Distillation:

Extracting the Essence of Programs

G.W. Hamilton
School of Computing, Dublin City University
hamilton@computing.dcu.ie

Abstract

In this paper, we present a new transformation algorithm called *distillation* which can automatically transform higher-order functional programs into equivalent tail-recursive programs. Using this algorithm, it is possible to produce superlinear improvement in the run-time of programs. This represents a significant advance over the supercompilation algorithm, which can only produce a linear improvement. Outline proofs are given that the distillation algorithm is correct and that it always terminates.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—optimization; I.2.2 [*Artificial Intelligence*]: Automatic Programming—program transformation

General Terms Languages, Performance

Keywords Program transformation, tail-recursion, superlinear improvement, generalisation, termination.

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 [10], 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* [11], *deforestation* [30] and *supercompilation* [26]. Although supercompilation is strictly more powerful than both partial evaluation and deforestation, Sørensen has shown that supercompilation (and hence also partial evaluation and deforestation) can only produce a linear speedup in programs [24]. A more powerful transformation algorithm should be able to produce a superlinear speedup in programs.

EXAMPLE 1. Consider the program shown in Figure 1. This program reverses the list xs, but the recursive function call $rev\ xs$ is an intermediate data structure, so in terms of time and space usage, it is quadratic in the length of the list xs. A more efficient program which is linear in the length of the list xs is shown in Figure 2. A number of algebraic transformations have been proposed which can perform this transformation (e.g. [29]) by appealing to a specific law stating the associativity of the app function. However,

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

Figure 1. Example Program

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

Figure 2. Example Program Transformed

none of the generic program transformation techniques mentioned above are capable of performing this transformation. □

In this paper, we present a transformation algorithm called distillation which will allow transformations such as the above to be performed. Many of the rules of distillation resemble those of supercompilation; however, supercompilation may *over-generalize* by extracting sub-terms which are intermediate data structures. In distillation, we try to avoid this over-generalization as much as possible.

The extra power of the distillation algorithm is obtained through the use of a more powerful matching mechanism prior to folding. In supercompilation, matching is performed on flat terms; functions are considered to match only if they have the same name. In the distillation algorithm, matching is performed on recursive terms, so different functions are considered to match if their corresponding recursive definitions also match.

The remainder of this paper is structured as follows. In Section 2, we define the higher-order language on which the described transformations are performed. In Section 3, we give an overview of supercompilation; this overview is largely based on [23]. In Section 4, we show how the termination of supercompilation can be ensured. In Section 5, we describe how supercompilation can be extended to give distillation, and give some examples of the application of distillation. In Section 6, we give outline proofs that the distillation algorithm is correct and that it always terminates. Section 7 considers related work and concludes.

2. Language

In this section, we describe the language which will be used throughout this paper.

DEFINITION 2.1 (Language). The language for which the described transformations are to be performed is a simple higher-order functional language as shown in Figure 3.

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

Figure 3. Language Grammar

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 the language is typed using the Hindley-Milner polymorphic typing system [17, 6] (so erroneous terms such as $(c\ e_1 \ldots e_n)\ e$ and **case** $(\lambda v.e)$ **of** $p_1 : e_1 \mid \cdots \mid p_k : e_k$ 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 $fv(e)$ to denote the free variables of expression e. We require that each function has exactly one definition and that all variables within a definition are bound. 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. We allow the usual notation [] for *Nil*, $x :: xs$ for *Cons* $x\ xs$ and $[x_1, \ldots, x_n]$ for *Cons* $x_1 \ldots$ *Cons* x_n *Nil*. We also allow the notation 0 for *Zero*, 1 for *Succ Zero* and $n+1$ for *Succ* n.

Within the expression **case** e_0 **of** $p_1 : e_1 \mid \cdots \mid p_k : e_k$, 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. Methods to transform **case** expressions with nested patterns to ones without nested patterns are described in [3, 28]. No variables may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive. □

DEFINITION 2.2 (Intermediate Data Structure). An intermediate data structure is a structure which is created using a constructor application, and is subsequently decomposed as the selector in a **case** expression. □

3. Supercompilation

In this section, we give an overview of the supercompilation algorithm. This is largely based on the presentation given in [23]. Transformation within supercompilation is perfomed by a number of rewrite rules which take a term (possibly containing free variables) and a program, and apply normal order reduction rules to produce a *partial process tree*, from which a residual program can be constructed.

We define the rules for supercompilation 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 3.1 (Redexes, Contexts and Observables). Redexes, contexts and observables are defined by the grammar shown in Figure 4, where red ranges over redexes, con ranges over contexts and obs ranges over observables. □

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

Figure 4. Grammar of Redexes, Contexts and Observables

The expression $con\langle e \rangle$ denotes the result of replacing the 'hole' $\langle\rangle$ in con by e.

LEMMA 3.2 (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 3.3 (Normal Order Reduction). The core set of transformation rules for supercompilation are the normal order reduction rules shown in Figure 5 which defines the map $\mathcal{N}$ from expressions to ordered sequences of expressions $[e_1, \ldots, e_n]$.

$$
\begin{array}{lcl}
\mathcal{N}[\![v\ e_1 \ldots e_n]\!] & = & [e_1, \ldots, e_n] \\
\mathcal{N}[\![c\ e_1 \ldots e_n]\!] & = & [e_1, \ldots, e_n] \\
\mathcal{N}[\![\lambda v.e]\!] & = & [e] \\
\mathcal{N}[\![con\langle f \rangle]\!] & = & [unfold\ (con\langle f \rangle)] \\
\mathcal{N}[\![con\langle(\lambda v.e_0)\ e_1\rangle]\!] & = & [con\langle e_0\{v := e_1\}\rangle]
\end{array}
$$

$$
\begin{array}{l}
\mathcal{N}[\![con\langle\ \textbf{case}\ (v\ e_1 \ldots e_n)\ \textbf{of} \\
\qquad\qquad p_1 \ :\ e_1'\{v' := v\ e_1 \ldots e_n\} \\
\qquad\qquad \vdots \\
\qquad\qquad p_k \ :\ e_k'\{v' := v\ e_1 \ldots e_n\}\rangle]\!] \\
= [v\ e_1 \ldots e_n, con\langle e_1'\{v' := p_1\}\rangle, \ldots, con\langle e_k'\{v' := p_k\}\rangle] \\
\mathcal{N}[\![con\langle\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 : e_1' \mid \cdots \mid p_k : e_k'\rangle]\!] \\
= [con\langle e_i\{v_1 := e_1, \ldots, v_n := e_n\}\rangle] \\
\quad \text{where } p_i = c\ v_1 \ldots v_n
\end{array}
$$

Figure 5. Normal Order Reduction Rules for Supercompilation

We use the notation $e\{v_1 := e_1, \ldots, v_n := e_n\}$ to represent the simultaneous substitution of the sub-expressions $e_1, \ldots, e_n$ for the free occurrences of variables $v_1, \ldots, v_n$, respectively, within e. The function *unfold* unfolds the function in the redex of its argument expression as follows:

$$unfold\ (con\langle f \rangle) = con\langle e \rangle \text{ where } f \text{ is defined by } f = e$$

