A Graph-Based Definition of Distillation

G.W. Hamilton and G. Mendel-Gleason

School of Computing and Lero@DCU
Dublin City University
Ireland
e-mail: {*hamilton,ggleason*}*@computing.dcu.ie*

Abstract. In this paper, we give a graph-based definition of the distillation transformation algorithm. This definition is made within a similar framework to the positive supercompilation algorithm, thus allowing for a more in-depth comparison of the two algorithms. We find that the main distinguishing characteristic between the two algorithms is that in positive supercompilation, generalization and folding are performed with respect to expressions, while in distillation they are performed with respect to graphs. We also find that while only linear improvements in performance are possible using positive supercompilation, super-linear improvements are possible using distillation. This is because computationally expensive terms can only be extracted from within loops when generalizing graphs rather than expressions.

1 Introduction

Supercompilation is a program transformation technique for functional languages which can be used for program specialization and for the removal of intermediate data structures. Supercompilation was originally devised by Turchin in what was then the USSR in the late 1960s, but did not become widely known to the outside world until a couple of decades later. One reason for this delay was that the work was originally published in Russian in journals which were not accessible to the outside world; it was eventually published in mainstream journals much later [1,2]. Another possible reason why supercompilation did not become more widely known much earlier is that it was originally formulated in the language Refal, which is rather unconventional in its use of a complex pattern matching algorithm. This meant that Refal programs were hard to understand, and describing transformations making use of this complex pattern matching algorithm made the descriptions quite inaccessible. This problem was overcome by the development of *positive supercompilation* [3,4], which is defined over a more familiar functional language. The positive supercompilation algorithm was further extended by the first author to give the *distillation* algorithm [5,6].

In this paper we give a graph-based definition of distillation which we believe gives the algorithm a more solid theoretical foundation. This definition is made within a similar framework to the positive supercompilation algorithm, thus allowing a more detailed comparison between the two algorithms to be made.

There are two reasons why we do this comparison with positive supercompilation rather than any other formulation of supercompilation. Firstly, positive supercompilation is defined on a more familiar functional language similar to that for which distillation is defined, thus facilitating a more direct comparison. Secondly, the original supercompilation algorithm is less clearly defined and has many variants, thus making comparison difficult. We find that the main distinguishing characteristic between the two algorithms is that in positive supercompilation, generalization and folding are performed with respect to expressions, while in distillation, they are performed with respect to graphs. We find that super-linear improvements in performance are possible using distillation, but not using positive supercompilation, because computationally expensive terms can only be extracted from within loops when generalizing graphs rather than expressions.

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 define the positive supercompilation algorithm. In Section 4 we define the distillation algorithm by using graphs to determine when generalization and folding should be performed. In Section 5 we show how programs can be extracted from the graphs generated by positive supercompilation and distillation, and Section 6 concludes.

2 Language

In this section, we describe the higher-order functional language which will be used throughout this paper. The syntax of this language is given in Fig. 1.

$$
\begin{array}{lll}
prog & ::= e_0 \ \mathbf{where} \ f_1 = e_1 \ldots f_k = e_k & \text{Program} \\
 & & \\
e & ::= v & \text{Variable} \\
 & \quad \mid c \ e_1 \ldots e_k & \text{Constructor} \\
 & \quad \mid f & \text{Function Call} \\
 & \quad \mid \lambda v.e & \lambda\text{-Abstraction} \\
 & \quad \mid e_0 \ e_1 & \text{Application} \\
 & \quad \mid \mathbf{case} \ e_0 \ \mathbf{of} \ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k & \text{Case Expression} \\
 & & \\
p & ::= c \ v_1 \ldots v_k & \text{Pattern}
\end{array}
$$

Fig. 1. Language Syntax

Programs in the language consist of an expression to evaluate and a set of function definitions. The intended operational semantics of the language is normal order reduction. It is assumed that erroneous terms such as $(c \ e_1 \ldots e_k) \ e$ and $\mathbf{case} \ (\lambda v.e) \ \mathbf{of} \ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$ cannot occur. The variables in the patterns of **case** expressions and the arguments of λ-abstractions are *bound*; all other variables are *free*. We use $fv(e)$ and $bv(e)$ to denote the free and bound variables respectively of expression e. We write $e \equiv e'$ if e and e' differ only in the names of bound variables. We require that each function has exactly one definition and that all variables within a definition are bound. We define a function *unfold* which replaces a function name with its definition.

Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. We allow the usual notation [] for *Nil*, $x : xs$ for $Cons\ x\ xs$ and $[e_1, \ldots, e_k]$ for $Cons\ e_1 \ldots (Cons\ e_k\ Nil)$.

Within the expression **case** e_0 **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$, e_0 is called the *selector*, and $e_1 \ldots e_k$ are called the *branches*. The patterns in **case** expressions may not be nested. No variables may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive.

We use the notation $[e'_1/e_1, \ldots, e'_n/e_n]$ to denote a *replacement*, which represents the simultaneous replacement of the expressions $e_1, \ldots, e_n$ by the corresponding expressions $e'_1, \ldots, e'_n$, respectively. We say that a replacement is a *substitution* if all of the expressions $e_1, \ldots, e_n$ are variables and define a predicate *is-sub* to determine whether a given replacement is a substitution. We say that an expression e is an *instance* of another expression e' iff there is a substitution θ s.t. $e \equiv e'\ \theta$.

Example 1. An example program for reversing the list xs is shown in Fig. 2.

```
nrev xs
where
nrev = λxs.case xs of
             []        ⇒ []
           | x' : xs'  ⇒ app (nrev xs') [x']
app  = λxs.λys.case xs of
                 []        ⇒ ys
               | x' : xs'  ⇒ x' : (app xs' ys)
```

Fig. 2. Example Program for List Reversal

3 Positive Supercompilation

In this section, we define the positive supercompilation algorithm; this is largely based on the definition given in [4], but has been adapted to define positive supercompilation within a similar framework to distillation. Within our formulation, positive supercompilation consists of three phases; *driving* (denoted by $\mathcal{D_S}$), *process graph* construction (denoted by $\mathcal{G_S}$) and *folding* (denoted by $\mathcal{F_S}$). The positive supercompilation $\mathcal{S}$ of an expression e is therefore defined as: $\mathcal{S}[\![e]\!] = \mathcal{F_S}[\![\mathcal{G_S}[\![\mathcal{D_S}[\![e]\!]]\!]]\!]$

3.1 Driving

At the heart of the positive supercompilation algorithm are a number of *driving* rules which reduce a term (possibly containing free variables) using normal-order reduction to produce a *process tree*. We define the rules for driving by identifying the next reducible expression (*redex*) within some *context*. An expression which cannot be broken down into a redex and a context is called an *observable*. These are defined as follows.

