Int J Parallel Prog (2018) 46:152–172
https://doi.org/10.1007/s10766-017-0510-5

Functional Program Transformation for Parallelisation Using Skeletons

Venkatesh Kannan[1] · **G. W. Hamilton**[1]

Received: 10 September 2016 / Accepted: 11 May 2017 / Published online: 23 May 2017

Abstract It can be challenging to use algorithmic skeletons in parallel program development as it is tedious to manually identify parallel computations in an algorithm and there may be mismatches between the algorithm and skeletons. Also, parallel programs defined using skeletons often employ inefficient intermediate data structures. In this paper, we present a program transformation method to address these issues by using an existing technique called *distillation* to reduce the use of intermediate data structures and an encoding technique to combine the inputs of a program into a single input whose structure matches that of the program. This facilitates automatic identification of skeletons that suit the algorithmic structure of the transformed program.

Keywords Program transformation · Parallelisation · Algorithmic skeletons

1 Introduction

While parallel programming is vital to effectively utilise the computing power, it is challenging as it requires manual analysis of an algorithm to identify potential parallel computations. Parallel program development often uses algorithmic skeletons to hide the complexity of implementing the parallelism from the developer. In this work, we

This work was supported, in part, by the Science Foundation Ireland Grant 10/CE/I1855 to Lero - the Irish Software Research Centre (www.lero.ie).

✉ Venkatesh Kannan
vkannan@computing.dcu.ie

G. W. Hamilton
hamilton@computing.dcu.ie

[1] School of Computing, Dublin City University, Dublin, Ireland

are particularly interested in data parallel skeletons, such as *map*, *map-reduce* and *accumulate* [1]. While skeleton-based programming eases the task of the developer, it still requires manual analysis of a program to identify computations that can be defined using the skeletons.

In the case of data parallel skeletons, a given operation is applied in parallel on each element in a given input. Consequently, there is a match between the structure of the input and the recursive structure of the skeleton definition. However, the structure of the data types and the algorithmic structure of a given program may not match with those of the skeletons, which is challenging to resolve [2,3]. Further, programs that are defined using skeletons often introduce inefficient intermediate data structures [4]. Consider the programs shown in Examples 1 and 2 which compute the product of two matrices and the dot-product of two binary trees, respectively. In Example 1, (*transpose yss*) and (*zipWith* (∗) *xs ys*) are intermediate data structures that are subsequently decomposed by *map* and *foldr*, respectively.

Example 1 (Matrix multiplication—original program)

$$
\begin{array}{l}
mMul :: [[a]] \rightarrow [[a]] \rightarrow [[a]] \\
mMul\ xss\ yss \\
\textbf{where} \\
\begin{array}{lcl}
mMul\ [\,]\ yss & = & [\,] \\
mMul\ (xs : xss)\ yss & = & (map\ (transpose\ yss)\ (dotp\ xs)) : (mMul\ xss\ yss) \\
dotp\ xs\ ys & = & foldr\ (+)\ 0\ (zipWith\ (*)\ xs\ ys) \\
transpose\ xss & = & transpose'\ xss\ [\,] \\
transpose'\ [\,]\ yss & = & yss \\
transpose'\ (xs : xss)\ yss & = & transpose'\ xss\ (rotate\ xs\ yss) \\
rotate\ [\,]\ yss & = & yss \\
rotate\ (x : xs)\ [\,] & = & [x] : (rotate\ xs\ yss) \\
rotate\ (x : xs)\ (ys : yss) & = & (ys +\!\!+ [x]) : (rotate\ xs\ yss)
\end{array}
\end{array}
$$

Example 2 (Dot-product of binary trees—original program)

$$
\begin{array}{l}
\textbf{data}\ BTree\ a ::= E \mid B\ a\ (BTree\ a)\ (BTree\ a) \\
dotP :: (BTree\ a) \rightarrow (BTree\ a) \rightarrow (BTree\ a) \\
dotP\ xt\ yt \\
\textbf{where} \\
\begin{array}{lcl}
dotP\ E\ yt & = & 0 \\
dotP\ (B\ x\ xt_1\ xt_2)\ E & = & 0 \\
dotP\ (B\ x\ xt_1\ xt_2)\ (B\ y\ yt_1\ yt_2) & = & (x * y) + (dotP\ xt_1\ yt_1) + (dotP\ xt_2\ yt_2)
\end{array}
\end{array}
$$

Even though these programs can be parallelised using the skeletons *map*, *reduce* and *zipWith* (defined over lists and binary trees with parallel implementations), Example 3 retails the original intermediate data structures from Example 1, while Example 4 introduces a new intermediate data structure (*zipWith* (∗) *xt yt*) due to the used of skeletons.

