The Next 700 Program Transformers

Geoff Hamilton[000−0001−5954−6444]

School of Computing, Dublin City University, Dublin, Ireland
geoffrey.hamilton@dcu.ie

Abstract. In this paper, we describe a hierarchy of program transformers, capable of performing fusion to eliminate intermediate data structures, in which the transformer at each level of the hierarchy builds on top of those at lower levels. The program transformer at level 1 of the hierarchy corresponds to positive supercompilation, and that at level 2 corresponds to distillation. We give a number of examples of the application of our transformers at different levels in the hierarchy and look at the speedups that are obtained. We determine the maximum speedups that can be obtained at each level, and prove that the transformers at each level terminate.

Keywords: transformation hierarchy · supercompilation · distillation · speedups

1 Introduction

It is well known that programs written using functional programming languages often make use of intermediate data structures and thus can be inefficient. Several program transformation techniques have been proposed to eliminate some of these intermediate data structures; for example *partial evaluation* [14], *deforestation* [30] and *supercompilation* [27]. *Positive supercompilation* [26] is a variant of Turchin's supercompilation [27] that was introduced in an attempt to study and explain the essentials of Turchin's supercompiler. Although strictly more powerful than both partial evaluation and deforestation, Sørensen has shown that positive supercompilation (without the identification of common sub-expressions in generalisation), and hence also partial evaluation and deforestation, can only produce a linear speedup in programs [24]. Even with the identification of common sub-expressions in generalisation, superlinear speedups are obtained for very few interesting programs, and many obvious improvements cannot be made without the use of so-called 'eureka' steps [4].

Example 1. Consider the function call $nrev\ xs$ shown in Fig. 1. This reverses the list xs, but the recursive function call $(nrev\ xs')$ is an intermediate data structure, so in terms of time and space usage, it is quadratic with respect to the length of the list xs. A more efficient function that is linear with respect to the length of the list xs is the function $qrev$ shown in Fig. 1.

A number of algebraic transformations have been proposed that can perform this transformation (e.g. [29]), making essential use of eureka steps requiring

$$
\begin{array}{l}
nrev\ xs\\
\textbf{where}\\
nrev\ xs = \textbf{case}\ xs\ \textbf{of}\\
\quad Nil \Rightarrow Nil\\
\quad Cons\ x'\ xs' \Rightarrow append\ (nrev\ xs')\ (Cons\ x'\ Nil)\\
append\ xs\ ys = \textbf{case}\ xs\ \textbf{of}\\
\quad Nil \Rightarrow ys\\
\quad Cons\ x'\ xs' \Rightarrow Cons\ x'\ (append\ xs'\ ys)
\end{array}
$$

$$
\begin{array}{l}
qrev\ xs\\
\textbf{where}\\
qrev\ xs = qrev'\ xs\ Nil\\
qrev'\ xs\ ys = \textbf{case}\ xs\ \textbf{of}\\
\quad Nil \Rightarrow ys\\
\quad Cons\ x'\ xs' \Rightarrow qrev'\ xs'\ (Cons\ x'\ ys)
\end{array}
$$

Fig. 1. Alternative Definitions of List Reversal

human insight and not easy to automate; for the given example this can be achieved by appealing to a specific law stating the associativity of the *append* function. However, none of the generic program transformation techniques mentioned above are capable of performing this transformation.

The *distillation* algorithm [9, 11] was originally motivated by the need for automatic techniques that avoid the reliance on eureka steps to perform transformations such as the above. In positive supercompilation, generalisation and folding are performed only on expressions, while in distillation, generalisation and folding are also performed on recursive function representations (*process trees*). This allows a number of improvements to be obtained using distillation that cannot be obtained using positive supercompilation.

The process trees that are generalised and folded in distillation are in fact those produced by positive supercompilation, so we can see that the definition of distillation is built on top of positive supercompilation. This suggests the existence of a hierarchy of program transformers, where the transformer at each level is built on top of those at lower levels, and more powerful transformations are obtained as we move up through this hierarchy. In this paper, we define such a hierarchy inductively, with positive supercompilation at level 1, distillation at level 2 and each new level defined in terms of the previous ones. Each of the transformers is capable of performing *fusion* to eliminate intermediate data structures by fusing nested function calls. As we move up through the hierarchy, deeper nestings of function calls can be fused, thus removing more intermediate data structures.

The remainder of this paper is structured as follows. In Section 2, we define the higher-order functional language on which the described transformations are performed. In Section 3, we give an overview of process trees and define a number of operations on them. In Section 4, we define the program transformer hierarchy,

where the transformer at level 0 corresponds to the identity transformation, and each successive transformer is defined in terms of the previous ones. In Section 5, we give examples of transformations that can be performed at different levels in our hierarchy. In Section 6, we consider the efficiency improvements that can be obtained as we move up through this hierarchy. In Section 7, we prove that each of the transformers in our hierarchy terminates. In Section 8, we consider related work and Section 9 concludes and considers possibilities for further work.

2 Language

In this section, we describe the call-by-name higher-order functional language that will be used throughout this paper.

Definition 1 (Language Syntax). The syntax of this language is as shown in Fig. 2.

$prog$	$::= e_0$ **where** $h_1 = e_1 \ldots h_n = e_n$	Program
e	$::= x$	Variable
	$\mid c\ e_1 \ldots e_n$	Constructor Application
	$\mid \lambda x.e$	λ-Abstraction
	$\mid f$	Function Call
	$\mid e_0\ e_1$	Application
	$\mid$ **case** e_0 **of** $p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n$	Case Expression
h	$::= f\ x_1 \ldots x_n$	Function Header
p	$::= c\ x_1 \ldots x_n$	Pattern

Fig. 2. Language Syntax

Programs in the language consist of an expression to evaluate and a set of function definitions. An expression can be a variable, constructor application, λ-abstraction, function call, application or **case**. Variables introduced by function definitions, λ-abstractions and **case** patterns are *bound*; all other variables are *free*. We assume that bound variables are represented using De Bruijn indices. An expression that contains no free variables is said to be *closed*. We write $e \equiv e'$ if e and e' differ only in the names of bound variables.

Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. In an expression $c\ e_1 \ldots e_n$, n must equal the arity of c. The patterns in **case** expressions may not be nested. No variable may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive. It is also assumed that erroneous terms such as $(c\ e_1 \ldots e_n)\ e$ where c is of arity n and **case** $(\lambda x.e)$ **of** $p_1 \Rightarrow e_1 \ldots p_k \Rightarrow e_k$ cannot occur.

Definition 2 (Substitution). We use the notation $\theta = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ to denote a *substitution*. If e is an expression, then $e\theta = e\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ is the result of simultaneously substituting the expressions $e_1, \ldots, e_n$ for the corresponding variables $x_1, \ldots, x_n$, respectively, in the expression e while ensuring that bound variables are renamed appropriately to avoid name capture. A *renaming* denoted by σ is a substitution of the form $\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$.