The above reduction rules are mutually exclusive and exhaustive by the unique decomposition property. The rules simply perform normal order reduction, with information propagation within **case** expressions giving the assumed outcome of the test (this is called *unification-based* information propagation in [23]). □

DEFINITION 3.4 (Process Trees). A *process tree* is a directed acyclic graph 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. 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, and

$t\{\alpha := t'\}$ denotes the tree obtained by replacing the subtree with root α in t by the tree t'. Finally, the tree $e \rightarrow t_1, \ldots, t_n$ is the tree with root labelled e and n children which are the subtrees $t_1, \ldots, t_n$ respectively. □

DEFINITION 3.5 (Construction of Process Trees). A process tree is constructed from an expression e using the following rule:

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

□

DEFINITION 3.6 (Partial Process Trees). A *partial process tree* is a process tree which may contain *repeat nodes*. A repeat node has a dashed edge to an ancestor within the process tree. □

DEFINITION 3.7 (Instance). An expression e is an *instance* of expression e', denoted by $e' \lessdot e$, if there is a substitution θ such that $e'\theta \equiv e$. □

Repeat nodes correspond to a fold step during transformation. When a term is encountered which is an instance of an ancestor term within the process tree, a repeat node is created. This matching ancestor is called a *function node*.

Thus, if the current expression is e, and there is an ancestor node α within the process tree which is labelled with an instance of e, then a dashed edge $e \dashrightarrow \alpha$ is created within the process tree, representing the occurrence of a repeat node. As any infinite sequence of transformation steps must involve the unfolding of a function, we only check for the occurrence of a repeat node when the redex of the current expression is a function.

EXAMPLE 2. Consider the program shown in Figure 6. Transformation of this program produces the partial process tree given in Figure 7[1]. □

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

Figure 6. Example Program

DEFINITION 3.8 (Residual Program Construction). A residual program can be constructed from the partial process tree resulting from supercompilation using the rules $\mathcal{C}$ as shown in Figure 8. □

The recursive functions which are introduced within residual programs are defined using local function definitions of the form **letrec** $f = e_0$ **in** e_1. The parameters of these functions are those variables within the function node which are instantiated with a different value within the corresponding repeat node, so the functions may contain non-local variables. The residual program constructed from the partial process tree in Figure 7 is shown in Figure 9.

4. Termination

If the transformation rules for supercompilation were left unsupervised, possible non-termination could arise, even in the presence of folding. In this section, we show how to ensure termination of the distillation algorithm through the use of *generalization*. To represent the result of generalization, we introduce a **let** construct of the

[1] This process tree, and later ones presented in this paper, have been simplified for ease of presentation. In particular nodes which contain labels of the form $con\langle(\lambda v.e_0)\ e_1\rangle$ and $con\langle\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of} \ldots\rangle$ have been omitted.

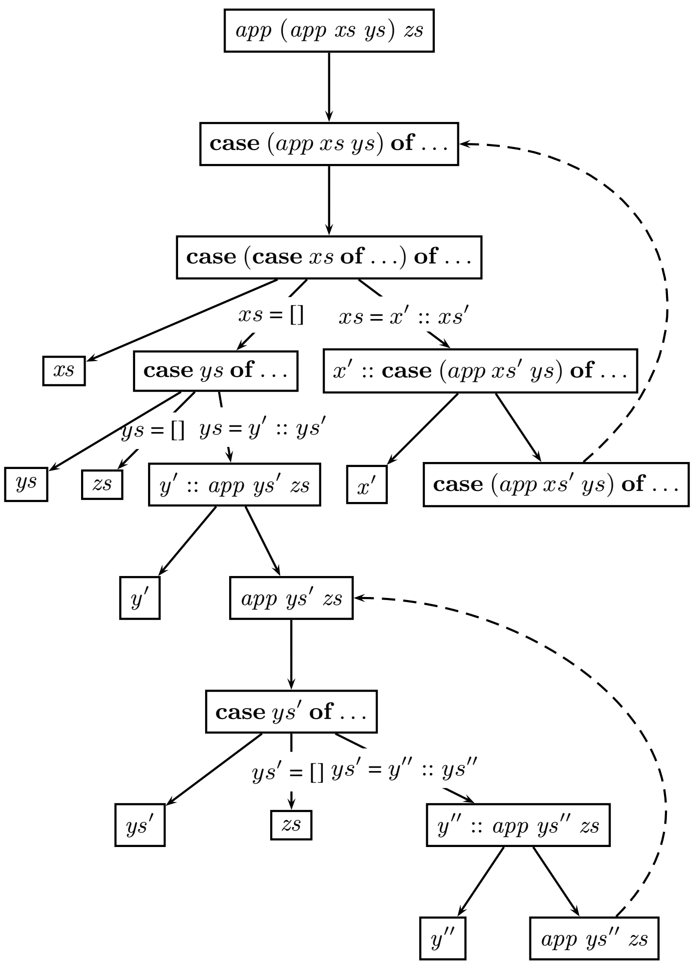

Figure 7. Example Partial Process Tree

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 reduction rule for this new construct is as follows:

$$\mathcal{N}[\![e\langle\textbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \textbf{in}\ e_0\rangle]\!] = [e_0, e_1, \ldots, e_n]$$

Partial process trees are extended to include **let** constructs and the rules for constructing a residual program from a partial process tree can also be extended for **let** constructs as follows:

$$\begin{array}{l}\mathcal{C}[\![(e\langle\textbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \textbf{in}\ e_0\rangle) \rightarrow t_0, \ldots, t_n]\!] \\ \quad = \quad \mathcal{C}[\![t_0]\!]\{v_1 := \mathcal{C}[\![t_1]\!], \ldots, v_n := \mathcal{C}[\![t_n]\!]\}\end{array}$$

In [24], three different possible causes of non-termination are identified within a first-order functional language: *obstructing function calls*, *accumulating parameters* and *accumulating side-effects*[2] [3]. We now give examples of each of these causes of non-termination.

EXAMPLE 3 (Obstructing Function Call). Consider the program shown in Figure 1. During the transformation of this program, we

[2] Since functional languages do not admit side-effects, we prefer to call this possible cause of non-termination *accumulating patterns*.