Definition 1 (Redexes, Contexts and Observables). Redexes, contexts and observables are defined as shown in Fig. 3, where *red* ranges over redexes, *con* ranges over contexts and *obs* ranges over observables (the expression $con\langle e\rangle$ denotes the result of replacing the 'hole' $\langle\rangle$ in *con* by e).

$$
\begin{array}{rl}
red ::= & f \\
& \mid (\lambda v.e_0)\ e_1 \\
& \mid \textbf{case}\ (v\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k' \\
& \mid \textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k' \\
\\
con ::= & \langle\rangle \\
& \mid con\ e \\
& \mid \textbf{case}\ con\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k \\
\\
obs ::= & v\ e_1 \ldots e_n \\
& \mid c\ e_1 \ldots e_n \\
& \mid \lambda v.e
\end{array}
$$

Fig. 3. Syntax of Redexes, Contexts and Observables

Lemma 1 (Unique Decomposition Property). For every expression e, either e is an observable or there is a unique context *con* and redex e' s.t. $e = con\langle e'\rangle$. □

Definition 2 (Process Trees). A *process tree* is a directed tree where each node is labelled with an expression, and all edges leaving a node are ordered. One node is chosen as the *root*, which is labelled with the original expression to be transformed. We use the notation $e \rightarrow t_1, \ldots, t_n$ to represent the tree with root labelled e and n children which are the subtrees $t_1, \ldots, t_n$ respectively. Within a process tree t, for any node α, $t(\alpha)$ denotes the label of α, $anc(t, \alpha)$ denotes the set of ancestors of α in t, $t\{\alpha := t'\}$ denotes the tree obtained by replacing the subtree with root α in t by the tree t' and $root(t)$ denotes the label at the root of t.

Definition 3 (Driving). The core set of transformation rules for positive supercompilation are the driving rules shown in Fig. 4, which define the map $\mathcal{D}_{\mathcal{S}}$

from expressions to process trees. The rules simply perform normal order reduction, with information propagation within **case** expressions giving the assumed outcome of the test. Note that the driving rules are mutually exclusive and exhaustive by the unique decomposition property.

$$
\begin{array}{ll}
\mathcal{D}_\mathcal{S}[\![v\ e_1 \ldots e_n]\!] & = v\ e_1 \ldots e_n \rightarrow \mathcal{D}_\mathcal{S}[\![e_1]\!], \ldots, \mathcal{D}_\mathcal{S}[\![e_n]\!] \\
\mathcal{D}_\mathcal{S}[\![c\ e_1 \ldots e_n]\!] & = c\ e_1 \ldots e_n \rightarrow \mathcal{D}_\mathcal{S}[\![e_1]\!], \ldots, \mathcal{D}_\mathcal{S}[\![e_n]\!] \\
\mathcal{D}_\mathcal{S}[\![\lambda v.e]\!] & = \lambda v.e \rightarrow \mathcal{D}_\mathcal{S}[\![e]\!] \\
\mathcal{D}_\mathcal{S}[\![con\langle f\rangle]\!] & = con\langle f\rangle \rightarrow \mathcal{D}_\mathcal{S}[\![con\langle unfold\ f\rangle]\!] \\
\mathcal{D}_\mathcal{S}[\![con\langle(\lambda v.e_0)\ e_1\rangle]\!] & = con\langle(\lambda v.e_0)\ e_1\rangle \rightarrow \mathcal{D}_\mathcal{S}[\![con\langle e_0[e_1/v]\rangle]\!] \\
\multicolumn{2}{l}{\mathcal{D}_\mathcal{S}[\![con\langle \mathbf{case}\ (v\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k'\rangle]\!]} \\
 & = con\langle \mathbf{case}\ (v\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k'\rangle \rightarrow \\
 & \mathcal{D}_\mathcal{S}[\![v\ e_1 \ldots e_n]\!], \mathcal{D}_\mathcal{S}[\![e_1'[p_1/v\ e_1 \ldots e_n]]\!], \ldots, \mathcal{D}_\mathcal{S}[\![e_k'[p_k/v\ e_1 \ldots e_n]]\!] \\
\multicolumn{2}{l}{\mathcal{D}_\mathcal{S}[\![con\langle \mathbf{case}\ (c\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k'\rangle]\!]} \\
 & = con\langle \mathbf{case}\ (c\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k'\rangle \rightarrow \\
 & \mathcal{D}_\mathcal{S}[\![con\langle e_i[e_1/v_1, \ldots, e_n/v_n]\rangle]\!] \\
 & \text{where } p_i = c\ v_1 \ldots v_n
\end{array}
$$

Fig. 4. Driving Rules

As process trees are potentially infinite data structures, they should be lazily evaluated.

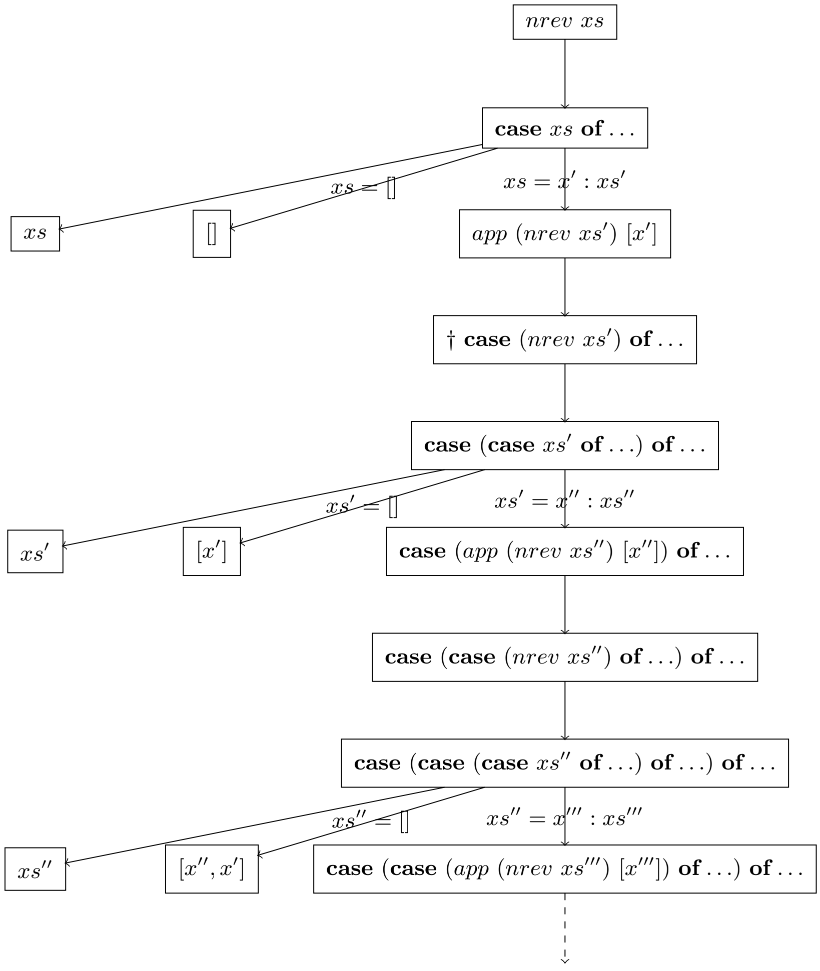
Example 2. A portion of the process tree generated from the list reversal program in Fig. 2 is shown in Fig. 5.