Example 3 (Matrix multiplication—hand-parallel program using skeletons)

```
mMul xss yss
where
mMul [] yss              = []
mMul (xs : xss) yss      = (map (dotp xs) (transpose yss)) : (mMul xss yss)
dotp xs ys               = reduce (+) 0 (zipWith (∗) xs ys)
transpose xss            = transpose′ xss []
transpose′ [] yss        = yss
transpose′ (xs : xss) yss = transpose′ xss (rotate xs yss)
rotate xs yss            = zipWith (λx.λys.(ys ++ [x])) xs yss
```

Example 4 (Dot-product of binary trees—hand-parallel program using skeletons)

```
dotP xt yt = reduce (+) 0 (zipWith (∗) xt yt)
```

The dot-product of binary trees program can also be parallelised by evaluating the two recursive calls ($dotP\ xt_1\ yt_1$) and ($dotP\ xt_2\ yt_2$) simultaneously. An example of such a parallel definition of the dot-product program using the Glasgow Parallel Haskell (GpH) is shown in Example 5 and this avoids the use of intermediate data structures.

Example 5 (Dot-product of binary trees—hand-parallel program using GpH)

```
data BTree a ::= E | B a (BTree a) (BTree a)
dotP :: (BTree a) → (BTree a) → (BTree a)
dotP xt yt
where
dotP E yt                      = 0
dotP (B x xt1 xt2) E           = 0
dotP (B x xt1 xt2) (B y yt1 yt2) = runEval $ do
                                     z1 ← rpar (dotP xt1 yt1)
                                     z2 ← rseq (dotP xt2 yt2)
                                     return (x ∗ y) + z1 + z2
```

However, such parallelisation requires manual analysis of a given program which is tedious for non-trivial programs. Also, note that there is no obvious skeleton corresponding to the definition of this program. Therefore, we observe that it is challenging to obtain a parallel version for a given program that uses fewer intermediate data structures and is defined using skeletons.

In this paper, we propose a transformation method with the following aspects to resolve these issues:

1. Reduce the number of intermediate data structures in a given program using an existing transformation called *distillation* [5].
2. Automatically create a new data type for the program so that the structure of the new data type matches the algorithmic structure of the program.
3. Automatically create skeletons that operate over the new data type and identify potential instances of the skeletons in the program. The transformed program can then be executed in parallel using efficient implementations for the skeletons.

In our earlier works, we presented techniques to transform programs defined over polytypic [6] and list inputs [7] individually. In this paper, we integrate the two transformations and present it along with evaluation results. In Sect. 2, we introduce the language used in this work. We present the proposed parallelisation transformation in Sect. 3. In Sect. 4, we present the results of evaluating three example programs that are transformed using our proposed method. In Sect. 5, we present concluding remarks on our transformation method and discuss related work.

2 The Language

The higher-order language used in this work is shown in Definition 1 and uses call-by-name evaluation.

Definition 1 (*Language grammar*)

$$
\begin{array}{lll}
\textbf{data}\ T\ \alpha_1 \ldots \alpha_M & ::= & c_1\ t_1^1 \ldots t_N^1 \mid \ldots \mid c_K\ t_1^K \ldots t_N^K \quad \text{Type Declaration} \\
t & ::= & \alpha_m \mid T\ t_1 \ldots t_M \quad \text{Type Component}
\end{array}
$$

$$
\begin{array}{lll}
e & ::= & x \quad \text{Variable} \\
& \mid & c\ e_1 \ldots e_N \quad \text{Constructor Application} \\
& \mid & e_0 \quad \text{Function Definition} \\
& & \textbf{where} \\
& & f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = e_1\ \ldots\ f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = e_K \\
& \mid & f \quad \text{Function Call} \\
& \mid & e_0\ e_1 \quad \text{Application} \\
& \mid & \textbf{let}\ x_1 = e_1\ \ldots\ x_N = e_N\ \textbf{in}\ e_0 \quad \textbf{let}\text{–Expression} \\
& \mid & \lambda x.e \quad \lambda\text{–Abstraction} \\
p & ::= & x \mid c\ p_1 \ldots p_N \quad \text{Pattern}
\end{array}
$$

A program can contain data type declarations of the form shown in Definition 1. Here, T is the name of the data type, which can be polymorphic, with type parameters $\alpha_1, \ldots, \alpha_M$. A data constructor c_k may have zero or more components, each of which may be a type parameter or a type application. An expression e of type T is denoted by $e :: T$.