Definition 3 (Shallow Reduction Context). A shallow reduction context $\mathcal{C}$ is an expression containing a single hole $\bullet$ in the place of the redex, which can have one of the two following possible forms:

$$\mathcal{C} ::= \bullet\ e \mid \textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n$$

Definition 4 (Evaluation Context). An evaluation context $\mathcal{E}$ is represented as a sequence of shallow reduction contexts (known as a *zipper* [13]), representing the nesting of these contexts from innermost to outermost within which the redex is contained. An evaluation context can therefore have one of the two following possible forms:

$$\mathcal{E} ::= \langle\rangle \mid \langle \mathcal{C} : \mathcal{E} \rangle$$

Definition 5 (Insertion into Evaluation Context). The insertion of an expression e into an evaluation context κ, denoted by $\kappa \bullet e$, is defined as follows:

$$\begin{array}{l} \langle\rangle \bullet e \qquad\qquad\ = e \\ \langle(\bullet\ e') : \kappa\rangle \bullet e = \kappa \bullet (e\ e') \\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n) : \kappa\rangle \bullet e \\ \qquad\qquad\qquad = \kappa \bullet (\textbf{case}\ e\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n) \end{array}$$

Definition 6 (Language Semantics). The normal order reduction semantics for programs in our language is defined by $\mathcal{N}_p[\![p]\!]$ as shown in Fig. 3, where it is assumed the program p contains no free variables. Within the rules $\mathcal{N}_e$, κ denotes the context of the expression under scrutiny and Δ is the set of function definitions. We always evaluate the redex of an expression within the context κ.

$$\mathcal{N}_p[\![e_0\ \textbf{where}\ h_1 = e_1 \ldots h_n = e_n]\!] = \mathcal{N}_e[\![e_0]\!]\ \langle\rangle\ \{h_1 = e_1, \ldots, h_n = e_n\}$$

$$\begin{array}{l} \mathcal{N}_e[\![c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \Delta = c\ (\mathcal{N}_e[\![e_1]\!]\ \langle\rangle\ \Delta) \ldots (\mathcal{N}_e[\![e_n]\!]\ \langle\rangle\ \Delta) \\ \mathcal{N}_e[\![c\ e_1 \ldots e_n]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e'_1 \ldots p_k \Rightarrow e'_k) : \kappa\rangle\ \Delta = \\ \quad \mathcal{N}_e[\![e'_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa\ \Delta \\ \quad \text{where } \exists i \in \{1 \ldots k\}.p_i = c\ x_1 \ldots x_n \\ \mathcal{N}_e[\![\lambda x.e]\!]\ \langle\rangle\ \Delta = \lambda x.(\mathcal{N}_e[\![e]\!]\ \langle\rangle\ \Delta) \\ \mathcal{N}_e[\![\lambda x.e]\!]\ \langle(\bullet\ e') : \kappa\rangle\ \Delta = \mathcal{N}_e[\![e\{x \mapsto e'\}]\!]\ \kappa\ \Delta \\ \mathcal{N}_e[\![f]\!]\ \kappa\ \Delta = \mathcal{N}_e[\![\lambda x_1 \ldots x_n.e]\!]\ \kappa\ \Delta \\ \quad \text{where } (f\ x_1 \ldots x_n = e) \in \Delta \\ \mathcal{N}_e[\![e_0\ e_1]\!]\ \kappa\ \Delta = \mathcal{N}_e[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \Delta \\ \mathcal{N}_e[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n]\!]\ \kappa\ \Delta = \\ \quad \mathcal{N}_e[\![e_0]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n) : \kappa\rangle\ \Delta \end{array}$$

Fig. 3. Language Semantics

3 Process Trees

The output of each of the transformers in our hierarchy are represented by *process trees*, as defined in [25]. Within these process trees, the nodes are labelled with expressions. We write $e \rightarrow t_1, \ldots, t_n$ for a process tree where the root node is labelled with the expression e, and $t_1, \ldots, t_n$ are the sub-trees of this root node. We also write $e \rightarrow \epsilon$ for a terminal node that has no sub-trees. We use $root(t)$ to denote the expression labelling the root node of process tree t. Process trees may also contain three special kinds of node:

- *Unfold nodes*: these are of the form $h \rightarrow t$, where h is a function header and t is the process tree resulting from transforming an expression after unfolding.
- *Fold nodes*: these are of the form $h \rightarrow \epsilon$, where folding has been performed with respect to a previous unfold node and the corresponding function headers are renamings of each other.
- *Generalisation nodes*: these are of the form $(x_0\ x_1 \ldots x_n) \rightarrow t_0, t_1, \ldots, t_n$. This represents the result of a process tree generalisation, where the sub-trees $t_1 \ldots t_n$ have been extracted and replaced by the corresponding variables $x_1 \ldots x_n$ in the process tree t_0 that is represented by x_0.

Variables introduced by λ-abstractions, case patterns and generalisation nodes in process trees are bound; all other variables are free. We assume that bound variables are represented using De Bruijn indices. We use $fv(t)$ to denote the free variables of process tree t.

Definition 7 (Renaming of Process Trees). If t is a process tree, then the renaming $t\sigma$ is obtained by applying the renaming σ to the expressions labelling all nodes in t, while ensuring that bound variables are renamed appropriately to avoid name capture.

When transforming an expression with a function in the redex at level $k+1$, the expression is first transformed using a level k transformer. The resulting process tree is then compared to previously encountered process trees generated at level k. If it is a *renaming* of a previous one, then *folding* is performed, and if it is an *embedding* of a previous one, then *generalisation* is performed. The use of process trees in this comparison allows us to abstract away from the number and order of the parameters in functions, and instead focus on their recursive structure. We therefore define renaming, embedding and generalisation on process trees.

Definition 8 (Process Tree Renaming). Process tree t is a *renaming* of process tree t' if there is a renaming σ (which also renames functions) such that $t\sigma \cong t'$, where the relation $\cong$ is defined as follows:

$$(\phi(e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n) \cong (\phi(e'_1 \ldots e'_n) \rightarrow t'_1, \ldots, t'_n), \text{ if } \forall i \in \{1 \ldots n\}.t_i \cong t'_i$$

Two process trees are therefore related by this equivalence relation if the pair of expressions in the corresponding root nodes have the same top-level syntactic constructor ϕ (a variable, constructor, lambda-abstraction, function name, application or **case**), and the corresponding sub-trees are also related. This includes

the pathological case where the nodes have no sub-trees (such as free variables which must have the same name, and bound variables which must have the same de Bruijn index).

In order to ensure the termination of our transformation, we have to perform *generalisation*. This generalisation is performed when a process tree is encountered that is an *embedding* of a previous one. The form of embedding which we use to determine whether to perform generalisation is known as *homeomorphic embedding*. The homeomorphic embedding relation was derived from results by Higman [12] and Kruskal [19] and was defined within term rewriting systems [5] for detecting the possible divergence of the term rewriting process. Variants of this relation have been used to ensure termination within positive supercompilation [25], partial evaluation [21] and partial deduction [3, 20]. The homeomorphic embedding relation is a *well-quasi-order*.

Definition 9 (Well-Quasi Order). A well-quasi order on a set S is a reflexive, transitive relation $\leq_S$ such that for any infinite sequence $s_1, s_2, \ldots$ of elements from S there are numbers i, j with $i < j$ and $s_i \leq_S s_j$.

The homeomorphic embedding relation on process trees is defined as follows.

Definition 10 (Process Tree Embedding). Process tree t is *embedded* in process tree t' if there is a renaming σ (which also renames functions) such that $t\sigma \trianglelefteq t'$, where the relation $\trianglelefteq$ is defined as follows:

$$(\phi(e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n) \trianglelefteq (\phi(e'_1 \ldots e'_n) \rightarrow t'_1, \ldots, t'_n), \text{ if } \forall i \in \{1 \ldots n\}.t_i \trianglelefteq t'_i$$

$$t \trianglelefteq (e \rightarrow t_1, \ldots, t_n), \text{ if } \exists i \in \{1 \ldots n\}.t \trianglelefteq t_i$$

The first rule is a *coupling* rule, where the pair of expressions in the root nodes must have the same top-level syntactic constructor ϕ, and the corresponding sub-trees of the root nodes must also be related to each other. This includes the pathological case where the root nodes have no sub-trees (such as free variables which must have the same name, and bound variables which must have the same de Bruijn index). The second rule is a *diving* rule; this relates a process-tree with a sub-tree of a larger process tree. We write $t \preceq t'$ if $t \trianglelefteq t'$ and the coupling rule can be applied at the top level.