[3] A further possible cause of non-termination has also been identified in [21] for higher-order functional languages: *accumulating spines*. In the Hindley-Milner polymorphic typing system assumed for our language, this situation cannot occur, so we can safely ignore it.

$$
\begin{array}{lcl}
\mathcal{C}[\![(v\ e_1 \ldots e_n) \rightarrow t_1,\ldots,t_n]\!] & & \\
 & = & v\ (\mathcal{C}[\![t_1]\!]) \ldots (\mathcal{C}[\![t_n]\!]) \\
\mathcal{C}[\![(c\ e_1 \ldots e_n) \rightarrow t_1,\ldots,t_n]\!] & & \\
 & = & c\ (\mathcal{C}[\![t_1]\!]) \ldots (\mathcal{C}[\![t_n]\!]) \\
\mathcal{C}[\![(\lambda v.e)\ \rightarrow\ t]\!] & = & \lambda v.(\mathcal{C}[\![t]\!]) \\
\mathcal{C}[\![\alpha = (con\langle f\rangle)\ \rightarrow\ t]\!] & = & \mathbf{letrec}\, f' \ =\ \lambda v_1 \ldots v_n.\mathcal{C}[\![t]\!] \\
 & & \mathbf{in}\, f'\ v_1 \ldots v_n,\ \text{if } \exists\beta \in t.\beta \dashrightarrow \alpha \\
 & & \beta \equiv \alpha\{v_1 := e_1,\ldots,v_n := e_n\} \\
 & = & \mathcal{C}[\![t]\!], \text{ otherwise} \\
\mathcal{C}[\![\beta = (con\langle f\rangle) \dashrightarrow \alpha]\!] & = & (f'\ e_1 \ldots e_n) \\
 & & \beta \equiv \alpha\{v_1 := e_1,\ldots,v_n := e_n\} \\
\mathcal{C}[\![(con\langle(\lambda v.e_0)\ e_1\rangle)\ \rightarrow\ t]\!] & & \\
 & = & \mathcal{C}[\![t]\!] \\
\mathcal{C}[\![(con\langle\mathbf{case}\ (c\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 : e_1' \mid \cdots \mid p_k : e_k'\rangle)\ \rightarrow\ t]\!] & & \\
 & = & \mathcal{C}[\![t]\!] \\
\mathcal{C}[\![(con\langle\mathbf{case}\ (v\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 : e_1 \mid \cdots \mid p_n : e_n\rangle) \rightarrow t_0,\ldots,t_n]\!] & & \\
 & = & \\
\multicolumn{3}{r}{\mathbf{case}\ (\mathcal{C}[\![t_0]\!])\ \mathbf{of}\ p_1 : \mathcal{C}[\![t_1]\!] \mid \cdots \mid p_n : \mathcal{C}[\![t_n]\!]}
\end{array}
$$

Figure 8. Rules For Constructing Residual Programs

```
letrec f =
λxs.case xs of
      []      : case ys of
                  []          :  zs
                  Cons y' ys'  :
                y' :: (letrec g =
                       λys.case ys of
                             []      : zs
                           | y' :: ys' : y' :: (g ys')
                       in g ys')
    | x' :: xs' : x' :: (f xs')
in f xs
```

Figure 9. Constructed Residual Program

encounter the progressively larger terms: $rev\ xs$, **case** $(rev\ xs)$ **of** $\cdots$, **case** (**case** $(rev\ xs)$ **of** $\cdots$) **of** $\cdots$, etc. The call to rev thus prevents the surrounding context from being reduced, so this context continues to grow. This call to rev is therefore an obstructing function call. In supercompilation, this problem is avoided by extracting the obstructing function call from its surrounding context, and transforming it separately. □

EXAMPLE 4 (Accumulating Parameter). Consider the program shown in Figure 2. During transformation of this program, we encounter the progressively larger terms: $rev'\ xs\ []$, $rev'\ xs'\ [x']$, $rev'\ xs''\ [x'', x']$, etc. The second parameter in each recursive call to rev' therefore accumulates a progressively larger term. In supercompilation, this problem is avoided by extracting the accumulating parameter and transforming it separately. □

EXAMPLE 5 (Accumulating Pattern). Consider the program shown in Figure 10.

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

Figure 10. Example Accumulating Pattern

During transformation of this program, we encounter the progressively larger terms: $app\,xs\,xs$, $app\,xs'\,(x' : xs')$, $app\,xs''\,(x' : x'' : xs'')$, etc. The second parameter in each recursive call to app therefore also accumulates a progressively larger term, but in this case the accumulation is caused by unification-based information propagation, which would not occur within deforestation. In supercompilation, this problem is avoided in the same way as for accumulating parameters by extracting the accumulating variable and transforming it separately. □

In all of the above cases, a previously encountered term becomes embedded within the current term. We therefore allow transformation to continue until an embedding of a previously encountered term is encountered within the current one, at which point generalization is performed to ensure termination of the transformation process.

4.1 Homeomorphic Embedding

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 [9] and Kruskal [14] and was defined within term rewriting systems [7] for detecting the possible divergence of the term rewriting process. Variants of this relation have been used to ensure termination within supercompilation [23], partial evaluation [16] and partial deduction [4, 15]. It can be shown that the homeomorphic embedding relation $\trianglelefteq$ is a *well-quasi-order*, which is defined as follows.

DEFINITION 4.1 (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. If $e_i \trianglelefteq e_j$ then all of the sub-expressions of e_i are present in e_j embedded in extra sub-expressions. This is defined more formally as follows.

DEFINITION 4.2 (Homeomorphic Embedding Relation).

$$
\begin{array}{ccc}
\text{Variable} & \text{Diving} & \text{Coupling} \\
x \trianglelefteq y & \dfrac{e \trianglelefteq e_i \text{ for some } i}{e \trianglelefteq \phi(e_1,\ldots,e_n)} & \dfrac{e_i \trianglelefteq e_i' \text{ for all } i}{\phi(e_1,\ldots,e_n) \trianglelefteq \phi(e_1',\ldots,e_n')}
\end{array}
$$

Diving detects a sub-expression embedded in a larger expression, and coupling matches all the sub-expressions of two expressions which have the same top-level functor. 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 corresponding binders within the two expressions must also match up. □

EXAMPLE 6. Some examples of the homeomorphic embedding relation are as follows.

1.	$f_1\ x \trianglelefteq f_2\ (f_1\ y)$	5.	$f\ (g\ x) \not\trianglelefteq f\ y$
2.	$f_1\ x \trianglelefteq f_1\ (f_2\ y)$	6.	$f\ (g\ x) \not\trianglelefteq g\ y$
3.	$f_1\ (f_3\ x) \trianglelefteq f_1\ (f_2\ (f_3\ y))$	7.	$f\ (g\ x) \not\trianglelefteq g\ (f\ y)$
4.	$f_1(x, x) \trianglelefteq f_1(f_2\ y, f_2\ y)$	8.	$f\ (g\ x) \not\trianglelefteq f\ (h\ y)$ □

DEFINITION 4.3 (Strict Embedding). An expression e is *strictly embedded* within an expression $\phi(e_1, \ldots, e_n)$, denoted by $e \triangleleft \phi(e_1, \ldots, e_n)$, if $e \trianglelefteq e_i$ for some i. □

For example, in the first item above, $f_1\ x \triangleleft f_2\ (f_1\ y)$.

4.2 Generalization

In supercompilation, generalization is performed when an expression e is encountered which is an embedding of an expression e' in an ancestor node within the partial process tree. This generalization of e and e' is the *most specific generalization*, denoted by $e \sqcap e'$, as defined in term algebra [7]. When an expression is generalized, sub-expressions within it are replaced with variables, which implies a loss of knowledge about the expression. The most specific generalization therefore entails the least possible loss of knowledge.

