ h \bar{y} = e_1 \ \mathbf{in} \ e_1 \succ_s \mathbf{letrec} \ h \bar{y} = e_1 \ \mathbf{in} \ (T[[e_1]]_{\rho'})\rho$$

Claim 23 $(T[[e_1]]_{\rho'})\rho' \triangleleft \mathbf{letrec} \ h \bar{y} = e_1 \ \mathbf{in} \ (T[[e_1]]_{\rho'})\rho$

From this claim, and the fact that $\mathbf{letrec} \ h \bar{y} = e_1 \ \mathbf{in} \ e_1 \triangleleft e_1$, this is equivalent to showing that

$$e_1 \succ_s (T[[e_1]]_{\rho'})\rho'$$

– which follows from the induction hypothesis, since $T[[e_1]]_{\rho'}$ is a shorter transformation.

It just remains to prove the Claim; By inspection of the rules for $T[[_]]$, all free occurrences of h in $T[[e_1]]_{\rho'}$ must occur in sub-expressions of the form $h \bar{y} \bar{x}$. Suppose there are k such occurrences, which we can write as $h \bar{y} \theta_1 \dots h \bar{y} \theta_k$, where the θ_i are just renamings of the variables \bar{y} . So $T[[e_1]]_{\rho'}$ can be written as $e'\{z_1 \dots z_k := h \bar{y} \theta_1 \dots h \bar{y} \theta_k\}$, where e' contains no free occurrences of h . Then

⁶ This is the *only* step of the proof which depends on the strict-improvement property. All other steps could be made with operational equivalence relation in place of strict improvement, but this step cannot be justified by operational equivalence alone.

we reason as follows (where application of substitutions associates to the left):

$$\begin{aligned}
(T\llbracket e_1 \rrbracket_{\rho'})\rho' &\equiv (T\llbracket e_1 \rrbracket_{\rho'})\rho\{h := \lambda\bar{y}.e_0\} \\
&\rightsquigarrow (e'\{z_1 \dots z_k := h\bar{y}\theta_1 \dots h\bar{y}\theta_k\})\rho\{h := \lambda\bar{y}.e_0\} \\
&\rightsquigarrow (e'\{z_1 \dots z_k := e_0\theta_1 \dots e_0\theta_k\})\rho \\
&\rightsquigarrow \text{(by Lemma 22)} \\
&\quad \mathbf{letrec } h\bar{y} = e_1 \mathbf{ in } (e'\{z_1 \dots z_k := h\bar{y}\theta_1 \dots h\bar{y}\theta_k\})\rho \\
&\equiv \mathbf{letrec } h\bar{y} = e_1 \mathbf{ in } (T\llbracket e_1 \rrbracket_{\rho'})\rho
\end{aligned}$$

5 Generalisations: Stepwise Transformation and Higher Order Functions

In the previous section we considered Wadler’s original formulation of deforestation and showed that correctness could be argued using a local version of the improvement theorem. In this section we will consider a number of generalisations. The key generalisations come from a new “stepwise” formulation of the deforestation transformation. The stepwise formulation expresses the transformation in terms of a one-step rewriting relation on programs, based on a novel strategy for describing the transformation process. This involves identifying the following components of the transformation:

- *reduction contexts*, as in our standard operational semantics (and as implicit in some other formulations of deforestation [7]),
- *passive contexts* which enable transformations to be pushed deeper into a term, and
- *basic rewrites* which mimic those of ordinary evaluation, plus rules which also perform *driving* [35].

This strategy yields a very simple and uniform extension of the transformation to richer languages, including higher-order functions, since extensions to the language (and its operational semantics) can be expressed in terms of additions to reduction contexts, passive contexts and basic rewrites.

More importantly, the stepwise formulation has a simpler, more modular correctness proof. The correctness proof is addressed in the next section. By making use of global recursive definitions and the corresponding Improvement Theorem, correctness reduces to showing that each transformation step is in the strict improvement relation. The number and order of transformation steps does not affect correctness.

5.1 Stepwise Deforestation

By inspection of the basic deforestation rules of Figure 3, we can classify them into basic classes:

Passive rules (1), (2), and (4), which just drive the transformation deeper into a term

Reduction rules (3), (5), and (6), which mimic the actions of the operational semantics, and

Nested-case rule (7) which allows propagation of context (specifically, a case expression) into the branches of a case-expression.

Based on this analysis, we will break down the transformation $T[\llbracket _ \rrbracket]$ into the repeated application of a certain reduction relation. To mimic the effect of the passive rules, we define the *passive contexts* as the contexts in which transformation steps are permitted.

Definition 24 (First-order Passive Contexts) *The first-order passive contexts, ranged over by \mathbb{P} , are single-holed contexts given by*

$$\mathbb{P} = [] \quad | \quad \mathbf{case} \ v \ \mathbf{of} \ \dots \ \mathit{pat}_i : \mathbb{P} \ \dots \quad | \quad c(\dots \mathbb{P} \ \dots)$$

A key property is that $T[\llbracket _ \rrbracket]$ is compositional with respect to passive contexts. In other words:

Lemma 25

$$\begin{aligned} T[\llbracket \mathbb{P}[e] \rrbracket] &\equiv e' \iff \\ \exists e_0, e_1. T[\llbracket \mathbb{P}[x] \rrbracket] = e_0 \ (x \ \mathit{fresh}) \ \mathit{and} \ T[\llbracket e \rrbracket] = e_1 \ \mathit{and} \ e_0\{x := e_1\} &\equiv e' \end{aligned}$$

Recall the class of contexts $E ::= [] \quad | \quad \mathbf{case} \ [] \ \mathbf{of} \ \dots$ which was used to combine rules (3) and (6) into one rule-schema (Fig. 4). (An analogous simplification is present in [7,3]).

In Figure 5 we present the stepwise version of the deforestation rules. Note that the inference rule (s0) could, alternatively, be eliminated by replacing expressions of the form $E[e]$ by $\mathbb{P}[E[e]]$ in the other three rules.