The use of this embedding relation ensures that in any infinite sequence of process trees $t_0, t_1, \ldots$ encountered during transformation there definitely exists some $i < j$ where t_i is embedded in t_j, so an embedding must eventually be encountered and transformation will not continue indefinitely without the need for generalisation or folding.

Definition 11 (Non-Decreasing Variable). Variable x is *non-decreasing* between process trees t and t' if there is a renaming σ such that $t\sigma \preceq t'$ and $(x \mapsto x) \in \sigma$.

Example 2. Consider the two process trees in Fig. 4 that are produced by our level 1 transformer for the expressions *append xs* (*Cons x Nil*) and *append*

($append\ xs'$ $(Cons\ x'\ Nil))$ $(Cons\ x\ Nil)$ respectively. The renaming $\{f \mapsto f', x \mapsto x, xs \mapsto xs'\}$ can be applied to process tree (1) so that it is embedded in process tree (2) by the relation $\preceq$. The variable x is therefore non-decreasing between these two process trees.

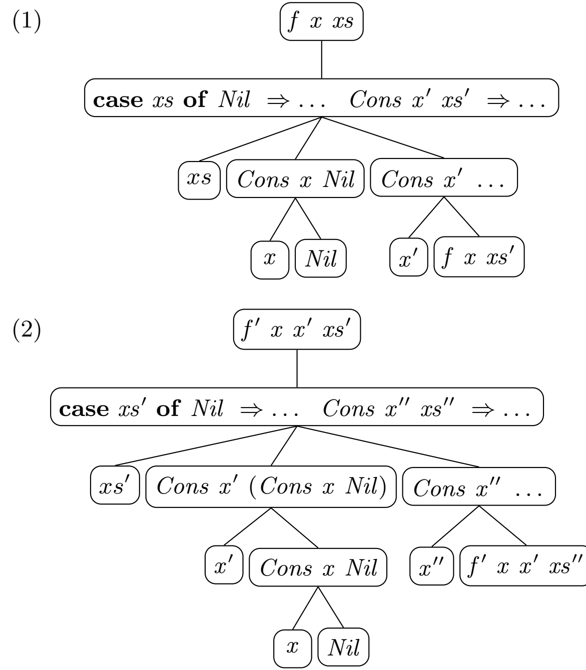

Fig. 4. Embedded Process Trees

The generalisation of a process tree involves replacing sub-trees with generalisation variables and creating *process tree substitutions*.

Definition 12 (Process Tree Substitution). We use the notation $\varphi = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ to denote a *process tree substitution*. If t is an process tree, then $t\varphi = t\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ is the result of simultaneously substituting the sub-trees $t_1, \ldots, t_n$ for the corresponding tree variables $X_1, \ldots, X_n$, respectively, in the process tree t while ensuring that bound variables are renamed appropriately to avoid name capture.

Definition 13 (Process Tree Instance). Process tree t' is an *instance* of process tree t if there is a process tree substitution φ such that $t\varphi \cong t'$.

Definition 14 (Generalisation). A *generalisation* of process trees t and t' is a triple (t_g, φ, φ') where φ and φ' are process tree substitutions such that $t_g\varphi \cong t$ and $t_g\varphi' \cong t'$.

Definition 15 (Most Specific Generalisation). A *most specific generalisation* of process trees t and t' is a generalisation $(t_g, \varphi_1, \varphi_2)$ such that for every other generalisation $(t'_g, \varphi'_1, \varphi'_2)$ of t and t', t_g is an instance of t'_g. When a process tree is generalised, sub-trees within it are replaced with variables which implies a loss of information, so the most specific generalisation therefore entails the least possible loss of information.

Definition 16 (The Generalisation Operator $\sqcap$). The most specific generalisation of two process trees t and t', where t and t' are related by $\preceq$, is given by $t \sqcap t'$, where the following rewrite rules are repeatedly applied to the initial triple $(X, \{X \mapsto t\}, \{X \mapsto t'\})$:

$$\left(\begin{array}{c} t_g, \\ \{X \mapsto (\phi(e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n)\} \cup \varphi, \\ \{X \mapsto (\phi(e'_1 \ldots e'_n) \rightarrow t'_1, \ldots, t'_n)\} \cup \varphi' \end{array}\right)$$
$$\Downarrow$$
$$\left(\begin{array}{c} t_g\{X \mapsto (\phi(e'_1 \ldots e'_n) \rightarrow X_1, \ldots, X_n)\}, \\ \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\} \cup \varphi, \\ \{X_1 \mapsto t'_1, \ldots, X_n \mapsto t'_n\} \cup \varphi' \end{array}\right)$$

$$(t_g, \{X \mapsto t, X' \mapsto t\} \cup \varphi, \{X \mapsto t', X' \mapsto t'\} \cup \varphi')$$
$$\Downarrow$$
$$(t_g\{X \mapsto X'\}, \{X' \mapsto t\} \cup \varphi, \{X' \mapsto t'\} \cup \varphi')$$

In the first rule, if the process trees associated with the same variable in each environment have the same top-level syntactic constructor ϕ, then the root node of one of the process trees is added into the generalised tree. New generalisation variables are then added for the corresponding sub-trees of these root nodes. Note that it does not matter which of the original two process trees the expressions in the resulting generalised process tree come from, so long as they all come from one of them (so the corresponding unfold and fold nodes still match); the resulting residualised program will be the same. The second rule identifies common substitutions that were previously given different names.

Theorem 1 (Most Specific Generalisation). If process trees t and t' are related by $\preceq$, then the generalisation procedure $t \sqcap t'$ terminates and calculates the most specific generalisation.

Proof. To prove that the generalisation procedure terminates, we show that within each rewrite rule, either the size of the environments φ and φ' is reduced, or the size of the terms contained in these environments is reduced. Since the values are well-founded, the rewrite rules can only be applied finitely many times.

The proof that the result of the procedure is indeed a generalisation is by induction. The initial triple is trivially a generalisation, and for each of the rewrite rules, if the input triple is a generalisation, then the output triple must also be a generalisation.

The proof that the result of the procedure is a most specific generalisation is by contradiction. If the resulting triple $(t, \varphi_1, \varphi_2)$ is not a most specific generalisation, then there must exist a most specific generalisation $(t', \varphi'_1, \varphi'_2)$ and a tree substitution φ such that $t\varphi \cong t'$, but no tree substitution φ' such that $t'\varphi' \cong t$. This will be the case if either φ is not a renaming, so contains a substitution of the form $X \mapsto (\phi(e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n$, or it identifies two variables within t. In the first case, the first rewrite rule would have been applied to further generalise, and in the second case, the second rewrite rule would have been applied to identify the variables. Thus there is a contradiction, so the generalisation computed by the procedure must be the most specific.

Example 3. The result of generalising the two process trees in Fig. 4 is shown in Fig. 5, with the mismatched nodes replaced by the generalisation variable X.

