Proving the Correctness of Recursion-Based Automatic Program Transformations *

David Sands

DIKU, University of Copenhagen **

Abstract This paper shows how the *Improvement Theorem*—a semantic condition for the total correctness of program transformation on higher-order functional programs—has practical value in proving the correctness of automatic techniques, including deforestation and supercompilation. This is aided by a novel formulation (and generalisation) of deforestation-like transformations, which also greatly adds to the modularity of the proof with respect to extensions to both the language and the transformation rules.

1 Introduction

Transformation of recursive programs Source-to-source transformation methods for functional programs, such as *partial evaluation* [JGS93] and *deforestation* [Wad90,Chi90], perform equivalence preserving modifications to the definitions in a given program. These methods fall in to a class which have been called called *generative set transformations* [PS83]: 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 [BD77]; many automatic transformations of this class can be viewed as specialised instances of unfold-fold rules.

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 (in order to get a speedup of more than an additive constant factor), it is desirable that such transformations have the ability to introduce recursion. Transformations such as deforestation (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*" principal [JGS93].

The Problem of Correctness Program transformations should preserve the the *extensional* meaning of programs 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

* To Appear: TAPSOFT '95 (Formal Approaches in Software Eng.) Springer LNCS

** Universitetsparken 1, 2100 København Ø; dave@diku.dk

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):

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

This example fits into the framework of the unfold-fold method (first apply the *law* $42 \cong 0 + 42$, and *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.

A Solution, in Principal 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 [Sa95a] we present such a condition, *improvement*, and show that if the local steps of a transformation are improvements (in a formal sense) then the transformation will be correct, and, *a fortiori*, yield an "improved" program. The method applies to call-by-name and call-by-value functional languages, including higher-order functions and lazy data structures. In [Sa95a] 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 of more automatic transformation methods (without need for restrictions).

In this paper we substantiate this claim.

A Solution, in Practice We consider two "automatic" program transformations to illustrate the application of the improvement theorem.

The first application [which we only have space to outline] is to a simple systematisation (due to Wadler [Wad89]) of a well-known transformation to eliminate instances of the concatenate operator from functional programs. With only minor changes in the presentation, the improvement theorem is directly applicable, and thus correctness and improvement are guaranteed.

The main application of the improvement theorem illustrated in this paper is more involved. We provide a total correctness proof for an automatic transformation based on a higher-order variant of the deforestation method [Wad90] (which implies the correctness of the transformations performed by the well-known first-order algorithm). To reason about the folding process, and to apply the improvement theorem, we need to reformulate the usual inductive style of definition to provide a *stepwise* account. With this new formulation (extended naturally to deal with higher-order functions) the proof of correctness[3] becomes strikingly simple, since it amounts to showing that each "step" is an improvement; the proof is robust with respect to the folding strategy, and is modular with respect to the transformation steps, so we also consider a generalisation of

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

the "positive supercompilation" rule from [SGJ94]. To our knowledge this is the first proof of correctness for the results of recursive deforestation (for a first-order language or otherwise) which explicitly considers the essential folding steps.

Related Work In the study of the correctness issues (in program transformation of the kind addressed in this paper) it is typical to *ignore* the folding or memoisation aspects of the algorithms. This often 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 [Gom92][Pal93][Wan93] [CK93] 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 [Ste94] are quite orthogonal to the ones studied here, since they concern local optimisations which are (only) justified by dataflow properties of the program in which they are performed. To the author's knowledge, the only other correctness proofs (of which we are aware) for automatic transformations of recursive programs which use some form of folding are in the study of related logic-program transformation, eg. [LS91] [Kom92]. For an extensive comparison of the improvement theorem with other general techniques for correct transformations, see [Sa95b].

The remainder of the paper is organised as follows. **Section 2** deals with syntax, operational semantics and definition of operational approximation and equivalence for a higher-order functional language. In **Section 3** the definition and properties of improvement are given, and the improvement theorem is stated. **Section 4** outlines the application of the improvement theorem to a concatenate-elimination transformation. **Section 5** applies the improvement theorem to prove correctness of the deforestation-like transformations.

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 (and its call-by-name operational semantics), but the inclusion of a strict application operator and arbitrary strict primitive functions (which could include constructors and destructors for strict data structures) should be sufficient to convince the reader that similar results carry over to call-by-value languages.

We assume a flat set of mutually recursive function definitions of the form $\mathtt{f}\, x_1 \ldots x_{\alpha_\mathtt{f}} \triangleq e_\mathtt{f}$ where $\alpha_\mathtt{f}$, the arity of function $\mathtt{f}$, is greater than zero. (For an indexed set of functions we will sometimes refer to the arity by index, $\alpha_\mathtt{i}$, rather than function name.) $\mathtt{f}$, $\mathtt{g}$, $\mathtt{h}$ ..., range over function names, $x, y, z \ldots$ over variables and e, e_1, $e_2 \ldots$ over expressions. The syntax of expressions is as follows:

$$
\begin{array}{lll}
e = x \mid \mathtt{f} \mid e_1\, e_2 & & \text{(Variable; Function name; Application)} \\
\quad \mid e_1 @ e_2 & & \text{(Strict application)} \\
\quad \mid \mathtt{case}\ e\ \mathtt{of} & & \text{(Case expressions)} \\
\qquad c_1(\vec{x}_1) : e_1 \ \ldots\ c_n(\vec{x}_n) : e_n & & \\
\quad \mid c(\vec{e}) & & \text{(Constructor expressions and constants)} \\
\quad \mid p(\vec{e}) & & \text{(Strict primitive functions)}
\end{array}
$$