In rule (s3), as before, we assume that the variables of $\mathit{pat}_1 \dots \mathit{pat}_n$ are made distinct from the free variables of E . Let \rightsquigarrow^* denote the transitive and reflexive closure of the \rightsquigarrow relation, and write $e \not\rightsquigarrow$ if there is no e' such that $e \rightsquigarrow e'$. The correspondence between the one-step rules and the original rules can now be stated:

Proposition 26 (i) *If $T[\llbracket e \rrbracket] = e'$ then $e \rightsquigarrow^* e' \not\rightsquigarrow$.*

$$\begin{aligned}
(s0) \quad & \frac{e \rightsquigarrow e'}{IP[e] \rightsquigarrow IP[e']} \\
(s1) \quad & E[\mathbf{f} \bar{e}] \rightsquigarrow E[\mathbf{e}_f \{\bar{x} := \bar{e}\}] \\
(s2) \quad & E[\mathbf{case} \ c_i(\bar{e}) \ \mathbf{of} \ \dots c_i(\bar{x}_i) : e_i \ \dots] \\
& \rightsquigarrow E[e_i \{\bar{x}_i := \bar{e}\}] \\
(s3) \quad & E[\mathbf{case} \ e_0 \ \mathbf{of} \ pat_1 : e_1 \ \dots \ pat_n : e_n] \\
& \rightsquigarrow \mathbf{case} \ e_0 \ \mathbf{of} \ pat_1 : E[e_1] \ \dots \ pat_n : E[e_n] \quad E \neq []
\end{aligned}$$

Fig. 5. Stepwise Deforestation Rules

(ii) If $e \rightsquigarrow e_1 \rightsquigarrow \dots \rightsquigarrow e_i \rightsquigarrow \dots$, then $T[e]$ is undefined.

The proof, with the help of the above Lemma, is left as an exercise. Note that, if we imposed a simple type discipline on the language to ensure that we do not get badly-formed case-expressions where the constructor-expression does not match any of the patterns, then the converse of these properties would also hold.

5.2 Folding with stepwise deforestation

We have established a tight correspondence between the deforestation rules and the stepwise formulation. We now consider what happens when we add memoisation and folding. For the stepwise formulation it does not make sense to add local recursion in the manner of Figure 4. Instead we use global recursion, by allowing the transformation steps to introduce new top-level definitions. This approach is already common in describing deforestation, see e.g. [13] [4].

First we modify rule (s1) to introduce a call to a new function:

$$(s1') \quad IR[\mathbf{f} \bar{e}] \rightsquigarrow \mathbf{f}^\diamond \bar{y}$$

where $\bar{y} = F\bar{V}(IR[\mathbf{f} \bar{e}])$

and \mathbf{f}^\diamond is a new function given by

$$\mathbf{f}^\diamond \bar{y} \triangleq IR[e_f \{\bar{x} := \bar{e}\}]$$

Now we can add a “memo table” as before: we associate the function call $\mathbf{f}^\diamond \bar{y}$ with the expression $IR[\mathbf{f} \bar{e}]$. Whenever a renaming of $IR[\mathbf{f} \bar{e}]$ is encountered at

a later step in the transformation, the corresponding renaming of $\mathbf{f}^\circ \bar{y}$ can be introduced.

Definition 27 (Stepwise Deforestation Algorithm) *Using memoisation as described above, the stepwise deforestation algorithm applied to an expression e_0 is defined as follows:*

First abstract the free variables from e_0 to form a new (non-recursive) definition $\mathbf{f}_0^\circ \bar{x} \triangleq e_0$.

Maintaining a distinction between the original functions in the program (ranged over by $\mathbf{f}, \mathbf{g} \dots$), and the new functions introduced by the transformation steps (henceforth ranged over by $\mathbf{f}^\circ, \mathbf{g}^\circ \dots$) including \mathbf{f}_0° , transform the right-hand sides of the new functions by repeated (nondeterministic) application of the rules but never applying rule (s1') in order to unfold a new function.

We now conjecture that whenever the original deforestation algorithm terminates, then so will the above algorithm (assuming that the rules are applied exhaustively). We might then be tempted to conjecture that the outcomes of the two versions of the algorithm will be syntactically the same. However, the correspondence is not that tight. The stepwise view of deforestation is in fact more general (it is nondeterministic). The generality comes from the fact that the memo-table which records the expressions seen before in the application of rule (s3) is now *global*.

To explain the difference, consider the deforestation rule for constructors:

$$T[[c(e_1, \dots, e_k)]]_\phi = c(T[[e_1]]_\phi, \dots, T[[e_k]]_\phi)$$

The expressions e_1, \dots, e_k are transformed independently, each with their own copy of the memo-list ϕ . This means that if, in the transformation of two of the sub-expressions e_i and e_j , some common sub-expressions arise, then transformation work will be duplicated. In the stepwise account, each argument of the constructor occurs in a passive context, so the transformation rules can be applied to any of these expressions. But in this case, any new function introduced in the transformation of one sub-expression e_i can be subsequently used in the transformation of some other sub-expression e_j .

The practical consequences of this difference is that the stepwise transformation has the ability to terminate more quickly, and produce more compact definitions. An illustrative example, too involved to reproduce here, can be found in [10][§6 (cf. footnote 4)].

The “theoretical” difference is that the stepwise deforestation algorithm is not deterministic. We do not consider this to be a problem of the formulation, since we will show that *any* outcome of the transformation is correct. In other

words, the choice of “transformation order” is orthogonal to the correctness issue. What is more, the correctness argument does not depend on the use of the memo-list, so it is possible to allow the transformation sometimes to apply rule (s1’) blindly, without bothering to check if the expression has been encountered before. Of course, in general this might have adverse effects on the termination of the algorithm, but not on the correctness of the outcome.

5.3 Higher-Order Deforestation

Let us add function names \mathbf{f} , and general application $e_1 e_2$ back into the language, so that now we can have partially applied functions, and thus full higher-order capabilities.

From the formulation we have given, our strategy for generalisation of the deforestation rules to this language is fairly straightforward. Before we can proceed, we make a small modification to the original algorithm. First, consider the contexts E , representing either the trivial context $[\]$, or a case-context (**case** $[\]$ **of** \dots). A small generalisation which is commonly (but implicitly) used in the description of the standard deforestation algorithm (eg. [7,4]) is to allow the contexts E to be nested, i.e.,

$$E ::= [\] \mid \mathbf{case} E \mathbf{of} pat_1 : e_1 \dots pat_n : e_n .$$

With this generalisation, the contexts E are now just the *reduction contexts* for this first-order language (an observation first made explicit in [31]).

The first step in our strategy for generalising to the higher-order case is to use the reduction contexts for this language, i.e.:

$$\mathbb{R} = [\] \mid \mathbf{case} \mathbb{R} \mathbf{of} c_1(\bar{x}_1) : e_1 \dots c_n(\bar{x}_n) : e_n \mid \mathbb{R} e$$

Now we must generalise the passive contexts. We propose the following:

Definition 28 (Higher-Order Passive Contexts) *The (higher-order) passive contexts, ranged over by \mathbb{P} , are single-holed contexts given by*

$$\begin{aligned} \mathbb{P} ::= [\] \mid \mathbf{case} d \mathbf{of} \dots (pat_i : \mathbb{P}) \dots \\ \mid c(\dots \mathbb{P} \dots) \mid d \mathbb{P} e_1 \dots e_k \end{aligned}$$

where d ranges over the simple dynamic expressions, given by $d ::= x \mid d e$.