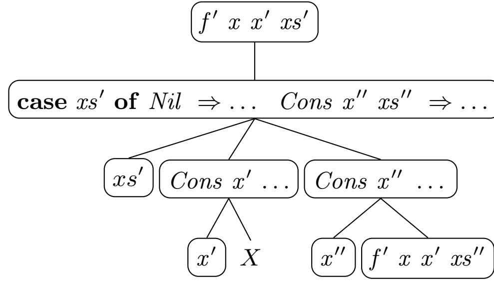

Fig. 5. Generalised Process Tree

Definition 17 (Generalisation Node Construction). The construction of a generalisation node for process tree t', where there is a process tree t such that $t \preceq t'$, is given by $t \uparrow t'$, which is defined as follows.

$$
\begin{array}{l}
t \uparrow t' = (x_0\ x_1 \ldots x_n) \rightarrow t_0, t_1, \ldots, t_n \\
\text{where } t \sqcap t' = (t'_0, \{X_1 \mapsto t'_1, \ldots, X_n \mapsto t'_n\}, \{X_1 \mapsto t''_1, \ldots, X_n \mapsto t''_n\}) \\
\qquad \{x_0 \mapsto t_0, x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} = \mathcal{G}^n(t \sqcap t')
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}^0(t_0, \{\}, \{\}) = \{x_0 \mapsto t_0\} \ (x_0 \text{ is fresh}) \\
\mathcal{G}^k(t_0, \{X_k \mapsto x\ x_1\ \ldots x_n\} \cup \varphi, \{X_k \mapsto t\} \cup \varphi') = \\
\quad \{x \mapsto \lambda x_1 \ldots x_n.t\} \cup \mathcal{G}^{k-1}(t_0\{X_k \mapsto (x\ x_1 \ldots x_n) \rightarrow \epsilon\}, \varphi, \varphi'), \\
\hfill \text{if } x \text{ is non-decreasing} \\
\mathcal{G}^k(t_0, \{X_k \mapsto t\} \cup \varphi, \{X_k \mapsto t'\} \cup \varphi') = \\
\quad \{x_k \mapsto \lambda x_1 \ldots x_n.t'\} \cup \mathcal{G}^{k-1}(t_0\{X \mapsto (x_k\ x_1 \ldots x_n) \rightarrow \epsilon\}, \varphi, \varphi') \ (x_k \text{ is fresh}) \\
\quad \text{where } \{x_1 \ldots x_n\} = fv(t)
\end{array}
$$

The rules $\mathcal{G}^n$ return an environment $\{x_0 \mapsto t_0, x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ from which the corresponding generalisation node $(x_0\ x_1 \ldots x_n) \rightarrow t_0, t_1, \ldots, t_n$ is

constructed. The rules are applied to the triple (t_0, φ, φ') resulting from the generalisation of the process trees t and t'. They work through each of the corresponding generalisation variables in the environments φ and φ' in turn. In the first rule, when both generalisation environments are exhausted, we are left with the generalised process tree t_0 in which appropriate values have been substituted for the generalisation variables; this is associated with the fresh variable x_0. In the final rule, the extracted sub-tree is abstracted over its variables (so that these are not extracted outside of their binders), and associated with the fresh variable x_k; the generalisation variable in the generalised process tree is replaced with a corresponding application of x_k. In the second rule, if we have an instance of the application of the variable x, and x is non-decreasing, then the same variable is reused in the generalisation.

Example 4. The result of applying the generalisation node constructor to the result of generalising the two process trees in Fig. 4 is shown in Fig. 6.

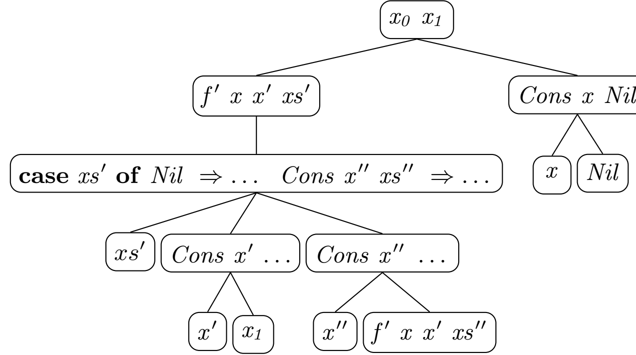

Fig. 6. Generalised Process Tree With Generalisation Node

We now show how a program can be residualised from a process tree.

Definition 18 (Residualisation). A program can be residualised from a process tree t as $\mathcal{R}_p[\![t]\!]$ using the rules as shown in Fig. 7.

Within the rules $\mathcal{R}_e$, the parameter ρ contains the unfold node function headers and the corresponding new function headers that are created for them. The rules return a residual expression along with a set of newly created function definitions. In rule (2), on encountering an unfold node, a new function header is created, associated with the unfold node function header, and added to ρ. Note that this new function header may not have the same variables as the one in the unfold node, as new variables may have been added to the sub-tree as a result of generalisation. In rule (3), on encountering a fold node, a recursive call

$$
\begin{array}{l}
(1)\ \mathcal{R}_p[\![t]\!] = e_0 \textbf{ where } h_1 = e_1, \ldots, h_n = e_n \\
\quad \text{where } \mathcal{R}_e[\![t]\!]\ \{\} = (e_0, \{h_1 = e_1, \ldots, h_n = e_n\}) \\
\\
(2)\ \mathcal{R}_e[\![h \to t]\!]\ \rho = (h', \{h' = e\} \cup \Delta) \\
\quad \text{where } \mathcal{R}_e[\![t]\!]\ (\rho \cup \{h = h'\}) = (e, \Delta) \\
\qquad\qquad h' = f\ x_1 \ldots x_n\ (f \text{ is fresh}, \{x_1 \ldots x_n\} = fv(t)) \\
(3)\ \mathcal{R}_e[\![h \to \epsilon]\!]\ \rho = (h''\sigma, \{\}) \\
\quad \text{where } (h' = h'') \in \rho \wedge h \equiv h'\sigma \\
(4)\ \mathcal{R}_e[\![x \to \epsilon]\!]\ \rho = (x, \{\}) \\
(5)\ \mathcal{R}_e[\![(c\ e_1 \ldots e_n) \to t_1, \ldots, t_n]\!]\ \rho = (c\ e'_1 \ldots e'_n, \biguplus_{i=1}^{n} \Delta_i) \\
\quad \text{where } \forall i \in \{1 \ldots n\}.\mathcal{R}_e[\![t_i]\!]\ \rho = (e'_i, \Delta_i) \\
(6)\ \mathcal{R}_e[\![(\lambda x.e) \to t]\!]\ \rho = (\lambda x.e', \Delta) \\
\quad \text{where } \mathcal{R}_e[\![t]\!]\ \rho = (e', \Delta) \\
(7)\ \mathcal{R}_e[\![(e_0\ \ e_1) \to t_0, t_1]\!]\ \rho = (e'_0\ e'_1, \biguplus_{i=1}^{2} \Delta_i) \\
\quad \text{where } \forall i \in \{0 \ldots 1\}.\mathcal{R}_e[\![t_i]\!]\ \rho = (e'_i, \Delta_i) \\
(8)\ \mathcal{R}_e[\![(\textbf{case } e_0 \textbf{ of } p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n) \to t_0, \ldots, t_n]\!]\ \rho = \\
\quad (\textbf{case } e'_0 \textbf{ of } p_1 \Rightarrow e'_1 \ldots p_n \Rightarrow e'_n, \biguplus_{i=0}^{n} \Delta_i) \\
\quad \text{where } \forall i \in \{0 \ldots n\}.\mathcal{R}_e[\![t_i]\!]\ \rho = (e'_i, \Delta_i) \\
(9)\ \mathcal{R}_e[\![(x_0\ x_1 \ldots x_n) \to t_0, t_1, \ldots, t_n]\!]\ \rho = e_0\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}, \biguplus_{i=0}^{n} \Delta_i) \\
\quad \text{where } \forall i \in \{0 \ldots n\}.\mathcal{R}_e[\![t_i]\!]\ \rho = (e_i, \Delta_i)
\end{array}
$$

Fig. 7. Rules For Residualisation

of the function associated with the unfold node function header in ρ is created. In rule (9), on encountering a generalisation node, the sub-trees of the node are residualised separately, and then the expressions residualised from the extracted sub-trees $t_1 \ldots t_n$ are substituted back into the result of residualising the main body t_0.