3.2 Generalization

In positive supercompilation, generalization is performed when an expression is encountered which is an *embedding* of a previously encountered expression. The form of embedding which we use to inform this process is known as *homeomorphic embedding*. The homeomorphic embedding relation was derived from results by Higman [7] and Kruskal [8] and was defined within term rewriting systems [9] for detecting the possible divergence of the term rewriting process. Variants of this relation have been used to ensure termination within positive supercompilation [10], partial evaluation [11] and partial deduction [12,13]. It can be shown that the homeomorphic embedding relation $\lesssim_e$ is a *well-quasi-order*, which is defined as follows.

Definition 4 (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 $s_i \leq_S s_j$.

This ensures that in any infinite sequence of expressions $e_0, e_1, \ldots$ there definitely exists some $i < j$ where $e_i \lesssim_e e_j$, so an embedding must eventually be encountered and transformation will not continue indefinitely.



$nrev\ xs$
case xs **of** ...
$xs = []$
$xs = x' : xs'$
xs
$[]$
$app\ (nrev\ xs')\ [x']$
† **case** $(nrev\ xs')$ **of** ...
case (**case** xs' **of** ...) **of** ...
$xs' = []$
$xs' = x'' : xs''$
xs'
$[x']$
case $(app\ (nrev\ xs'')\ [x''])$ **of** ...
case (**case** $(nrev\ xs'')$ **of** ...) **of** ...
case (**case** (**case** xs'' **of** ...) **of** ...) **of** ...
$xs'' = []$
$xs'' = x''' : xs'''$
xs''
$[x'', x']$
case (**case** $(app\ (nrev\ xs''')\ [x'''])$ **of** ...) **of** ...

Fig. 5. Portion of Process Tree Resulting From Driving *nrev xs*

Definition 5 (Homeomorphic Embedding of Expressions). To define the homeomorphic embedding relation on expressions $\lesssim_e$, we firstly define a relation $\trianglelefteq_e$ which requires that all of the free variables within the two expressions match up as follows:

$$\frac{e_1 \lhd_e e_2}{e_1 \trianglelefteq_e e_2} \qquad \frac{e_1 \bowtie_e e_2}{e_1 \trianglelefteq_e e_2} \qquad \frac{e \trianglelefteq_e (e'[v/v'])}{\lambda v.e \bowtie_e \lambda v'.e'}$$

$$\frac{\exists i \in \{1 \ldots n\}.e \trianglelefteq_e e_i}{e \lhd_e \phi(e_1, \ldots, e_n)} \qquad \frac{\forall i \in \{1 \ldots n\}.e_i \trianglelefteq_e e'_i}{\phi(e_1, \ldots, e_n) \bowtie_e \phi(e'_1, \ldots, e'_n)}$$

$$\frac{e_0 \trianglelefteq_e e'_0 \quad \forall i \in \{1 \ldots n\}.\exists \theta_i.p_i \equiv (p'_i\ \theta_i) \wedge e_i \trianglelefteq_e (e'_i\ \theta_i)}{(\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 | \ldots | p_n : e_n) \bowtie_e (\textbf{case}\ e'_0\ \textbf{of}\ p'_1 : e'_1 | \ldots | p'_n : e'_n)}$$

An expression is embedded within another by this relation if either *diving* (denoted by $\lhd_e$) or *coupling* (denoted by $\bowtie_e$) can be performed. Diving occurs when an expression is embedded in a sub-expression of another expression, and coupling occurs when two expressions have the same top-level functor and all the corresponding sub-expressions of the two expressions are embedded. This embedding relation is extended slightly to be able to handle constructs such as λ-abstractions and **case** expressions which may contain bound variables. In these instances, the bound variables within the two expressions must also match up. The homeomorphic embedding relation $\lesssim_e$ can now be defined as follows:

$$e_1 \lesssim_e e_2 \text{ iff } \exists \theta.\textit{is-sub}(\theta) \wedge e_1\ \theta \bowtie_e e_2$$

Thus, within this relation, the two expressions must be coupled, but there is no longer a requirement that all of the free variables within the two expressions match up.

Definition 6 (Generalization of Expressions). The generalization of two expressions e and e' (denoted by $e \sqcap_e e'$) is a triple (e_g, θ, θ') where θ and θ' are substitutions such that $e_g\theta \equiv e$ and $e_g\theta' \equiv e'$, as defined in term algebra [9][1]. This generalization is defined as follows:

$$e \sqcap_e e' = \begin{cases} (\phi(e^g_1, \ldots, e^g_n), \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i), & \text{if } e \lesssim_e e' \\ \quad \text{where } e = \phi(e_1, \ldots, e_n) & \\ \qquad\qquad e' = \phi(e'_1, \ldots, e'_n) & \\ \qquad\qquad (e^g_i, \theta_i, \theta'_i) = e_i \sqcap_e e'_i & \\ (v, [e/v], [e'/v]), & \text{otherwise} \end{cases}$$

Within these rules, if both expressions have the same functor at the outermost level, this is made the outermost functor of the resulting generalized expression, and the corresponding sub-expressions within the functor applications are then generalized. Otherwise, both expressions are replaced by the same variable. The rewrite rule $(e, \theta[e'/v_1, e'/v_2], \theta'[e''/v_1, e''/v_2]) \Rightarrow (e[v_2/v_1], \theta[e'/v_2], \theta[e''/v_2])$ is exhaustively applied to the triple resulting from generalization to minimize the substitutions by identifying common substitutions which were previously given different names.