A program can also contain an expression which can be a variable, constructor application, function definition, function call, application, **let**-expression or λ-abstraction. Variables introduced in a function definition, **let**-expression or λ-abstraction are *bound*, while all other variables are *free*. The free variables in an expression e are denoted by $fv(e)$. Each constructor has a fixed arity. In an expression $c\ e_1 \ldots e_N$, N must be equal to the arity of the constructor c. For ease of presentation, patterns in function definition headers are grouped into two – $p_1^k \ldots p_M^k$ are inputs that are pattern-matched, and $x_{(M+1)}^k \ldots x_N^k$ are inputs that are not pattern-matched. The series of patterns $p_1^k \ldots p_M^k$ in a function definition must be non-overlapping and exhaustive. We use [] and (:) as shorthand notations for the Nil and $Cons$ constructors of a *cons*-list.

Definition 2 (*Context*) A context E is an expression with *hole*s in place of sub-expressions. $E[e_1, \ldots, e_N]$ is the expression obtained by filling holes in context E with the expressions $e_1, \ldots, e_N$.

3 Parallelisation Transformation

Given a program defined in the language presented in Definition 1, our proposed parallelisation transformation is performed in three stages:

1. Apply the *distillation transformation* on the given program to reduce the number of intermediate data structures used. This produces a *distilled program*. (Sect. 3.1)
2. Apply the *encoding transformation* on the distilled program to combine the inputs into a data structure that matches the algorithmic structure of the distilled program. This produces an *encoded program*. (Sect. 3.2)

3. Identify potential skeleton instances in the encoded program. This produces an *encoded parallel program*. (Sect. 3.4)

3.1 Distillation Transformation

Given a program in the language from Definition 1, *distillation* [5] is an unfold/fold-based technique that transforms the program to remove intermediate data structures and yields a *distilled program* that can potentially have super-linear speedups. The syntax of a distilled program $de^{\{\}}$ is shown in Definition 3. Here, ρ is the set of variables introduced by **let**–expressions that are created during generalisation. The bound variables of **let**-expressions are not decomposed by pattern-matching in a distilled program. Consequently, $de^{\{\}}$ is an expression that has fewer intermediate data structures.

Definition 3 (*Distilled form grammar*)

$$
\begin{array}{rcll}
de^{\rho} & ::= & x\ de_1^{\rho} \ldots de_N^{\rho} & \text{Variable Application} \\
& | & c\ de_1^{\rho} \ldots de_N^{\rho} & \text{Constructor Application} \\
& | & de_0^{\rho} & \text{Function Definition} \\
& & \textbf{where} & \\
& & f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = de_1^{\rho}\ \ldots\ f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = de_K^{\rho} & \\
& | & f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N & \text{Function Application} \\
& & \text{s.t. } \forall x \in \{x_1, \ldots, x_M\} \cdot x \notin \rho & \\
& | & \textbf{let } x_1 = de_1^{\rho}\ \ldots\ x_N = de_N^{\rho} \textbf{ in } de_0^{\rho \cup \{x_1, \ldots, x_N\}} & \textbf{let}\text{–Expression} \\
& | & \lambda x.de^{\rho} & \lambda\text{–Abstraction} \\
p & ::= & x \mid c\ p_1 \ldots p_N & \text{Pattern}
\end{array}
$$

Example 6 presents the distilled version of the matrix multiplication program from Example 1 and does not use intermediate data structures.