DEFINITION 4.4 (Generalization). A generalization of expressions e and e' is a triple (e_g, θ, θ') where θ and θ' are substitutions such that $e_g\theta \equiv e$ and $e_g\theta' \equiv e'$. $\square$

DEFINITION 4.5 (Most Specific Generalization). A most specific generalization of expressions e and e' is a generalization (e_g, θ, θ') such that for every other generalization $(e'_g, \theta'', \theta''')$ of e and e', e_g is an instance of e'_g. The most specific generalization, denoted by $e \sqcap e'$, of two expressions e and e' is computed by exhaustively applying the following rewrite rules to the initial triple $(v, \{v := e\}, \{v := e'\})$.

$$\left(\begin{array}{c} e, \\ \{v := \phi(e_1, \ldots, e_n)\} \cup \theta, \\ \{v := \phi(e'_1, \ldots, e'_n)\} \cup \theta' \end{array}\right) \Downarrow \left(\begin{array}{c} e\{v := \phi(v_1, \ldots, v_n)\}, \\ \{v_1 := e_1, \ldots, v_n := e_n\} \cup \theta, \\ \{v_1 := e'_1, \ldots, v_n := e'_n\} \cup \theta' \end{array}\right)$$

$$\left(\begin{array}{c} e, \\ \{v_1 := e', v_2 := e'\} \cup \theta, \\ \{v_1 := e'', v_2 := e''\} \cup \theta' \end{array}\right) \Downarrow \left(\begin{array}{c} e\{v_1 := v_2\}, \\ \{v_2 := e'\} \cup \theta, \\ \{v_2 := e''\} \cup \theta' \end{array}\right)$$

$\square$

The first of these rewrite rules is for the case where both expressions have the same functor at the outermost level. In this case, this is made the outermost functor of the resulting generalized expression, and this functor is removed from each of the two expressions. The second rule identifies common sub-expressions within an expression. The results of applying this most specific generalization to items 1-4 in Example 6 are as follows:

1. $(v, \{v := f_1\ x\}, \{v := f_2\ (f_1\ y)\})$
2. $(f_1\ v, \{v := x\}, \{v := f_2\ y\})$
3. $(f_1\ v, \{v := f_3\ x\}, \{v := f_2\ (f_3\ y)\})$
4. $(f_1(v, v), \{v := x\}, \{v := f_2\ y\})$

When we encounter an expression e which is an embedding of a previously encountered expression e', we perform generalization. If e' is strictly embedded within e, then there is an obstructing function call, so the partial process subtree rooted at e is replaced by the result of transforming a generalized form of e in which the obstructing function call is extracted.

DEFINITION 4.6 (Extraction). The *extraction* of an obstructing function call e' from a term e, denoted by $e' \leftarrow e$, is defined by applying the following rewrite rules to the initial triple $(\langle\rangle, e', e)$.

$$\begin{array}{c} (con, f(e_1, \ldots, e_n), f'(e'_1, \ldots, e'_n)) \\ \Downarrow \\ (con, f(e_1, \ldots, e_n), unfold(f'(e'_1, \ldots, e'_n))) \end{array}$$

$$\begin{array}{c} (con, f(e_1, \ldots, e_n), \textbf{case}\ e'_0\ \textbf{of}\ p_1 : e'_1 \mid \cdots \mid p_k : e'_k) \\ \Downarrow \\ (\textbf{case}\ con\ \textbf{of}\ p_1 : e'_1 \mid \cdots \mid p_k : e'_k, f(e_1, \ldots, e_n), e'_0) \end{array}$$

$\square$

The generalized form of an expression e resulting from extracting an obstructing function call e' from it is constructed using the $extract$ operation.

DEFINITION 4.7 (Extract Operation).

$extract(e', e) = \textbf{let}\ v\ =\ e_g\ \textbf{in}\ con\langle v\rangle$
where $e' \leftarrow e = (con, e', e_g)$ $\square$

If e is a non-strict embedding of e', then we calculate the most specific generalization of e and e'. The partial process subtree rooted at e' is then replaced by the result of transforming the generalized form of e', which is constructed using the $abstract$ operation.

DEFINITION 4.8 (Abstract Operation).

$abstract(e, e') = \textbf{let}\ v_1\ =\ e_1, \ldots, v_n\ =\ e_n\ \textbf{in}\ e_g$
where $e \sqcap e' = (e_g, \theta, \{v_1 := e_1, \ldots, v_n := e_n\})$ $\square$

If a sub-expression which is to be extracted using a **let** expression contains bound variables, then the sub-expression is not moved outside of the bound variable bindings.

We now give a more formal definition of supercompilation. If the partially constructed process tree t contains a node β in which the redex is a function, we process this node as follows:

```
if $\exists\alpha \in anc(t,\beta).t(\alpha) \leq t(\beta)$
then $t\{\beta := t(\beta) \dashrightarrow \alpha\}$
else if $\exists\alpha \in anc(t,\beta).t(\alpha) \lhd t(\beta)$
    then $t\{\beta := \mathcal{T}[\![extract(t(\alpha), t(\beta))]\!]\}$
    else if $\exists\alpha \in anc(t,\beta).t(\alpha) \unlhd t(\beta)$
        then $t\{\alpha := \mathcal{T}[\![abstract(t(\alpha), t(\beta))]\!]\}$
        else $t\{\beta := t(\beta) \rightarrow \mathcal{T}[\![unfold(t(\beta))]\!]\}$
```

All other nodes are processed as shown in Definition 3.5.

5. Distillation

In this section, we present the distillation algorithm, which is more powerful than supercompilation. This extra power is obtained through the use of more powerful matching prior to folding. In supercompilation, matching is performed on flat terms only; functions are considered to match only if they have the same name. In the distillation algorithm, matching is also performed on recursive terms, so different functions are considered to match if their corresponding recursive definitions also match. The output of the distillation algorithm is also a partial process tree, from which a residual program can be constructed.

Many of the sub-expressions which are extracted using generalization within supercompilation may actually be intermediate within the resulting generalized expression, but will not be transformed away. In distillation, we therefore further transform the program which would be obtained by supercompilation to try and remove these intermediate structures. The new rule for transforming a node β within a partially constructed tree t, where the label of β contains a function in the redex position is therefore as follows:

```
if $\exists\alpha \in anc(t,\beta).t(\alpha) \leq t(\beta)$
then $(t\{\beta := t(\beta) \dashrightarrow \alpha\})\{\alpha := \mathcal{T}[\![\mathcal{C}[\![\alpha]\!]]\!]\}$
else if $\exists\alpha \in anc(t,\beta).t(\alpha) \lhd t(\beta)$
    then $t\{\beta := \mathcal{T}[\![extract(t(\alpha), t(\beta))]\!]\}$
    else if $\exists\alpha \in anc(t,\beta).t(\alpha) \unlhd t(\beta)$
```

```
        then $t\{\alpha := \mathcal{T}[\![\mathcal{C}[\![\mathcal{T}[\![abstract(t(\alpha), t(\beta))]\!]]\!]]\!]\}$
        else $t\{\beta := t(\beta) \rightarrow \mathcal{T}[\![unfold\ (t(\beta))]\!]\}$
```