The expression written $e\{\vec{e}'/\vec{x}\}$ will denote simultaneous (capture-free) substitution of a sequence of expressions $\vec{e}'$ for free occurrences of a sequence of variables $\vec{x}$, respectively, in the expression e. The term $\text{FV}(e)$ will denote the free variables of expression e. Sometimes we will (informally) write substitutions of the form $\{\vec{e}/\vec{\mathtt{g}}\}$ to represent the replacement of occurrences of function symbols $\vec{\mathtt{g}}$ by expressions $\vec{e}$. 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).

Operational Semantics The operational semantics defines an evaluation relation (a partial function) $\Downarrow$. If $e \Downarrow w$ for some closed expression e then we say that *e evaluates to weak head normal form w*, or *e converges*. The weak head normal forms, $w, w_1, w_2, \ldots \in \text{WHNF}$ are just the constructor-expressions $c(\vec{e})$, and the partially applied functions, $\mathtt{f}\, e_1 \ldots e_k$, $0 \leq k < \alpha_{\mathtt{f}}$. For a given closed e, if there is no such w then we say the *e diverges*. We make no finer distinctions between divergent expressions, so "errors" and "loops" are identified. The operational semantics is a standard call-by-name one, and $\Downarrow$ is defined in terms of a one-step evaluation relation using the notion of a *reduction context* [FFK87]. Reduction contexts, ranged over by $\mathbb{R}$, are contexts containing a single hole which is used to identify the next expression to be evaluated (reduced).

Definition 2.1 *A reduction context $\mathbb{R}$ is given inductively by the following grammar*

$$\mathbb{R} = [\,] \mid \mathbb{R}\ e \mid \mathbb{R}@e \mid w@\mathbb{R} \mid \mathtt{case}\ \mathbb{R}\ \mathtt{of}\ c_1(\vec{x}_1) : e_1 \ldots c_n(\vec{x}_n) : e_n \mid p(\vec{c}, \mathbb{R}, \vec{e})$$

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

Definition 2.2 *One-step reduction $\mapsto$ is the least relation on closed expressions satisfying the rules given in Figure 1.*

$$\begin{aligned}
\mathbb{R}[\mathtt{f}\ e_1 \ldots e_{\alpha_{\mathtt{f}}}] &\mapsto \mathbb{R}[e_{\mathtt{f}}\{e_1 \ldots e_{\alpha_{\mathtt{f}}}/x_1 \ldots x_{\alpha_{\mathtt{f}}}\}] \quad (*)\\
\mathbb{R}[w@w'] &\mapsto \mathbb{R}[w\ w']\\
\mathbb{R}[\mathtt{case}\ c_i(\vec{e})\ \mathtt{of} \ldots c_i(\vec{x}_i) : e_i \ldots] &\mapsto \mathbb{R}[e_i\{\vec{e}/\vec{x}_i\}]\\
\mathbb{R}[p(\vec{c})] &\mapsto \mathbb{R}[c'] \qquad (\textit{if } [\![p]\!]\vec{c} = c')
\end{aligned}$$

Fig. 1. *One-step reduction rules*

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 $I\!R$ and a redex e' such that $e_1 = I\!R[e']$.

Definition 2.3 *Closed expression e converges to weak head normal form w, $e{\Downarrow}w$, if and only if $e \mapsto^* w$ (where $\mapsto^*$ is the transitive reflexive closure of $\mapsto$).*

From this we define the standard notions of operational approximation and equivalence. The operational approximation we use is the standard Morris-style contextual ordering, or *observational approximation* eg. [Plo75].The notion of "observation" we take is just the fact of convergence, as in the lazy lambda calculus [Abr90]. Operational equivalence equates two expressions if and only if in all closing contexts they give rise to the same observation - ie. either they both converge, or they both diverge[4].

Definition 2.4 *(i) e* observationally approximates *e', $e \lesssim e'$, if for all contexts C such that $C[e]$, $C[e']$ are closed, if $C[e]{\Downarrow}$ then $C[e']{\Downarrow}$.*

(ii) e is observationally equivalent *to e', $e \cong e'$, if $e \lesssim e'$ and $e' \lesssim e$.*

3 Improvement

In this section we outline the main technical result from [Sa95a], which says that if transformation steps are guided by certain optimisation concerns (a fairly natural condition for a transformation), then correctness of the transformation follows.

The above notion of optimisation is based on 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, in terms of the number of non-primitive function calls computed. 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 the improvement relation a more tractable formulation and some related proof techniques are used.

The *improvement theorem* shows that if e is improved by e' (in addition to e being operationally equivalent to e') then a transformation which replaces e by e' (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).

Definition 3.1 *Closed expression e converges in n ($\in I\!N$) -steps to weak head normal form w, $e{\Downarrow}^n w$ if $e{\Downarrow}w$, and this computation requires n reductions of non-primitive functions (rule $(*)$, Fig. 1).*

We will be convenient to adopt the following abbreviations:

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

[4] For this language if we choose to observe more – such as the actual constructor produced – or if we choose to observe only convergence to a constant rather than any WHNF, the observational approximation and equivalence relations will be unchanged.