Example 5. The program shown in Fig. 8 is obtained by applying the residualisation rules to the process tree shown in Fig. 6.

$$
\begin{array}{l}
f\ x'\ xs'\ (Cons\ x\ Nil) \\
\textbf{where} \\
f\ x\ xs\ ys = \textbf{case } xs \textbf{ of} \\
\qquad\qquad\qquad Nil \qquad\quad\ \Rightarrow Cons\ x\ ys \\
\qquad\qquad\qquad Cons\ x'\ xs' \Rightarrow Cons\ x'\ (f\ x\ xs'\ ys)
\end{array}
$$

Fig. 8. Result of Residualisation

4 A Hierarchy of Program Transformers

In this section, we define our hierarchy of transformers. The level k transformer is defined as $\mathcal{T}_p^k[\![p]\!]$, where p is the program to be transformed. It is assumed that the input program contains no λ-abstractions; these can be replaced by named functions. The output of the transformer is a process tree from which the transformed program can be residualised.

4.1 Level 0 Transformer

Level 0 in our hierarchy just maps a program to a corresponding process tree without performing any reductions as shown in Fig. 9.

$$
\begin{array}{l}
(1)\ \mathcal{T}_p^0[\![e_0 \ \textbf{where}\ h_1 = e_1, \ldots, h_n = e_n]\!] = \mathcal{T}_e^0[\![e_0]\!]\ \{\}\ \{h_1 = e_1, \ldots, h_n = e_n\} \\[1em]
(2)\ \mathcal{T}_e^0[\![x]\!]\ \rho\ \Delta = x \rightarrow \epsilon \\
(3)\ \mathcal{T}_e^0[\![c\ e_1 \ldots e_n]\!]\ \rho\ \Delta = (c\ e_1 \ldots e_n) \rightarrow (\mathcal{T}_e^0[\![e_1]\!]\ \rho\ \Delta), \ldots, (\mathcal{T}_e^0[\![e_n]\!]\ \rho\ \Delta) \\
(4)\ \mathcal{T}_e^0[\![\lambda x.e]\!]\ \rho\ \Delta = (\lambda x.e) \rightarrow (\mathcal{T}_e^0[\![e]\!]\ \rho\ \Delta) \\
(5)\ \mathcal{T}_e^0[\![f]\!]\ \rho\ \Delta = \begin{cases} f \rightarrow \epsilon, & \text{if } f \in \rho \\ f \rightarrow (\mathcal{T}_e^0[\![\lambda x_1 \ldots x_n.e]\!]\ (\rho \cup \{f\})\ \Delta), & \text{otherwise} \\ \text{where } (f\ x_1 \ldots x_n = e) \in \Delta & \end{cases} \\
(6)\ \mathcal{T}_e^0[\![e_0\ e_1]\!]\ \rho\ \Delta = (e_0\ e_1) \rightarrow (\mathcal{T}_e^0[\![e_0]\!]\ \rho\ \Delta), (\mathcal{T}_e^0[\![e_1]\!]\ \rho\ \Delta) \\
(7)\ \mathcal{T}_e^0[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_k \Rightarrow e_k]\!]\ \rho\ \Delta = \\
\qquad (\textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_k \Rightarrow e_k) \rightarrow (\mathcal{T}_e^0[\![e_0]\!]\ \rho\ \Delta), \ldots, (\mathcal{T}_e^0[\![e_k]\!]\ \rho\ \Delta)
\end{array}
$$

Fig. 9. Level 0 Transformation Rules

Within the rules $\mathcal{T}_e^0$, ρ is the set of previously encountered function calls and Δ is the set of function definitions. If a function call is re-encountered, no further nodes are added to the process tree. Thus, the constructed process tree will always be a finite representation of the program.

4.2 Level $k+1$ Transformers

Each subsequent level $(k+1)$ in our hierarchy is built on top of the previous levels. The rules for level $k+1$ transformation of program p are defined by $\mathcal{T}_p^{k+1}[\![p]\!]$ as shown in Fig. 10.

Within these rules, κ denotes the context of the expression under scrutiny and ρ contains memoised process trees and their associated new function headers. For most of the level $k+1$ transformation rules, normal order reduction is applied to the current term, as for the semantics given in Fig. 3.

In rule (3), if the context surrounding a variable redex is a **case**, then information is propagated to each branch of the **case** to indicate that this variable has the value of the corresponding branch pattern.

$$
(1)\quad \mathcal{T}_p^{k+1}[\![e_0 \textbf{ where } h_1 = e_1, \ldots, h_n = e_n]\!] = \mathcal{T}_e^{k+1}[\![e_0]\!]\ \langle\rangle\ \{\}\ \{h_1 = e_1, \ldots, h_n = e_n\}
$$

$$
\begin{array}{ll}
(2) & \mathcal{T}_e^{k+1}[\![x]\!]\ \langle\rangle\ \rho\ \Delta = x \rightarrow \epsilon \\
(3) & \mathcal{T}_e^{k+1}[\![x]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n) : \kappa\rangle\ \rho\ \Delta = \\
 & \quad \mathcal{T}_\kappa^{k+1}[\![x \rightarrow \epsilon]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow (\kappa\bullet e_1)\{x \mapsto p_1\} \ldots p_n \Rightarrow (\kappa\bullet e_n)\{x \mapsto p_n\}) : \langle\rangle\rangle\ \rho\ \Delta \\
(4) & \mathcal{T}_e^{k+1}[\![x]\!]\ \langle(\bullet\ e) : \kappa\rangle\ \rho\ \Delta = \mathcal{T}_\kappa^{k+1}[\![x \rightarrow \epsilon]\!]\ \langle(\bullet\ e) : \kappa\rangle\ \rho\ \Delta \\
(5) & \mathcal{T}_e^{k+1}[\![c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \Delta = \\
 & \quad (c\ e_1 \ldots e_n) \rightarrow (\mathcal{T}_e^{k+1}[\![e_1]\!]\ \langle\rangle\ \rho\ \Delta), \ldots, (\mathcal{T}_e^{k+1}[\![e_n]\!]\ \langle\rangle\ \rho\ \Delta) \\
(6) & \mathcal{T}_e^{k+1}[\![c\ e_1 \ldots e_n]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1' \ldots p_k \Rightarrow e_k') : \kappa\rangle\ \rho\ \Delta = \\
 & \quad \mathcal{T}_e^{k+1}[\![e_i'\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa\ \rho\ \Delta \\
 & \quad \text{where } \exists i \in \{1 \ldots k\}.p_i = c\ x_1 \ldots x_n \\
(7) & \mathcal{T}_e^{k+1}[\![\lambda x.e_0]\!]\ \langle\rangle\ \rho\ \Delta = (\lambda x.e_0) \rightarrow (\mathcal{T}_e^{k+1}[\![e_0]\!]\ \langle\rangle\ \rho\ \Delta) \\
(8) & \mathcal{T}_e^{k+1}[\![\lambda x.e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \Delta = \mathcal{T}_e^{k+1}[\![e_0\{x \mapsto e_1\}]\!]\ \kappa\ \rho\ \Delta \\
(9) & \mathcal{T}_e^{k+1}[\![f]\!]\ \kappa\ \rho\ \Delta = \left\{ \begin{array}{ll} h\sigma \rightarrow \epsilon, & \text{if } \exists(h = t') \in \rho, \sigma.t'\sigma \cong t \\ \mathcal{T}_\varphi^{k+1}[\![t'\sigma \uparrow t]\!], & \text{if } \exists(h = t') \in \rho, \sigma.t'\sigma \preceq t \\ h \rightarrow \mathcal{T}_e^{k+1}[\![\lambda x_1 \ldots x_n.e]\!]\ \kappa\ (\rho \cup \{h = t\})\ \Delta, & \text{otherwise} \\ \text{where } (f\ x_1 \ldots x_n = e) \in \Delta & \\ \qquad\quad h = f'\ x_1' \ldots x_k'\ (f' \text{ is fresh, } \{x_1' \ldots x_k'\} = fv(t)) & \end{array} \right. \\
 & \qquad\qquad\qquad\qquad \text{where } t = \mathcal{T}_e^{k}[\![f]\!]\ \kappa\ \{\}\ \Delta \\
(10) & \mathcal{T}_e^{k+1}[\![e_0\ e_1]\!]\ \kappa\ \rho\ \Delta = \mathcal{T}_e^{k+1}[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \Delta \\
(11) & \mathcal{T}_e^{k+1}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n]\!]\ \kappa\ \rho\ \Delta = \\
 & \quad \mathcal{T}_e^{k+1}[\![e_0]\!]\ \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n) : \kappa\rangle\ \rho\ \Delta
\end{array}
$$