Nodes which contain **letrec** expressions in the redex position are also used to determine when folding or generalization should be performed. The instance and embedding relations extend naturally to the **letrec** construct. If the node currently being processed contains a **letrec** expression in the redex position and is an instance of an expression within an ancestor node, then a repeat node is created. If the node contains a **letrec** expression in the redex position and is an embedding of an expression within an ancestor node, then generalization is performed as before; however, this generalization is permanent and is not further transformed. Otherwise, the **letrec** function is unfolded, and the resulting expression is further transformed. The rule for transforming a node β within a partially constructed tree t, where the label of β contains a **letrec** expression in the redex position is therefore as follows:

```
if $\exists\alpha \in anc(t, \beta).t(\alpha) \lessdot t(\beta)$
then $(t\{\beta := t(\beta) \dashrightarrow \alpha\})$
else if $\exists\alpha \in anc(t, \beta).t(\alpha) \trianglelefteq t(\beta)$
    then $t\{\alpha := \mathcal{T}[\![abstract(t(\alpha), t(\beta))]\!]\}$
    else $t\{\beta := t(\beta) \rightarrow \mathcal{T}[\![unfold\ t(\beta)]\!]\}$
```

The rules for constructing a residual program from a node in which the label contains a **letrec** expression in the redex position are as follows:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}[\![\alpha = con\langle \mathbf{letrec}\, f\ =\ e_0\ \mathbf{in}\ e_1\rangle\ \rightarrow\ t]\!]} \\
= & \mathbf{letrec}\, f'\ =\ \lambda v_1 \ldots v_n.\mathcal{C}[\![t]\!] \\
 & \mathbf{in}\, f'\ v_1 \ldots v_n,\ \text{if } \exists\beta \in t.\beta \dashrightarrow \alpha \\
 & \beta \equiv \alpha\{v_1 := e_1, \ldots, v_n := e_n\} \\
= & \mathcal{C}[\![t]\!],\ \text{otherwise} \\
\multicolumn{2}{l}{\mathcal{C}[\![\beta = con\langle \mathbf{letrec}\, f\ =\ e_0\ \mathbf{in}\ e_1\rangle \dashrightarrow \alpha]\!]} \\
= & (f'\ e_1 \ldots e_n) \\
 & \beta \equiv \alpha\{v_1 := e_1, \ldots, v_n := e_n\}
\end{array}$$

We now give some examples to show how distillation copes with the different causes of non-termination identified in Section 4. We go through one example in detail to illustrate our technique, but due to space constraints, only the reults obtained for the other examples are given.

EXAMPLE 7 (Obstructing Function Call). Consider the program shown in Figure 1. Starting from the original term:

$$rev\ xs \tag{1}$$

We subsequently encounter the following term:

$$app\ (rev\ xs')\ [x'] \tag{2}$$

Term (1) is strictly embedded within term (2), so the obstructing function call is extracted from term (2) using the $extract$ operation to obtain the following term:

$$\mathbf{let}\ vs\ =\ rev\ xs'\ \mathbf{in}\ app\ vs\ [x'] \tag{3}$$

The subterm $rev\ xs'$ is intermediate within this term, but will not be removed by supercompilation. After supercompilation of the original term, the partial process tree shown in Figure 11 is obtained. The following residual program is constructed from this partial process tree:

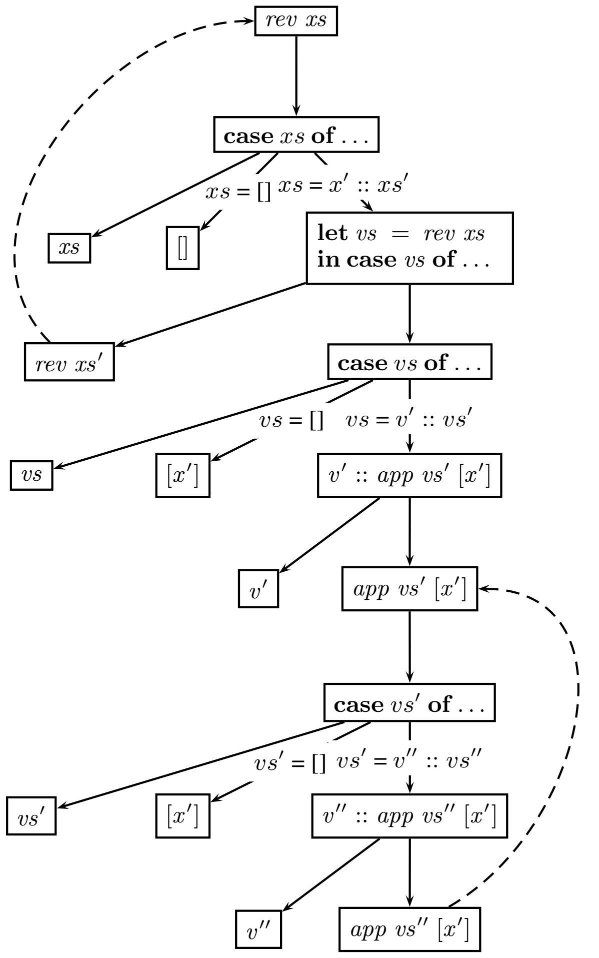

Figure 11. Partial Process Tree During Transformation (1)

$$\begin{array}{l}
\mathbf{letrec}\, f\ = \\
\lambda xs.\mathbf{case}\ xs\ \mathbf{of} \\
\quad\quad []\quad\quad : [] \\
\quad\ \mid x' :: xs' : \mathbf{case}\ vs\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad [] \quad\quad : \quad [x'] \\
\quad\quad\quad\quad\quad\quad v' :: vs' \quad : \\
\quad\quad\quad\quad\quad v' :: \mathbf{letrec}\ g\ = \\
\quad\quad\quad\quad\quad\quad \lambda vs'.\mathbf{case}\ vs'\ \mathbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad [] \quad\quad : [x'] \\
\quad\quad\quad\quad\quad\quad\quad \mid v'' :: vs'' : v'' :: (g\ vs'') \\
\quad\quad\quad\quad\quad\quad \mathbf{in}\ g\ (f\ xs') \\
\mathbf{in}\, f\ xs
\end{array} \tag{4}$$

The function f is then unfolded, and after further transformation the following term is obtained:

$$\begin{array}{l}
\mathbf{case}\ (f\ xs')\ \mathbf{of} \\
\quad [] \quad\quad : [x'] \\
\ \mid v' :: vs' : v' :: (g\ vs')
\end{array} \tag{5}$$

After further transformation, the following term is obtained:

$$\begin{array}{l} \textbf{case}\ (\textbf{case}\ (f\ xs'')\ \textbf{of} \\ \qquad [\,] \qquad : [x''] \\ \qquad \mid v' :: vs' : v' :: (g\ vs'))\ \textbf{of} \\ \quad [\,] \qquad : [x'] \\ \quad \mid v' :: vs' : v' :: (g\ vs') \end{array} \tag{6}$$

Term (5) is strictly embedded within term (6), so we extract this more deeply embedded term to give the following:

$$\begin{array}{l} \textbf{let}\ vs\ =\ \textbf{case}\ (f\ xs'')\ \textbf{of} \\ \qquad\qquad [\,] \qquad : [x''] \\ \qquad\qquad \mid v' :: vs' : v' :: (g\ vs') \\ \textbf{in case}\ vs\ \textbf{of} \\ \qquad [\,] \qquad : [x'] \\ \qquad \mid v' :: vs' : v' :: (g\ vs') \end{array} \tag{7}$$