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

Definition 3.2 *(Improvement)* e is improved by e', $e \mathrel{\underset{\sim}{\rhd}} e'$, *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 that $\mathrel{\underset{\sim}{\rhd}}$ is a *precongruence* (transitive, reflexive, closed under contexts, ie. $e \mathrel{\underset{\sim}{\rhd}} e' \Rightarrow C[e] \mathrel{\underset{\sim}{\rhd}} C[e']$) and is a refinement of operational approximation, ie. $e \mathrel{\underset{\sim}{\rhd}} e' \Rightarrow e \mathrel{\underset{\sim}{\sqsubset}} e'$.

3.1 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 $\mathtt{f}\,x \triangleq e$ to a new version $\mathtt{f}\,x \triangleq e'$ will be represented by the derivation of a new function $\mathtt{g}\,x \triangleq e'\{\mathtt{g}/\mathtt{f}\}$. In this way we do not need to explicitly parameterise operational equivalence and improvement by the intended set of function definitions.

Theorem 3.3 ([Sa95a]) *Given a set of function definitions,*
$\{\mathtt{f}_i\,x_1 \ldots x_{\alpha i} \triangleq e_i\}_{i\in I}$ *and a set* $\{e'_i\}_{i\in I}$ *such that* $\mathrm{FV}(e'_i) \subseteq \{x_1 \ldots x_{\alpha i}\}$,
if $e_i \mathrel{\underset{\sim}{\rhd}} e'_i$ *then* $\mathtt{f}_i \mathrel{\underset{\sim}{\rhd}} \mathtt{g}_i$ *where* $\vec{\mathtt{g}}$ *are new functions:* $\{\mathtt{g}_i\,x_1 \ldots x_{\alpha i} \triangleq e'_i\{\vec{\mathtt{g}}/\vec{\mathtt{f}}\}\}_{i\in I}$

The "standard" partial correctness result (see eg. [Kot78][Cou79]), which follows easily from a least fixed-point theorem for $\mathrel{\underset{\sim}{\sqsubset}}$, (the full details are given in [Sa95b]) says that if $\mathtt{f}\,x \triangleq e$ and $e \cong e'$ then $\mathtt{g} \mathrel{\underset{\sim}{\sqsubset}} \mathtt{f}$ where $\mathtt{g}\,x \triangleq e'\{\mathtt{g}/\mathtt{f}\}$. Combining this with the improvement theorem, we get a condition for total correctness for transformations which are built by (repeated) application of a set of source-to-source transformations to the bodies of function definitions:

Corollary 3.4 *If the basic steps of a transformation are equivalence-preserving, and are also contained in the improvement relation (with respect to the original definitions) then the resulting transformation will be correct, an moreover, the resulting program will be an improvement over the original.*

3.2 Proving Improvement

Finding a more tractable characterisation of improvement (than that provided by Def. 3.2) is essential in establishing improvement laws (and in the proof of the improvement theorem). 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. [Abr90,How89], and the proof of the characterisation uses previous technical results concerning a more general class of improvement relations [San91].

Definition 3.5 *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) $w_1 \equiv c(e_1 \ldots e_n)$, $w_2 \equiv c(e'_1 \ldots e'_n)$, *and* $e_i\ \mathcal{IR}\ e'_i$, $(i \in 1 \ldots n)$, or
(ii) $w_1 \in$ *Closures*[5], $w_2 \in$ *Closures, and for all closed* e_0, $(w_1\,e_0)\ \mathcal{IR}\ (w_2\,e_0)$

[5] *Closures* is the set of function-valued results, ie. partially applied functions.

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

The key to reasoning about the improvement relation is the fact that $\underset{\sim}{\rhd}$, restricted to closed expressions, is itself an improvement simulation (and is in fact the *maximal* 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 3.6 (Improvement Context-Lemma) *For all e, e', $e \underset{\sim}{\rhd} e'$ 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, sometimes called *co-induction*:

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

We conclude the section with some example laws which follow directly from this characterisation, or can be proved by exhibiting appropriate improvement simulations:

Proposition 3.7 (i) $e \mapsto e' \Rightarrow e \underset{\sim}{\rhd} e'$ (ii) $\mathtt{f}\ x \underset{\sim}{\rhd} e$ *if* $\mathtt{f}\ x \triangleq e$

(iii) $$I\!R[\mathtt{case}\ x\ \mathtt{of} \quad c_1(\vec{y}_1) : e_1 \quad \cdots \quad c_n(\vec{y}_n) : e_n] \underset{\sim}{\rhd} \mathtt{case}\ x\ \mathtt{of} \quad c_1(\vec{y}_1) : I\!R[e_1] \quad \cdots \quad c_n(\vec{y}_n) : I\!R[e_n]$$

The first rule follows easily by showing that the relation containing (e, e') together with all syntactic equivalences is an improvement simulation. It is also easy to see that if the reduction step is not the function call case $(*)$, then $e' \underset{\sim}{\rhd} e$ also holds. (ii) says that *unfolding* is an improvement; this is just a consequence of (i), since each closed instance is in the one-step reduction relation. For the third law we construct a simulation relation in the manner of the first case, and reason from the operational semantics.

