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 runtime 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 \( L \), but the recursive function call \( \text{rev } L \) is an intermediate data structure, so in terms of time and space usage, it is quadratic in the length of the list \( L \). A more efficient program which is linear in the length of the list \( L \) 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 \( \text{app} \) function. However, 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.

![Figure 3. Language Grammar](image)

**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 the language is typed using the Hindley-Milner polymorphic typ-...92x504]expression are non-overlapping and exhaustive.

Each constructor has a unique decomposition property. The rules simply perform simultaneous substitution of the sub-expressions of a rightmost redex.

**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 performed 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.

![Figure 4. Grammar of Redexes, Contexts and Observables](image)

The expression con(e) denotes the result of replacing the ‘hole’ ⟨⟩ 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(e’).

**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 N from expressions to ordered sequences of expressions [e1, ..., en].

![Figure 5. Normal Order Reduction Rules for Supercompilation](image)

We use the notation ε(t1 := e1, ..., tn := en) to represent the simultaneous substitution of the sub-expressions e1, ..., en for the free occurrences of variables t1, ..., tn, respectively, within e. The function unfold unfolds the function in the redex of its argument expression as follows:

\[
\text{unfold}(\text{con}(f)) = \text{con}(e) \quad \text{where } f \text{ is defined by } f = e
\]

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(α) denotes the label of α, anc(t, α) denotes the set of ancestors of α in t, and t(α := t’) denotes the tree obtained by replacing the subtree with root α in t by the tree t’. Finally, the tree e → t1, ..., tn is the tree with root labelled e and n children which are the subtrees t1, ..., tn respectively.
Definition 3.5 (Construction of Process Trees). A process tree is constructed from an expression $e$ using the following rule:

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

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 $e' \leq e$, if there is a substitution $\theta$ 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 $\alpha$ within the process tree which is labelled with an instance of $e$, then a dashed edge $e \rightarrow \alpha$ 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.

$$\text{app } (\text{app } x s y s) y s$$

where

$$\text{app } = \lambda x s. \lambda y s. \text{case } x s \text{ of }$$

$$| \quad \quad \quad \text{:: } y s \quad | \quad x : x s : x : (\text{app } x s y s)$$

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 $C$ as shown in Figure 8.

The recursive functions which are introduced within residual programs are defined using local function definitions of the form

$$\text{let } f = e_0 \text{ in } e_1$$

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 unvisored, 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 form

$$\text{let } v_1 = e_1, \ldots, v_n = e_n \text{ in } e_0$$

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:

$$N[e (\text{let } v_1 = e_1, \ldots, v_n = e_n \text{ in } e_0)] = [e_0, e_1, \ldots, e_n]$$

The 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:

$$C[e (\text{let } v_1 = e_1, \ldots, v_n = e_n \text{ in } e_0)] \rightarrow t_0, \ldots, t_n$$

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. 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 encounter the progressively larger terms: $\text{rev } x s, \text{case } (\text{rev } x s) \text{ of } \cdots, \text{case } (\text{rev } x s) \text{ of } \cdots$ of $\cdots$, etc. The call to $\text{rev}$ thus prevents the surrounding context from being reduced, so this context continues to grow. This call to $\text{rev}$ is therefore an obstructing

---

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 $\text{con}(\langle\lambda v. e_0 \rangle e_1)$ and $\text{con}(\text{case } (c \ e_1 \ldots e_n) \text{ of } \ldots)$ have been omitted.

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.
During transformation of this program, we encounter the progressively larger terms: \( \text{app } x y z s \), \( \text{app } x y z s' \), \( \text{app } x y z s'' \), etc. The second parameter in each recursive call to \( \text{app} \) 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 \( \sqsubseteq \) 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 \( \sqsubseteq \) such that for any infinite sequence \( s_1, s_2, \ldots \) of elements from \( S \) there are numbers \( i, j \) with \( i < j \) and \( s_i \not\sqsubseteq s_j \).

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Diving</th>
<th>Coupling</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e \sqsubseteq e_i ) for some ( i )</td>
<td>( e \sqsubseteq \phi(e_1, \ldots, e_n) )</td>
<td>( e_i \sqsubseteq \phi(e_1', \ldots, e_n') )</td>
</tr>
</tbody>
</table>

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 lambda 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. \( f_1 \cdot f_2 \sqsubseteq f_3 (f_1 y) \)
2. \( f_1 \cdot f_2 \sqsubseteq f_3 (f_2 y) \)
3. \( f_1 (f_2 x) \sqsubseteq f_1 (f_2 (f_2 y)) \)
4. \( f_1 (x) \sqsubseteq f_1 (f_2 (f_2 y)) \)

**Definition 4.3** (Strict Embedding). An expression \( e \) is strictly embedded within an expression \( \phi(e_1, \ldots, e_n) \), denoted by \( e \prec \phi(e_1, \ldots, e_n) \), if \( e \sqsubseteq e_i \) for some \( i \).