Note that the passive case-expression is generalised using a simple grammar of dynamic expressions. The rationale of the dynamic expressions is that they

$$\begin{aligned}
(d0) \quad & \frac{e \rightsquigarrow e'}{\mathbb{P}[e] \rightsquigarrow \mathbb{P}[e']} \\
(d1) \quad & \mathbb{R}[\mathbf{f} \ e_1 \dots e_{\alpha_{\mathbf{f}}}] \rightsquigarrow \mathbf{f}^\diamond \bar{y} \\
& \text{where } \bar{y} = \text{FV}(\mathbb{R}[\mathbf{f} \ e_1 \dots e_{\alpha_{\mathbf{f}}}]) \\
& \text{and } \mathbf{f}^\diamond \bar{y} \triangleq \mathbb{R}[e_{\mathbf{f}}\{x_1 \dots x_{\alpha_{\mathbf{f}}}\} := e_1 \dots e_{\alpha_{\mathbf{f}}}] \\
(d2) \quad & \mathbb{R}[\mathbf{case} \ c_i(\bar{e}) \ \mathbf{of} \ \dots c_i(\bar{x}_i) : e_i \dots] \\
& \rightsquigarrow \mathbb{R}[e_i\{\bar{x}_i := \bar{e}\}] \\
(d3) \quad & \mathbb{R}[\mathbf{case} \ d \ \mathbf{of} \ pat_1 : e_1 \dots pat_n : e_n] \\
& \rightsquigarrow \mathbf{case} \ d \ \mathbf{of} \ pat_1 : \mathbb{R}[e_1] \dots pat_n : \mathbb{R}[e_n] \\
(d4) \quad & \mathbf{f} \ e_1 \dots e_{\alpha_{\mathbf{f}}-k} \rightsquigarrow \mathbf{f}^\diamond \bar{y} \quad (k > 0) \\
& \text{where } \bar{y} = \text{FV}(\mathbf{f} \ e_1 \dots e_{\alpha_{\mathbf{f}}-k}) \\
& \text{and } \mathbf{f}^\diamond \bar{y} \ z_1 \dots z_k \triangleq e_{\mathbf{f}}\{x_1 \dots x_{\alpha_{\mathbf{f}}}\} := e_1 \dots e_{\alpha_{\mathbf{f}}-k} \ z_1 \dots z_k, \quad z_1 \dots z_k \text{fresh}
\end{aligned}$$

Fig. 6. Stepwise Higher-Order Deforestation Rules

represent terms from which no information can be extracted. Two examples of passive contexts are:

$$\begin{array}{ll}
(x \ (\mathbf{f} \ y) \ \mathbf{cons}([], \mathbf{nil})) & \mathbf{case} \ x \ y \ \mathbf{of} \\
& \mathbf{nil} : \mathbf{nil} \\
& \mathbf{cons}(y, ys) : []
\end{array}$$

It turns out that in the correctness proof of the transformation, the details of the definition of passive contexts will play no part. In other words, correctness follows for *any* definition of the context \mathbb{P} . However, from the point of view of deforestation-like transformations, the present definition is more interesting: we are therefore treading a line between generality — from the point of view of the correctness argument — and practical relevance — from the point of view of the effects of the transformation.

The rules generalising the stepwise deforestation rules (Fig. 5) are given in Figure 6.

Partial application Rule (d4) has no analog in the first-order version, and requires some explanation. This rule deals with the case of a partially applied

function. Firstly, note that the rule contains no reduction context. This is because a partial application $\mathbf{f} e_1 \dots e_{\alpha_f - k}$ only makes sense in one kind of non-trivial reduction context, namely an application context $[] e'$. But in this case we are interested in handling the term $\mathbf{f} e_1 \dots e_{\alpha_f - k} e'$ (which may or may not be a partial application).

The motivation for the rule is that we want to produce a specialised version of the function $\mathbf{f} e_1 \dots e_{\alpha_f - k}$ which takes advantage of the specific arguments $e_1 \dots e_{\alpha_f - k}$. However, the function does not have sufficiently many arguments to unfold the call as it stands (as in rule (d1)), so we enable the unfold by the construction of an auxiliary function.

The effect of rule (d4) can be understood in terms of the corresponding lambda-expressions. If we included lambda-expressions, we would have passive contexts of the form $\lambda x.P$. The partial application $\mathbf{f} e_1 \dots e_{\alpha_f - k}$ would correspond to $\lambda z_1 \dots \lambda z_k. \mathbf{f} e_1 \dots e_{\alpha_f - k} z_1 \dots z_k$, and then it is easy to see that the effect of the transformation would be the same.

Folding in Higher-Order Deforestation Just as before, in order to get the above algorithm to terminate in some non trivial cases we need to add folding, or memoisation⁷. Folding is needed to produce useful results; if the rules are applied exhaustively then without folding the algorithm will hardly ever terminate. Both rules (d1) and (d4) introduce new function definitions (without these rules termination would be assured, but the effects of the transformation would be uninteresting). The basic idea is the same as before: to use a memo-table, which is accumulated during the transformation, to enable (d1) and (d4) to make use of previously defined functions.

When there is a possibility of applying the rule (d1) to an expression e_1 then we look into the memo-list. If there is an entry $\langle e_0, \mathbf{f}^\circ \bar{y} \rangle$ such that $e_1 \equiv e_0 \theta$, where θ is a *renaming* (a substitution mapping free variables to variables) then we replace e_1 by $\mathbf{f}^\circ \bar{y} \theta$. Otherwise we apply the rule as normal, introducing a new function call $\mathbf{f}^\circ \bar{z}$, where $\bar{z} = \text{FV}(e_1)$ and add the pair $\langle e_1, \mathbf{f}^\circ \bar{z} \rangle$ to the memo-table. We can use memoisation in rule (d4) in exactly the same way.

Example 29 The following example illustrates the transformation rules in action. Consider the definitions given in Figure 7.

Writing **compose** in the usual infix style ($e \circ e' \stackrel{\text{def}}{=} \mathbf{compose} e e'$) we wish to transform the expression $(\mathbf{map} f) \circ (\mathbf{filter} p)$. We begin with the new definition:

$$\mathbf{f}_0^\circ f p \triangleq (\mathbf{map} f) \circ (\mathbf{filter} p)$$

⁷ Although with this stepwise formulation we can simply stop the transformation at any point and we have a well-formed program.

```

filter  $p$   $xs$   $\triangleq$  case  $xs$  of
     $nil$  : nil
     $cons(y, ys)$  : case  $p$   $y$  of
         $true$  : cons( $y$ , filter  $p$   $ys$ )
         $false$  : filter  $p$   $ys$ 

map  $f$   $xs$   $\triangleq$  case  $xs$  of
     $nil$  : nil
     $cons(z, zs)$  : cons(( $f$   $z$ ), map  $f$   $zs$ )

compose  $f$   $g$   $x$   $\triangleq$   $f$  ( $g$   $x$ )

```

Fig. 7. Example Definitions

Now we transform the right-hand side of this new definition and of the right-hand sides of subsequently introduced definitions. The initial transformation steps are given in Fig. 8; each derivation step (\rightsquigarrow) refers to the right-hand side of the preceding definition. We have labelled the steps according to the rule applied, but we have elided the use of the (s0) inference rule.