Example 6 (Matrix multplication—distilled program)

$$
\begin{array}{l}
mMul\ xss\ yss \\
\textbf{where} \\
mMul\ xss\ yss\ =\ mMul_1\ xss\ yss\ yss \\
mMul_1\ []\ zss\ yss\ =\ [] \\
mMul_1\ xss\ []\ yss\ =\ [] \\
mMul_1\ (xs : xss)\ (zs : zss)\ yss\ =\ \textbf{let } v = \lambda xs.g\ xs \\
\qquad \textbf{where} \\
\qquad g\ []\ = 0 \\
\qquad g\ (x : xs)\ = x \\
\qquad \textbf{in } (mMul_2\ zs\ xs\ yss\ v) : (mMul_1\ xss\ zss\ yss) \\
mMul_2\ []\ xs\ yss\ v\ =\ [] \\
mMul_2\ (z : zs)\ xs\ yss\ v\ =\ \textbf{let } v' = \lambda xs.g\ xs \\
\qquad \textbf{where} \\
\qquad g\ []\ = 0 \\
\qquad g\ (x : xs)\ = v\ xs \\
\qquad \textbf{in } (mMul_3\ xs\ yss\ v) : (mMul_2\ zs\ xs\ yss\ v') \\
mMul_3\ []\ yss\ v\ =\ 0 \\
mMul_3\ (x : xs)\ []\ v\ =\ 0 \\
mMul_3\ (x : xs)\ (ys : yss)\ v\ =\ (x * (v\ ys)) + (mMul_3\ xs\ yss\ v)
\end{array}
$$

3.2 Encoding Transformation

A distilled program is defined over the original program data types. In order to transform these data types into a structure that reflects the structure of the distilled program, we apply the encoding transformation detailed in this section. In this transformation, we *encode* the pattern-matched arguments of each recursive function f in the distilled program into a single argument which is of a new data type T_f and whose structure reflects the algorithmic structure of function f.

Consider a recursive function f, with arguments $x_1, \ldots, x_M, x_{(M+1)}, \ldots, x_N$, of the form shown in Definition 4 in a distilled program. Here, a function body e_k corresponding to function header $f\ p_1^k \ldots p_M^k\ x_{(M+1)}^k \ldots x_N^k$ in the definition of f may contain one or more recursive calls to function f.

Definition 4 (*Recursive function in distilled program*)

$$
\begin{array}{l}
f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N \\
\textbf{where} \\
f\ p_1^1 \ldots p_M^1\ x_{(M+1)} \ldots x_N \ = \ e_1 \\
\vdots \qquad\qquad\qquad\qquad\quad \vdots \\
f\ p_1^K \ldots p_M^K\ x_{(M+1)} \ldots x_N \ = \ e_K \\
\text{where} \\
\exists k \in \{1, \ldots, K\} \cdot e_k = E_k \left[f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \ldots, f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \right]
\end{array}
$$

3.2.1 Encoding Into New Type

The three steps to encode the pattern-matched arguments of function f into a new type are as follows:

1. **Declare a new data type for the encoded argument:**
   First, we declare a new data type T_f for the new encoded argument. This new data type corresponds to the data types of the original pattern-matched arguments of function f. The definition of the new data type T_f is shown in Definition 5.

Definition 5 (*New encoded data type T_f*)

$$
\begin{array}{l}
\textbf{data}\ T_f\ \alpha_1 \ldots \alpha_G \ ::= \ c_1\ T_1^1 \ldots T_L^1\ (T_f\ \alpha_1 \ldots \alpha_G)_1^1 \ldots (T_f\ \alpha_1 \ldots \alpha_G)_J^1 \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad\quad \mid \ \ c_K\ T_1^K \ldots T_L^K\ (T_f\ \alpha_1 \ldots \alpha_G)_1^K \ldots (T_f\ \alpha_1 \ldots \alpha_G)_J^K \\
\text{where} \\
\alpha_1, \ldots, \alpha_G \text{ are type parameters of the data types of pattern-matched arguments} \\
\forall k \in \{1, \ldots, K\} \cdot c_k \text{ is a fresh constructor for } T_f \text{ corresponding to } p_1^k \ldots p_M^k \text{ of} \\
\qquad\qquad \text{the pattern-matched arguments} \\
\qquad\qquad f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \left[\begin{array}{l} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \ldots, \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{array} \right] \\
\qquad\qquad \left\{ (z_1 :: T_1^k), \ldots, (z_L :: T_L^k) \right\} = fv(E_k) \setminus \{x_{(M+1)}, \ldots, x_N\}
\end{array}
$$

Here, a new constructor c_k of the type T_f is created for each set $p_1^k \ldots p_M^k$ of the pattern-matched arguments $x_1 \ldots x_M$ of function f that are encoded. As stated above, our objective is to encode the arguments of function f into a new type whose structure reflects the recursive structure of f. To achieve this, the components bound by

constructor c_k correspond to the variables in $p_1^k \ldots p_M^k$ that occur in the context E_k and the encoded arguments of the recursive calls to function f.

