Extracting the Essence of Distillation

G.W. Hamilton

School of Computing
Dublin City University
Ireland
`hamilton@computing.dcu.ie`

Abstract. In this paper, we give a re-formulation of our previously defined *distillation* algorithm, which can automatically transform higher-order functional programs into equivalent tail-recursive programs. Our re-formulation simplifies the presentation of the transformation and hopefully makes it easier to understand. Using distillation, it is possible to produce superlinear improvement in the run-time of programs. This represents a significant advance over deforestation, partial evaluation and positive supercompilation, which can only produce a linear improvement.

1 Introduction

It is well known that programs which are written using lazy functional programming languages often tend to make use of intermediate data structures, and are therefore inefficient. A number of program transformation techniques have been proposed which can eliminate some of these intermediate data structures; for example *partial evaluation* [1], *deforestation* [2] and *supercompilation* [3]. *Positive supercompilation* [4] is a variant of Turchin's supercompilation which was introduced in an attempt to study and explain the essentials of Turchin's supercompiler. Although positive supercompilation is strictly more powerful than both partial evaluation and deforestation, Sørensen has shown that positive supercompilation (and hence also partial evaluation and deforestation) can only produce a linear speedup in programs [5]. A more powerful transformation algorithm should be able to produce a superlinear speedup in programs.

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 which is linear with respect to the length of the list xs is the function $arev$ shown in Fig. 1. A number of algebraic transformations have been proposed which can perform this transformation (e.g. [6]) by appealing to a specific law stating the associativity of the app function. However, none of the generic program transformation techniques mentioned above are capable of performing this transformation.

Previously, we defined a transformation algorithm called *distillation* [7] which will allow transformations such as the above to be performed. In our previous

```
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)

arev xs
where
arev  = λxs.arev′ xs []
arev′ = λxs.λys.case xs of
                     []        ⇒ ys
                   | x′ : xs′ ⇒ arev′ xs′ (x′ : ys)
```

Fig. 1. Alternative Definitions of List Reversal

work, the definition of distillation was dependent upon that of positive supercompilation. In this paper, we give a definition of distillation which is not dependent upon positive supercompilation, thus simplifying the algorithm and hopefully making it easier to understand.

The distillation algorithm was largely influenced by positive supercompilation, but also improves upon it. Both algorithms involve *driving* to produce a *process tree* representing all the possible states in the symbolic execution of a program, and *folding* to extract a (hopefully more efficient) program from this process tree. *Generalization* may also be required to ensure the termination of the algorithm. The extra power of the distillation algorithm over positive supercompilation is obtained through the use of a more powerful matching mechanism when performing folding and generalization. In positive supercompilation, folding and generalization are performed on flat terms; terms are considered to match only if they use the same functions. In distillation, folding and generalization are performed on process trees, so terms are considered to match only if they have the same recursive structure.

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 driving rules for this language which perform symbolic execution to produce a process tree. In Section 4 we define generalization on terms in this language and also on process trees. In Section 5 we show how folding can be performed on process trees to extract corresponding programs. In Section 6 we give some examples of the application of distillation and Section 7 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. 2.

$prog$	$::= e_0$ **where** $f_1 = e_1 \ldots f_k = e_k$	Program
e	$::= v$	Variable
	$\mid c\ e_1 \ldots e_k$	Constructor
	$\mid f$	Function Call
	$\mid \lambda v.e$	λ-Abstraction
	$\mid e_0\ e_1$	Application
	$\mid$ **case** e_0 **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$	Case Expression
p	$::= c\ v_1 \ldots v_k$	Pattern

Fig. 2. 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 **case** $(\lambda v.e)$ **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 $\{v_1 := e_1, \ldots, v_n := e_n\}$ to denote a *substitution*, which represents the simultaneous substitution of the expressions $e_1, \ldots, e_n$ for the corresponding variables $v_1, \ldots, v_n$, respectively. We say that an expression e is an *instance* of expression e' if there is a substitution θ such that $e \equiv e'\ \theta$. We also 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.

3 Driving

In this section, we define *driving* rules similar to those for positive supercompilation to reduce a term (possibly containing free variables) using normal-order reduction and 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).