After these steps the transformation can proceed to the two occurrences of the sub-term **filter** p (**map** f zs) (both of which occur in passive contexts) — but these expressions (modulo renaming) have been encountered above at the first application of rule (*d1*) (and therefore would occur in the memo-table), so we “fold”, introducing recursive calls to \mathbf{f}_2^\diamond , obtaining:

```

 $\mathbf{f}_1^\diamond$   $p$   $f$   $xs$   $\triangleq$   $\mathbf{f}_2^\diamond$   $p$   $f$   $xs$ 
 $\mathbf{f}_2^\diamond$   $p$   $f$   $xs$   $\triangleq$   $\mathbf{f}_3^\diamond$   $p$   $f$   $xs$ 
 $\mathbf{f}_3^\diamond$   $p$   $f$   $xs$   $\triangleq$  case  $xs$  of
     $nil$  : nil
     $cons(z, zs)$  : case  $p$  ( $f$   $z$ ) of
         $true$  : cons(( $f$   $z$ ),  $\mathbf{f}_2^\diamond$   $p$   $f$   $zs$ )
         $false$  :  $\mathbf{f}_2^\diamond$   $p$   $f$   $zs$ 

```

We can eliminate the trivial intermediate functions \mathbf{f}_1^\diamond and \mathbf{f}_2^\diamond by *post-unfolding* [15].

With regard to the “higher order” capabilities of the transformation, we suggest that the transformation copes equally well with ordinary recursive defi-

$$(\mathbf{filter} \ p) \circ (\mathbf{map} \ f) \xrightarrow{d4} \mathbf{f}_1^\diamond \ p \ f \quad \text{where}$$

$$\mathbf{f}_1^\diamond \ p \ f \ x s \triangleq \mathbf{filter} \ p \ (\mathbf{map} \ f \ x s)$$

$$\xrightarrow{d1} \mathbf{f}_2^\diamond \ p \ f \ x s \quad \text{where}$$

$$\mathbf{f}_2^\diamond \ p \ f \ x s \triangleq \mathbf{case} \ (\mathbf{map} \ f \ x s) \ \mathbf{of}$$

$$\mathit{nil} : \mathbf{nil}$$

$$\mathit{cons}(y, ys) : \mathbf{case} \ p \ y \ \mathbf{of}$$

$$\mathit{true} : \mathbf{cons}(y, \mathbf{filter} \ p \ ys)$$

$$\mathit{false} : \mathbf{filter} \ p \ ys$$

$$\xrightarrow{d1} \mathbf{f}_3^\diamond \ p \ f \ x s \quad \text{where}$$

$$\mathbf{f}_3^\diamond \ p \ f \ x s \triangleq \mathbf{case} \ (\mathbf{case} \ x s \ \mathbf{of}$$

$$\mathit{nil} : \mathbf{nil}$$

$$\mathit{cons}(z, zs) : \mathbf{cons}((f \ z), \mathbf{map} \ f \ zs)) \ \mathbf{of}$$

$$\mathit{nil} : \mathbf{nil}$$

$$\mathit{cons}(y, ys) : \mathbf{case} \ p \ y \ \mathbf{of}$$

$$\mathit{true} : \mathbf{cons}(y, \mathbf{filter} \ p \ ys)$$

$$\mathit{false} : \mathbf{filter} \ p \ ys$$

$$\xrightarrow{d3 \ d2 \ d2}$$

$$\mathbf{case} \ x s \ \mathbf{of}$$

$$\mathit{nil} : \mathbf{nil}$$

$$\mathit{cons}(z, zs) : \mathbf{case} \ p \ (f \ z) \ \mathbf{of}$$

$$\mathit{true} : \mathbf{cons}((f \ z), \mathbf{filter} \ p \ (\mathbf{map} \ f \ zs))$$

$$\mathit{false} : \mathbf{filter} \ p \ (\mathbf{map} \ f \ zs)$$

Fig. 8. Initial Deforestation Steps

nitions, or definitions expressed with an explicit fixed-point combinator **fix**. What is more, we conjecture that whenever transformation of an expression terminates it will also terminate on an explicit recursive representation using **fix** as the only recursive function. We leave it as an exercise to rework in this style the **flip**(**flip** x) example from Section 4.

The effects of the transformation, using this generalisation, are not substantially different from those previously introduced by Marlow and Wadler [19], but the presentation is more concise; in some sense this extension of the deforestation method to deal with higher-order functions is the canonical one, stemming from the fact that, in addition to the case-reduction context, the language now has an application reduction context (Re), plus an additional set of weak head normal forms—the partially applied functions. More recent higher-order generalisations are due to Hamilton [12] and also to Marlow [20]; notably both Hamilton and Marlow give a grammar of terms (a “treeless form”) which is used to characterise a set of expressions for which their respective algorithms always terminates.

Following [28], this style of stepwise transformation has been adopted by Nielsen and Sørensen [21] in a study of the relationship between deforestation and partial evaluation. They define a class of passive contexts (what they also call “dead contexts”) which makes their analog of \rightsquigarrow deterministic.

6 Correctness of Memoising Transformations

In this section we address the correctness issue for transformations in the style of the generalised deforestation of the previous section. We begin with an abstract definition of a memoising transformation algorithm, which is parameterised by some transformation relation on expressions. This definition captures the essential features of memoising stepwise transformations.

Then we give certain conditions on the transformations relation, with respect to the definitions transformed, which guarantee that any output of the recursion-based transformation algorithm is a strong improvement over (and hence equivalent to) the input.

Finally, we show the correctness of the higher-order stepwise transformation.