2. **Define a function to build encoded argument:**
   Given a function f of the form shown in Definition 4, we define a function $encode_f$, as shown in Definition 6, to build the encoded argument for function f.

Definition 6 (*Definition of function encode*$_f$)

$$
\begin{array}{l}
encode_f\ x_1 \ldots x_M \\
\textbf{where} \\
encode_f\ p_1^1 \ldots p_M^1 \ = \ e'_1 \\
\vdots \qquad\qquad\qquad\quad \vdots \\
encode_f\ p_1^K \ldots p_M^K \ = \ e'_K \\
\text{where} \\
\forall k \in \{1, \ldots, K\} \cdot e'_k = c_k\ z_1^k \ldots z_L^k\ (encode_f\ x_1^1 \ldots x_M^1) \ldots (encode_f\ x_1^J \ldots x_M^J) \\
\qquad\qquad f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \ldots, \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix} \\
\qquad\qquad \{z_1^k, \ldots, z_L^k\} = fv(E_k) \setminus \{x_{(M+1)}, \ldots, x_N\}
\end{array}
$$

Here, the original arguments $x_1 \ldots x_M$ of function f are pattern-matched and consumed by $encode_f$ in the same way as in the definition of f. For each pattern $p_1^k \ldots p_M^k$ of the arguments $x_1 \ldots x_M$, function $encode_f$ uses the corresponding constructor c_k whose components are the variables $z_1^k, \ldots, z_L^k$ in $p_1^k \ldots p_M^k$ that occur in the context E_k and the encoded arguments of the recursive calls to function f.

3. **Transform the distilled program:**
   After creating the encoded data type T_f and the $encode_f$ function for each function f, we transform the distilled program as shown in Definition 7 by defining a function f', which operates over the encoded argument, corresponding to function f.

Definition 7 (*Definition of transformed function over encoded argument*)

$$
\begin{array}{l}
f'\ x\ x_{(M+1)} \ldots x_N \\
\textbf{where} \\
f'\ \left(c_1\ z_1^1 \ldots z_L^1\ x_1^1 \ldots x_1^J\right)\ x_{(M+1)} \ldots x_N \ = \ e'_1 \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots \\
f'\ \left(c_K\ z_1^K \ldots z_L^K\ x_K^1 \ldots x_K^J\right)\ x_{(M+1)} \ldots x_N \ = \ e'_K \\
\text{where } \forall k \in \{1, \ldots, K\} \cdot e'_k \ = \ E_k \left[f'\ x_k^1\ x_{(M+1)}^1 \ldots x_N^1, \ \ldots, \ f'\ x_k^J\ x_{(M+1)}^J \ldots x_N^J \right] \\
\qquad\qquad f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \ldots, \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}
\end{array}
$$

The two steps to transform function f into function f' that operates over the encoded argument are:

(a) In each function definition header of f, replace the original pattern-matched arguments with the corresponding pattern of their encoded data type T_f. For instance, a function header $f\ p_1 \ldots p_M\ x_{(M+1)} \ldots x_N$ is transformed to the header $f'\ p\ x_{(M+1)} \ldots x_N$, where p is the pattern created by $encode_f$ corresponding to the pattern-matched arguments $p_1, \ldots, p_M$.

(b) In each call to function f, replace the original arguments with their corresponding encoded argument. For instance, a call $f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N$ is transformed to the function call $f'\ x\ x_{(M+1)} \ldots x_N$, where x is the encoded argument corresponding to the original arguments $x_1, \ldots, x_M$.