Definition 2 (Normal Order Reduction). The core set of transformation rules for distillation are the normal order reduction rules shown in Figure 4 which defines the map $\mathcal{N}$ from expressions to ordered sequences of expressions $[e_1, \ldots, e_n]$. The rules simply perform normal order reduction, with information propagation within **case** expressions giving the assumed outcome of the test.

$$\begin{array}{rl}
red ::= & f \\
& \mid (\lambda v.e_0)\ e_1 \\
& \mid \textbf{case}\ (v\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k \\
& \mid \textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k \\
& \mid \textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)\ \textbf{of}\ p'_1 \Rightarrow e'_1 \mid \cdots \mid p'_k \Rightarrow e'_k \\
\\
con ::= & \langle\rangle \\
& \mid con\ e \\
& \mid \textbf{case}\ \langle\rangle\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k \\
\\
obs ::= & v\ e_1 \ldots e_n \\
& \mid c\ e_1 \ldots e_n \\
& \mid \lambda v.e
\end{array}$$

Fig. 3. Syntax of Redexes, Contexts and Observables

$$\begin{array}{ll}
\mathcal{N}[\![v\ e_1 \ldots e_n]\!] & = [e_1, \ldots, e_n] \\
\mathcal{N}[\![c\ e_1 \ldots e_n]\!] & = [e_1, \ldots, e_n] \\
\mathcal{N}[\![\lambda v.e]\!] & = [e] \\
\mathcal{N}[\![con\langle f\rangle]\!] & = [con\langle unfold\ f\rangle] \\
\mathcal{N}[\![con\langle(\lambda v.e_0)\ e_1\rangle]\!] & = [con\langle e_0\{v := e_1\}\rangle] \\
\multicolumn{2}{l}{\mathcal{N}[\![con\langle\textbf{case}\ (v\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k\rangle]\!]} \\
& = [v\ e_1 \ldots e_n, con\langle e'_1[p_1/v\ e_1 \ldots e_n]\rangle, \ldots, con\langle e'_k[p_k/v\ e_1 \ldots e_n]\rangle] \\
\multicolumn{2}{l}{\mathcal{N}[\![con\langle\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k\rangle]\!]} \\
& = [con\langle e_i\{e_1 := v_1, \ldots, e_n := v_n\}\rangle]\ \text{where}\ p_i = c\ v_1 \ldots v_n \\
\multicolumn{2}{l}{\mathcal{N}[\![con\langle\textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)\ \textbf{of}\ p'_1 \Rightarrow e'_1 \mid \cdots \mid p'_k \Rightarrow e'_k\rangle]\!]} \\
& = [\textbf{case}\ e_0\ \textbf{of} \\
& \qquad p_1 \Rightarrow con\langle\textbf{case}\ e_1\ \textbf{of}\ p'_1 \Rightarrow e'_1 \mid \cdots \mid p'_k \Rightarrow e'_k\rangle \\
& \qquad \vdots \\
& \qquad p_n \Rightarrow con\langle\textbf{case}\ e_n\ \textbf{of}\ p'_1 \Rightarrow e'_1 \mid \cdots \mid p'_k \Rightarrow e'_k\rangle]
\end{array}$$

Fig. 4. Normal Order Reduction Rules

Definition 3 (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.

Definition 4 (Driving). Driving in distillation is defined by the following map $\mathcal{D}$ from expressions to process trees:

$$\mathcal{D}[\![e]\!] = e \rightarrow \mathcal{D}[\![e_1]\!], \ldots, \mathcal{D}[\![e_n]\!] \text{ where } \mathcal{N}[\![e]\!] = [e_1, \ldots, e_n]$$

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

Example 2. A portion of the process tree which would be generated as a result of driving the expression *nrev xs* as defined in Fig. 1 is shown in Fig. 5[1].

4 Generalization

In distillation, as for 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 guide generalization is known as *homeomorphic embedding*. The homeomorphic embedding relation was derived from results by Higman [8] and Kruskal [9] and was defined within term rewriting systems [10] for detecting the possible divergence of the term rewriting process. Variants of this relation have been used to ensure termination within positive supercompilation [11], partial evaluation [12] and partial deduction [13,14]. It can be shown that the homeomorphic embedding relation $\trianglelefteq$ is a *well-quasi-order*, which is defined as follows.

Definition 5 (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 $e_i \trianglelefteq e_j$, so an embedding must eventually be encountered and transformation will not continue indefinitely.

Definition 6 (Recursive Component). A variable v is called a recursive component of another variable v' (denoted by $v \sqsubset v'$) if v is a sub-component of v' and is of the same type. We also define $v \sqsubseteq v'$ if $v \sqsubset v'$ or $v = v'$.

Definition 7 (Homeomorphic Embedding Relation). The rules for the homeomorphic embedding relation are defined as follows:

[1] This process tree, and later ones presented in this paper, have been simplified for ease of presentation by removing some intermediate nodes.