4 A Simple Application

The simplest illustration of the application of the improvement theorem is to the verification of the correctness (and improvement) of a mechanisable transformation which aims to eliminate calls to the concatenate (or *append*) function. The effects of the transformation are well-known, such as the transformation of a naïve quadratic-time reverse function into a linear-time equivalent. The systematic definition of the transformation used is due to Wadler [Wad89]. Wadler's formulation of this well known transformation is completely mechanisable, and the transformation "algorithm" always terminates. Unlike many other mechanisable transformations (such as deforestation and partial evaluation), it can improve the asymptotic complexity of some programs.

After some initial definitions have been constructed, the core of the transfomation is a set of rewrite rules which are applied exhausively to the program. Without any essential change to the definition of the transformation, we have obtained a strikingly simple proof of correctness by showing that all of the rewrites are improvements.

[The details are omitted for lack of space—see [Sa95b]]

5 Deforestation and Positive Supercompilation

Deforestation [Wad90] is a transformation developed for first-order lazy functional programs, which aims to eliminate the construction of intermediate data structures (eg. trees, and hence the name). The aim of the transformation is the symbolic 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 systematic and thereby mechanisable.

The deforestation algorithm is described by simple expression-level transformation of the expressions in a program, plus implicit (but essential) folding steps, whereby previously encountered expressions are identified, and recursion is introduced. The main body of work on this transformation concerns the restriction of the transformation steps so that, assuming folding, the algorithm terminates [FW88,Chi90,Sø94a]. Other work considers extensions of the algorithm to richer languages eg. higher-order functions [MW92], and extensions to enable more powerful transformations [SGJ94].

The contributions of this section are:

- a new stepwise formulation of the deforestation transformation;
- a natural generalisation to higher-order functions;
- a correctness proof which includes folding steps, and shows that any folding strategy based on the transformation history (not excluding the possibility of mutual recursion) is correct.

Regarding the correctness of the transformed programs, Wadler originally argued that the expression level transformation is obviously correct (since it essentially uses just unfolding and simplifications which eliminate constructors). This fact is proved for a certain weak form of equivalence in [Sø94b] (unfortunately the "weakness" in question is that the equivalence relation is not closed under arbitrary substitution, so is not a congruence). But these properties, whilst necessary, do not in themselves imply correctness of the resulting programs, because the transformation uses a memoisation process to implement folding[6]. What remains to be achieved, and what we achieve in the remainder of this section, is to show that the resulting programs are equivalent to the originals – and in particular in the presence of folding.

The Deforestation Transformation

In most work studying deforestation (or fusion/driving), the language is a first-order subset of the language presented here, including just case-expressions (or equivalently, definition by pattern-matching) and recursive function calls. We will generalise this to a higher-order language but we will omit primitive function and strict application to simplify the presentation. The results of transformation using this generalisation are not substantially different from those achieved by Marlow and Wadler [MW92], 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 ($\mathbb{R}\, e$), plus an

[6] Non recursion based approaches to deforestation [GLJ93] [SF93] do not encounter this problem, but cannot handle recursive definitions.

additional set of weak head normal forms—the partially applied functions. A similar generalisation is given by Nielsen and Sørensen [NH94], where they study the relationship to partial evaluation.

For reasons of space we do not present the original recursive formulation of the deforestation algorithm.

5.1 Stepwise Deforestation

For the most part the folding process is left *implicit* in the definition of the transformation. But from the point of view of proving total correctness this is the meat of the problem. Combining folding with the standard deforestation algorithm is notationally rather complicated. In essence, the steps of the transformation must be sequentialised in some way, and a "memo-table" of terms encountered so-far must be threaded through the transformation. However, such a description commits the method to using a particular "transformation order" – eg. should the sub-expressions in the term $c(e_1 \ldots e_n)$ be transformed left-to-right, breadth-first, or as is usual in the implicit definitions, independently. Not all transformation orders will lead to the same transformed program, but from the point of view of the correctness of the resulting programs, these issues are orthogonal.

We will prove correctness for a finer-grained description of the transformation (than is usually given) via a *one-step deforestation relation* on terms, analogous to the one-step reduction relation ($\mapsto$). This is based on a novel combination of *reduction contexts* (as implicit in earlier formulations of deforestation [FW88]) and *passive contexts* which enable transformations to be pushed deeper into a term.

Unlike the one-step reduction relation, the transformation rules we specify are not deterministic. This is because the resulting correctness results do not depend on the the transformation order; any strategies of applying the given rules (eg. breadth-first to obtain more possible folds) including restrictions (eg. to improve termination behaviour on a wider class of programs) will give correct programs. Folding can be described simply in terms of the transformation history - the sequences of expressions rewritten (and new function definitions introduced). A second benefit of the new presentation will be that additional rules can be added and the correctness proof will be completely modular.

The basic deforestation rules are presented in Figure 2. To simplify presentation, we define a class of *simple dynamic expressions* – these are variables or variables applied to some expressions. Using this we define the *passive contexts.* These are the contexts which take no further part in the transformation and allow the transformations to be pushed further into an expression.

Definition 5.1

(i) The simple dynamic expressions, *ranged over by d are given by* $d = x \mid d\,e$.