Definition 30 *A transformation relation, \rightarrow , is a ternary relation between two expressions, and a set of new definitions. If $(e, e', D) \in \rightarrow$ then we write $e \rightarrow e', D'$. Furthermore, we assume that D' are some new definitions upon which e does not depend.*

Definition 31 *A memoising transformation of a set of definitions D , using a transformation relation \rightarrow is defined as any outcome of the following non-*

deterministic algorithm:

```

M := ∅  \*  M is a binary relation on expressions * \
for any number of iterations do
  \* Choose any function f, and any sub-expression which
  can be viewed as an instance of some expression e0  * \
  choose (f  $\bar{x} \triangleq C[e_0\theta]$ ) from D;
  if (e0, e1) ∈ M then
    D := (D \ {f  $\bar{x} \triangleq C[e_0\theta]$ }) ∪ {f  $\bar{x} \triangleq C[e_1\theta]$ };
  elseif e0 → e1, D' then
    D := (D \ {f  $\bar{x} \triangleq C[e_0\theta]$ }) ∪ {f  $\bar{x} \triangleq C[e_1\theta]$ } ∪ D';
    M := M ∪ {(e0, e1)};
  end if;
end for;
return D;

```

Notice that the algorithm permits us to add new definitions to the set D , and the memo-table M allows us to shortcut a transformation step that we have done before, rather than introduce further new definitions.

Definition 32 (Independence) *The call-graph of a function \mathbf{f} is inductively defined as the the set of function names occuring in the body of \mathbf{f} , together with the call-graphs of these functions.*

Let D be a subset of the function definitions in a given program. An expression e is independent of D , the function names contained in e and their respective call graphs are disjoint from the functions in D .

A transformation relation \rightarrow is defined to be domain independent of D if whenever $e_0 \rightarrow e_1, D'$ then the following conditions hold:

There exists a substitution θ , and expressions e'_0, e'_1 such that

- (i) $e_0 \equiv e'_0\theta, e_1 \equiv e'_1\theta,$
- (ii) $e'_0 \rightarrow e'_1, D',$ and
- (iii) e'_0 is independent of D .

So, for example, if some transformation relation \rightarrow is domain independent of $D = \{\mathbf{g} \triangleq e_{\mathbf{g}}\}$, then we could not have $\mathbf{g} \rightarrow e, \emptyset$ unless we also had $x \rightarrow e', \emptyset$ for some x and e' such that e is an instance of $e'\{x := \mathbf{g}\}$.

Theorem 33 *Let D' denote any result of applying the memoising transformation algorithm to definitions D , using some relation \rightarrow .*

- (i) *If $e \rightarrow e', D''$ implies $e \overset{\Delta}{\rightsquigarrow} e'$ with respect to the definitions $D \cup D''$ then the definitions in D are cost-equivalent to the corresponding definitions*

in D' .

- (ii) If for all $e, e \rightarrow e', D''$ implies $e \succsim_s e'$ with respect to the definitions in $D \cup D''$, and \rightarrow is domain independent of D then the definitions in D are strongly improved by the corresponding definitions in D' .

The domain independence condition in part (ii) maintains the invariant property that it is contained in the strict-improvement relation with respect to the current definitions (rather than just with respect to the definitions at the time the respective entries were added). The symmetry of cost equivalence means that we do not need this condition in part (i). In the proof we will distinguish between the definitions at each iteration of the transformation loop.

PROOF. Consider some transformation of $D = \{\mathbf{f}_i \triangleq e_i\}_{i \in I}$ (we assume functions of zero-arity just to simplify the presentation). Suppose that $\{\mathbf{f}_i \triangleq e_{ij}\}$ is the value of D after $j \geq 0$ iterations of the loop. Let $D_j = \{\mathbf{f}_i^j \triangleq e_i^j \{\mathbf{f}_i := \mathbf{f}_i^j\}_{i \in I}\}$, and let M_j be the value of M after j iterations. Now we consider the two parts of the transformation in turn.

- (i) Assume that $e \rightarrow e', D''$ implies $e \triangleleft e'$. Now we establish the following invariant:

$$\mathbf{f}_i^j \triangleleft \mathbf{f}_i \text{ and } M_j \subseteq (\triangleleft)$$

We proceed by induction on j ; we omit the details as they are similar to (but simpler than) the following argument for the second part of the theorem.

- (ii) Assume $e \rightarrow e', D''$ implies $e \succsim_s e'$, and that \rightarrow is domain independent of D . We proceed by induction on j , establishing the following loop invariant:
- (a) $\mathbf{f}_i \succsim_s \mathbf{f}_i^j$, and
 - (b) $\forall (e, e') \in M_j, \exists e'', e''', \phi$.

$$(e''\phi, e'''\phi) \equiv (e, e'), e'' \succsim_s e''' \text{ and } e'' \text{ is independent of } D$$

The base case is immediate. For the induction assume that it holds for $j = k$ for some k . Assume without loss of generality that $\mathbf{f}_0 \triangleq C[e_0\theta]$ is the chosen function (context, subexpression and substitution) from the $f_i \triangleq e_{ik}$. We consider the cases $e_0 \rightarrow e_1, D''$ or $(e_0, e_1) \in M_k$ in turn (and if neither case holds the functions are unchanged and we are done).

Case $e_0 \rightarrow e_1, D''$: then $e_0 \equiv e'_0\phi$ and $e'_1\phi \equiv e_1$, for some e'_0, e'_1 and ϕ such that e'_0 is independent of D , and $e'_0 \rightarrow e'_1, D''$. By the assumption we know that $e'_0 \succsim_s e'_1$, and hence by the first part of the induction hypothesis that $e'_0 \succsim_s e'_1 \{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}$. Let ϕ' denote the result of applying replacement $\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}$ to the range of ϕ . Since \succsim_s is closed under substitution we know that $e'_0\phi' \succsim_s e'_1 \{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}\phi'$. But $e'_1 \{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}\phi'$ can be written as $e'_1\phi \{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}$, and since e'_0 is independent of D we have that

$e'_0\phi' \equiv e'_0\phi\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\} \equiv e_0\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}$. Hence we know that

$$e_0\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\} \succsim_s e_1\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}. \quad (4)$$

Let $C' = C\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}$ (this is well defined since this is a replacement operation, not a proper substitution) and let $\theta' = (\theta\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\})$. By the substitutivity and congruence properties of \succsim_s , we have from (4) that

$$C'[e_0\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}\theta'] \succsim_s C'[e_1\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}\theta']. \quad (5)$$

But $e_i\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}\theta' \equiv e_i\theta\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}$, $i = 1, 2$ and so we can write (5) as $C[e_0\theta]\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\} \succsim_s C[e_1\theta]\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}$. Now by the corollary of the Improvement Theorem (Cor. 10) we know that \mathbf{f}_0^k , given by $\mathbf{f}_0^k \triangleq C[e_0\theta]\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^k\}$, is strictly improved by the function $\mathbf{f}_0^{k+1} \triangleq C[e_1\theta]\{\bar{\mathbf{f}} := \bar{\mathbf{f}}^{k+1}\}$ (the other $\mathbf{f}_i^{k+1}, i \neq 0$ are just renamings of the \mathbf{f}_i^k). Thus we can conclude that $\mathbf{f}_i^k \succsim_s \mathbf{f}_i^{k+1}$. By part (a) of the induction hypothesis and transitivity it follows that $\mathbf{f}_i \succsim_s \mathbf{f}_i^{k+1}$. Now for the second conjunct (b): $M_{k+1} = M_k \cup (e_0, e_1)$, so we need to verify that the property b holds for the new pair (e_0, e_1) , but this follows directly from the fact that \rightarrow is “contained in” strict improvement, and domain-independent of D .