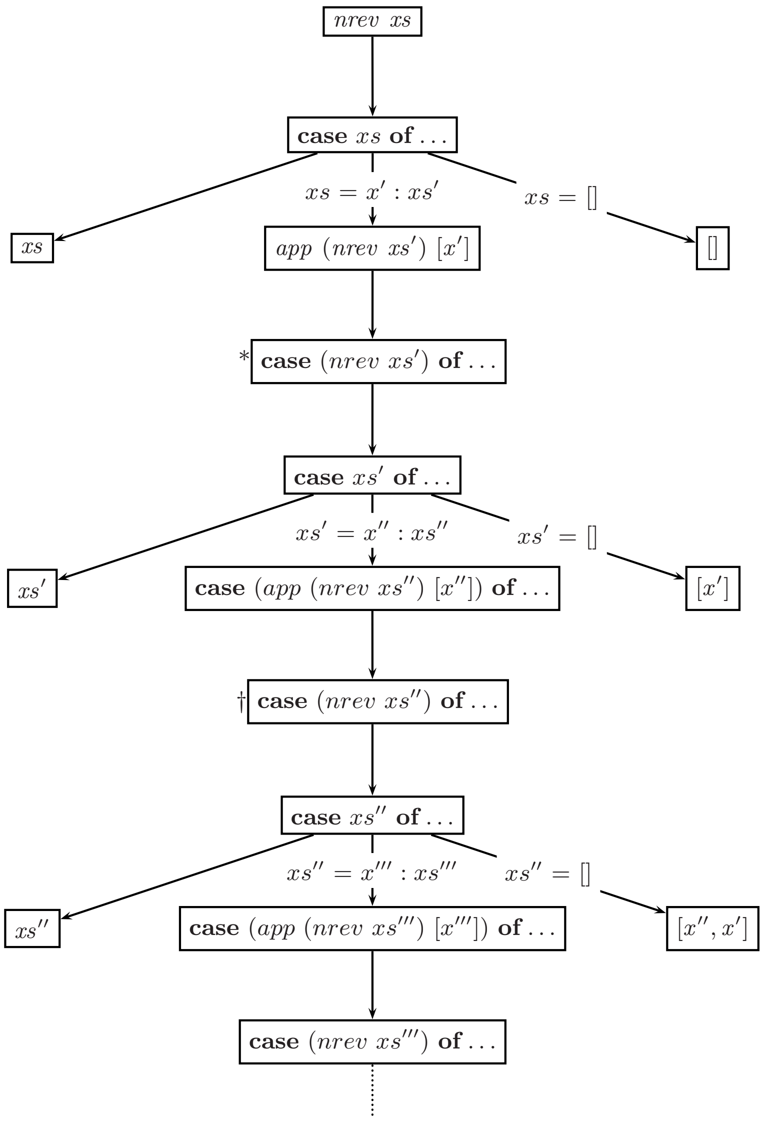


Fig. 5. Portion of Process Tree Resulting From Driving *nrev xs*

$$\frac{e_1 \lhd e_2}{e_1 \unlhd e_2} \qquad \frac{e_1 \bowtie e_2}{e_1 \unlhd e_2}$$

$$\frac{fv \sqsubseteq fv'}{fv \bowtie fv'} \qquad \frac{bv = bv'}{bv \bowtie bv'}$$

$$\frac{f = f'}{f \bowtie f'} \qquad \frac{c = c' \quad \forall i.e_i \unlhd e'_i}{(c\ e_1 \ldots e_n) \bowtie (c'\ e'_1 \ldots e'_n)}$$

$$\frac{e \unlhd (e'\{v' := v\})}{\lambda v.e \bowtie \lambda v'.e'} \qquad \frac{e_0 \bowtie e'_0 \quad e_1 \unlhd e'_1}{(e_0\ e_1) \bowtie (e'_0\ e'_1)}$$

$$\frac{e \bowtie e' \quad \forall i.p_i \equiv (p'_i\ \theta_i) \wedge e_i \unlhd (e'_i\ \theta_i)}{(\mathbf{case}\ e\ \mathbf{of}\ p_1 : e_1 | \ldots | p_n : e_n) \bowtie (\mathbf{case}\ e'\ \mathbf{of}\ p'_1 : e'_1 | \ldots | p'_n : e'_n)}$$

$$\frac{\exists i.e \unlhd e_i}{e \lhd (c\ e_1 \ldots e_n)} \qquad \frac{e \unlhd e'}{e \lhd \lambda v.e'}$$

$$\frac{\exists i.e \unlhd e_i}{e \lhd (e_0\ e_1)} \qquad \frac{\exists i.e \unlhd e_i}{e \lhd (\mathbf{case}\ e_0\ \mathbf{of}\ p_1 : e_1 | \ldots | p_n : e_n)}$$

An expression is homeomorphically embedded within another if either *diving* (denoted by $\lhd$) or *coupling* (denoted by $\bowtie$) 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. Free variables are considered to be embedded if they are related by the $\sqsubseteq$ relation, and the corresponding bound variables within expressions must also match up.