To represent the result of generalization, we introduce a **let** construct of the form **let** $v_1 = e_1, \ldots, v_n = e_n$ **in** e_0 into our language. This represents the permanent extraction of the expressions $e_1, \ldots, e_n$, which will be transformed separately. The driving rule for this new construct is as follows:

$$\begin{array}{l}\mathcal{D}_{\mathcal{S}}[\![con\langle \textbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \textbf{in}\ e_0 \rangle]\!] = \\ \quad con\langle \textbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \textbf{in}\ e_0 \rangle \rightarrow \mathcal{D}_{\mathcal{S}}[\![e_1]\!], \ldots, \mathcal{D}_{\mathcal{S}}[\![e_n]\!], \mathcal{D}_{\mathcal{S}}[\![con\langle e_0 \rangle]\!]\end{array}$$

[1] Note that, in a higher-order setting, this is no longer a most specific generalization, as the most specific generalization of the terms $f\ (g\ x)$ and $f\ (h\ x)$ would be $(f\ (v\ x), [g/v], [h/v])$, whereas $f\ (g\ x) \sqcap_e f\ (h\ x) = (f\ v, [(g\ x)/v], [(h\ x)/v])$.

We now define an *abstract* operation on expressions which extracts the sub-terms resulting from generalization using **let** expressions.

Definition 7 (Abstract Operation).

$abstract_e(e, e') = \textbf{let}\ v_1\ =\ e_1, \ldots, v_n\ =\ e_n\ \textbf{in}\ e_g$
where $e \sqcap_e e' = (e_g, [e_1/v_1, \ldots, e_n/v_n], \theta)$

3.3 Process Graph Construction

In our formulation of positive supercompilation, the potentially infinite process tree produced by driving is converted into a finite *process graph.*

Definition 8 (Process Graph). A process graph is a process tree which may in addition contain *replacement nodes*. A replacement node has the form $e \overset{\theta}{\dashrightarrow} \alpha$ where α is an ancestor node in the tree and θ is a replacement s.t. $t(\alpha)\ \theta \equiv e$.

Definition 9 (Process Graph Substitution). Substitution in a process graph is performed by applying the substitution pointwise to all the node labels within it as follows.

$$(e \rightarrow t_1, \ldots, t_n)\ \theta = e\ \theta \rightarrow t_1\ \theta, \ldots, t_n\ \theta$$

Definition 10 (Process Graph Equivalence). Two process graphs are equivalent if the following relation is satisfied.

$$con\langle e\rangle \rightarrow t_1, \ldots, t_n \equiv con'\langle e'\rangle \rightarrow t'_1, \ldots, t'_n, \text{ iff } e \lesssim_e e' \wedge \forall i \in \{1 \ldots n\}.t_i \equiv t'_i$$

$$e \overset{\theta}{\dashrightarrow} t \equiv e' \overset{\theta'}{\dashrightarrow} t', \text{ iff } t \equiv t'$$

Within this relation, there is therefore a requirement that the redexes within corresponding nodes are coupled.

Definition 11 (Process Graph Construction in Positive Supercompilation). The rules for the construction of a process graph from a process tree in positive supercompilation t are as follows.

$$\mathcal{G_S}[\![\beta = con\langle f\rangle \rightarrow t']\!] = \begin{cases} con\langle f\rangle \overset{[e'_i/e_i]}{\dashrightarrow} \alpha, & \text{if } \exists \alpha \in anc(t, \beta).t(\alpha) \lesssim_e t(\beta) \\ con\langle f\rangle \rightarrow \mathcal{G_S}[\![t']\!], & \text{otherwise} \end{cases}$$
$$\text{where}$$
$$t(\alpha) \sqcap_e t(\beta) = (e_g, [e_i/v_i], [e'_i/v_i])$$

$$\mathcal{G_S}[\![e \rightarrow t_1, \ldots, t_n]\!] \quad = e \rightarrow \mathcal{G_S}[\![t_1]\!], \ldots, \mathcal{G_S}[\![t_n]\!]$$

A process graph is considered to be *folded* when all of the replacements within it are substitutions. This folding is performed as follows.

Definition 12 (Folding in Positive Supercompilation). The rules for folding a process graph t using positive supercompilation are as follows.

$$\mathcal{F}_{\mathcal{S}}[\![e \overset{\theta}{\dashrightarrow} \alpha]\!] = \begin{cases} e \overset{\theta}{\dashrightarrow} \alpha, & \text{if } \mathit{is\text{-}sub}(\theta) \\ t\{\alpha := \mathcal{S}[\![\mathit{abstract}_e(t(\alpha), e)]\!]\}, & \text{otherwise} \end{cases}$$

$$\mathcal{F}_{\mathcal{S}}[\![e \to t_1, \ldots, t_n]\!] = e \to \mathcal{F}_{\mathcal{S}}[\![t_1]\!], \ldots, \mathcal{F}_{\mathcal{S}}[\![t_n]\!]$$

Example 3. The process graph constructed from the process tree in Fig. 5 is shown in Fig. 6 where the replacement θ is equal to $[app\ (nrev\ xs'')\ [x'']/nrev\ xs']$.

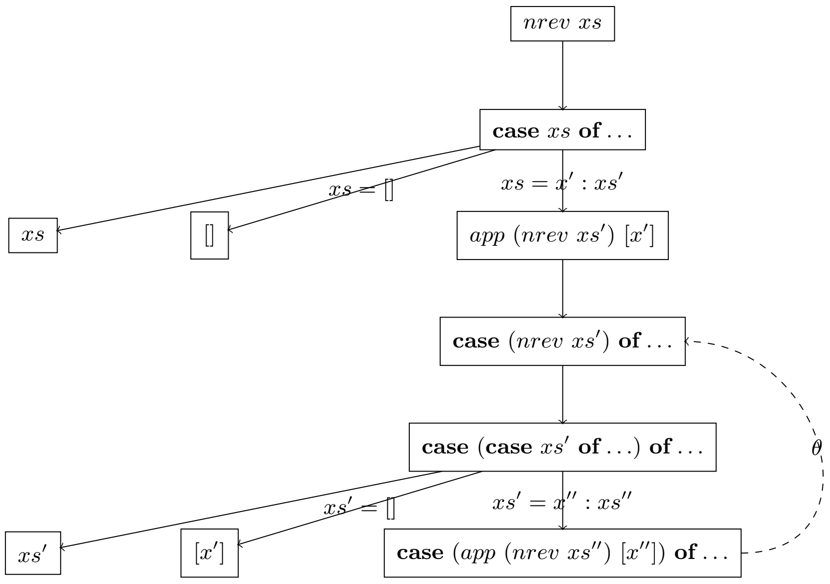


Fig. 6. Process Graph Constructed for *nrev xs*

The folded process graph constructed from the process graph in Fig. 6 is shown in Fig. 7.