3.2.2 Encoding Into List

From Definition 5, we observe that the new encoded data type T_f that is created is essentially an n-ary tree. If we observe that T_f is a linear data type (i.e. $n = 1$), then we perform the encoding transformation such that the pattern-matched inputs are encoded into a *cons*-list instead of into a new data type. This will allow us to potentially parallelise the encoded program using existing parallel list-based skeletons as discussed in Sect. 3.4. In this section, we present the version of our encoding transformation to combine the inputs into a list of type $[T'_f]$. Note that, if the type T_f created by Definition 5 is linear, then each function body e_k for the recursive function f will contain at most one recursive call. The three steps to encode the pattern-matched inputs of function f are as follows:

1. **Declare a new data type for elements of the encoded list:**
   First, we declare a new data type T'_f for the elements of the encoded list. This new data type corresponds to the data types of the original pattern-matched arguments of function f. The definition of the new data type T'_f is shown in Definition 8.

Definition 8 (*New encoded data type T'_f for list*)

$$\textbf{data}\ T'_f\ \alpha_1 \ldots \alpha_G\ ::=\ c_1\ T_1^1 \ldots T_L^1 \mid \ldots \mid c_K\ T_1^K \ldots T_L^K$$

where

$\alpha_1, \ldots, \alpha_G$ are type parameters of the data types of pattern-matched arguments

$\forall k \in \{1, \ldots, K\} \cdot$ c_k is a fresh constructor for T'_f corresponding to $p_1^k \ldots p_M^k$ of the pattern-matched arguments

$$f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = e_k$$

$$\left\{(z_1 :: T_1^k), \ldots, (z_L :: T_L^k)\right\} =$$

$$\begin{cases} fv(E_k) \setminus \{x_{(M+1)}, \ldots, x_N\}, \text{ if } e_k = E_k\left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k\right] \\ fv(e_k) \setminus \{x_{(M+1)}, \ldots, x_N\}, \text{ otherwise} \end{cases}$$

Here, a new constructor c_k of the type T'_f is created for each set $p_1^k \ldots p_M^k$ of the pattern-matched arguments $x_1 \ldots x_M$ of function f that are encoded. As stated above, our objective is to encode the arguments of function f into a list where each element contains pattern-matched variables consumed in an iteration of f. To achieve this, the variables bound by constructor c_k correspond to the variables $z_1, \ldots, z_L$ in $p_1^k \ldots p_M^k$ that occur in the context E_k (if e_k contains a recursive call to f) or the expression e_k (otherwise). Consequently, the type components of constructor c_k are the data types of the variables $z_1, \ldots, z_L$.

2. **Define a function *encode*$_f$:**
   Given a recursive function f, we define a function $encode_f$ as shown in Definition 9 to build the encoded list in which each element is of type T'_f.

Definition 9 (*Definition of function encode$_f$ for list*)

$$
\begin{array}{l}
encode_f\ x_1 \ldots x_M \\
\textbf{where} \\
encode_f\ p_1^1 \ldots p_M^1 \ = \ e'_1 \\
\vdots \qquad\qquad\qquad\quad \vdots \\
encode_f\ p_1^K \ldots p_M^K \ = \ e'_K \\
\text{where} \\
\forall k \in \{1, \ldots, K\}. \\
e'_k = \begin{cases}
\left[c_k\ z_1^k \ldots z_L^k\right] +\!\!+ (encode_f\ x_1^k \ldots x_M^k), & \text{if } e_k = E_k \left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k\right] \\
\text{where } \{z_1^k, \ldots, z_L^k\} = fv(E_k) \setminus \{x_{(M+1)}, \ldots, x_N\} & \\
\left[c_k\ z_1^k \ldots z_L^k\right], & \text{otherwise} \\
\text{where } \{z_1^k, \ldots, z_L^k\} = fv(e_k) \setminus \{x_{(M+1)}, \ldots, x_N\} &
\end{cases} \\
\qquad \text{where } f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = e_k
\end{array}
$$

Here, for each pattern $p_1^k \ldots p_M^k$ of the pattern-matched arguments, the *encode*$_f$ function creates a list element. This element is composed of a new constructor c_k of type T'_f that binds $z_1^k, \ldots, z_L^k$, which are the variables in $p_1^k \ldots p_M^k$ that occur in the context E_k (if e_k contains a recursive call to f) or the expression e_k (otherwise). The encoded input of the recursive call $f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k$ is then computed by $encode_f\ x_1^k \ldots x_M^k$ and appended to the element to build the complete encoded list for function f.