Example 3. Some examples of these embedding relations are as follows:

1. $f_2\ (f_1\ x) \unlhd f_3(f_2\ (f_1\ y))$ 2. $f_1\ (f_2\ x) \unlhd f_1\ (f_2\ (f_3\ y))$
3. $f_2\ (f_1\ x) \lhd f_3(f_2\ (f_1\ y))$ 4. $f_1\ (f_2\ x) \ntriangleleft f_1\ (f_2\ (f_3\ y))$
5. $f_2\ (f_1\ x) \not\bowtie f_3(f_2\ (f_1\ y))$ 6. $f_1\ (f_2\ x) \bowtie f_1\ (f_2\ (f_3\ y))$
7. $\lambda x.x \unlhd \lambda y.y$ 8. $\lambda x.x \ntrianglelefteq \lambda y.x$

Definition 8 (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 [10][2]. This generalization is defined as follows:

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

$$e \sqcap_e e' = \begin{cases} (\phi(e_1^g,\ldots,e_n^g), \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta_i'), \text{ if } e \bowtie e' \\ \text{where } \ e = \phi(e_1,\ldots,e_n) \\ \qquad\qquad e' = \phi(e_1',\ldots,e_n') \\ \qquad\qquad (e_i^g, \theta_i, \theta_i') = e_i \sqcap_e e_i' \\ (v\ v_1 \ldots v_k, \{v := \lambda v_1 \ldots v_k.e\}, \{v := \lambda v_1 \ldots v_k.e'\}), \text{ otherwise} \\ \text{where } \ \{v_1 \ldots v_k\} = bv(e) \cup bv(e') \end{cases}$$

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 application. The arguments of this application are the bound variables of the extracted expressions; this ensures that these bound variables are not extracted outside their binders. The introduced variable application is a *higher-order pattern* [15]; any term which contains the same bound variables as one of these patterns will therefore be an instance of it, as described in [16].

Definition 9 (Generalization of Process Trees). Generalization is extended to process trees using the $\sqcap_t$ operator which is defined as follows:

$$t \sqcap_t t' = \begin{cases} (e_0^g \rightarrow t_1^g,\ldots,t_n^g, \bigcup_{i=0}^n \theta_i, \bigcup_{i=0}^n \theta_i'), \text{ if } e_0 \bowtie e_0' \\ \text{where } \ t = e_0 \rightarrow t_1,\ldots,t_n \\ \qquad\qquad t' = e_0' \rightarrow t_1',\ldots,t_n' \\ \qquad\qquad (e_0^g, \theta_0, \theta_0') = e_0 \sqcap_e e_0' \\ \qquad\qquad (t_i^g, \theta_i, \theta_i') = t_i \sqcap_t t_i' \\ (v\ v_1 \ldots v_k, \{v := \lambda v_1 \ldots v_k.e_0\}, \{v := \lambda v_1 \ldots v_k.e_0'\}), \text{ otherwise} \\ \text{where } \ \{v_1 \ldots v_k\} = bv(e_0) \cup bv(e_0') \end{cases}$$

The following rewrite rule is exhaustively applied to the triple resulting from this generalization to minimize the substitutions by identifying common substitutions which were previously given different names:

$$\begin{pmatrix} e, \\ \{v_1 := e', v_2 := e'\} \cup \theta, \\ \{v_1 := e'', v_2 := e''\} \cup \theta' \end{pmatrix} \Rightarrow \begin{pmatrix} e\{v_1 := v_2\}, \\ \{v_2 := e'\} \cup \theta, \\ \{v_2 := e''\} \cup \theta' \end{pmatrix}$$

5 Folding

In this section, we describe how folding is performed in distillation. This folding is performed on process trees, rather than the flat terms used in positive supercompilation. As process trees are potentially infinite data structures, we use co-induction to define a finite method for determining whether one process tree is an instance of another.

Definition 10 (Process Tree Instance). The following co-inductive rules are used to determine whether one process tree is an instance of another:

$$\frac{\Gamma, con\langle f\rangle \equiv con'\langle f\rangle\ \theta \vdash t \equiv t'\ \theta}{\Gamma \vdash (con\langle f\rangle \rightarrow t) \equiv (con'\langle f\rangle \rightarrow t')\ \theta}\ \text{IND}$$

$$\frac{}{\Gamma, con\langle f\rangle \equiv con'\langle f\rangle\ \theta \vdash (con\langle f\rangle\ \theta \rightarrow t) \equiv (con'\langle f\rangle\ \theta \rightarrow t')\ \theta}\ \text{HYP}$$

$$\frac{\Gamma \vdash e \equiv e'\ \theta, t_i \equiv t'_i\ \theta}{\Gamma \vdash (e \rightarrow t_1, \ldots, t_n) \equiv (e' \rightarrow t'_1, \ldots, t'_n)\ \theta}\ \text{NON-IND}$$

The environment Γ here relates previously encountered corresponding expressions which have a function as their redex. To match the recursive structure of process trees, the corresponding previously encountered expressions are initially assumed to match if one is an instance of the other in the rule IND. In rule HYP, if corresponding expressions are subsequently encountered which are an instance of previously encountered ones, then we have a recursive match.

