Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 190 (2007) 17–32
www.elsevier.com/locate/entcs

Distilling Programs for Verification

G.W. Hamilton [1,2]

School of Computing
Dublin City University
Dublin, IRELAND

Abstract

In this paper, we show how our program transformation algorithm called *distillation* can not only be used for the optimisation of programs, but can also be used to facilitate program verification. Using the distillation algorithm, programs are transformed into a specialised form in which functions are tail recursive, and very few intermediate structures are created. We then show how properties of this specialised form of program can be easily verified by the application of inductive proof rules. We therefore argue that the distillation algorithm is an ideal candidate for inclusion within compilers as it facilitates the two goals of program optimization and verification.

Keywords: transformation, optimization, proof, verification

1 Introduction

In 2004, the UK Computing Research Committee initiated a number of 'Grand Challenges' aimed at stimulating long term research in key areas of computing science. The sixth challenge which was identified was that of dependable systems evolution, which was inspired by the idea of a *verifying compiler* [8], which is a compiler that guarantees the correctness of a program before running it.

In this paper we present a program transformation algorithm called *distillation* which we argue provides a major step towards the dream of a verifying compiler. The distillation algorithm [5] was originally devised with the goal of eliminating intermediate data structures from functional programs. A number of program transformation techniques have been proposed which can eliminate some of these intermediate data structures; for example *partial evaluation* [9], *deforestation* [25] and *supercompilation* [22]. 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

[1] Email: hamilton@computing.dcu.ie
[2] Fax: +353 1 7005442

doi:10.1016/j.entcs.2007.09.005

speedup in programs [19]. Distillation, however, can produce a superlinear speedup in programs.

Example 1.1 Consider the program shown in Figure 1.

$$
\begin{array}{l}
rev\ xs \\
\textbf{where} \\
rev\ = \lambda xs.\, \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad [] \quad\ \Rightarrow [] \\
\qquad\qquad\quad \mid x : xs \Rightarrow app\ (rev\ xs)\ [x] \\
app = \lambda xs.\lambda ys.\, \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad [] \quad\ \Rightarrow ys \\
\qquad\qquad\qquad\ \mid x : xs \Rightarrow x : (app\ xs\ ys)
\end{array}
$$

Fig. 1. Example Program

This program reverses the list xs, but in terms of time and space usage, it is quadratic in the length of the list xs. Applying the distillation algorithm to this program, we obtain the program shown in Figure 2, which is linear in the length of the list xs.

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

Fig. 2. Example Program Transformed

The programs resulting from distillation are in a specialised form in which functions are tail recursive and very few intermediate structures are created. We show that this specialised form is very amenable to the automatic verification of properties of programs through the application of inductive proof rules. We therefore argue that the distillation algorithm is an ideal candidate for inclusion within a compiler as it enables both powerful optimization and program verification.

The remainder of this paper is structured as follows. In Section 2, we define the higher-order language on which the described transformation and verification are performed. In Section 3, we give an overview of the distillation algorithm. In

Section 4, we show how the programs resulting from distillation can be verified. Section 5 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
	|	$c\ e_1 \ldots e_n$	Constructor Application
	|	f	User-Defined Function
	|	$\lambda v.e$	λ-Abstraction
	|	$e_0\ e_1$	Application
	|	**case** e_0 **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$	Case Expression
p	::=	$c\ v_1 \ldots v_n$	Pattern

Fig. 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 (so erroneous terms such as $(c\ e_1 \ldots e_n)\ e$ 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 $fv(e)$ 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** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$, 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. Methods to transform **case** expressions with nested patterns to ones without nested patterns are described in [1,24]. No variables may appear more than once within a pattern. We assume that the patterns in a **case** expression are

non-overlapping and exhaustive.

3 Distillation

In this section, we give an overview of the distillation algorithm; full details of the algorithm can be found in [5]. The distillation algorithm is a significant advance over the supercompilation algorithm. Using the supercompilation algorithm, it is only possible to obtain a linear improvement in the run-time performance of programs; with distillation it is possible to produce a superlinear improvement.

We define the rules for distillation 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.

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

Fig. 4. Grammar of Redexes, Contexts and Observables

The expression $\mathit{con}\langle e\rangle$ 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. $e = \mathit{con}\langle e'\rangle$.

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

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

Fig. 5. Normal Order Reduction Rules for Disitllation

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

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 [20]).