4 Distillation

In this section, we define the distillation algorithm within a similar framework to that used to define positive supercompilation in the previous section. Distillation consists of two phases; driving (the same as for positive supercompilation) and folding (denoted by $\mathcal{F}_{\mathcal{D}}$). The distillation $\mathcal{D}$ of an expression e is therefore defined as: $\mathcal{D}[\![e]\!] = \mathcal{F}_{\mathcal{D}}[\![\mathcal{D}_{\mathcal{S}}[\![e]\!]]\!]$. Folding in distillation is performed with respect to process graphs. We therefore define what it means for one process graph to be an instance or a homeomorphic embedding of another.

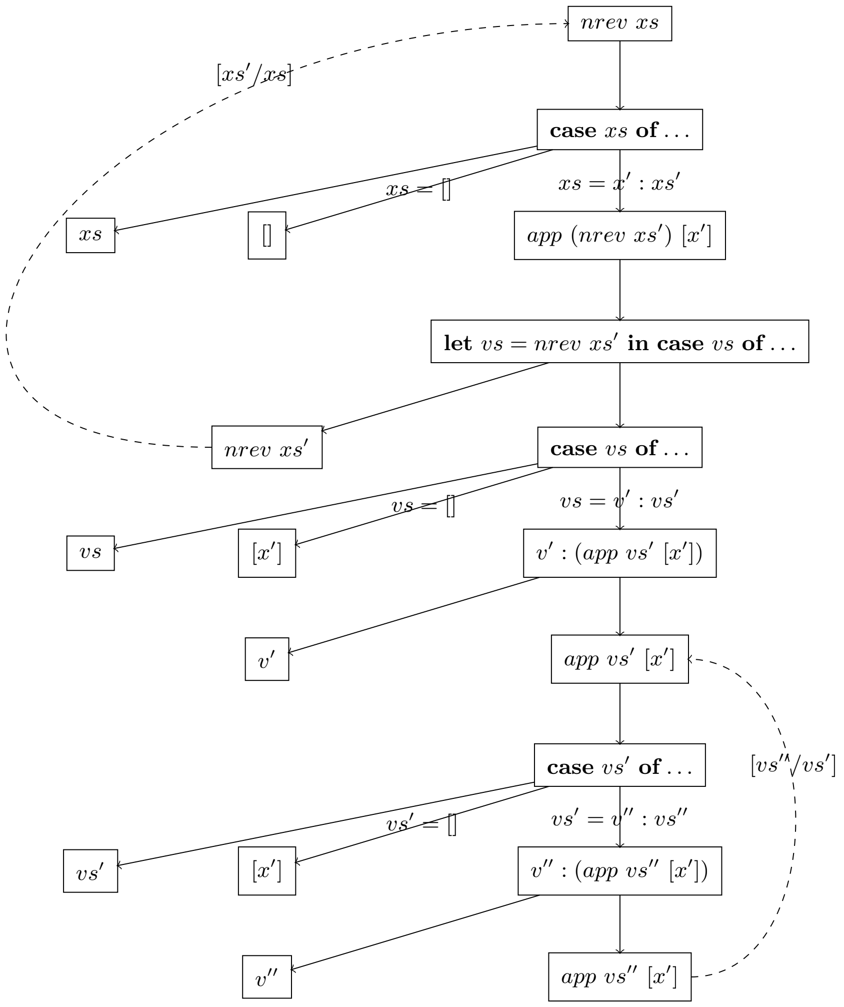


Fig. 7. Folded Process Graph for *nrev xs*

Definition 13 (Process Graph Instance). A process graph t' is an instance of another process graph t (denoted by $t \lessdot_\theta t'$) iff there is a substitution θ s.t. $t \equiv t'\ \theta$.

Definition 14 (Homeomorphic Embedding of Process Graphs). To define the homeomorphic embedding relation on process graphs $\lesssim_t$, we firstly define a relation $\trianglelefteq_t$ which requires that all the free variables in the two process graphs match up as follows:

$$\frac{t_1 \lhd_t t_2}{t_1 \trianglelefteq_t t_2} \qquad \frac{t_1 \bowtie_t t_2}{t_1 \trianglelefteq_t t_2} \qquad \frac{t \trianglelefteq_t (t'[v/v'])}{\lambda v.e \rightarrow t \bowtie_t \lambda v'.e' \rightarrow t'}$$

$$\frac{e \bowtie_e e' \quad \forall i \in \{1 \ldots n\}. t_i \trianglelefteq_t t'_i}{con\langle e\rangle\!\rightarrow t_1, \ldots, t_n \bowtie_t con'\langle e'\rangle\!\rightarrow t'_1, \ldots, t'_n} \qquad \frac{\exists i \in \{1 \ldots n\}. t \trianglelefteq_t t_i}{t \lhd_t e \rightarrow t_1, \ldots, t_n}$$

$$\frac{t \bowtie_t t'}{e \overset{\theta}{\dashrightarrow} t \bowtie_t e' \overset{\theta'}{\dashrightarrow} t'}$$

$$\frac{t_0 \trianglelefteq_t t'_0 \quad \forall i \in \{1 ... n\}. \exists \theta_i . p_i \equiv (p'_i \; \theta_i) \wedge t_i \trianglelefteq_t (t'_i \; \theta_i)}{(\textbf{case } e_0 \textbf{ of } p_1 : e_1 | ... | p_n : e_n) \rightarrow t_0, ..., t_n \bowtie_t (\textbf{case } e'_0 \textbf{ of } p'_1 : e'_1 | ... | p'_n : e'_n) \rightarrow t'_0, ..., t'_n}$$

A tree is embedded within another by this relation if either *diving* (denoted by $\lhd_t$) or *coupling* (denoted by $\bowtie_t$) can be performed. Diving occurs when a tree is embedded in a sub-tree of another tree, and coupling occurs when the redexes of the root expressions of two trees are coupled. As for the corresponding embedding relation on expressions, this embedding relation is extended slightly to be able to handle constructs such as λ-abstractions and **case** expressions which may contain bound variables. In these instances, the bound variables within the two process graphs must also match up. The homeomorphic embedding relation on process graphs $\lesssim_t$ can now be defined as follows:

$$t_1 \lesssim_t t_2 \text{ iff } \exists \theta. \textit{is-sub}(\theta) \wedge t_1 \; \theta \bowtie_t t_2$$

Within this relation, there is no longer a requirement that all of the free variables within the two process graphs match up.