For example, in the first item above, \( f_1 \cdot f_2 \sqsubseteq f_3 (f_1 y) \).

### 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 \( e \sqsubseteq e' \).
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 $(e_g, \theta, \theta')$ where $\theta$ and $\theta'$ are substitutions such that $e_\theta \equiv e$ and $e_\theta' \equiv e'$.

**Definition 4.5 (Most Specific Generalization).** A most specific generalization of expressions $e$ and $e'$ is a generalization $(e_g, \theta, \theta')$ 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 $(e, \{v := e\}, \{v := e'\})$.

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. $(e, \{v := e_1\}, \{v := e_2\})$.
2. $(f(e_1, \{v := e_2\}), \{v := f_1\})$.
3. $(f(e_1, \{v := e_2\}), \{v := f_2\})$.
4. $(f(e_1, \{v := e_2\}), \{v := f_2\})$.

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 $(\{\}, e', e)$. 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).**

$$\text{extract}(e', e) = \text{let}\ v = e_g\ \text{in}\ \text{con}(v)$$

where $e' \leftarrow e = (e, e')$.

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).**

$$\text{abstract}(e, e') = \text{let}\ v_1 = e_1, \ldots, v_n = e_n\ \text{in}\ e_g$$

where $e \equiv e' = (e_g, \theta, \{v_1 := e_1, \ldots, v_n := e_n\})$.

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 $\beta$ in which the redex is a function, we process this node as follows:

- if $\exists \alpha \in \text{anc}(t, \beta).t(\alpha) \leq t(\beta)$
  - then $t[\beta := \text{fold}(t(\beta))]$
- else if $\exists \alpha \in \text{anc}(t, \beta).t(\alpha) > t(\beta)$
  - then $t[\beta := \text{fold}(t(\beta))]$
- else if $\exists \alpha \in \text{anc}(t, \beta).t(\alpha) \leq t(\beta)$
  - then $t[\alpha := \text{fold}(t(\beta))]$
- else $t[\beta := \text{fold}(t(\beta))]$

All other nodes are processed as shown in Definition 3.5.
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 $\beta$ within a partially constructed tree $t$, where the label of $\beta$ contains a `letrec` expression in the redex position is therefore as follows:

$$\begin{align*}
\text{if } \exists \alpha & \in \text{anc}(t, \beta), t(\alpha) \subseteq t(\beta) \\
\text{then } & (t(\beta := t(\beta) \rightarrow \alpha)) \\
\text{else if } & \exists \alpha \in \text{anc}(t, \beta), t(\alpha) \nsubseteq t(\beta) \\
\text{then } & t(\alpha := T[\text{abstract}(t(\alpha), t(\beta))]) \\
\text{else } & t(\beta := t(\beta) \rightarrow T[\text{unfold}(t(\beta))])
\end{align*}$$

The rules for constructing a residual program from a node in which the 
subterm obtained, the partial process tree shown in Figure 11 is 
transformed. The rule for transforming a node $\beta$ within a partially 
constructed tree $t$, where the label of $\beta$ contains a `letrec` expression in the redex position is therefore as follows:

$$\begin{align*}
C[\alpha = \text{con}(\text{letrec } f = e_0 \text{ in } e_1) \rightarrow t] \\
= \text{letrec } f' = \lambda v_1, \ldots, v_n. C[t] \\
in f', v_1, \ldots, v_n, \text{ if } \exists \beta \in t, \beta \rightarrow \alpha \\
\beta \equiv \alpha(v_1 := e_1, \ldots, v_n := e_n) \\
\beta \equiv \alpha(v_1 := e_1, \ldots, v_n := e_n)
\end{align*}$$

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 results obtained for the other examples 
are given.

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

$$\text{rev } xs$$ (1)

We subsequently encounter the following term:

$$\text{app (rev } xs') \ [x']$$ (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:

$$\text{let } vs = \text{rev } xs' \text{ in app vs } [x']$$ (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:

$$\begin{align*}
\text{letrec } f &= \lambda vs. \text{case } xs \text{ of} \\
&\quad | \ x' :: xs' : \text{case } vs \text{ of} \\
&\quad &\quad v' :: vs' : \ [x'] \\
&\quad &\quad v'' : \text{letrec } g = \\
&\quad &\quad \lambda vs'. \text{case } vs' \text{ of} \\
&\quad &\quad | \ x' :: vs'' : \text{case } vs'' \text{ of} \\
&\quad &\quad &\quad | v'' :: vs'' : v'' :: (g \ vs'') \\
&\quad &\quad in \ f \ xs
\end{align*}$$ (4)

![Figure 11. Partial Process Tree During Transformation (1)](image)