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.

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 3.5 [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 3.6 [Instance] An expression e is an *instance* of expression e', denoted by $e' \preceq e$, 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 labelled with e' where e is 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 3.7 Consider the program shown in Figure 6.
Transformation of this program produces the partial process tree given in Figure

$$app\ (app\ xs\ ys)\ zs$$

where

$$app = \lambda xs.\lambda ys.\,\textbf{case}\ xs\ \textbf{of}$$
$$[] \quad \Rightarrow ys$$
$$\mid x : xs \Rightarrow x : (app\ xs\ ys)$$

Fig. 6. Example Program

7 [3].

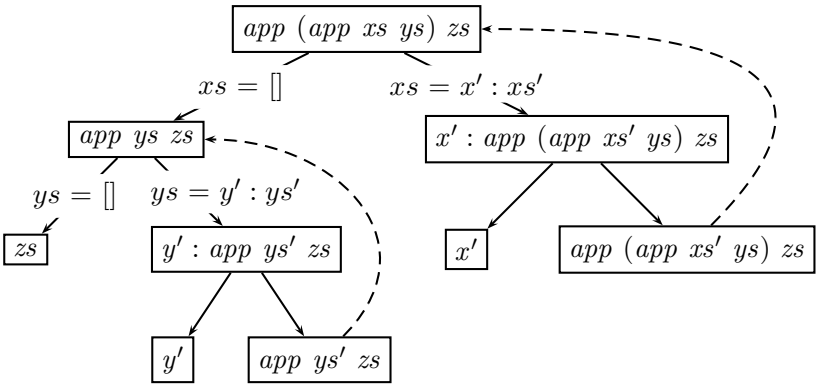

Fig. 7. Example Partial Process Tree

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 residual program constructed from the partial process tree in Figure 7 is shown in Figure 9

If the transformation rules presented so far were left unsupervised, non-termination could arise, even in the presence of folding. This non-termination will always involve encountering expressions which are *embeddings* of previously encountered expressions. 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.

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 [7] and Kruskal [11] and was defined within term rewriting systems [4] for detecting the possible divergence of the term rewriting process. Variants of this relation have been used to ensure termination within supercompilation [20],

[3] This process tree, and later ones presented in this paper, have been simplified for ease of presentation.

$$
\begin{array}{lcl}
\mathcal{C}[\![(v\ e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n]\!] & = & v\ (\mathcal{C}[\![t_1]\!]) \ldots (\mathcal{C}[\![t_n]\!]) \\
\mathcal{C}[\![(c\ e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n]\!] & = & c\ (\mathcal{C}[\![t_1]\!]) \ldots (\mathcal{C}[\![t_n]\!]) \\
\mathcal{C}[\![(\lambda v.e)\ \rightarrow\ t]\!] & = & \lambda v.(\mathcal{C}[\![t]\!]) \\
\mathcal{C}[\![\alpha = (con\langle f \rangle)\ \rightarrow\ t]\!] & = & \textbf{letrec}\ f'\ =\ \lambda v_1 \ldots v_n.\mathcal{C}[\![t]\!] \\
 & & \textbf{in}\ f'\ v_1 \ldots v_n,\ \text{if}\ \exists \beta \in t.\beta \dashrightarrow \alpha \\
 & & \text{where}\ \beta \equiv \alpha\{v_1 := e_1, \ldots, v_n := e_n\} \\
 & = & \mathcal{C}[\![t]\!],\ \text{otherwise} \\
\mathcal{C}[\![\beta = (con\langle f \rangle)\ \dashrightarrow\ \alpha]\!] & = & f'\ e_1 \ldots e_n \\
 & & \text{where}\ \beta \equiv \alpha\{v_1 := e_1, \ldots, v_n := e_n\} \\
\mathcal{C}[\![(con\langle(\lambda v.e_0)\ e_1\rangle)\ \rightarrow\ t]\!] & = & \mathcal{C}[\![t]\!]
\end{array}
$$

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

$$\mathcal{C}[\![(con\langle \textbf{case}\ (v\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e_1\ |\cdots|\ p_n \Rightarrow e_n\rangle) \rightarrow t_0, \ldots, t_n]\!]$$

$$= \textbf{case}\ (\mathcal{C}[\![t_0]\!])\ \textbf{of}\ p_1 \Rightarrow \mathcal{C}[\![t_1]\!]\ |\cdots|\ p_n \Rightarrow \mathcal{C}[\![t_n]\!]$$