Definition 11 (Embedding Process Trees). We define the *embedding process trees* of an expression e within a process tree t (denoted by $e \overset{\emptyset}{\Rightarrow} t$) to be the finite set of subtrees of t where the root expression is coupled with e. This can be defined more formally as follows:

$$e \overset{\sigma}{\Rightarrow} (e_0 \rightarrow t_1, \ldots, t_n) = \begin{cases} \emptyset, & \text{if } \exists e'_0 \in \sigma . e'_0 \bowtie e_0 \\ \{e_0 \rightarrow t_1, \ldots, t_n\}, & \text{if } e \bowtie e_0 \\ \bigcup_{i=1}^{n} e \overset{\sigma'}{\Rightarrow} t_i, & \text{otherwise, where } \sigma' = \sigma \cup \{e_0\} \end{cases}$$

The parameter σ contains the set of expressions previously encountered within the nodes of the process tree, and will be empty initially. If the root expression of the current subtree is coupled with an expression in σ, then nothing further is added to the result set. If the root expression of the current subtree is coupled with the given expression, then the subtree is added to the result set and nothing further is added. Otherwise, the subtrees of the current node are searched for embedding process trees, and the expression in the current node is added to σ.

Definition 12 (Folding). Folding in distillation is defined as the map $\mathcal{F}$ from process trees to expressions, as defined in Fig. 6.

Within these rules, the parameter ρ contains a set of newly defined function calls and the previously encountered process trees they replaced. The rules descend through the nodes of the process tree until an expression is encountered in which the redex is a function. If the process tree rooted at this expression is an instance of a previously encountered process tree in ρ, then it is replaced by a corresponding call of the associated function in ρ. If there are no embeddings of the root expression of the current process tree, then this root node is ignored and its subtree is further folded. If there are embeddings of the root expression and at least one of them is not an instance, then the process tree is generalized and further folded; the sub-terms extracted as a result of generalization are then further distilled and substituted back in. If all of the embeddings of the root expression are instances, then a call to a newly defined function is created, and this function call is associated with the current process tree in ρ.

$$\mathcal{F}[\![(v\ e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n]\!]\ \rho = v\ (\mathcal{F}[\![t_1]\!]\ \rho) \ldots (\mathcal{F}[\![t_n]\!]\ \rho)$$

$$\mathcal{F}[\![(c\ e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n]\!]\ \rho = c\ (\mathcal{F}[\![t_1]\!]\ \rho) \ldots (\mathcal{F}[\![t_n]\!]\ \rho)$$

$$\mathcal{F}[\![(\lambda v.e)\ \rightarrow\ t]\!]\ \rho = \lambda v.(\mathcal{F}[\![t]\!]\ \rho)$$

$$\mathcal{F}[\![(con\langle(\lambda v.e_0)\ e_1\rangle)\ \rightarrow\ t]\!]\ \rho = \mathcal{F}[\![t]\!]\ \rho$$

$$\mathcal{F}[\![(con\langle \textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k\rangle)\ \rightarrow\ t]\!]\ \rho = \mathcal{F}[\![t]\!]\ \rho$$

$$\begin{array}{l} \mathcal{F}[\![(con\langle \textbf{case}\ (v\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n\rangle) \rightarrow t_0, \ldots, t_n]\!]\ \rho \\ \qquad\qquad = \textbf{case}\ (\mathcal{F}[\![t_0]\!]\ \rho)\ \textbf{of}\ p_1 \Rightarrow (\mathcal{F}[\![t_1]\!]\ \rho) \mid \cdots \mid p_n \Rightarrow (\mathcal{F}[\![t_n]\!]\ \rho) \end{array}$$

$$\mathcal{F}[\![con\langle \textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)\ \textbf{of}\ p'_1 \Rightarrow e'_1 \mid \cdots \mid p'_k \Rightarrow e'_k\rangle \rightarrow t]\!]\ \rho = \mathcal{F}[\![t]\!]\ \rho$$