(ii) The passive contexts, *ranged over by $\mathbb{P}$, are single-holed contexts given by*

$$\mathbb{P} = [\,] \quad | \quad \texttt{case}\ d\ \texttt{of}\ \ldots c_n(\vec{x}_i) : \mathbb{P} \ldots \quad | \quad c(\ldots \mathbb{P} \ldots)$$

Recall that the reduction contexts for this subset of the language are given by

$$\mathbb{R} = [\,] \quad | \quad \texttt{case}\ \mathbb{R}\ \texttt{of}\ c_1(\vec{x}_1) : e_1 \ldots c_n(\vec{x}_n) : e_n \quad | \quad \mathbb{R}\,e$$

$$\mathbb{P}.\mathbb{R}[\mathtt{f}\ e_1 \ldots e_{\alpha_{\mathtt{f}}}] \rightsquigarrow \mathbb{P}[\mathtt{f}^\diamond\ y_1 \ldots y_n] \qquad (d1)$$
$$\textit{where}\ \{y_1 \ldots y_n\} = \mathrm{FV}(\mathbb{R}[\mathtt{f}\ e_1 \ldots e_{\alpha_{\mathtt{f}}}])$$
$$\textit{and}\ \mathtt{f}^\diamond\ y_1 \ldots y_n \triangleq \mathbb{R}[e_{\mathtt{f}}\{{}^{e_1 \cdots e_{\alpha_{\mathtt{f}}}}/_{x_1 \ldots x_{\alpha_{\mathtt{f}}}}\}]$$

$$\mathbb{P}.\mathbb{R}[\mathtt{case}\ c_i(\vec{e})\ \mathtt{of}\ \ldots c_i(\vec{x}_i) : e_i \ldots] \rightsquigarrow \mathbb{P}.\mathbb{R}[e_i\{{}^{\vec{e}}/_{\vec{x}_i}\}] \qquad (d2)$$

$$\mathbb{P}.\mathbb{R}[\mathtt{case}\ d\ \mathtt{of}\ c_1(\vec{x}_1) : e_1 \ldots c_n(\vec{x}_n) : e_n\ ] \qquad (d3)$$
$$\rightsquigarrow \mathbb{P}[\mathtt{case}\ d\ \mathtt{of}\ c_1(\vec{x}_1) : \mathbb{R}[e_1] \ldots c_n(\vec{x}_n) : \mathbb{R}[e_n]\ ]$$

$$\mathbb{P}.\mathbb{R}[\mathtt{f}\ e_1 \ldots e_{\alpha_{\mathtt{f}-k}}] \rightsquigarrow \mathbb{P}.\mathbb{R}[\mathtt{f}^\diamond\ y_1 \ldots y_j] \qquad (d4)$$
$$\textit{where}\ \ \{y_1 \ldots y_j\} = \mathrm{FV}(e_1 \ldots e_{\alpha_{\mathtt{f}-k}})$$
$$\textit{and}\ \ \mathtt{f}^\diamond\ y_1 \ldots y_j\ z_1 \ldots z_k \triangleq e_{\mathtt{f}}\{{}^{e_1 \cdots e_{\alpha_{\mathtt{f}-k}}\ z_1 \ldots z_k}/_{x_1 \ldots x_{\alpha_{\mathtt{f}}}}\}$$

Fig. 2. One-Step Deforestation Rules

We write $\mathbb{P}.\mathbb{R}$ to denote the composition of passive and reduction contexts, so $\mathbb{P}.\mathbb{R}[e]$ will denote the term $\mathbb{P}[\mathbb{R}[e]]$. Note that contexts $\mathbb{P}.\mathbb{R}$ include both the passive and the reduction contexts.

Some comments are appropriate. The rules essentially mimic the action of the operational semantics, but for terms containing free variables. Rule **(d1)** unfolds a function call occurring in a reduction context; the result of the unfolding of the call in the reduction context is represented indirectly by the introduction of a new function definition (introduced for the purpose of folding, in case this expression is re-encountered). It is assumed that new function names are fresh. Rule **(d2)** is the standard case-reduction rule from the operational semantics. Rule **(d3)** is the "propagate-context" rule. Here a case expression, with an un-resolvable (dynamic) test, occurs in a reduction position, and so the reduction context is pushed into the branches. This is the generalisation of the "case-case" law in the original formulation of deforestation. Rule **(d4)** is the case of a partially applied function where we force an unfolding via an auxiliary function.

The role of the passive context is to allow the rules to drive deeper into a term, in the situation where the outer layer of the term cannot be transformed further. For example, if the outermost construct of the term is an un-resolvable case expression, then transformation can proceed to the branches.

Applying the Rules in Deforestation The deforestation algorithm begins with a top level expression, e_0, which represents the program (containing some free variables) to be transformed.

The one step deforestation rules can implement the deforestation process by applying them in the following manner. First abstract the free variables from e_0 to form a new (non-recursive) definition $\mathtt{f}_0^\diamond\,\vec{x} \triangleq e_0$. Maintaining a distinction between the original functions in the program (ranged over by $\mathtt{f}, \mathtt{g} \ldots$), and the new functions introduced by the transformation steps (henceforth ranged over by $\mathtt{f}^\diamond, \mathtt{g}^\diamond \ldots$) including $\mathtt{f}_0^\diamond$, transform the right-hand sides of the new functions by repeated (nondeterministic) application of the rules *but never applying rule*

$(d1)$ *in order to unfold a new function.* Rule (d3) is not applied when $\mathbb{R} = [\,]$, since in this case the rule is the identity, and rule (d4) is not used if $\mathbb{R}$ is an application context.