Fig. 8. Rules For Constructing Residual Programs

```
letrec
  f0 = λxs.λys.λzs. case xs of
                      []      ⇒ letrec
                                  f1 = λys.λzs. case ys of
                                                  []      ⇒ zs
                                                | y' : ys' ⇒ y' : (f1 ys' zs)
                                  in f1 ys zs
                    | x' : xs' ⇒ x' : (f0 xs' ys zs)
in f0 xs ys zs
```

Fig. 9. Constructed Residual Program

partial evaluation [14] and partial deduction [2,12]. It can be shown that the homeomorphic embedding relation $\trianglelefteq$ is a *well-quasi-order*, which is defined as follows.

Definition 3.9 [Well-Quasi Order] A well-quasi order on a set S is a reflexive, transitive relation $\leq_S$ such that for any infinite sequence $s_1, s_2, \ldots$ of elements from S there are numbers i, j with $i < j$ and $s_i \leq_S s_j$. □

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 3.10 [Homeomorphic Embedding Relation]

$$\begin{array}{lll} \text{Variable} & \text{Diving} & \text{Coupling} \\ & \dfrac{e \trianglelefteq e_i \text{ for some } i}{e \trianglelefteq \phi(e_1, \ldots, e_n)} & \dfrac{e_i \trianglelefteq e'_i \text{ for all } i}{\phi(e_1, \ldots, e_n) \trianglelefteq \phi(e'_1, \ldots, e'_n)} \\ x \trianglelefteq y & & \end{array}$$

This embedding relation is extended slightly to be able to handle constructs such as λ-abstraction and **case** which may contain bound variables. In these instances, the corresponding binders within the two expressions must also match up. □

Example 3.11 Some examples of the homeomorphic embedding relation are as follows.

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

In distillation, generalization is performed when an expression is encountered which is an embedding of a previously encountered expression. To represent the result of generalization, we introduce a **let** construct of the form **let** $v_1 = e_1, \ldots, v_n = e_n$ **in** e_0 into our language. This represents the extraction of the expressions $e_1, \ldots, e_n$, which will be transformed separately. The normal-order reduction for the **let** construct is as follows:

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

The rule for constructing a residual program from a sub-tree which contains a **let** construct in the redex of the root node is as follows:

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

If an expression e is encountered which is an embedding of a previously encountered expression e', generalization is also performed. This generalization of e and e' is the *most specific generalization*, denoted by $e \sqcap e'$, as defined in term algebra [4]. 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 3.12 [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 $e_g\theta' \equiv e'$. □

Definition 3.13 [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 $(v, \{v := e\}, \{v := e'\})$.

$$(e, \{v := \phi(e_1, \ldots, e_n)\} \cup \theta, \{v := \phi(e'_1, \ldots, e'_n)\} \cup \theta')$$

$$\Downarrow$$

$$(e\{v := \phi(v_1, \ldots, v_n)\}, \{v_1 := e_1, \ldots, v_n := e_n\} \cup \theta, \{v_1 := e'_1, \ldots, v_n := e'_n\} \cup \theta')$$

$$(e, \{v_1 := e', v_2 := e'\} \cup \theta, \{v_1 := e'', v_2 := e''\} \cup \theta')$$

$$\Downarrow$$

$$(e\{v_1 := v_2\}, \{v_2 := e'\} \cup \theta, \{v_2 := e''\} \cup \theta') \qquad \square$$

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 3.11 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 calculate the most specific generalization of e and e'. If the redex of this most specific generalization is a variable, then the partial process subtree rooted at e is replaced by the result of transforming the generalized form of e. Otherwise, the partial process subtree rooted at e' is replaced by the result of transforming the generalized form of e'. The generalized forms of these expressions are constructed using the *abstract* operation.

Definition 3.14 [Abstract Operation]
$abstract(e, e') =$ **let** $v_1 = e_1, \ldots, v_n = e_n$ **in** e_g
where $e \sqcap e' = (e_g, \{v_1 := e_1, \ldots, v_n := e_n\}, \theta)$ $\square$

Many of the sub-terms which are extracted by generalization may actually be intermediate, but will not be removed if they are permanently extracted. We therefore futher transform these generalized terms to remove these possibly intermediate structures. Thus, if a node within the partial process tree is labelled with a term which has been generalized, we replace this node with a new one which has the program constructed from this node as its label. This new node is then further transformed. Generalizations which are performed on nodes labelled with constructed

programs are permanent and are not further transformed.