After further transformation, the partial process tree shown in Figure 12 is obtained. The following residual program is constructed from this partial process tree:

$$\begin{array}{l} \textbf{letrec}\ f\ = \\ \lambda xs'.\textbf{case}\ xs'\ \textbf{of} \\ \qquad [\,] \qquad : [x'] \\ \qquad \mid x'' :: xs'' : \textbf{case}\ (f\ xs'')\ \textbf{of} \\ \qquad\qquad\qquad [\,] \qquad : \quad [x'', x'] \\ \qquad\qquad\qquad v' :: vs' \quad : \\ \qquad\qquad\qquad v' :: (\textbf{letrec}\ g\ = \\ \qquad\qquad\qquad\quad \lambda vs'.\textbf{case}\ vs'\ \textbf{of} \\ \qquad\qquad\qquad\qquad [\,] \qquad : [x'', x'] \\ \qquad\qquad\qquad\qquad \mid v'' :: vs'' : v'' :: (g\ vs'') \\ \qquad\qquad\qquad\quad \textbf{in}\ g\ vs') \\ \textbf{in}\ f\ xs' \end{array} \tag{8}$$

We can now see that term (8) is an embedding of term (4), so the most specific generalization of term (4) is performed, which results in the permanent extraction of the sub-term [] from the tail of the list $[x']$, giving the following term:

$$\begin{array}{l} \textbf{let}\ v\ =\ [\,] \\ \textbf{in letrec}\ f\ = \\ \quad \lambda xs.\textbf{case}\ xs\ \textbf{of} \\ \qquad [\,] \qquad : v \\ \qquad \mid x' :: xs' : \textbf{case}\ (f\ xs')\ \textbf{of} \\ \qquad\qquad\qquad [\,] \qquad : \quad x' :: v \\ \qquad\qquad\qquad v' :: vs' \quad : \\ \qquad\qquad\qquad v' :: (\textbf{letrec}\ g\ = \\ \qquad\qquad\qquad\quad \lambda vs'.\textbf{case}\ vs'\ \textbf{of} \\ \qquad\qquad\qquad\qquad [\,] \qquad : x' :: v \\ \qquad\qquad\qquad\qquad \mid v'' :: vs'' : v'' :: (g\ vs'') \\ \qquad\qquad\qquad\quad \textbf{in}\ g\ vs') \\ \quad \textbf{in}\ f\ xs \end{array} \tag{9}$$

The remaining generalized term is therefore as follows:

$$\begin{array}{l} \textbf{letrec}\ f\ = \\ \lambda xs.\textbf{case}\ xs\ \textbf{of} \\ \qquad [\,] \qquad : v \\ \qquad \mid x' :: xs' : \textbf{case}\ (f\ xs')\ \textbf{of} \\ \qquad\qquad\qquad [\,] \qquad : \quad x' :: v \\ \qquad\qquad\qquad v' :: vs' \quad : \\ \qquad\qquad\qquad v' :: (\textbf{letrec}\ g\ = \\ \qquad\qquad\qquad\quad \lambda vs'.\textbf{case}\ vs'\ \textbf{of} \\ \qquad\qquad\qquad\qquad [\,] \qquad : x' :: v \\ \qquad\qquad\qquad\qquad \mid v'' :: vs'' : v'' :: (g\ vs'') \\ \qquad\qquad\qquad\quad \textbf{in}\ g\ vs') \\ \textbf{in}\ f\ xs \end{array} \tag{10}$$

After further transformation of term (10), the partial process tree in Figure 13 is obtained. The following residual program is constructed from this partial process tree:

$$\begin{array}{l} \textbf{letrec}\ f\ = \\ \lambda xs'.\textbf{case}\ xs'\ \textbf{of} \\ \qquad [\,] \qquad : x' :: v \\ \qquad \mid x'' :: xs'' : \textbf{case}\ (f\ xs'')\ \textbf{of} \\ \qquad\qquad\qquad [\,] \qquad : \quad x'' :: x' :: v \\ \qquad\qquad\qquad v' :: vs' \quad : \\ \qquad\qquad\qquad v' :: (\textbf{letrec}\ g\ = \\ \qquad\qquad\qquad\quad \lambda vs'.\textbf{case}\ vs'\ \textbf{of} \\ \qquad\qquad\qquad\qquad [\,] \qquad : x'' :: x' :: v \\ \qquad\qquad\qquad\qquad \mid v'' :: vs'' : v'' :: (g\ vs'') \\ \qquad\qquad\qquad\quad \textbf{in}\ g\ vs') \\ \textbf{in}\ f\ xs \end{array} \tag{11}$$

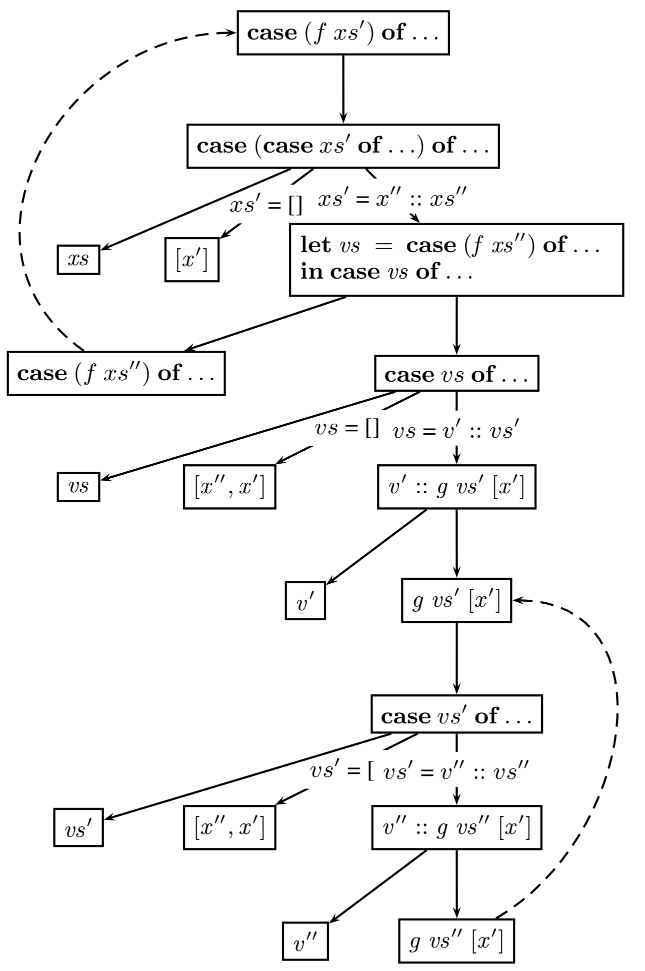

Figure 12. Partial Process Tree During Transformation (2)