Folding in Deforestation In order to get the above algorithm to terminate in some non trivial cases we need to add folding, or memoization[7]. Both rules (d1) and (d4) introduce new function definitions (without these rules termination would be assured, but uninteresting). The basic idea is 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 of the form $\mathbb{P}.\mathbb{R}[\mathtt{f}\ e_1 \ldots e_{\alpha_{\mathtt{f}}}]$ then we look into the memo-list. If there is an entry $\langle \mathbb{R}'[\mathtt{f}\ e'_1 \ldots e'_{\alpha_{\mathtt{f}}}], \mathtt{f}^\diamond \vec{y} \rangle$ such that $\mathbb{R}'[\mathtt{f}\ e'_1 \ldots e'_{\alpha_{\mathtt{f}}}]\theta \equiv \mathbb{R}[\mathtt{f}\ e_1 \ldots e_{\alpha_{\mathtt{f}}}]$ where θ is a *renaming* (a substitution mapping variables to variables) then we transform $\mathbb{P}.\mathbb{R}[\mathtt{f}\ e_1 \ldots e_{\alpha_{\mathtt{f}}}]$ to $\mathbb{P}[(\mathtt{f}^\diamond \vec{y})\theta]$. Otherwise we apply the rule as normal, introducing a new function name $\mathtt{f}^\diamond$, and add the pair $\langle \mathbb{R}[\mathtt{f}\ e_1 \ldots e_{\alpha_{\mathtt{f}}}], \mathtt{f}^\diamond \vec{y} \rangle$ to the memo-table. We memoise use of rule $(d4)$ in the same way.

Example 5.2 Consider the following definitions:

$$
\begin{array}{l}
\mathtt{filter}\ p\ xs \triangleq \mathtt{case}\ xs\ \mathtt{of} \\
\qquad\qquad\qquad\quad nil : \mathtt{nil} \\
\qquad\qquad\qquad\quad cons(y, ys) : \mathtt{case}\ p\ y\ \mathtt{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad true : \mathtt{cons}(y, \mathtt{filter}\ p\ ys) \\
\qquad\qquad\qquad\qquad\qquad\qquad false : \mathtt{filter}\ p\ ys \\
\qquad\quad \mathtt{map}\ f\ xs \triangleq \mathtt{case}\ xs\ \mathtt{of} \\
\qquad\qquad\qquad\quad nil : \mathtt{nil} \\
\qquad\qquad\qquad\quad cons(z, zs) : \mathtt{cons}((f\ z), \mathtt{map}\ f\ zs) \\
\mathtt{compose}\ f\ g\ x \triangleq f\ (g\ x)
\end{array}
$$

Writing `compose` in the usual infix style ($e \circ e' \equiv \mathtt{compose}\ e\ e'$) we wish to transform the initial definition:

$$\mathtt{f}^\diamond_0\ f\ p \triangleq (\mathtt{map}\ f) \circ (\mathtt{filter}\ p)$$

Now we transform the right hand side of the initial definition and the right-hand sides of subsequently introduced definitions. The initial transformation steps are given in Fig. 3; each derivation step ($\rightsquigarrow$) refers to the right-hand side of the preceding definition.

After these steps the transformation can proceed to the two occurrences of the sub-term $\mathtt{filter}\ p\ (\mathtt{map}\ f\ zs))$ (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

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

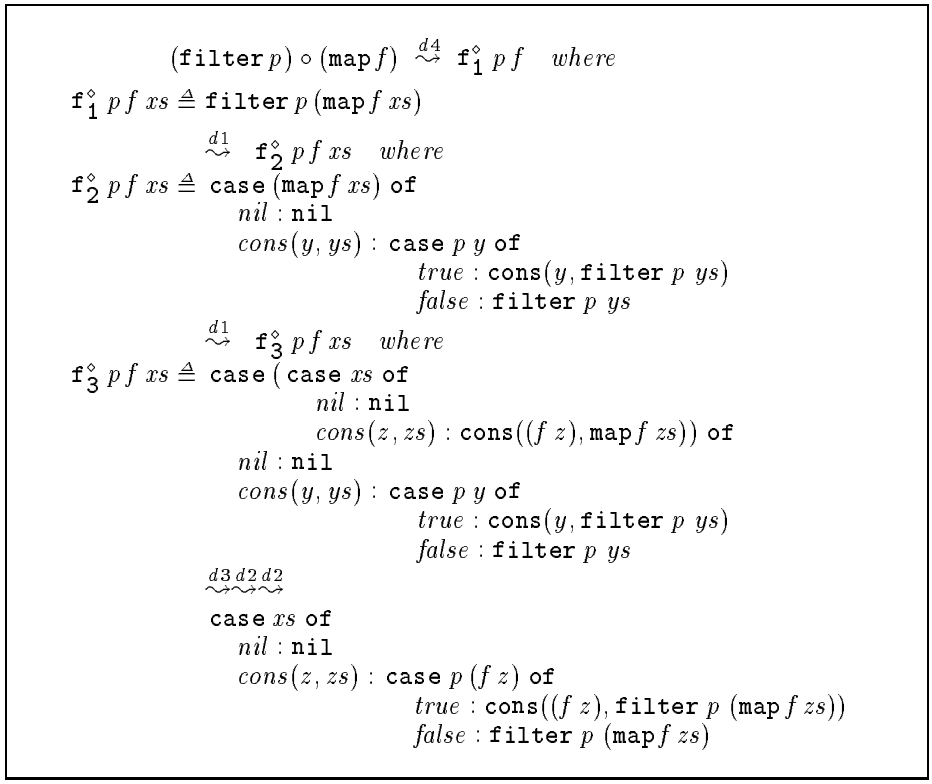

Fig. 3. Initial Deforestation Steps

we "fold", introducing a recursive calls to $\mathtt{f}_2^\diamond$, obtaining:

$\mathtt{f}_1^\diamond\ p\ f\ xs \triangleq \mathtt{f}_2^\diamond\ p\ f\ xs$
$\mathtt{f}_2^\diamond\ p\ f\ xs \triangleq \mathtt{f}_3^\diamond\ p\ f\ xs$

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

As is usual, we can eliminate the trivial intermediate functions $\mathtt{f}_1^\diamond$ and $\mathtt{f}_2^\diamond$ by *post-unfolding* [JGS93].

5.2 Correctness

Using the improvement theorem, to prove correctness it will be sufficient to prove that each transformation step is an improvement. This property holds because, as observed by Chin [Chi90], each new function call introduced by the transformation comes together with an unfolding step. This is, in turn, sufficient to justify the folding steps, since these are guaranteed to be improvements.

Proposition 5.3 *$e \rightsquigarrow e'$ implies $e \trianglerighteq_{\sim} e'$.*

PROOF. Straightforward using the congruence properties of improvement. □

There are generally considered to be three aspects to the correctness of deforestation [Sø94b]: (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 the procedure does 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 a "treeless form") These issue is outside the scope of this paper. The improvement theorem deals with aspects (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 $\trianglerighteq$) under call-by-name evaluation.

Proposition 5.4 *Deforestation yields totally correct programs in that any result of applying the deforestation steps (including folding) to a program will result in an improved program.*

PROOF. From the previous proposition, the basic steps are all in the improvement relation. Clearly they are also operational equivalences. Taking a "virtual" view of the transformation [TS84] in which we consider that the initial definitions of the new functions (introduced by the transformation steps) are already present at the beginning of the transformation, then the folding steps are essentially no different from any other rule: they replace an expression by an improved one. So by corollary 3.4 the result of the transformation equivalent to, and an improvement over, the original. □

On Efficiency Improvement relates to call-by-name. Under a call-by-need implementation the usual restrictions of the transformation seem sufficient to regain the improvement result under call-by-need. These restrictions are that only functions which are *linear* in their arguments should be transformed — see [Wad90], [Chi90]. Alternatively, duplication of sub-expressions (eg. Example 5.2 (*fz*) is duplicated in $\mathtt{f}^{\circ}_{3}$) can be avoided by the use of let-bindings, in the obvious way.

On Robustness The correctness proof is dependent on the fact that the individual steps (and hence the folding steps) are improvements, but not on the overall structure of the transformation. This means that the application of the transformation steps can be constrained, for example by use of annotations (eg. "blazing" from [Wad90]) without any additional proof obligation. Similarly, given any particular sequentialisation of the transformation steps, the memoisation process can be arbitrarily constrained in terms of both lookups and writes. In this sense any memoisation strategy is covered.

It also means that we can add or replace transformation rules to increase the power of the method (eg. allowing folding to take advantage of more general expressions) and the only property that needs to be verified is that the new rule is an improvement. Language extensions, such as addition of primitive functions, are easily incorporated by extending the classes of dynamic expressions, passive and reduction contexts appropriately, and by adding any new reduction rules from the operational semantics.

5.3 Driving and Positive Supercompilation

In terms of transformational power (but ignoring termination issues) Turchin's *driving* techniques [Tur86] subsume deforestation. This increased power is due,

in part, to increased information-propagation in the transformation. Propagation of the so-called "positive" information [GK93] can be easily added to the the one-step deforestation rules along the lines of [SGJ94]. The basic idea is that when a case-expression has a variable in the test position, as in $\texttt{case}\ y\ \texttt{of}\ \ldots c_i(\vec{x}_i) : e_i \ldots$, within the ith branch we know that free occurrences of y are equivalent to $c_i(\vec{x}_i)$. The effect of "positive information propagation" is achieved by substituting $c_i(\vec{x}_i)$ for all free occurrences of y in e_i. The transformation seems trivial, but cannot be achieved by preprocessing because it is applied to terms generated on the fly by earlier unfolding steps. The effect of this extra power is illustrated in [SGJ94].

We achieve the natural higher order variant of this transformation rule by generalising the propagation from the single variable case, to any free occurrences of a simple dynamic expression d (Def. 5.1). Positive information propagation is implemented by adding the following rule.

Definition 5.5 *Define the following transformation rule* (**d5**)*:*

$$\mathbb{P}.\mathbb{R}[\texttt{case}\ d\ \texttt{of}\ \ldots c_i(\vec{x}_i) : e\{{}^{d}/_{z}\}\ldots] \rightsquigarrow \mathbb{P}.\mathbb{R}[\texttt{case}\ d\ \texttt{of}\ \ldots c_i(\vec{x}_i) : e\{{}^{c_i(\vec{x}_i)}/_{z}\}\ldots]$$

where we assume the free variables in d and $\vec{x}_i$ are all distinct, and that we allow renaming of bound variables in a term.