We now give a more formal definition of distillation. The rule for transforming a node β within a partially constructed tree t, where the label of β is an expression with a function in the redex position is as follows:

if $\exists \alpha \in anc(t,\beta).t(\alpha) \trianglelefteq t(\beta)$
then $(t\{\beta := t(\beta) \dashrightarrow \alpha\})\{\alpha := \mathcal{T}[\![\mathcal{C}[\![\alpha]\!]]\!]\}$
else if $\exists \alpha \in anc(t,\beta).t(\alpha) \trianglelefteq t(\beta)$
  then if $t(\alpha) \sqcap t(\beta) = con\langle v \rangle$
    then $t\{\beta := \mathcal{T}[\![\mathcal{C}[\![\mathcal{T}[\![abstract(t(\beta), t(\alpha))]\!]]\!]]\!]\}$
    else $t\{\alpha := \mathcal{T}[\![\mathcal{C}[\![\mathcal{T}[\![abstract(t(\alpha), t(\beta))]\!]]\!]]\!]\}$
  else $t\{\beta := t(\beta) \ \rightarrow \ \mathcal{T}[\![unfold\ (t(\beta))]\!]\}$

The rule for transforming a node β within a partially constructed tree t, where the label of β is a constructed program is as follows:

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

Definition 3.15 [Distilled Form] The expressions resulting from distillation are in *distilled form dt* which is defined as follows:

$$
\begin{array}{rcl}
dt & ::= & v\ dt_1 \ldots dt_n \\
& | & c\ dt_1 \ldots dt_n \\
& | & \lambda v.dt \\
& | & \textbf{letrec}\ f\ =\ \lambda v_1 \ldots v_n.dt\ \textbf{in}\ f\ v'_1 \ldots v'_n \\
& | & f\ dt_1 \ldots dt_n \\
& | & \textbf{case}\ (v\ dt_1 \ldots dt_n)\ \textbf{of}\ p_1 \Rightarrow dt'_1\ |\cdots|\ p_k \Rightarrow dt'_k \\
& | & \textbf{let}\ v\ =\ dt_0\ \textbf{in}\ dt_1
\end{array}
$$

□

Proofs of the correctness and termination of the distillation can be found in [5].

4 Verifying Distilled Programs

In this section, we show how programs can be verified using the distillation algorithm. In order to prove a property p of a program e_0 **where** $f_1 = e_1 \ldots f_n = e_n$, we apply the distillation algorithm to the program $p\ e_0$ **where** $f_1 = e_1 \ldots f_n = e_n$. The result of this transformation will be a boolean expression which is in distilled form. Inductive proof rules are then applied to this expression to verify it. The functions within the boolean expression are all potential inductive hypotheses. If one of

the parameters in a recursive call of one of these functions is *decreasing*, then this inductive hypothesis can be applied, and the value *True* returned. If, however, all of the parameters in a recursive call of one of these functions are *non-decreasing*, then the function is potentially non-terminating, so the undefined value $\perp$ is returned.

Definition 4.1 [Decreasing Parameter] A parameter is decreasing from value e to value e', denoted by $e' \sqsubset e$, if e' is a sub-component of e.

Definition 4.2 [Non-Decreasing Parameter] A parameter is non-decreasing from value e to value e', denoted by $e \sqsubseteq e'$, if $e \sqsubset e'$ or $e = e'$.

The inductive proof rules are shown in Figure 10. Within these rules, ϕ contains the set of previously encountered function calls which are the potential inductive hypotheses.