Case $(e_0, e_1) \in M_k$: we apply the induction hypothesis to get e'_0, e'_1 and ϕ such that e'_0 is independent of D , $e'_0\phi \equiv e_0$, $e'_1\phi \equiv e_1$, $e'_0 \succsim_s e'_1$, and we proceed as in the last case.

□

6.1 Correctness of Higher Order Deforestation

We now argue that any sequence of steps of the higher order deforestation algorithm corresponds to a sequence obtainable by an instance of the memoising transformation algorithm of Definition 31: the transformation relation is the stepwise relation \rightsquigarrow , where new definitions are only generated by rule (s1') (and (s4) is we choose to memoise this rule also). Notice that when representing an application of the deforestation rules as an instance of the memoising transformation algorithm, the substitution θ in the algorithm will always correspond to a (possibly trivial) renaming.

To prove correctness using the Improvement Theorem, it will be sufficient to prove that each transformation step is an improvement. The key to this fact, in the case where the memo-table is used, is that each new function call introduced by the transformation comes together with an unfolding step in the body of that function's definition. First we consider the individual transformation steps:

Proposition 34 $e \rightsquigarrow e'$ implies $e \triangleleft e'$.

PROOF. We need to reason by induction on the rules. The only non-axiom is rule (d0), but this case is easy since \simeq is a congruence (note that this would be sound for *any* context). Rules (d1)–(d2) and (d4) are easily established using Prop. 15(iv) by showing that e and e' reduce in the same number of $((\mathbf{fun}))$ -steps to syntactically equivalent (and hence cost-equivalent) expressions. Rule (d3) was proved in Prop. 14. \square

There are generally considered to be three aspects to the correctness of deforestation [31]: (i) termination of the algorithm, (ii) correctness of the resulting program, and (iii) non degradation of efficiency. It is not difficult to construct example programs for which an attempt to apply the transformation rules exhaustively would not terminate, so the effort in point (i) must be, eg., to find some syntactic characterisation of the programs for which the algorithm terminates (such as “treeless form”). This issue is outside the scope of this paper. The improvement theorem deals with aspect (ii) and to some extent (iii); from the previous proposition it is a small step to show that the transformation yields equivalent programs, and these will be, formally, equally efficient (in terms of \succsim) under call-by-name evaluation.

Proposition 35 *The higher order deforestation algorithm yields totally correct programs in that any result of applying the transformation steps (including folding) to an initial definition $\mathbf{f}^\circ_0 \bar{x} \triangleq e_0$ will result in a set of new definitions in which the new version of \mathbf{f}°_0 will be cost-equivalent to the original.*

PROOF. From the previous proposition, the basic steps are all cost-equivalences. Then using Proposition 34 we can apply Theorem 33(i) to show that the result of the transformation of the initial definition is cost-equivalent to, and hence operationally equivalent to, the original. \square

On Efficiency Improvement is defined in terms of a call-by-name execution model. We do not have a corresponding Improvement Theorem for call-by-need. Under a call-by-need implementation the usual restrictions of the transformation seem sufficient to ensure that the result is also a “call-by-need improvement” over the original. These restrictions are that only functions which are *linear* in their arguments should be transformed — see [38], [3]. Alternatively, duplication of sub-expressions (eg. Example 29 (fz) is duplicated in \mathbf{f}_3°) can be avoided by the use of let-bindings, in the obvious way.

We have shown that the resulting programs are cost-equivalent to the originals, so we might ask whether there *is* any optimisation achieved by the algorithm. Many of the new function definitions introduced by the transformation will not be recursively called, and so can easily be in-lined during a post-processing

stage. Of course, it should be remembered that the main purpose of the algorithm is to eliminate the construction of intermediate data structures. This property becomes evident from the syntactic form of the output, and is not the measure upon which our improvement theory is based.⁸

One aspect of efficiency which is not addressed in any way by the improvement theory is the size of the resulting code. Even using linearity to avoid slowdown of more than a constant factor, we can get code explosion due to the “case-case” rule in the original transformation, and in its generalisation (d3). This problem is outside the scope of this paper.

7 Further Variations

In this section we consider a number of systematic extensions and variations of the higher order deforestation rules, for which there is little or no additional work to be done with regard to the correctness proof.

The correctness proof is dependent on the fact that the individual steps (and hence the folding steps) are cost-equivalences, but not on the overall structure of the transformation, or on exactly how the memo-table is utilised. Theorem 33 provides a modular approach to correctness. We can add or replace transformation rules to increase the power of the method, and the only property that needs to be checked is that the new rule is a cost equivalence. However, some rules that we might wish to add are not cost-equivalences. To handle these cases we need to show that the correctness result for higher order deforestation can also be obtained using part (ii) of Theorem 33. All we need is the following:

Proposition 36 *In any application of the higher-order deforestation rules, the stepwise transformation relation \rightsquigarrow is domain independent of the new functions, \mathbf{f}°_0 etc.*

PROOF. Since rules (d1) and (d4) do not apply to new functions, and since the definition of reduction contexts does not depend on function definitions, then this follows easily by inspection of the rules. \square

The consequence of this proposition is that we can add any rule schema of the form $e_1 \rightsquigarrow e_2$, and it will be sufficient to verify that the rule schema is independent of the new definitions, and that $e_1 \succ_s e_2$.

⁸ With good reason: the improvement theorem does not hold for an improvement theory based only on the number of constructor-expressions built.

In the remainder of this section we consider a number of variations of this form, concluding with a look at a rule which generalises the positive information propagation found in Turchin’s supercompiler [35,33].

7.1 Language Extensions

We can systematically extend the transformation rules to cover new constructs. New (functional) constructs will typically be defined by the addition of one or both of: basic rewrites and reduction contexts. It is natural to extend the transformation rules to include the new rewrite rule (and, of course, the new reduction context), and possibly a new class of weak head normal forms. For the transformation rules we also need to add the new passive contexts. As we have mentioned before, from the perspective of correctness it is safe to allow any context, but following our strategy for extending the deforestation rules we are led to some more focussed choices.