Proposition 5.6 *$e \overset{d5}{\rightsquigarrow} e'$ implies $e \mathrel{\underset{\sim}{\trianglerighteq}} e'$*

Proof. Straightforward using the fact that $e_1 \Downarrow e_2$ implies $e_1 \mathrel{\underset{\sim}{\trianglerighteq}} e_2$, together with congruence properties of improvement. □

In conclusion we mention an additional feature of Turchin's supercompilation, namely *generalisation*. This is a familiar concept in inductive proofs, and has a fairly direct analogy in program transformation (see eg. [BD77] [Tur86]), 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 ($d1$) 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 $\texttt{f}^{\diamond}$. (We leave a discussion of *what* should be abstracted to a long version of the paper.) There is a corresponding generalisation of the folding process. The correctness of these variations are also easily proved from the congruence properties of the improvement relation.

Acknowledgements 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 a number of clarifications and improvements.

References

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

[BD77] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24:44–67, January 1977.

[Chi90] W. N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, 1990.

[CK93] C. Consel and S. Khoo. On-line and off-line partial evaluation: Semantic specification and correctness proofs. Tech. Report, Yale, April 1993.

[Cou79] B. Courcelle. Infinite trees in normal form and recursive equations having a unique solution. *Math. Systems Theory*, 13:131–180, 1979.

[FFK87] M. Felleisen, D. Friedman, and E. Kohlbecker. A syntactic theory of sequential control. *TCS*, 52:205–237, 1987.

[FW88] A. Ferguson and P. Wadler. When will deforestation stop. In *1988 Glasgow Workshop on Functional Programming*, Research Rep. 89/R4, 1988.

[GK93] R. Glück and A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In *Static Analysis Syposium*, *LNCS* 724, 1993.

[GLJ93] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *FPCA '93*. ACM Press, 1993.

[Gom92] C. Gomard. A self-applicable partial evaluator for the lambda calculus: correctness and pragmatics. *ACM TOPLAS*, 14(2):147–172, 1992.

[How89] D. J. Howe. Equality in lazy computation systems. In *4th LICS*. IEEE, 1989.

[JGS93] N. D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[Kom92] J. Komorowski. An introduction to partial deduction. In *Third Int. Workshop on Meta-Programming in Logic*, *LNCS* 649, 1992.

[Kot78] L. Kott. About transformation system: A theoretical study. In B. Robinet, editor, *Program Transformations*. Dunod, 1978.

[LS91] J. W. Lloyd and J. Shepherdson. Partial evaluation in logic programming. *J. Logic Programming*, 3–4(11), 1991.

[MW92] S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Functional Programming, Glasgow 1992*, Springer Workshop Series, 1992.

[NH94] K. Nielsen and M. Heine Sørensen Deforestation, partial exaluation and evaluation orders. Unpublished, DIKU, Copenhagen, 1994.

[Pal93] J. Palsberg. Correctness of binding time analysis. *J. Functional Programming*, 3(3), 1993.

[Plo75] G. D. Plotkin. Call-by-name, Call-by-value and the λ-calculus. *TCS*, 1(1):125–159, 1975.

[PS83] P. Partsch and R. Steinbruggen. Program transformation systems. *Computing Surveys*, 15:199–236, 1983.

[San91] D. Sands. Operational theories of improvement in functional languages (extended abstract). In *Fourth Glasgow Workshop on Functional Programming*, Springer Workshop Series, 1991.

[Sa95a] D. Sands. Total correctness by local improvement in program transformation. In *22nd POPL*. ACM Press, 1995.

[Sa95b] D. Sands. Total correctness by local improvement in the transformation of functional programs. DIKU, University of Copenhagen, 48pages, January 1995.

[SF93] T. Sheard and L. Fegaras. A fold for all seasons. In *FPCA '93*. ACM Press, 1993.

[SGJ94] 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.

[Sø94a] M H Sørensen. A grammar-based data-flow analysis to stop deforestation. In *CAAP'94*, LNCS 787, 1994.

[Sø94b] M H Sørensen. Turchin's supercompiler revisited: An operational theory of positive information propagation. Master's thesis, DIKU, University of Copenhagen, (RR 94/9) 1994.

[Ste94] P. Steckler. *Correct Higher-Order Program Transformations*. PhD thesis, Northeastern University, Boston, 1994.

[TS84] H. Tamaki and T. Sato. Unfold/-fold transformation of logic programs. In*2nd Int. Logic Programming Conf.*, 1984.

[Tur86] V. F. Turchin. The concept of a supercompiler. *ToPLaS*, 8:292–325, July 1986.

[Wad89] P. Wadler. The concatenate vanishes. University of Glasgow. Unpublished (preliminary version circulated on the fp mailing list, 1987), November 1989.

[Wad90] P. Wadler. Deforestation: transforming programs to eliminate trees. *TCS*, 73:231–248, 1990. (Preliminary version in ESOP 88, LNCS 300).

[Wan93] M. Wand. Specifying the correctness of binding time analysis. *J. Functional Programming*, 3(3), 1993.