$$
\begin{array}{lll}
(1) & \mathcal{P}[\![v]\!]\ \phi & = \ \mathit{False} \\
(2) & \mathcal{P}[\![c]\!]\ \phi & = \ c \\
(3) & \mathcal{P}[\![\textbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \textbf{in}\ e_0]\!]\ \phi & \\
 & & = \ (\mathcal{P}[\![e_0]\!]\ \phi) \wedge \ldots \wedge (\mathcal{P}[\![e_n]\!]\ \phi) \\
(4) & \mathcal{P}[\![\textbf{case}\ (v\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_n \Rightarrow e_n']\!]\ \phi & \\
 & & = \ (\mathcal{P}[\![e_1']\!]\ \phi) \wedge \ldots \wedge (\mathcal{P}[\![e_n']\!]\ \phi) \\
(5) & \mathcal{P}[\![\textbf{letrec}\ f = \lambda v_1 \ldots v_n.e_0\ \textbf{in}\ f\ v_1' \ldots v_n']\!]\ \phi & \\
 & & = \ \mathcal{P}[\![e_0[v_1'/v_1, \ldots, v_n'/v_n]]\!]\ (\phi \cup \{f\ v_1' \ldots v_n'\}) \\
(6) & \mathcal{P}[\![f\ e_1 \ldots e_n]\!]\ \phi & = \ \begin{cases} \mathit{True}, & \text{if } \exists (f\ e_1' \ldots e_n') \in \phi . \exists i \in \{1 \ldots n\} . e_i \sqsubset e_i' \\ \perp, & \text{if } \exists (f\ e_1' \ldots e_n') \in \phi . \forall i \in \{1 \ldots n\} . e_i' \sqsubseteq e_i \\ e, & \text{otherwise} \end{cases}
\end{array}
$$

where

$$
\begin{array}{lll}
f & = & \lambda v_1 \ldots v_n.e_0 \\
e & = & \mathcal{P}[\![e_0[e_1/v_1, \ldots, e_n/v_n]]\!]\ (\phi \cup \{f\ e_1 \ldots e_n\})
\end{array}
$$

Fig. 10. Inductive Proof Rules

These rules can be explained as follows. In rule (1), if we encounter a variable, the value *False* is returned, as this is one possible value of this variable (it must be a boolean). In rule (2), if we encounter a constructor, then we return the value of this constructor (again, this must be a boolean). In rule (3), if we encounter a **let** expression, then we need to apply the proof rules to all the sub-terms within this expression to show that they are all true. In rule (4), if we encounter a **case** expression, we need to apply the proof rules to all the branches of the **case** to show that they are all true. In rule (5), if we encounter a **letrec** expression, we add the

$$
\begin{array}{l}
sort\ xs \\
\textbf{where} \\
sort \quad = \lambda xs.\, \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad [] \quad\ \Rightarrow [] \\
\qquad\qquad\quad \mid x : xs \Rightarrow insert\ x\ (sort\ xs) \\
insert = \lambda y.\lambda xs.\, \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\quad [] \quad\ \Rightarrow [y] \\
\qquad\qquad\qquad \mid x' : xs' \Rightarrow \textbf{case}\ (less\ x'\ y)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad True \Rightarrow x' : insert\ y\ xs' \\
\qquad\qquad\qquad\qquad\qquad\quad \mid False \Rightarrow y : xs \\
less \quad = \lambda x.\lambda y.\, \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\quad Zero \quad\ \Rightarrow False \\
\qquad\qquad\qquad \mid Succ\ y' \Rightarrow \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Zero \quad\ \Rightarrow True \\
\qquad\qquad\qquad\qquad\qquad\quad \mid Succ\ x' \Rightarrow less\ x'\ y'
\end{array}
$$

Fig. 11. Program for Sorting Lists

function call to the set ϕ and further apply the proof rules to the unfolded function call. In rule (6), if we encounter a function call, then we look within ϕ for previous calls of this function. If one of the parameters in the current call is decreasing, then we apply the inductive hypothesis and return the value *True*. If all of the parameters in the current call are non-decreasing, then the function is potentially non-terminating so we return the value $\bot$. Otherwise, we add the function call to the set ϕ and further apply the proof rules to the unfolded function call. Note that there are no rules for expressions of the form $v\ e_1 \ldots e_n$, $c\ e_1 \ldots e_n$ or $\lambda v.e$ as the proof rules are only applied to expressions of boolean type.

It is possible that overgeneralization can occur, thus turning a theorem into a non-theorem (but not the converse). This means that our theorem prover may determine that a correct program is not actually correct. However, if our theorem prover determines that a program is correct, then this is definitely the case.

Example 4.3 Consider the program shown in Figure 11 for sorting lists of natural numbers.

If we want to verify this program, we need to show that the list resulting from the program is *sorted*, so we define a property to this effect as shown in Figure 12.