$$\begin{array}{llllll}
\mathcal{F}[\![con\langle f\rangle\ \rightarrow\ t]\!]\ \rho = & \text{if} & \multicolumn{4}{l}{\exists(f'\ v_1 \ldots v_n = t') \in \rho.(con\langle f\rangle\ \rightarrow\ t) \equiv t'\ \theta} \\
& \text{then} & \multicolumn{4}{l}{(f'\ v_1 \ldots v_n)\ \theta} \\
& \text{else} & \text{if} & \multicolumn{3}{l}{(con\langle f\rangle \overset{\emptyset}{\Rightarrow} t) = \emptyset} \\
& & \text{then} & \multicolumn{3}{l}{\mathcal{F}[\![t]\!]\ \rho} \\
& & \text{else} & \text{if} & \multicolumn{2}{l}{\exists\ t' \in (con\langle f\rangle \overset{\emptyset}{\Rightarrow} t).\nexists\theta.t' \equiv (con\langle f\rangle\ \rightarrow\ t)\ \theta} \\
& & & \text{then} & \multicolumn{2}{l}{(\mathcal{F}[\![t^g]\!]\ \rho)\ \theta''} \\
& & & & \multicolumn{2}{l}{\text{where}} \\
& & & & \multicolumn{2}{l}{(con\langle f\rangle\ \rightarrow\ t)\ \sqcap_t\ t' = (t^g, \theta, \theta')} \\
& & & & \multicolumn{2}{l}{\theta = \{v_i := e_i\}} \\
& & & & \multicolumn{2}{l}{\theta'' = \{v_i := \mathcal{F}[\![\mathcal{D}[\![e_i]\!]]\!]\ \rho\}} \\
& & & \text{else} & \multicolumn{2}{l}{f'\ v_1 \ldots v_n} \\
& & & & \multicolumn{2}{l}{\text{where}} \\
& & & & \multicolumn{2}{l}{f' = \lambda v_1 \ldots v_n.\mathcal{F}[\![t]\!]\ \rho'} \\
& & & & \multicolumn{2}{l}{\rho' = \rho \cup \{f'\ v_1 \ldots v_n = con\langle f\rangle\ \rightarrow\ t\}} \\
& & & & \multicolumn{2}{l}{\{v_1 \ldots v_n\} = fv(con\langle f\rangle \rightarrow t)}
\end{array}$$

Fig. 6. Folding Rules for Distillation

6 Examples

In this section, we give some examples of the application of the distillation algorithm.

Example 4. The result of applying the driving rules to the expression *nrev xs* defined in Fig. 1 is shown in Fig. 5. When the folding rules are applied to this output, it is found that the subtree with root labelled * is coupled with the subtree with root labelled †. Generalization is therefore performed to obtain the process tree given in Fig. 7, where the extracted variable v has the value *Nil*. The subtree with root labelled † is now an instance of the subtree with root labelled *. Folding is therefore performed to obtain the program shown in Fig. 8. This program has a run-time which is linear with respect to the length of the input list, while the original program is quadratic.