4.1 Generalization

Generalization is performed on two process trees if their corresponding process graphs are homeomorphically embedded as follows.

Definition 15 (Generalization of Process Trees). Generalization is performed on process trees using the $\sqcap_t$ operator which is defined as follows:

$$t \sqcap_t t' = \begin{cases} (e \rightarrow t^g_1, \ldots, t^g_n, \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i), & \text{if } t \lesssim_t t' \\ \quad \text{where } t = e \rightarrow t_1, \ldots, t_n & \\ \qquad\qquad t' = e' \rightarrow t'_1, \ldots, t'_n & \\ \qquad\qquad (t^g_i, \theta_i, \theta'_i) = t_i \sqcap_t t'_i & \\ (\mathcal{D_S}[\![e^g]\!], \theta, \theta'), & \text{otherwise} \\ \quad \text{where } (e^g, \theta, \theta') = root(t) \sqcap_e root(t') & \end{cases}$$

Within these rules, if two trees are coupled then their corresponding sub-trees are generalized. Otherwise, the expressions in the corresponding root nodes are generalized. As the process trees being generalized are potentially infinite, this generalization should also be performed lazily. As is done for the generalization of expressions, the rewrite rule $(e, \theta[e'/v_1, e'/v_2], \theta'[e''/v_1, e''/v_2]) \Rightarrow (e[v_2/v_1], \theta[e'/v_2], \theta[e''/v_2])$ is also exhaustively applied to the triple resulting from generalization to minimize the substitutions by identifying common substitutions which were previously given different names. Note that the use of this rewrite rule is essential for the correctness of the distillation algorithm. We now define an *abstract* operation on process trees which extracts the sub-terms resulting from generalization using **let** expressions.

Definition 16 (Abstract Operation on Process Trees).
$abstract_t(t, t') = (\mathbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \mathbf{in}\ root(t)) \rightarrow \mathcal{D_S}[\![e_1]\!], ..., \mathcal{D_S}[\![e_n]\!], t_g$
where $t \sqcap_t t' = (t_g, [e_1/v_1, \ldots, e_n/v_n], \theta)$

4.2 Folding

In distillation, process graphs are used to determine when to perform folding and generalization. These process graphs are constructed slightly differently than those in positive supercompilation, with replacement nodes being added when an expression is encountered which is an embedding (rather than a coupling) of an ancestor expression. To facilitate this, a new relation $\lesssim'_e$ is defined as follows:

$$e_1 \lesssim'_e e_2 \text{ iff } \exists \theta. is\text{-}sub(\theta) \wedge e_1\ \theta \trianglelefteq_e e_2$$

Definition 17 (Process Graph Construction in Distillation). The rules for the construction of a process graph from a process tree in distillation t are as follows.

$$\mathcal{G_D}[\![\beta = con\langle f\rangle \rightarrow t']\!] = \begin{cases} con\langle f\rangle \overset{[e'_i/e_i]}{\dashrightarrow} \alpha, & \text{if } \exists \alpha \in anc(t, \beta). t(\alpha) \lesssim'_e t(\beta) \\ con\langle f\rangle \rightarrow \mathcal{G_D}[\![t']\!], & \text{otherwise} \end{cases}$$
where
$t(\alpha) \sqcap_e t(\beta) = (e_g, [e_i/v_i], [e'_i/v_i])$

$$\mathcal{G_D}[\![e \rightarrow t_1, \ldots, t_n]\!] \quad = e \rightarrow \mathcal{G_D}[\![t_1]\!], \ldots, \mathcal{G_D}[\![t_n]\!]$$

Definition 18 (Folding in Distillation). The rules for folding a process tree t using distillation are as follows.

$$\mathcal{F_D}[\![\beta = con\langle f\rangle \rightarrow t']\!] = \begin{cases} con\langle f\rangle \overset{\theta}{\dashrightarrow} \alpha, & \text{if } \exists \alpha \in anc(t, \beta). \mathcal{G_D}[\![\alpha]\!] \lessdot_\theta \mathcal{G_D}[\![\beta]\!] \\ t\{\alpha := \mathcal{F_D}[\![abstract_t(\alpha, \beta)]\!]\}, & \\ & \text{if } \exists \alpha \in anc(t, \beta). \mathcal{G_D}[\![\alpha]\!] \lesssim_t \mathcal{G_D}[\![\beta]\!] \\ con\langle f\rangle \rightarrow \mathcal{F_D}[\![t']\!], & \text{otherwise} \end{cases}$$

$$\mathcal{F_D}[\![e \rightarrow t_1, \ldots, t_n]\!] \quad = e \rightarrow \mathcal{F_D}[\![t_1]\!], \ldots, \mathcal{F_D}[\![t_n]\!]$$

Example 4. The process graph constructed from the root node of the process tree in Fig. 5 is shown in Fig. 8, where the replacement θ is $[app\ (nrev\ xs')\ [x']/nrev\ xs]$.

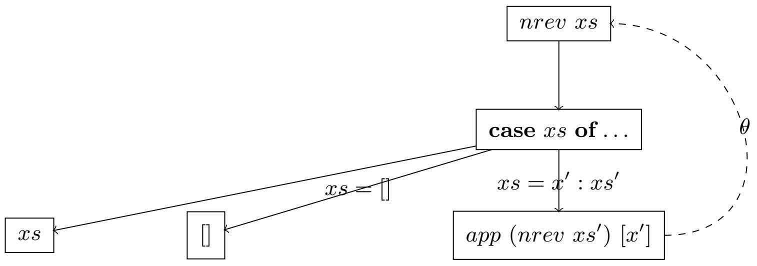
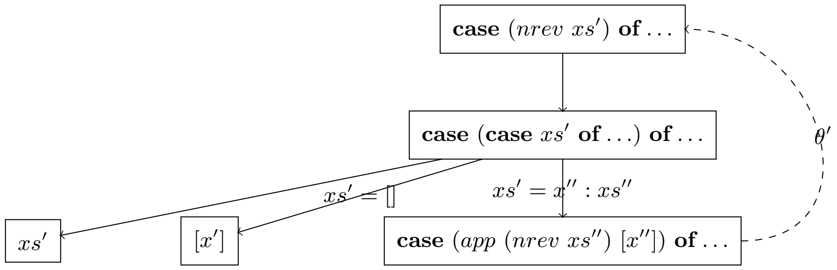


Fig. 8. Process Graph

Similarly, the process graph constructed from the node labelled † in the process tree in Fig. 5 is shown in Fig. 9, where the replacement θ' is equal to $[app\ (nrev\ xs'')\ [x'']/nrev\ xs']$.



case (*nrev xs'*) **of** ...
case (**case** *xs'* **of** ...) **of** ...
θ'
$xs' = []$
$xs' = x'' : xs''$
xs'
[*x'*]
case (*app* (*nrev xs''*) [*x''*]) **of** ...