Consider, for example, extensions to handle primitive functions as given in the full language of Section 2. Now we can systematically extend the transformation:

- reduction contexts are extended with the clause $p(\bar{c}, \mathbb{R}, \bar{e})$,
- the reduction rules are extended with $\mathbb{R}[p(\bar{c})] \rightsquigarrow \mathbb{R}[c']$ if $p(\bar{c}) \mapsto c'$
- the grammar of dynamic expressions is extended with the production $p(\bar{e}, d, \bar{e}')$; that is to say, if any argument of a primitive function is dynamic, then the whole application can be considered dynamic. This is consistent with the other forms of passive context, since primitive functions need all of their arguments.
- the passive contexts are extended with $p(\bar{e}, \mathbb{P}, \bar{e}')$ where either \bar{e} or \bar{e}' contain a dynamic expression. This is consistent with other passive context since the primitive function can never be reduced, so the transformation can proceed to any of the arguments.

Correctness follows easily since the rewrite rule is independent of the function definitions.

7.2 Generalisation and Control of Termination

The stepwise formulation of the algorithm has an advantage when we come to discuss termination issues. We are at liberty to control (restrict) where the rules are applied or simply to stop the transformation after a certain number of steps, and the result will be a well-formed program. Furthermore, the correctness argument is completely independent of these choices.

Generalisation is a familiar concept in inductive proofs, and has a fairly direct analogy in program transformation (see eg. [2] [35]), where in order to be able to fold one must proceed by transforming a more general function. In the transformation studied here we can model generalisation as follows. Rule (d1) (and also (d4)) abstracts the free variables from a term and introduces a new function which replaces the term. Generalisation is enabled if we allow abstraction of sub-terms other than just the free variables, thereby creating a *more general* new function \mathbf{f}° . There is a corresponding generalisation of the folding process: if we encounter a term of the form e_1 , and our memo-list contains a pair $\langle e, \mathbf{f}^\circ \bar{y} \rangle$ such that $e_1 \equiv e\sigma$ for any substitution σ , then we can replace e_1 by $\mathbf{f}^\circ \bar{y}\sigma$.

Note that the memoising transformation algorithm does not need to be extended to handle these forms of generalisation, since the definition itself allows us to use any instance of a previously encountered expression. Therefore the correctness of these variations is also easily proved from the congruence properties of the improvement relation.

In the original deforestation algorithm an annotation scheme (called “blazing”) was given to ensure termination of the algorithm for a wider class of programs (this was subsequently generalised to ensure termination for *all* first-order programs by e.g. Chin [4] and Hamilton and Jones [13]). These annotations achieve the effect of indicating which sub-expressions should be generalised. Following [38], we can represent this by an extension of the algorithm to handle let-expressions of the form $\mathbf{let} \ x = e \ \mathbf{in} \ e'$. To achieve generalisation we do not add the obvious reduction rule for this expression, but instead we add the passive contexts $\mathbf{let} \ x = IP \ \mathbf{in} \ e'$ and $\mathbf{let} \ x = e \ \mathbf{in} \ IP$. In this way, the two sub-expressions will be transformed independently. Since \mathbf{let} -expressions can not be eliminated, it also makes sense to add a context-propagation rule analogous to (d3): $\mathbb{R}[\mathbf{let} \ x = e \ \mathbf{in} \ e'] \rightsquigarrow \mathbf{let} \ x = e \ \mathbf{in} \ \mathbb{R}[e']$

Of course, too much generalisation can prevent interesting transformation from occurring. Too little, and it can be hard to ensure termination. The question of when and where to generalise is beyond the scope of this paper. See e.g. [36,31,32] [36].

7.3 Adding Laws

In principle, any strong improvement laws $e_0 \rightsquigarrow_s e_1$, can be added, and correctness follows providing that the law is domain independent of the definitions thus transformed.

As a simple example, we can add properties about the list-concatenate function

append such as:

$$\begin{aligned} & \mathbf{append} (\mathbf{append} \ x \ y) \ z \ \succsim_s \ \mathbf{append} \ x (\mathbf{append} \ y \ z) \\ & \mathbf{append} (\mathbf{case} \ e \ \mathbf{of} \ pat_1 : e_1 \cdots pat_n : e_n) \ y \\ & \quad \succsim_s \ \mathbf{case} \ e \ \mathbf{of} \ pat_1 : (\mathbf{append} \ e_1 \ y) \cdots pat_n : (\mathbf{append} \ e_n \ y) \end{aligned}$$

Along these lines we can enable transformations which yield non-linear speed-ups (see for example [37]), in contrast to the transformations considered so far, and the following generalisation.

7.4 *Driving and Positive Supercompilation*

In terms of transformational power (but ignoring termination issues) Turchin’s *driving* techniques, as realised in the *supercompiler* (for: *supervised compilation*) [35] subsume deforestation. This increased power is due to a *dynamic* generalisation strategy (which means that decisions as to when and how to generalise are taken at transformation-time) together with increased information-propagation in the transformation. Propagation of the so-called “positive” information [9] can be easily added to the one-step deforestation rules along the lines of [33]. The basic idea is that when a case-expression has a variable in the test position, as in (**case** y **of** $\dots c_i(\bar{x}_i) : e_i \dots$), then within the i^{th} branch we know that free occurrences of y are equivalent to $c_i(\bar{x}_i)$. The effect of “positive information propagation” is achieved by substituting $c_i(\bar{x}_i)$ for all free occurrences of y in e_i . Thus we have transmitted the “positive” information that y has value $c_i(\bar{x}_i)$ in the i^{th} branch (the corresponding *negative* information is that $y \not\equiv c_j(\bar{x}_j)$, and this information could be used, e.g. to prune redundant branches of case-expression). In a language with conditional expressions, this information propagation is achieved by unification.

The transformation seems trivial (for this language at least), but cannot be achieved in any obvious way by preprocessing the original program, because it is applied to terms generated on the fly by earlier unfolding steps. The effect of this extra power is illustrated in [33], where this addition is sufficient to enable the transformation to automatically specialise a naïve pattern matcher to achieve the effect of automata-construction in the classic Knuth-Morris-Pratt pattern matching algorithm.

We introduce a natural generalisation of this transformation rule by generalising the propagation from the case of a single variable, to propagate information to free occurrences of a simple dynamic expression d (Def. 28). Positive

information propagation is implemented by adding the following rule

Definition 37 Define the following transformation rule (**d5**):

$$\begin{aligned} & \mathbb{R}[\mathbf{case\ } d \mathbf{ of\ } \dots c_i(\bar{x}_i) : e\{z := d\} \dots] \\ \rightsquigarrow & \mathbb{R}[\mathbf{case\ } d \mathbf{ of\ } \dots c_i(\bar{x}_i) : e\{z := c_i(\bar{x}_i)\} \dots] \end{aligned}$$

where we assume: the free variables in d and \bar{x}_i are all distinct; we allow renaming of bound variables in a term, and that there is at least one free occurrence of z in e .