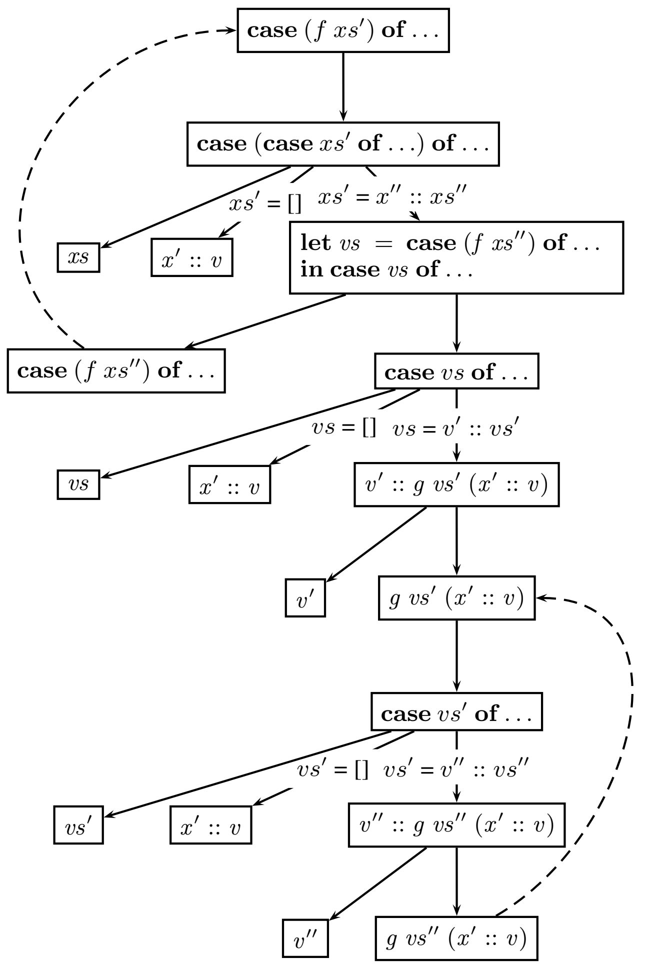

Figure 13. Partial Process Tree During Transformation (3)

We can now see that term (11) is an instance of term (10). A repeat node is therefore created, and the final program is constructed as shown in Figure 14. This program has a run-time which is linear in

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

Figure 14. Example Program Distilled

the length of the input list, while the original program is quadratic. □

EXAMPLE 8 (Accumulating Parameter). Consider the program shown in Figure 15. When this program is transformed without generalization, the successively larger terms *app* (*rev xs* []) *ys*, *app* (*rev xs*′ [*x*′]) *ys*, *app* (*rev xs*″ [*x*″, *x*′]) *ys*, ... are encountered. In supercompilation, the accumulating parameter would be extracted as a result of generalization, and would not be transformed away even though it is an intermediate data structure. In distillation, this intermediate data structure is removed, resulting in the program shown in Figure 16.

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

Figure 15. Accumulating Parameter Example

```
letrec f = λxs.λys.case xs of
                      []        : ys
                    | x :: xs   : f xs (x :: ys)
in f xs ys
```

Figure 16. Accumulating Parameter Example Transformed

EXAMPLE 9 (Accumulating Pattern). Consider the program shown in Figure 17.

```
leq x (add y x)
where
leq = λx.λy.case x of
               0       : True
             | x′ + 1  : case y of
                            0       : False
                          | y′ + 1  : leq x′ y′
add = λx.λy.case x of
               0       : y
             | x′ + 1  : (add x′ y) + 1
```

Figure 17. Accumulating Pattern Example

When this program is transformed without generalization, the successively larger terms *leq x* (*add y x*), *leq x*′ (*add y*′ (x' + 1)), *leq x*″ (*add y*″ (x'' + 2)), ... are encountered. In supercompilation, the accumulating patterm would be extracted as a result of generalization, and would not be transformed away even though it is an intermediate data structure. In distillation, the intermediate data structure is removed, resulting in the program shown in Figure 18.

```
letrec f =
λx.λy.case x of
         0       : True
       | x′ + 1  : case y of
                      0       : letrec g =
                                λx′.case x′ of
                                      0       : True
                                    | x″ + 1  : g x″
                                in g x′
                    | y′ + 1  : f x′ y′
in f x y
```

Figure 18. Accumulating Pattern Example Transformed

EXAMPLE 10. As a final example, consider the program shown in Figure 19 for calculating fibonacci numbers. The function *fib* has two recursive calls, which are both intermediate structures. This program is transformed into the program shown in Figure 20 as a result of distillation. We can see that the resulting function *f* now

```
fib n
where
fib  = λn.case n of
            0        : 1
          | (n' + 1) : case n' of
                          0         : 1
                        | (n'' + 1) : add (fib n') (fib n'')
add = λx.λy.case x of
              0        : y
            | (x' + 1) : (add x' y) + 1
```

Figure 19. Program for Calculating Fibonacci Numbers

```
f n 1 1
where
f = λn.λx.λy.case n of
                0        : x
              | (n' + 1) : f n' y (g x y)
g = λx.λy.case x of
            0        : y
          | (x' + 1) : (g x' y) + 1
```

Figure 20. Distilled Program for Calculating Fibonacci Numbers

contains just one recursive call which is not intermediate, and is therefore more efficient. A similar transformation can be performed using the *tupling* transformation [5]. However, the program resulting from tupling transformation is not tail-recursive (the recursive function call is intermediate), and so it will be less efficient. □

6. The Distillation Theorem

In this section, we state and give an outline proof of the *distillation theorem*, which is the main result of the paper. More details of the proof can be found in a forthcoming longer version of this paper.

THEOREM 6.1 (Distillation Theorem). Every program can be transformed by the distillation algorithm to an equivalent program without loss of efficiency. □

PROOF 1. This follows immediately from Lemmata 6.2, 6.4 and 6.5. □

LEMMA 6.2 (Efficiency). The distillation algorithm produces programs which are no less efficient than the original. □

PROOF 2 (Sketch). In order to prove that our distillation algorithm does not result in a loss of efficiency, we need to have a measure of the *cost* of expressions. This measure should relate to an operational semantics of our language, and should indicate the number of reduction steps required to reduce an expression to normal form. An appropriate measure for a call-by-name semantics can be found in [19, 20]. In this work, the one-step reduction relation within the semantics is denoted by $\mapsto$, and a closed expression e is said to converge to weak head normal form w, written $e \Downarrow w$, if and only if $e \mapsto^* w$, where $\mapsto^*$ denotes the transitive closure of $\mapsto$. A closed expression e is also said to converge in n steps to weak head normal form w, written $e \Downarrow^n w$, if $e \mapsto_n w$, where $\mapsto_n$ denotes a sequence of n reductions. In [19, 20] the notion of *improvement* is defined as follows.

DEFINITION 6.3 (Improvement). *An expression e is improved by e', written $e \gtrsim e'$, if for all contexts C such that $C[e]$ and $C[e']$ are closed, if $C[e] \Downarrow^n$, $C[e'] \Downarrow^m$ and $m \leq n$.* □

Here, a context $C[]$ is a term with a single hole $[]$ in it, and $C[e]$ denotes the term that results from replacing the hole in $C[]$ with e.