$$
\begin{array}{ll}
(12) & \mathcal{T}_\kappa^{k+1}[\![t]\!]\ \langle\rangle\ \rho\ \Delta = t \\
(13) & \mathcal{T}_\kappa^{k+1}[\![t]\!]\ (\kappa = \langle(\bullet\ e) : \kappa'\rangle)\ \rho\ \Delta = \\
 & \quad \mathcal{T}_\kappa^{k+1}[\![(\kappa\bullet root(t)) \rightarrow t, (\mathcal{T}_e^{k+1}[\![e]\!]\ \langle\rangle\ \rho\ \Delta)]\!]\ \kappa'\ \rho\ \Delta \\
(14) & \mathcal{T}_\kappa^{k+1}[\![t]\!]\ (\kappa = \langle(\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n) : \kappa'\rangle)\ \rho\ \Delta = \\
 & \quad (\kappa\bullet root(t)) \rightarrow t, (\mathcal{T}_e^{k+1}[\![e_1]\!]\ \kappa'\ \rho\ \Delta), \ldots, (\mathcal{T}_e^{k+1}[\![e_n]\!]\ \kappa'\ \rho\ \Delta)
\end{array}
$$

$$
\begin{array}{ll}
(15) & \mathcal{T}_\varphi^{k+1}[\![(x_0\ x_1 \ldots x_n) \rightarrow t_0, t_1, \ldots, t_n]\!] = (x_0\ x_1 \ldots x_n) \rightarrow t_0', t_1', \ldots, t_n' \\
 & \quad \text{where } \forall i \in \{0 \ldots n\}.t_i' = \mathcal{T}_p^{k+1}[\![\mathcal{R}_p[\![t_i]\!]]\!]
\end{array}
$$

Fig. 10. Level $k+1$ Transformation Rules

In rule (6), if the context surrounding a constructor application redex is a **case**, then pattern matching is performed and the appropriate branch of the **case** is selected, thus removing the constructor application. This is where our transformers actually remove intermediate data structures.

In rule (9), if the redex of the current term is a function, then it is transformed by the transformer one level lower in the hierarchy (level k) producing a process tree; this is therefore where the transformer builds on all the transformers at lower levels. This level k process tree is compared to the previous process trees produced at level k (contained in ρ). If the process tree is a *renaming* of a previous one, then *folding* is performed, and a fold node is created using a recursive call of the function associated with the renamed process tree in ρ. If the process tree is an *embedding* of a previous one, then *generalisation* is performed; the result of

this generalisation is then further transformed. Otherwise, the current process tree is memoised by being associated with a new function call in ρ; an unfold node is created with this new function call in the root node, with the result of transforming the unfolding of the current term as its sub-tree.

The rules $\mathcal{T}_{\kappa}^{k+1}$ are defined on a process tree and a surrounding context. These rules are applied when the normal-order reduction of the input program becomes 'stuck' as a result of encountering a variable in the redex position. In this case, the surrounding context is further transformed.

The rule $\mathcal{T}_{\varphi}^{k+1}$ is applied to a newly constructed generalisation node; all the sub-trees of the node are residualised and further transformed.

5 Examples

In this section, we look at some examples of the transformations that can be performed at different levels in our program transformation hierarchy.

Example 6. Consider the following program from [6]:

$$\begin{array}{l} f\ x\ x \\ \textbf{where} \\ f\ x\ y = \textbf{case}\ x\ \textbf{of} \\ \qquad\qquad \begin{array}{ll} Zero & \Rightarrow y \\ Succ(x) & \Rightarrow f\ (f\ x\ x)\ (f\ x\ x) \end{array} \end{array}$$

This program takes exponential time $O(2^n)$, where n is the size of the input value x. If we transform this program at level 1 in our hierarchy, we obtain the following program:

$$\begin{array}{l} f\ x \\ \textbf{where} \\ f\ x = \textbf{case}\ x\ \textbf{of} \\ \qquad\qquad \begin{array}{ll} Zero & \Rightarrow Zero \\ Succ(x) & \Rightarrow f\ (f\ x) \end{array} \end{array}$$

This program takes linear time $O(n)$ on the same input, so an exponential speedup has been achieved. If we transform the original program at level 2 in our hierarchy, we obtain the following program:

$$\begin{array}{l} f\ x \\ \textbf{where} \\ f\ x = \textbf{case}\ x\ \textbf{of} \\ \qquad\qquad \begin{array}{ll} Zero & \Rightarrow Zero \\ Succ(x) & \Rightarrow f\ x \end{array} \end{array}$$

A very slight further improvement has therefore been obtained. No further improvements are obtained at higher levels in the hierarchy.

Example 7. Consider the transformation of the naïve reverse program shown in Fig. 1, which has $O(n^2)$ runtime where n is the length of the list xs. If we transform this program at level 1 in our hierarchy, we obtain the following program:

```
f xs
where
f xs    = case xs of
            Nil        ⇒ Nil
            Cons x xs ⇒ case (f xs) of
                           Nil        ⇒ Cons x Nil
                           Cons x' xs ⇒ Cons x' (f' xs x)
f' xs x = case xs of
            Nil        ⇒ Cons x Nil
            Cons x' xs ⇒ Cons x' (f' xs x)
```

There is very little improvement in the performance of this program over the original; it still has $O(n^2)$ runtime. However, if we transform the naïve reverse program at level 2 in our hierarchy, we obtain the following program:

```
case xs of
  Nil        ⇒ Nil
  Cons x xs ⇒ f xs x Nil
where
f xs x ys = case xs of
              Nil        ⇒ Cons x ys
              Cons x' xs ⇒ f xs x' (Cons x ys)
```

This program takes linear time $O(n)$ on the same input, so a superlinear speedup has been achieved. No further improvements are obtained at higher levels in the hierarchy.

Example 8. Consider the following program:

```
map inc (qrev xs)
where
map f xs    = case xs of
                Nil        ⇒ Nil
                Cons x xs ⇒ Cons (f x) (map f xs)
inc n       = Succ n
qrev xs     = qrev' xs Nil
qrev' xs ys = case xs of
                Nil        ⇒ ys
                Cons x xs ⇒ qrev' xs (Cons x ys)
```

This program requires $2n$ allocations, where n is the length of the list xs. If we transform the original program at level 1 in our hierarchy, we obtain the following program:

$$
\begin{array}{l}
f\ xs\ Nil \\
\mathbf{where} \\
\begin{array}{lll}
f\ xs\ ys & = \mathbf{case}\ xs\ \mathbf{of} & \\
& \quad Nil & \Rightarrow f'\ ys \\
& \quad Cons\ x\ xs & \Rightarrow f\ xs\ (Cons\ x\ ys) \\
f'\ xs & = \mathbf{case}\ xs\ \mathbf{of} & \\
& \quad Nil & \Rightarrow Nil \\
& \quad Cons\ x\ xs & \Rightarrow Cons\ (Succ\ x)\ (f'\ xs)
\end{array}
\end{array}
$$