Proposition 38 $e \xrightarrow{d5} e'$ implies $e \succsim_s e'$

PROOF. Straightforward using the fact that $e_1 \Downarrow e_2$ implies $e_1 \succsim_s e_2$, (so in particular, if $d\theta \Downarrow c_i(\bar{e})$ then $d\theta \succsim_s c_i(\bar{e})$) together with congruence properties of improvement. \square

8 Conclusions

The following lists some contributions of this article:

- We have shown that the improvement theorem has practical value in proving the correctness of existing recursion-based transformations, using the example of deforestation; moreover,
- we have provided the first correctness proof for deforestation which correctly includes a treatment of folding (Section 4).
- We have devised a strategy for extending the deforestation algorithm to richer languages, including higher-order functions. This strategy is based on a novel stepwise description of the algorithm (Section 5).
- We have defined a general “memoising transformation algorithm”, parameterised by a transformation relation, and derived conditions for its correctness in terms of the transformation relation (Section 6).
- We have shown how a number of further variations of the stepwise deforestation algorithm can be made, including generalisation and the information propagation found in supercompilation (Section 7).

Topics for further work include an investigation into the applicability of these methods for automatic transformation on call-by-value languages; alternatively, one can attempt to understand call-by-value transformations by translating them to call-by-name [21]. Recursion-based transformations are also relevant in languages with side-effects, such as Lisp, Scheme or Standard ML. It remains to be seen whether the improvement theorem can

be shown to hold for these kinds of languages, and whether it is equally applicable to proving the correctness of transformations.

Acknowledgement

Thanks to Robert Glück, John Hatcliff, Morten Heiner Sørensen, Kristian Nielson and Phil Wadler for a number of invaluable discussions and feedback on earlier drafts, and to the referees for suggesting many clarifications and improvements.

References

- [1] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison Wesley, 1990.
- [2] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24:44–67, January 1977.
- [3] W. N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, University of London, 1990.
- [4] Wei-Ngan Chin. Safe fusion of functional expressions II: further improvements. *Journal of Functional Programming*, 4(4):515–555, October 1994.
- [5] B. Courcelle. Infinite trees in normal form and recursive equations having a unique solution. *Mathematical Systems Theory*, 13:131–180, 1979.
- [6] M. Felleisen, D. Friedman, and E. Kohlbecker. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.
- [7] A. Ferguson and P. Wadler. When will deforestation stop. In *Proceedings of the 1988 Glasgow Workshop on Functional Programming*, Glasgow Computer Science Department Research Report 89/R4, 1988.
- [8] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *FPCA '93, Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 1993.
- [9] R. Glück and A. V. Klimov. Occam’s razor in metacomputation: the notion of a perfect process tree. In G.Filè P.Cousot, M.Falaschi and A.Rauzy, editors, *Static Analysis. Proceedings*, volume 724 of *LNCS*, pages 112–123. Springer-Verlag, 1993.
- [10] Robert Glück and Morten Heine Sørensen. Partial deduction and driving are equivalent. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*, pages 165–181. Springer-Verlag, 1994.

- [11] C. Gomard. A self-applicable partial evaluator for the lambda calculus: correctness and pragmatics. *ACM TOPLAS*, 14(2):147–172, 1992.
- [12] G. W. Hamilton. Deforestation for higher order functional programs. Unpublished, Keele University, UK, April 1995.
- [13] G. W. Hamilton and S. B. Jones. Extending deforestation for first order functional programs. In *Proceedings of the Fourth Glasgow Workshop on Functional Programming*, Workshops in Computing Series, pages 134–145, Skye, August 1991. Springer-Verlag.
- [14] D. J. Howe. Equality in lazy computation systems. In *Fourth annual symposium on Logic In Computer Science*, pages 198–203. IEEE, 1989.
- [15] N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
- [16] J. Komorowski. An introduction to partial deduction. In *Proceedings of the Third International Workshop on Meta-Programming in Logic*, volume 649 of *LNCS*, pages 49–69. Springer-Verlag, 1992.
- [17] L. Kott. About transformation system: A theoretical study. In B. Robinet, editor, *Program Transformations*, pages 232–247. Dunod, 1978.
- [18] J. W. Lloyd and J. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 3–4(11), November 1991.
- [19] S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Functional Programming, Glasgow 1992*, Springer Workshops in Computer Science, pages 154–165. Springer-Verlag, 1992.
- [20] S. D. Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, Dept. of Computing Science, University of Glasgow, 1995.
- [21] K. Nielsen and M. H. Sørensen. Call-by-name CPS-translation as a binding-time improvement. In *Static Analysis Symposium (SAS '95)*, volume 983 of *LNCS*, pages 296–313. Springer-Verlag, 1995.
- [22] J. Palsberg. Correctness of binding time analysis. *Journal of Functional Programming*, 3(3), 1993.
- [23] P. Partsch and R. Steinbruggen. Program transformation systems. *Computing Surveys*, 15:199–236, 1983.
- [24] A. Pettorossi and M Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 1995.
- [25] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd, London, 1987.
- [26] G. D. Plotkin. Call-by-name, Call-by-value and the λ -calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.

- [27] D. Sands. Operational theories of improvement in functional languages (extended abstract). In *Proceedings of the Fourth Glasgow Workshop on Functional Programming*, pages 298–311, Skye, August 1991. Springer Workshop Series.
- [28] D. Sands. Correctness of recursion-based automatic program transformations. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT/FASE '95)*, number 915 in LNCS. Springer-Verlag, 1995.
- [29] D. Sands. Total correctness by local improvement in program transformation. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, January 1995.
- [30] D. Sands. Total correctness by local improvement in the transformation of functional programs. Technical report, DIKU, University of Copenhagen, January 1995. 48pages (Submitted for Publication).
- [31] M H Sørensen. Turchin's supercompiler revisited: An operational theory of positive information propagation. Master's thesis, Department of Computer Science, University of Copenhagen, 1994.
- [32] M H Sørensen and R Glück. An algorithm for generalization in positive supercompilation. In *International Logic Programming Symposium (ILPS'95)*, Portland, Oregon, 1995. MIT Press.
- [33] M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *ESOP'94*. LNCS 788, Springer Verlag, 1994.
- [34] P. Steckler. *Correct Higher-Order Program Transformations*. PhD thesis, College of Computer Science, Northeastern University, Boston, 1994. Tech Report NU-CCS-94-15.
- [35] V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8:292–325, July 1986.
- [36] V. F. Turchin. The algorithm of generalization. In D. Bjørner, Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, Gammel Avernæs, Denmark, October 1987*, pages 531–549. North-Holland, 1988.
- [37] P. Wadler. The concatenate vanishes. University of Glasgow. Unpublished (preliminary version circulated on the FP mailing list, 1987), November 1989.
- [38] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990. Preliminary version in ESOP 88, LNCS 300.
- [39] M. Wand. Specifying the correctness of binding time analysis. *Journal of Functional Programming*, 3(3), 1993.