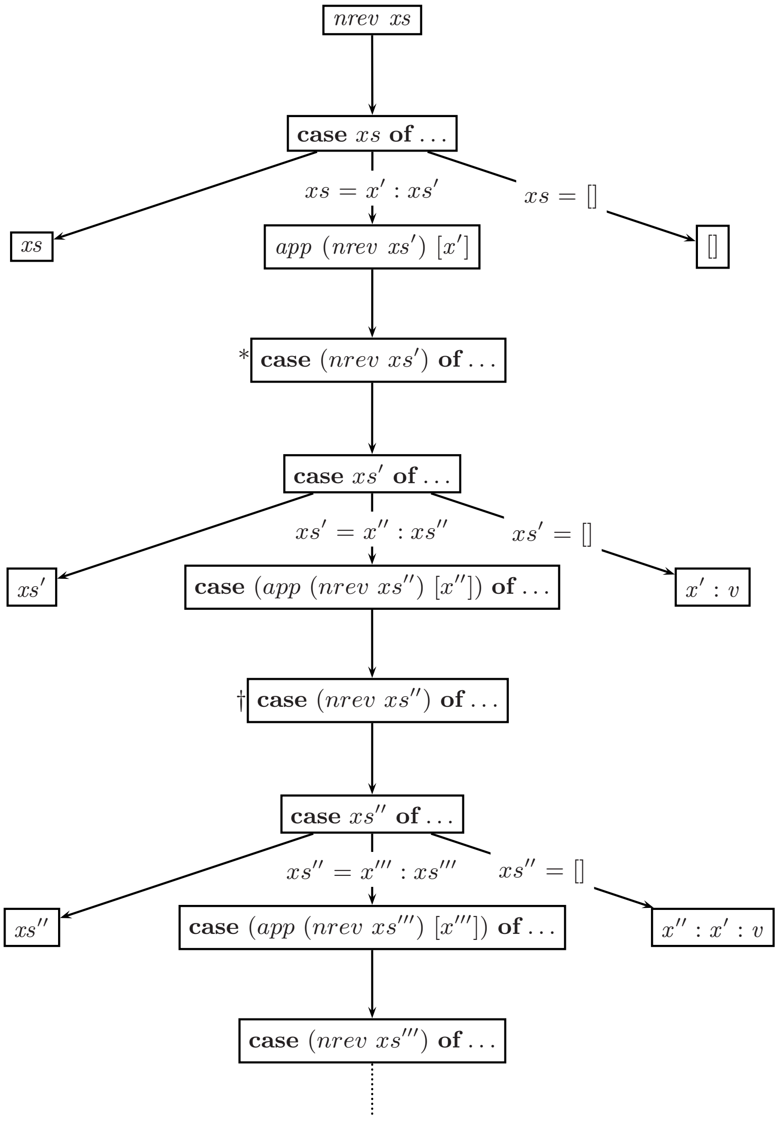


Fig. 7. Result of Generalizing *nrev xs*

Example 5. Consider the expression *app* (*arev′ xs ys*) *zs* where the functions *app* and *arev′* are as defined in Fig. 1. The result of applying the driving rules to this expression is shown in Fig. 9. When the folding rules are applied to this output, it is found that the subtree with root labelled * is coupled with the subtree with root labelled †. Generalization is therefore performed to obtain the process tree given in Fig. 10, where the extracted variable v has the value

```
case xs of
   []       ⇒ []
 | x′ : xs′ ⇒ f x′ xs′ []
where
f = λx′.λxs′.λv.case xs′ of
                   []         ⇒ x′ : v
                 | x″ : xs″ ⇒ f x″ xs″ (x′ : v)
```

Fig. 8. Result of Distilling *nrev xs*

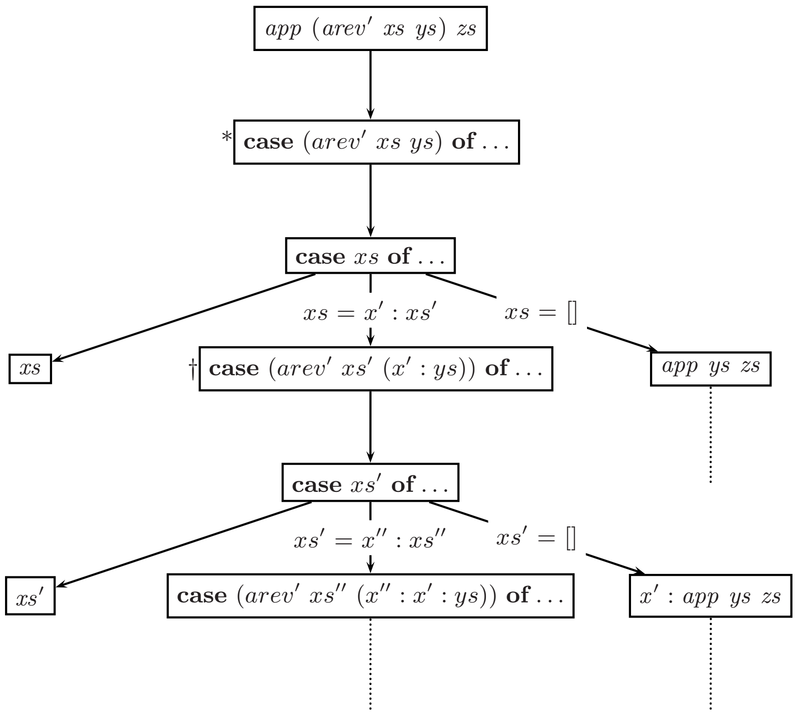
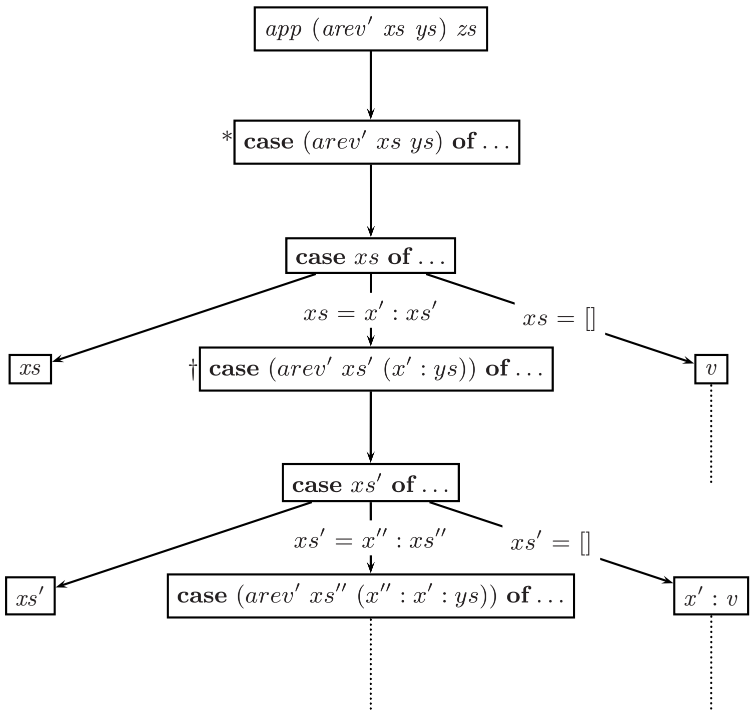


Fig. 9. Result of Driving *app* (*arev*′ *xs ys*) *zs*

app ys zs. We can now see that the subtree with root labelled † is an instance of the subtree with root labelled *. Folding is therefore performed to obtain the program shown in Fig. 11. The intermediate list (*arev*′ *xs ys*) within the initial program has therefore been eliminated. This intermediate list is not removed using positive supercompilation.