In [19], this notion of improvement was used to prove that there is no efficiency loss with respect to a call-by-name semantics resulting from supercompilation. We therefore only need to extend this proof to handle our new rules for transforming nodes which contain recursive terms. To do this, we need to show that each new function call which is introduced by transformation comes together with an unfolding step in the body of that function definition. This is quite straightforward, as after first encountering a node which contains a **letrec** expression, we unfold the function. When a node with a matching **letrec** expression is subsequently encountered, folding is performed, but an unfolding step will have been performed in the body of the constructed function. □

LEMMA 6.4 (Correctness). The distillation algorithm produces programs which are equivalent to the original. □

PROOF 3 (Sketch). In order to prove the correctness of the distillation algorithm, we follow the work of Sands [19, 20] which makes use of an *improvement theorem* which says that if each transformation step is equivalence preserving then a transformation which replaces e by e' is totally correct if e is improved by e'. The equivalence of each transformation step can be proved with respect to the operational semantics of the language, and the improvement of the expression e follows immediately from Lemma 6.2 (on efficiency). □

LEMMA 6.5 (Termination). The distillation algorithm always terminates. □

PROOF 4 (Sketch). In order to prove that the distillation algorithm terminates, we follow the language-independent framework for proving termination of abstract program transformers in [22]. For a transformer to fit the framework, it is sufficient to ensure that:

- in the sequence of trees produced by the transformation, for any depth d, there must be some point from which every two consecutive trees are identical down to depth d
- only finite trees are produced.

By induction on the depth of the trees produced, the former can be proved by the fact that the algorithm does one of the following:

- adds new leaves to a tree which trivially makes consecutive trees identical at an increasing depth
- replaces a subtree by a node containing a **let** expression. Each node can be generalised at most twice in this way; once when the label is a flat term, and once when the label is a recursive term.

The latter is ensured because, in every proces tree:

- a path that consists of **let** expressions only must be finite, since each **let** expression e will have sub-expressions of e as children; thus the size of the nodes in such a path strictly decreases.
- all other nodes are not allowed to homeomorphically embed an ancestor. □

7. Conclusion and Related Work

We have presented the distillation transformation algorithm for higher-order functional languages. The algorithm is largely based on the supercompilation transformation algorithm, but can produce a superlinear speedup in programs, which is not possible using supercompilation.

Supercompilation was originally formulated in the early seventies by Turchin and has been further developed in the eighties [26]. The form of generalization used by Turchin is described in [27].

This involves looking at the call stack to detect recurrent patterns of function calls. Interest in supercompilation was revived in the nineties through the *positive supercompiler* [24], and the homeomorphic embedding relation was later proposed to guide generalization and ensure termination of positive supercompilation [23].

It has previously been noted in a number of works [12, 2, 1, 31, 24] that the unfold/fold transformation methodolgy is incomplete; there are programs which cannot be synthesized from each other. This is due to the linear relationship which must exist between the original and transformed programs. This incompleteness motivated Kott to define *second order replacement* [13] which allows replacement of equivalent expressions in the context of recursive definitions. A similar mechanism has also been proposed in the area of logic program transformation in the SCOUT transformation system [18], which allows folding in the presence of recursion. However, the matching which is performed in each of these cases is not as general as the matching which we perform on recursive terms. This is because our recursive terms are parameterised only by those variables which change, so they abstract away from the number of parameters within recursive definitions and concentrate purely on the recursive structure.

There are a number of possible directions for further work. Firstly, although the distillation algorithm has already been implemented, it is intended to develop a re-implementation in its own input language which will allow the transformer to be self-applicable. Secondly, the distillation algorithm is being incorporated into an automatic inductive theorem prover called Poitín; some preliminary results of this are reported in [8] [4]. Finally, it is intended to incorporate the distillation algorithm into a full 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 using our theorem prover.

References

[1] T. Amtoft. *Sharing of Computations*. PhD thesis, DAIMI, Aarhus University, 1993.

[2] Lars Ole Andersen and Carsten K. Gomard. Speedup Analysis in Partial Evaluation: Preliminary Results. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–7, 1992.

[3] L. Augustsson. Compiling Pattern Matching. In *Functional Programming Languages and Computer Architecture*, pages 368–381, 1985.

[4] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1–2):25–46, 1993.

[5] Wei-Ngan Chin. Towards an Automated Tupling Strategy. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, 1993.

[6] L. Damas and R. Milner. Principal Type Schemes for Functional Programs. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[7] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier, MIT Press, 1990.

[8] G. W. Hamilton. Poitín: Distilling Theorems From Conjectures. *Electronic Notes in Theoretical Computer Science*, 151(1):143–160, 2006.

[9] G. Higman. Ordering by Divisibility in Abstract Algebras. *Proceedings of the London Mathemtical Society*, 2:326–336, 1952.

[10] R. J. M. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, April 1989.

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

[12] Laurent Kott. A System for Proving Equivalences of Recursive Programs. In *5th Conference on Automated Deduction*, pages 63–69, 1980.

[13] Laurent Kott. Unfold/Fold Transformations. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 12, pages 412–433. CUP, 1985.

[14] J.B. Kruskal. Well-Quasi Ordering, the Tree Theorem, and Vazsonyi's Conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[15] M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. In *Proceedings of the International Static Analysis Symposium*, pages 230–245, 1998.

[16] R. Marlet. *Vers une Formalisation de l'Évaluation Partielle*. PhD thesis, Université de Nice - Sophia Antipolis, 1994.

[17] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[18] Abhik Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. An Unfold/Fold Transformation Framework for Definite Logic Programs. *ACM Transactions on Programming Language Systems*, 26(3):464–509, 2004.

[19] D. Sands. Proving the Correctness of Recursion-Based Automatic Program Transformations. *Theoretical Computer Science*, 1–2(167), 1996.

[20] D. Sands. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Transactions on Programming Languages and Systems*, 18(2), March 1996.

[21] H. Seidl and M. H. Sørensen. Constraints to Stop Higher-Order Deforestation. In *Proceedings of the Twelfth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 400–413, 1997.

[22] M. H. Sørensen. Convergence of Program Transformers in the Metric Space of Trees. *Lecture Notes in Computer Science*, 1422:315–337, 1998.

[23] M. H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. *Lecture Notes in Computer Science*, 787:335–351, 1994.

[24] Morten Heine Sørensen. Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.

[25] V.F. Turchin. The Use of Metasystem Transition in Theorem Proving and Program Optimization. *Lecture Notes in Computer Science*, 85:645–657, 1980.

[26] V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):90–121, July 1986.

[27] V.F. Turchin. The Algorithm of Generalization in the Supercompiler. In *Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 531–549, 1988.

[28] P. Wadler. Efficient Compilation of Pattern Matching. In S.L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, pages 78–103. Prentice Hall, 1987.

[29] P. Wadler. The Concatenate Vanishes. FP Electronic Mailing List, December 1987.

[30] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *European Symposium on Programming*, pages 344–358, 1988.

[31] Hong Zhu. How Powerful are Folding/Unfolding Transformations? *Journal of Functional Programming*, 4(1):89–112, 1994.

[4] It has previously been shown in [25] how supercompilation can be used in inductive theorem proving.