This property is applied to the program for sorting lists to obtain the boolean expression *sorted* (*sort xs*). This expression is transformed by the distillation al-

$$
\begin{aligned}
sorted \;&= \lambda xs.\, \textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad Nil \;\Rightarrow True \\
&\quad \mid Cons\ x\ xs \Rightarrow sorted'\ x\ xs \\
sorted' \;&= \lambda x.\lambda xs.\, \textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad Nil \;\Rightarrow True \\
&\quad \mid Cons\ y\ ys \Rightarrow \textbf{case}\ (less\ x\ y)\ \textbf{of} \\
&\qquad\qquad\qquad\qquad True \Rightarrow sorted'\ y\ ys \\
&\qquad\qquad\qquad\quad \mid False \Rightarrow False
\end{aligned}
$$

Fig. 12. Required Property for List Sorting Program

$$
\begin{aligned}
&\textbf{case}\ xs\ \textbf{of} \\
&\quad [] \;\Rightarrow True \\
&\quad \mid x : xs \Rightarrow \textbf{letrec}\ f0 \;=\; \lambda xs.\, \textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad [] \;\Rightarrow True \\
&\qquad\qquad\qquad\qquad \mid x' : xs' \Rightarrow f0\ xs' \\
&\qquad \textbf{in}\ f0\ xs
\end{aligned}
$$

Fig. 13. Resulting Program

gorithm into the program shown in Figure 13.

The verification of this program now proceeds as shown in Figure 14

5 Conclusion and Related Work

In this paper, we have presented a novel transformation algorithm called distillation, which can produce a superlinear speedup in programs. This represents a major advance over existing unfold/fold transformation techniques, which can only produce a linear improvement. We have shown that, not only is distillation useful for performing program optimization, it also facilitates the relatively straightforward verification of the resulting programs. We therefore argue that the distillation algorithm is an ideal candidate for inclusion within a compiler as it enables both powerful optimization and program verification.

The distillation algorithm was largely inspired by supercompilation, which was originally formulated in the early seventies by Turchin and has been further developed in the eighties [22]. The form of generalization used by Turchin is described in [23]. 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* [19], and the homeomorphic embedding relation was later proposed

$$
\begin{array}{lll}
 & \mathcal{P}[\![\, \textbf{case}\ xs\ \textbf{of} & \\
 & \qquad [] \quad\ \Rightarrow True & \\
 & \quad \mid x : xs \Rightarrow \textbf{letrec}\ f0\ =\ \lambda xs.\, \textbf{case}\ xs\ \textbf{of} & \\
 & \qquad\qquad\qquad\qquad [] \quad\ \Rightarrow True & \\
 & \qquad\qquad\qquad\qquad \mid x' : xs' \Rightarrow f0\ xs' & \\
 & \qquad\quad \textbf{in}\ f0\ xs]\!]\ \{\} & \\
= & (\mathcal{P}[\![True]\!]\ \{\}) \wedge (\mathcal{P}[\![\, \textbf{letrec}\ f0\ =\ \lambda xs.\, \textbf{case}\ xs\ \textbf{of} & \text{(by (4))} \\
 & \qquad\qquad\qquad\qquad [] \quad\ \Rightarrow True & \\
 & \qquad\qquad\qquad\qquad \mid x' : xs' \Rightarrow f0\ xs' & \\
 & \qquad\quad \textbf{in}\ f0\ xs]\!]\ \{\}) & \\
= & True \wedge (\mathcal{P}[\![\, \textbf{letrec}\ f0\ =\ \lambda xs.\, \textbf{case}\ xs\ \textbf{of} & \text{(by (2))} \\
 & \qquad\qquad\qquad\qquad [] \quad\ \Rightarrow True & \\
 & \qquad\qquad\qquad\qquad \mid x' : xs' \Rightarrow f0\ xs' & \\
 & \qquad\quad \textbf{in}\ f0\ xs]\!]\ \{\}) & \\
= & True \wedge (\mathcal{P}[\![\, \textbf{case}\ xs\ \textbf{of} & \text{(by (5))} \\
 & \qquad [] \quad\ \Rightarrow True & \\
 & \quad \mid x' : xs' \Rightarrow f0\ xs']\!]\ \{f0\ xs\}) & \\
= & True \wedge (\mathcal{P}[\![True]\!]\ \{f0\ xs\}) \wedge (\mathcal{P}[\![f0\ xs']\!]\ \{f0\ xs\}) & \text{(by (4))} \\
= & True \wedge True \wedge (\mathcal{P}[\![f0\ xs']\!]\ \{f0\ xs\}) & \text{(by (2))} \\
= & True \wedge True \wedge True & \text{(by (6))} \\
= & True &
\end{array}
$$

Fig. 14. Program Verification

to guide generalization and ensure termination of positive supercompilation [20]. Supercompilation has been used for the verification of infinite state systems [13], with some limited success.