This program also requires $2n$ allocations, and not much improvement has been made. If we transform the original program at level 2 in our hierarchy, we obtain the following program:

$$
\begin{array}{l}
f\ xs\ Nil\ (\lambda xs.f'\ xs) \\
\mathbf{where} \\
\begin{array}{lll}
f\ xs\ ys\ g & = \mathbf{case}\ xs\ \mathbf{of} & \\
& \quad Nil & \Rightarrow g\ ys \\
& \quad Cons\ x\ xs & \Rightarrow f\ xs\ (Cons\ x\ ys)\ g \\
f'\ xs & = \mathbf{case}\ xs\ \mathbf{of} & \\
& \quad Nil & \Rightarrow Nil \\
& \quad Cons\ x\ xs & \Rightarrow Cons\ (Succ\ x)\ (f'\ xs)
\end{array}
\end{array}
$$

This program still requires $2n$ allocations, so again not much improvement has been made. However, if we transform this program at level 3 in our hierarchy, we obtain the following program:

$$
\begin{array}{l}
f\ xs\ (\lambda xs.xs) \\
\mathbf{where} \\
\begin{array}{lll}
f\ xs\ g & = \mathbf{case}\ xs\ \mathbf{of} & \\
& \quad Nil & \Rightarrow g\ Nil \\
& \quad Cons\ x\ xs & \Rightarrow f\ xs\ (\lambda xs.Cons\ (Succ\ x)\ (g\ xs))
\end{array}
\end{array}
$$

This program now requires n allocations, so we can see that improvements can still be made as high as level 3 in our hierarchy (and indeed even higher in some cases). For this example, no further improvements are obtained at higher levels in the hierarchy.

6 Speedups

In this section, we look at the efficiency gains that can be obtained at different levels in our program transformation hierarchy.

Theorem 2 (Exponential Speedups). Exponential speedups can only be obtained above level 0 in our hierarchy if common sub-expression elimination is performed during generalisation.

Proof. An exponential speedup can only be obtained if a number of repeated computations are identified, so the computation need only be performed once. This can only happen in our transformations if the repeated computations are identified by the common sub-expression elimination that takes place during generalisation.

If we consider the transformation of the program at level 1 in our hierarchy given in Example 6, the term $(f\ x\ x)$ is extracted twice during generalisation, but then identified by common sub-expression elimination, thus allowing an exponential speedup to be achieved. In practice, we have found that such exponential improvements are obtained for very few useful programs; it is very uncommon for the same computation to be extracted more than once during generalisation to facilitate this improvement. It is also very unlikely that a programmer would write such an inefficient program when a much better solution exists.

We now look at the improvements in efficiency that can be obtained without common sub-expression elimination.

Theorem 3 (Non-Exponential Speedups). Without the use of common sub-expression elimination, the maximum speedup factor possible at level $k > 0$ in our hierarchy for input of size n is $O(n^{k-1})$.

Proof. The proof is by induction on the hierarchy level k. For level 1, the proof is as given in [24]; since there can only be a constant number of reduction steps removed between each successive call of a function, at most a linear speedup is possible. For level $k+1$, there will be a constant number of calls to functions that were transformed at level k between each successive call of a level $k+1$ function. By the inductive hypothesis, the maximum speedup factor for each level k function is $O(n^{k-1})$, so the maximum speedup factor at level $k+1$ is $O(n^k)$.

Consider the transformation of the naïve reverse program at level 2 in our hierarchy given in Example 7. During this transformation, we end up having to transform a term equivalent to the following at level 1:

$$\mathit{append}\ (\mathit{append}\ \mathit{xs}'\ (\mathit{Cons}\ x'\ \mathit{Nil}))\ (\mathit{Cons}\ x\ \mathit{Nil})$$

Within this term, the list xs' has to be traversed twice. This term is transformed to one equivalent to the following at level 1 (process tree (2) in Fig. 4 is the process tree produced as a result of this transformation):

$$\mathit{append}\ \mathit{xs}'\ (\mathit{Cons}\ x'\ (\mathit{Cons}\ x\ \mathit{Nil}))$$

Within this term, the list xs' has only to be traversed once, so a linear speedup has been obtained. This linear improvement will be made between each successive call of the naïve reverse function, thus giving an overall superlinear speedup and producing the resulting accumulating reverse program.

7 Termination

In order to prove that each of the transformers in our hierarchy terminate, we need to show that in any infinite sequence of process trees encountered during transformation $t_0, t_1, \ldots$ there definitely exists some $i < j$ where $t_i \preceq t_j$, so an embedding must eventually be encountered and transformation will not continue indefinitely without folding or generalising. This amounts to proving that the embedding relation $\preceq$ is a *well-quasi order*.

Lemma 1 ($\preceq$ is a Well-Quasi Order). The embedding relation $\preceq$ is a *well-quasi order* on any sequence of process trees that are encountered during transformation at level $k > 0$ in our hierarchy.

Proof. The proof is by induction on the hierarchy level k.

For level 1, the proof is similar to that given in [15]. This involves showing that there are a finite number of syntactic constructors in the language. The process trees encountered during transformation are those produced at level 0, so the function names will be those from the original program, so must be finite. Applications of different arities are replaced with separate constructors; we prove that arities are bounded, so there are a finite number of these. We also replace **case** expressions with constructors. Since bound variables are defined using de Bruijn indices, each of these are replaced with separate constructors; we also prove that de Bruijn indices are bounded. The overall number of syntactic constructors is therefore finite, so Kruskal's tree theorem can then be applied to show that $\preceq$ is a well-quasi-order at level 1 in our hierarchy.

At level $k+1$, the process trees encountered during transformation are those produced at level k and must be finite (by the inductive hypothesis). The number of functions in these process trees must therefore be finite, and the same argument given above for level 1 also applies here, so $\preceq$ is a well-quasi-order at level $k+1$ in our hierarchy.

Since we only check for embeddings for expressions which have a named function as redex, we need to show that every potentially infinite sequence of expressions encountered during transformation must include expressions of this form.

Lemma 2 (Function Unfolding During Transformation). Every infinite sequence of transformation steps must include function unfolding.

Proof. Every infinite sequence of transformation steps must include either function unfolding or λ-application. Since we do not allow λ-abstractions in our input program, the only way in which new λ-abstractions can be introduced is by function unfolding. Thus, every infinite sequence of transformation steps must include function unfolding.

Theorem 4 (Termination of Transformation). The transformation algorithm always terminates.

Proof. The proof is by contradiction. If the transformation algorithm did not terminate, then the set of memoised process trees in ρ must be infinite. Every new process tree which is added to ρ cannot have any of the previous process trees in ρ embedded within it by the homeomorphic embedding relation $\preceq$, since generalisation would have been performed instead. However, this contradicts the fact that $\preceq$ is a well-quasi-order (Lemma 1).

8 Related Work

The seminal work corresponding to level 1 in our hierarchy is that of Turchin on supercompilation [27], although our level 1 transformer more closely resembles positive supercompilation [26]. There have been several previous attempts to move beyond level 1 in our transformation hierarchy, the first one by Turchin himself using *walk grammars* [28]. In this approach, traces through residual graphs are represented by regular grammars that are subsequently analysed and simplified. This approach is also capable of achieving superlinear speedups, but no automatic procedure is defined for it; the outlined heuristics and strategies may not terminate.

A hierarchy of program specialisers is described in [7] that shows how programs can be metacoded and then manipulated through a *metasystem transition*, with a number of these metasystem transitions giving a metasytem hierarchy in which the original program may have several levels of metacoding. In the work described here, a process tree can be considered to be the metacoding of a program. However, we do not have the difficulties associated with metasystem transitions and multi-level metacoding, as our process trees are residualised back to the object level.

Distillation [9, 11] is built on top of positive supercompilation, so corresponds to level 2 in our hierarchy, but does not go beyond this level. Klyuchnikov and Romanenko [16] construct a hierarchy of supercompilers in which lower level supercompilers are used to prove lemmas about term equivalences, and higher level supercompilers utilise these lemmas by rewriting according to the term equivalences (similar to the "second order replacement method" defined by Kott [18]). Transformers in this hierarchy are capable of similar speedups to those in our hierarchy, but no automatic procedure is defined for it; the need to find and apply appropriate lemmas introduces infinite branching into the search space, and various heuristics have to be used to try to limit this search.