$app\ (arev'\ xs\ ys)\ zs$
$*$ **case** $(arev'\ xs\ ys)$ **of** ...
case xs **of** ...
$xs = x' : xs'$
$xs = []$
xs
$\dagger$ **case** $(arev'\ xs'\ (x' : ys))$ **of** ...
v
case xs' **of** ...
$xs' = x'' : xs''$
$xs' = []$
xs'
case $(arev'\ xs''\ (x'' : x' : ys))$ **of** ...
$x' : v$

Fig. 10. Result of Generalizing $app\ (arev'\ xs\ ys)\ zs$

$$
\begin{array}{l}
f\ xs\ (g\ ys\ zs) \\
\textbf{where} \\
f = \lambda xs.\lambda v.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\quad []\qquad\ \ \Rightarrow v \\
\qquad\qquad\qquad\ \mid x' : xs' \Rightarrow f\ xs'\ (x' : v) \\
g = \lambda ys.\lambda zs.\textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad\qquad\quad []\qquad\ \Rightarrow zs \\
\qquad\qquad\qquad\ \mid y' : ys \Rightarrow y' : (g\ ys'\ zs)
\end{array}
$$

Fig. 11. Result of Distilling $app\ (arev'\ xs\ ys)\ zs$

7 Conclusion

We have presented the distillation transformation algorithm for higher-order functional languages. The algorithm is influenced by the positive supercompilation transformation algorithm, but can produce a superlinear speedup in programs, which is not possible using positive supercompilation. 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. Firstly, we intend to incorporate the detection of non-termination into distillation and also into our theorem prover Poitín. Secondly, it has already been shown how distillation can be used to verify safety properties of programs [17]; work is now in progress to show how it can also be used to verify liveness properties. Finally, it is intended to incorporate the distillation algorithm into the Haskell programming language; this will not only allow a lot of powerful optimizations to be performed on programs in the language, but will also allow the automatic verification of properties of these programs. This will also allow the distillation algorithm to be made self-applicable as it has itself been implemented in Haskell.

References

1. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, Englewood Cliffs (1993)
2. Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees. In: Ganzinger, H. (ed.) ESOP 1988. LNCS, vol. 300, pp. 344–358. Springer, Heidelberg (1988)
3. Turchin, V.: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 90–121 (1986)
4. Sørensen, M., Glück, R., Jones, N.: A Positive Supercompiler. Journal of Functional Programming 6(6), 811–838 (1996)
5. Sørensen, M.: Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen, DIKU-rapport 94/17 (1994)
6. Wadler, P.: The Concatenate Vanishes. FP Electronic Mailing List (December 1987)
7. 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)
8. Higman, G.: Ordering by Divisibility in Abstract Algebras. Proceedings of the London Mathemtical Society 2, 326–336 (1952)
9. Kruskal, J.: Well-Quasi Ordering, the Tree Theorem, and Vazsonyi's Conjecture. Transactions of the American Mathematical Society 95, 210–225 (1960)
10. Dershowitz, N., Jouannaud, J.P.: Rewrite Systems. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 243–320. Elsevier, MIT Press (1990)
11. Sørensen, M., Glück, R.: An Algorithm of Generalization in Positive Supercompilation. In: Tison, S. (ed.) CAAP 1994. LNCS, vol. 787, pp. 335–351. Springer, Heidelberg (1994)
12. Marlet, R.: Vers une Formalisation de l'Évaluation Partielle. PhD thesis, Université de Nice - Sophia Antipolis (1994)
13. Bol, R.: Loop Checking in Partial Deduction. Journal of Logic Programming 16(1-2), 25–46 (1993)
14. Leuschel, M.: On the Power of Homeomorphic Embedding for Online Termination. In: Proceedings of the International Static Analysis Symposium, pp. 230–245 (1998)
15. Miller, D.: A Logic Programming Language with Lambda-Abstraction, Function Variables and Simple Unification. In: Schroeder-Heister, P. (ed.) ELP 1989. LNCS, vol. 475, pp. 253–281. Springer, Heidelberg (1991)
16. Nipkow, T.: Functional Unification of Higher-Order Patterns. In: Eighth Annual Symposium on Logic in Computer Science, pp. 64–74 (1993)
17. Hamilton, G.W.: Distilling Programs for Verification. Electronic Notes in Theoretical Computer Science 190(4), 17–32 (2007)