The distillation algorithm would be of equivalent power to the supercompilation algorithm if the terms which are extracted on performing generalization were not substituted back in to the generalized term. This means that over-generalization would occur quite frequently when using supercompilation, thus greatly limiting its power. Also, in order to show that the program resulting from supercompilation terminates, Turchin requires that all functions are total, so the onus is on the user to show that this really is the case. In order to show that the program resulting from distillation terminates, we simply need to show it is infinitely progressing [3],

which is much easier to check automatically.

A number of papers illustrate the relationship between the unfold/fold transformation technique and the proofs of program properties, both for functional and logic programs [10,15,16,18,17]. However, the folding mechanism which is used in this work is not as powerful as the folding mechanism presented here, so less program properties can be verified using these techniques.

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 has been incorporated into an automatic inductive theorem prover called Poitín; some preliminary results of this are reported in [6] [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] Augustsson, L., *Compiling Pattern Matching*, in: *Functional Programming Languages and Computer Architecture*, 1985, pp. 368–381.

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

[3] Brotherston, J., *Cyclic Proofs for First-Order Logic With Inductive Definitions*, Lecture Notes in Computer Science **3702** (2005), pp. 78–92.

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

[5] Hamilton, G., *Distillation: Extracting the Essence of Programs*, in: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2007, pp. 61–70.

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

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

[8] Hoare, C. A. R., *The Verifying Compiler: A Grand Challenge for Computing Research*, Journal of the ACM **50** (2003), pp. 63–69.

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

[10] Kott, L., *Unfold/Fold Transformations*, in: M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, CUP, 1985 pp. 412–433.

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

[12] Leuschel, M., *On the Power of Homeomorphic Embedding for Online Termination*, in: *Proceedings of the International Static Analysis Symposium*, 1998, pp. 230–245.

[13] Lisitsa, A. and A. P. Nemytykh, *Towards Verification via Supercompilation*, in: *Proceedings of the 29th Annual International Computer Software and Applications Conference*, 2005, pp. 9–10.

[14] Marlet, R., "Vers une Formalisation de l'Évaluation Partielle," Ph.D. thesis, Université de Nice - Sophia Antipolis (1994).

[4] It has previously been shown in [21] how supercompilation can be used in inductive theorem proving.

[15] Pettorossi, A. and M. Proietti, *Synthesis and Transformation of Logic Programs Using Unfold/Fold Proofs*, Journal of Logic Programming **41** (1999), pp. 197–230.

[16] Pettorossi, A. and M. Proietti, *Perfect Model Checking via Unfold/Fold Transformations*, in: *Proceedings of the First International Conference on Computational Logic*, 2000, pp. 613–628.

[17] Pettorossi, A., M. Proietti and V. Senni, *Proofs of Program Properties via Unfold/Fold Transformations of Constraint Logic Programs*, in: *Transformation Techniques in Software Engineering*, 2005.

[18] Roychoudhury, A., K. N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan and S. A. Smolka, *Verification of Parameterized Systems Using Logic Program Transformations*, in: *Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2000, pp. 172–187.

[19] Sørensen, M. H., "Turchin's Supercompiler Revisited," Master's thesis, Department of Computer Science, University of Copenhagen (1994), dIKU-rapport 94/17.

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

[21] Turchin, V., *The Use of Metasystem Transition in Theorem Proving and Program Optimization*, Lecture Notes in Computer Science **85** (1980), pp. 645–657.

[22] Turchin, V., *The Concept of a Supercompiler*, ACM Transactions on Programming Languages and Systems **8** (1986), pp. 90–121.

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

[24] Wadler, P., *Efficient Compilation of Pattern Matching*, in: S. P. Jones, editor, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987 pp. 78–103.

[25] Wadler, P., *Deforestation: Transforming Programs to Eliminate Trees*, in: *European Symposium on Programming*, 1988, pp. 344–358.