Fig. 9. Process Graph

The process graph in Fig. 8 is embedded in the process graph in Fig. 9, so the corresponding process trees are generalized to produce the process tree shown in Fig. 10. The process graph constructed for the node labelled † is now an instance of the process graph constructed for the root node of this process tree, so folding is performed to produce the folded process graph shown in Fig. 11

5 Program Residualization

A residual program can be constructed from a folded process graph using the rules $\mathcal{C}$ as shown in Fig. 12.

Example 5. The program constructed from the folded process graph resulting from the positive supercompilation of *nrev xs* shown in Fig. 7 is as shown in Fig. 13. The program constructed from the folded process graph resulting from the distillation of *nrev xs* shown in Fig. 11 is as shown in Fig. 14. We can see that the distilled program is a super-linear improvement over the original, while the supercompiled program has produced no improvement.

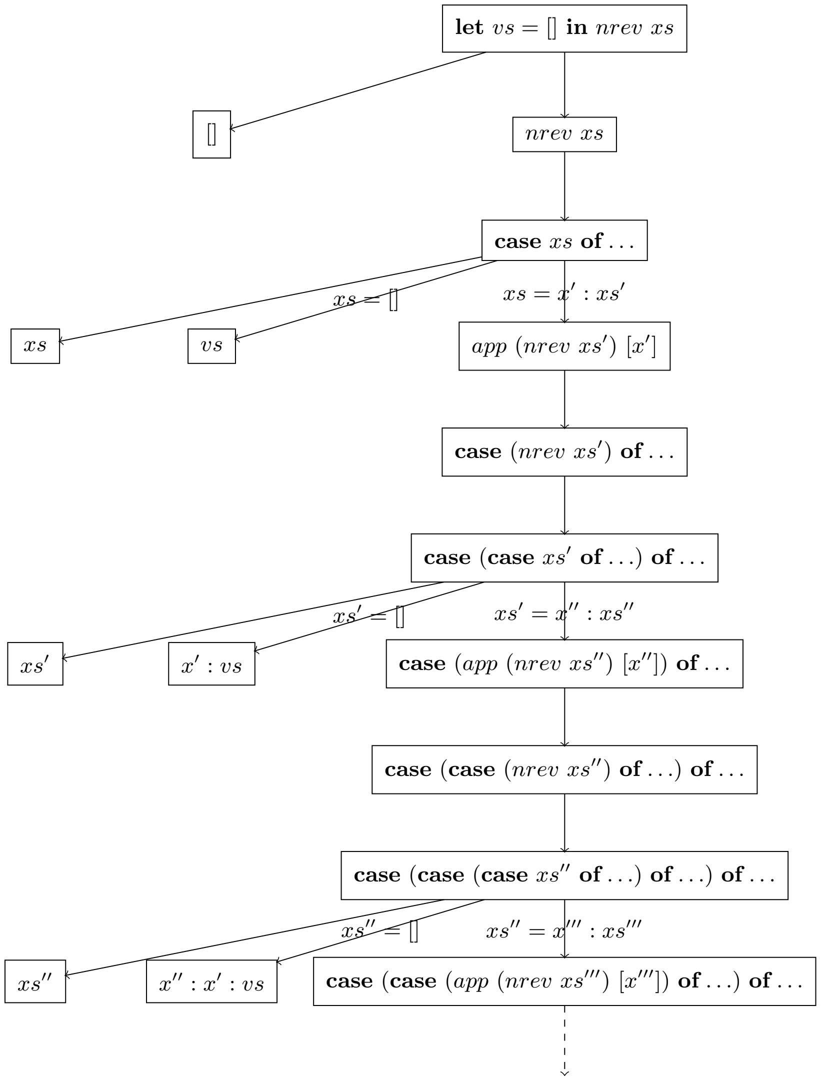

let $vs = []$ in $nrev\ xs$
$[]$
$nrev\ xs$
case xs of ...
$xs = []$
$xs = x' : xs'$
xs
vs
$app\ (nrev\ xs')\ [x']$
case $(nrev\ xs')$ of ...
case (case xs' of ...) of ...
$xs' = []$
$xs' = x'' : xs''$
xs'
$x' : vs$
case $(app\ (nrev\ xs'')\ [x''])$ of ...
case (case $(nrev\ xs'')$ of ...) of ...
case (case (case xs'' of ...) of ...) of ...
$xs'' = []$
$xs'' = x''' : xs'''$
xs''
$x'' : x' : vs$
case (case $(app\ (nrev\ xs''')\ [x'''])$ of ...) of ...

Fig. 10. Result of Generalizing *nrev xs*

6 Conclusion

We have presented a graph-based definition of the distillation transformation algorithm for higher-order functional languages. The definition is made within a similar framework to the positive supercompilation transformation algorithm, thus allowing for a more detailed comparison of the two algorithms. We have