Preliminary work on the hierarchy of transformers defined here was presented in [10]; this did not include analysis of the efficiency improvements that can be made at each level in the hierarchy. The work described here is a lot further developed than that described in [10], and we hope simpler and easier to follow.

Logic program transformation is closely related, and the equivalence of partial deduction and driving (as used in supercompilation) has been argued by Glück and Sørensen [8]. Superlinear speedups can be achieved in logic program transformation by *goal replacement* [22, 23]: replacing one logical clause with another to facilitate folding. Techniques similar to the notion of "higher level

supercompilation" [16] have been used to prove correctness of goal replacement, but have similar problems regarding the search for appropriate lemmas.

9 Conclusion and Further Work

We have presented a hierarchy of program transformers, capable of performing fusion to eliminate intermediate data structures, in which the transformer at each level of the hierarchy builds on top of those at lower levels. We have proved that the transformers at each level in the hierarchy terminate, and have characterised the speedups that can be obtained at each level. Previous works [17, 2, 1, 31, 24] have noted that the unfold/fold transformation methodology is incomplete; some programs cannot be synthesised from each other. It is our hope that this work will help to overcome this restriction.

There are many possible avenues for further work. Firstly, we need to determine what level in the hierarchy is sufficient to optimise a program as much as is possible using this approach. We have seen that it is not sufficient to just transform a program until no further improvement is obtained; improvements may still be still possible at higher levels. We would therefore like to find some analysis technique which would allow us to determine what level in the hierarchy is required. Ultimately, we would like the process trees produced by our transformers to be in what we call *distilled form* $t^{\{\}}$, which is defined as follows:

$$\begin{array}{ll} t^\rho ::= & (x_0\ x_1 \ldots x_n) \rightarrow t_0^{(\rho \cup \{x_1,\ldots,x_n\})}, t_1^\rho, \ldots, t_n^\rho \\ & \mid (\mathbf{case}\ x_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n) \rightarrow (x_0 \rightarrow \epsilon), t_1^\rho, \ldots, t_n^\rho\ (x_0 \notin \rho) \\ & \mid \phi(e_1 \ldots e_n) \rightarrow t_1^\rho, \ldots, t_n^\rho \end{array}$$

Within this definition, generalisation variables are added to the set ρ, and cannot be used in the selectors of **case** expressions, so the resulting programs must not create any intermediate data structures. Each of the example programs that we transformed using our transformation hierarchy are ultimately transformed into distilled form before no further improvement is obtained. We could therefore apply successively higher levels in our hierarchy until a process tree in distilled form is obtained. However, at present, we have no proof that this must eventually happen. Work is continuing in this area.

If we can obtain process trees that are in distilled form, then there are many areas in which our work can be applied, as distilled form is much easier to analyse and reason about. These areas include termination analysis, computational complexity analysis, theorem proving, program verification and constructing programs from specifications. Work is also continuing in all of these areas.

Acknowledgements

The author would like to thank the anonymous referees, who provided very useful feedback and suggested improvements to this paper. This work owes a lot to the input of Neil D. Jones, who provided many useful insights and ideas on the subject matter presented here.

References

1. Amtoft, T.: Sharing of Computations. Ph.D. thesis, DAIMI, Aarhus University (1993)
2. Andersen, L.O., Gomard, C.K.: Speedup Analysis in Partial Evaluation: Preliminary Results. In: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. pp. 1–7 (1992)
3. Bol, R.: Loop Checking in Partial Deduction. Journal of Logic Programming **16**(1–2), 25–46 (1993)
4. Burstall, R., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM **24**(1), 44–67 (Jan 1977)
5. Dershowitz, N., Jouannaud, J.P.: Rewrite Systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 243–320. Elsevier, MIT Press (1990)
6. Glück, R., Klimov, A., Nepeivoda, A.: Non-linear configurations for superlinear speedup by supercompilation. In: Proceedings of the Fifth International Workshop on Metacomputation in Russia (2016)
7. Glück, R., Hatcliff, J., Jørgensen, J.: Generalization in Hierarchies of Online Program Specialization Systems. In: Workshop on Logic-Based Program Synthesis and Transformation. pp. 179–198 (1998)
8. Glück, R., Jørgensen, J.: Generating Transformers for Deforestation and Supercompilation. In: Proceedings of the Static Analysis Symposium. Lecture Notes in Computer Science, vol. 864, pp. 432–448. Springer-Verlag (1994)
9. Hamilton, G.W.: Distillation: Extracting the Essence of Programs. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 61–70 (2007)
10. Hamilton, G.W.: A Hierarchy of Program Transformers. In: Proceedings of the Second International Workshop on Metacomputation in Russia (2012)
11. Hamilton, G.W., Jones, N.D.: Distillation with labelled transition systems. In: Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation. pp. 15–24. ACM (2012)
12. Higman, G.: Ordering by Divisibility in Abstract Algebras. Proceedings of the London Mathemtical Society **2**, 326–336 (1952)
13. Huet, G.: The Zipper. Journal of Functional Programming **7**(5), 549–554 (1997)
14. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
15. Klyuchnikov, I.: Supercompiler HOSC 1.1: Proof of Termination. Preprint 21, Keldysh Institute of Applied Mathematics, Moscow (2010)
16. Klyuchnikov, I.: Towards Higher-Level Supercompilation. In: Proceedings of the Second International Workshop on Metacomputation in Russia. pp. 82–101 (2010)
17. Kott, L.: A System for Proving Equivalences of Recursive Programs. In: 5th Conference on Automated Deduction. pp. 63–69 (1980)
18. Kott, L.: Unfold/Fold Transformations. In: Nivat, M., Reynolds, J. (eds.) Algebraic Methods in Semantics, chap. 12, pp. 412–433. CUP (1985)
19. Kruskal, J.: Well-Quasi Ordering, the Tree Theorem, and Vazsonyi's Conjecture. Transactions of the American Mathematical Society **95**, 210–225 (1960)
20. Leuschel, M.: On the Power of Homeomorphic Embedding for Online Termination. In: Proceedings of the International Static Analysis Symposium. pp. 230–245 (1998)
21. Marlet, R.: Vers une Formalisation de l'Évaluation Partielle. Ph.D. thesis, Université de Nice - Sophia Antipolis (1994)

22. Pettorossi, A., Proietti, M.: A Theory of Totally Correct Logic Program Transformations. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM). pp. 159–168 (2004)
23. Roychoudhury, A., Kumar, K., Ramakrishnan, C., Ramakrishnan, I.: An Unfold/Fold Transformation Framework for Definite Logic Programs. ACM Transactions on Programming Language Systems **26**(3), 464–509 (2004)
24. Sørensen, M.H.: Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen (1994), dIKU-rapport 94/17
25. Sørensen, M.H., Glück, R.: An Algorithm of Generalization in Positive Supercompilation. Lecture Notes in Computer Science **787**, 335–351 (1994)
26. Sørensen, M.H., Glück, R., Jones, N.D.: A Positive Supercompiler. Journal of Functional Programming **6**(6), 811–838 (1996)
27. Turchin, V.F.: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems **8**(3), 90–121 (Jul 1986)
28. Turchin, V.F.: Program Transformation With Metasystem Transitions. ACM Transactions on Programming Languages and Systems **3**(3), 283–313 (1993)
29. Wadler, P.: The Concatenate Vanishes (Dec 1987), fP Electronic Mailing List
30. Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees. Lecture Notes in Computer Science **300**, 344–358 (1988)
31. Zhu, H.: How Powerful are Folding/Unfolding Transformations? Journal of Functional Programming **4**(1), 89–112 (1994)