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

$$\begin{align*}
\text{case } (f \ vs') \text{ of} \\
&\quad | \ [x'] : [x'] \\
&\quad | v'' :: vs'' : v'' :: (g \ vs'')
\end{align*}$$ (5)

After further transformation, the following term is obtained:

$$\begin{align*}
\text{case } (\text{case } (f \ vs') \text{ of} \\
&\quad | \ [x'] : [x'] \\
&\quad | v'' :: vs'' : v'' :: (g \ vs'') \text{ of} \\
&\quad | v' :: vs' : v' :: (g \ vs')
\end{align*}$$ (6)

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

$$\begin{align*}
\text{let } vs = \text{case } (f \ vs'') \text{ of} \\
&\quad | \ [x''] : [x'] \\
&\quad | v' :: vs' : v' :: (g \ vs')
\end{align*}$$ (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:

letrec $f = \lambda \text{xs}. \text{case xs of}$
  $| [] : v$
  $| x :: \text{xs'} : \text{case (f xs') of}$
    $| [] : [x', x]$  
    $| v' :: \text{vs'} :$
    $v' :: \text{letrec g =}$
    $\lambda \text{vs'}. \text{case vs' of}$
    $| [] : [v'', x']$
    $| v'' :: \text{gs'} : v' : (g \text{vs''})$
  $\text{in g vs'}$

in $f \text{xs'}$

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:

let $v = []$

in letrec $f = \lambda \text{xs}. \text{case xs of}$
  $| x :: \text{xs'} : \text{case (f xs') of}$
    $| [] : x' :: v$
    $| v' :: \text{vs'} :$
    $v' :: \text{letrec g =}$
    $\lambda \text{vs'}. \text{case vs' of}$
    $| [] : x' :: v$
    $| v'' :: \text{vs''} : v'' :: (g \text{vs''})$
  $\text{in g vs'}$

in $f \text{xs}$

The remaining generalized term is therefore as follows:

letrec $f = \lambda \text{xs}. \text{case xs of}$
  $| x :: \text{xs'} : \text{case (f xs') of}$
    $| [] : x' :: v$
    $| v' :: \text{vs'} :$
    $v' :: \text{letrec g =}$
    $\lambda \text{vs'}. \text{case vs' of}$
    $| [] : x' :: v$
    $| v'' :: \text{vs''} : v'' :: (g \text{vs''})$
  $\text{in g vs'}$

in $f \text{xs}$

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:

letrec $f = \lambda \text{xs'}. \text{case xs of}$
  $| x' :: \text{xs''} : \text{case (f xs') of}$
    $| [] : x' :: v$
    $| v' :: \text{vs'} :$
    $v' :: \text{letrec g =}$
    $\lambda \text{vs'}. \text{case vs' of}$
    $| [] : x' :: v$
    $| v'' :: \text{vs''} : v'' :: (g \text{vs''})$
  $\text{in g vs'}$

in $f \text{xs}$

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 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 $\text{app (rev xs []) ys}$, $\text{app (rev xs' [x']) ys}$, $\text{app (rev xs'' [x'', x']) ys}$, $\text{...}$ 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.

**Example 9** (Accumulating Pattern). Consider the program shown in Figure 17. When this program is transformed without generalization, the suc-
cessively larger terms \(\leq x\) (add \(y\)), \(\leq x'\) (add \(y'\) (\(x' + 1\))), \(\leq x''\) (add \(y''\) (\(x'' + 2\))), . . . are encountered. In supercompilation, the accumulating pattern 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.

Example 10. As a final example, consider the program shown in Figure 19 for calculating fibonacci numbers. The function \(\text{fib}\) 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 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.
fib n
where
fib = \lambda n. case n of
  0 : 1
  | (n' + 1) : case n' of
          0 : 1
          | (n'' + 1) : add (fib n') (fib n'')
add = \lambda x.\lambda y. case x of
  0 : y
  | (x' + 1) : (add x' y) + 1

Figure 19. Program for Calculating Fibonacci Numbers

f n 1 1
where
f = \lambda n.\lambda x.\lambda y. case n of
  0 : x
  | (n' + 1) : f n' y (g x y)
g = \lambda x.\lambda y. case x of
  0 : y
  | (x' + 1) : (g x' y) + 1

Figure 20. Distilled Program for Calculating Fibonacci Numbers

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 \rightarrow, and a closed expression e is said to converge to weak head normal form \mathcal{w}, written e \Downarrow \mathcal{w}, if and only if e \rightarrow \mathcal{w}, where \rightarrow^* denotes the transitive closure of \rightarrow. A closed expression e is also said to converge in n steps to weak head normal form \mathcal{w}, written e \Downarrow^n \mathcal{w}, if e \rightarrow \mathcal{w}, where \rightarrow^* 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 \geq e', if for all contexts C such that C[e] and C[e'] are closed, if C[e] \Downarrow^m, 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 process 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 methodology 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 Poin; some preliminary results of this are reported in [8]. 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


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