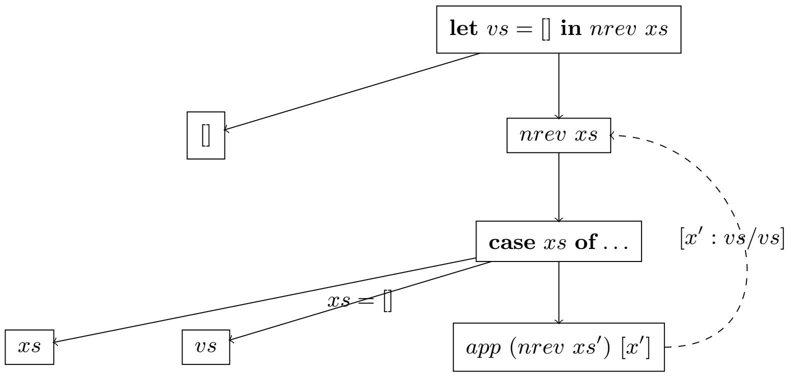


Fig. 11. Result of Folding *nrev xs*

$$
\begin{array}{ll}
\mathcal{C}[\![(v\ e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n]\!]\ \phi & = v\ (\mathcal{C}[\![t_1]\!]\ \phi) \ldots (\mathcal{C}[\![t_n]\!]\ \phi)\\
\mathcal{C}[\![(c\ e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n]\!]\ \phi & = c\ (\mathcal{C}[\![t_1]\!]\ \phi) \ldots (\mathcal{C}[\![t_n]\!]\ \phi)\\
\mathcal{C}[\![(\lambda v.e) \rightarrow t]\!]\ \phi & = \lambda v.(\mathcal{C}[\![t]\!]\ \phi)\\
\mathcal{C}[\![(con\langle f \rangle) \rightarrow t]\!]\ \phi & = f'\ v_1 \ldots v_n\\
 & \quad \text{where}\\
 & \quad f' = \lambda v_1 \ldots v_n . \mathcal{C}[\![t]\!]\ (\phi \cup \{f'\ v_1 \ldots v_n = con\langle f \rangle \rightarrow t\})\\
 & \quad \{v_1 \ldots v_n\} = fv(t)\\
\mathcal{C}[\![(con\langle f \rangle) \overset{\theta}{\dashrightarrow} t]\!]\ \phi & = (f\ v_1 \ldots v_n)\ \theta\\
 & \quad \text{where}\\
 & \quad (f'\ v_1 \ldots v_n = t) \in \phi\\
\multicolumn{2}{l}{\mathcal{C}[\![(con\langle \mathbf{case}\ (v\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n \rangle) \rightarrow t_0, \ldots, t_n]\!]\ \phi}\\
 & = \mathbf{case}\ (\mathcal{C}[\![t_0]\!]\ \phi)\ \mathbf{of}\ p_1 \Rightarrow \mathcal{C}[\![t_1]\!]\ \phi \mid \cdots \mid p_n \Rightarrow \mathcal{C}[\![t_n]\!]\ \phi\\
\multicolumn{2}{l}{\mathcal{C}[\![\mathbf{let}\ v_1\ =\ t_1, \ldots, v_n\ =\ t_n\ \mathbf{in}\ t]\!]\ \phi}\\
 & = (\mathcal{C}[\![t]\!]\ \phi)[(\mathcal{C}[\![t_1]\!]\ \phi)/v_1, \ldots, (\mathcal{C}[\![t_n]\!]\ \phi)/v_n]
\end{array}
$$

Fig. 12. Rules For Constructing Residual Programs

found that the main distinguishing characteristic between the two algorithms is that in positive supercompilation, generalization and folding are performed with respect to expressions, while in distillation they are performed with respect to graphs. We have also found that while only linear improvements in performance are possible using positive supercompilation, super-linear improvements are possible using distillation. This is because computationally expensive terms can only be extracted from within loops when generalizing graphs rather than expressions. Of course, this extra power comes at a price. As generalization and folding are now performed on graphs rather than flat terms, there may be an exponential increase in the number of steps required to perform these operations in the worst case.

There are a number of possible directions for further work. It has already been shown how distillation can be used to verify safety properties of programs

```
f xs
where
f  = λxs.case xs of
                []        ⇒ []
              | x′ : xs′ ⇒ case (f xs′) of
                                []          ⇒ [x′]
                              | x″ : xs″ ⇒ x″ : (f′ xs″ x′)
f′ = λxs.λy.case xs of
                   []        ⇒ [y]
                 | x′ : xs′ ⇒ x′ : (f′ xs′ y)
```

Fig. 13. Result of Applying Positive Supercompilation to *nrev xs*

```
f xs []
where
f = λxs.λvs.case xs of
                  []        ⇒ vs
                | x′ : xs′ ⇒ f xs′ (x′ : vs)
```

Fig. 14. Result of Applying Distilling to *nrev xs*

[14]; work is now in progress by the second author to show how it can also be used to verify liveness properties. Work is also in progress in incorporating the distillation algorithm into the Haskell programming language, so this will allow a more detailed evaluation of the utility of the distillation algorithm to be made. Distillation is being added to the York Haskell Compiler [15] in a manner similar to the addition of positive supercompilation to the same compiler in Supero [16]. Further work is also required in proving the termination and correctness of the distillation algorithm. Finally, it has been found that the output produced by the distillation algorithm is in a form which is very amenable to automatic parallelization. Work is also in progress to incorporate this automatic parallelization into the York Haskell Compiler.

Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie), and by the School of Computing, Dublin City University.

References

1. Turchin, V.: Program Transformation by Supercompilation. Lecture Notes in Computer Science **217** (1985) 257–281
2. Turchin, V.: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems **8**(3) (July 1986) 90–121

3. Sørensen, M.: Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen (1994) DIKU-rapport 94/17.
4. Sørensen, M., Glück, R., Jones, N.: A Positive Supercompiler. Journal of Functional Programming **6**(6) (1996) 811–838
5. Hamilton, G.W.: Distillation: Extracting the Essence of Programs. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. (2007) 61–70
6. Hamilton, G.W.: Extracting the Essence of Distillation. In: Proceedings of the Seventh International Andrei Ershov Memorial Conference: Perspectives of System Informatics (PSI '09). Volume 5947 of Lecture Notes in Computer Science. (2009) 151–164
7. Higman, G.: Ordering by Divisibility in Abstract Algebras. Proceedings of the London Mathemtical Society **2** (1952) 326–336
8. Kruskal, J.: Well-Quasi Ordering, the Tree Theorem, and Vazsonyi's Conjecture. Transactions of the American Mathematical Society **95** (1960) 210–225
9. Dershowitz, N., Jouannaud, J.P.: Rewrite Systems. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science. Elsevier, MIT Press (1990) 243–320
10. Sørensen, M., Glück, R.: An Algorithm of Generalization in Positive Supercompilation. Lecture Notes in Computer Science **787** (1994) 335–351
11. Marlet, R.: Vers une Formalisation de l'Évaluation Partielle. PhD thesis, Université de Nice - Sophia Antipolis (1994)
12. Bol, R.: Loop Checking in Partial Deduction. Journal of Logic Programming **16**(1–2) (1993) 25–46
13. Leuschel, M.: On the Power of Homeomorphic Embedding for Online Termination. In: Proceedings of the International Static Analysis Symposium. (1998) 230–245
14. Hamilton, G.W.: Distilling Programs for Verification. Electronic Notes in Theoretical Computer Science **190**(4) (2007) 17–32
15. Mitchell, N.: Yhc Manual (wiki)
16. Mitchell, N., Runciman, C.: A Supercompiler for Core Haskell. Lecture Notes in Computer Science **5083** (2008) 147–164