3. **Transform the distilled program:**
   After creating the data type T'_f for the encoded list and the *encode*$_f$ function for each function f, we transform the distilled program as shown in Definition 10 by defining a recursive function f', which operates over the encoded list, corresponding to function f.

Definition 10 (*Definition of transformed function over encoded list*)

$$
\begin{array}{l}
f'\ x\ x_{(M+1)} \ldots x_N \\
\textbf{where} \\
f'\ \left((c_1\ z_1^1 \ldots z_L^1) : x^1\right)\ x_{(M+1)} \ldots x_N \ = \ e'_1 \\
\vdots \qquad\qquad\qquad\qquad\qquad\quad \vdots \\
f'\ \left((c_K\ z_1^K \ldots z_L^K) : x^K\right)\ x_{(M+1)} \ldots x_N \ = \ e'_K \\
\text{where} \\
\forall k \in \{1, \ldots, K\} \cdot e'_k = \begin{cases}
E_k \left[f'\ x^k\ x_{(M+1)}^k \ldots x_N^k\right], & \text{if } e_k = E_k \left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k\right] \\
e_k, & \text{otherwise}
\end{cases} \\
\qquad \text{where } f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = e_k
\end{array}
$$

The two steps to transform function f into function f' that operates over the encoded list are:

- In each function definition header of f, replace the pattern-matched arguments with a pattern to decompose the encoded list, such that the first element in the encoded list is matched with the corresponding pattern of the encoded type T'_f.
- In each call to function f, replace the pattern-matched arguments with their encoding.

3.3 Steps to Apply Encoding Transformation

In summary, the steps to apply the two versions of our encoding transformation on a given program are as follows:

1. Apply the distillation transformation on a given program. This produces a *distilled program*.
2. For each recursive function f in the distilled program,
   (a) Create an encoded data type T_f as described in Definition 5. This new data type T_f is essentially an n-ary tree.
   (b) `IF` $n > 1$
   `THEN` Apply the encoding transformation in Sect. 3.2.1 to combine the inputs into a new data type T_f whose structure matches that of function f.
   `ELSE` Apply the encoding transformation in Sect. 3.2.2 to combine the inputs into a *cons*-list of type $[T'_f]$.
   This produces an *encoded program*.

For the distilled matrix multiplication program in Example 6, we observe that the encoded data types T_{mMul_1}, T_{mMul_2} and T_{mMul_3} for recursive functions $mMul_1$, $mMul_2$ and $mMul_3$ are all linear (i.e. $n = 1$) as shown below.

$$\begin{array}{l}
\textbf{data}\ T_{mMul_1}\ a\ ::=\ c_1 \mid c_2 \mid c_3\ [a]\ [a]\ (T_{mMul_1}\ a) \\
\textbf{data}\ T_{mMul_2}\ a\ ::=\ c_4 \mid c_5\ (T_{mMul_2}\ a) \\
\textbf{data}\ T_{mMul_3}\ a\ ::=\ c_6 \mid c_7 \mid c_8\ a\ [a]\ (T_{mMul_3}\ a)
\end{array}$$

Therefore, we apply the encoding transformation presented in Sect. 3.2.2 to obtain an encoded matrix multplication program that operates over encoded lists of types $[T'_{mMul_1}]$, $[T'_{mMul_2}]$ and $[T'_{mMul_3}]$ as shown in Example 7.

Example 7 (Matrix multplication—encoded program)

$$\begin{array}{l}
\textbf{data}\ T'_{mMul_1}\ a\ ::=\ c_1 \mid c_2 \mid c_3\ [a]\ [a] \\
\textbf{data}\ T'_{mMul_2}\ a\ ::=\ c_4 \mid c_5 \\
\textbf{data}\ T'_{mMul_3}\ a\ ::=\ c_6 \mid c_7 \mid c_8\ a\ [a] \\
encode_{mMul_1}\ []\ zss = [c_1] \\
encode_{mMul_1}\ xss\ [] = [c_2] \\
encode_{mMul_1}\ (xs : xss)\ (zs : zss) = [c_3\ xs\ zs] +\!\!+ (encode_{mMul_1}\ xss\ zss) \\
encode_{mMul_2}\ [] = [c_4] \\
encode_{mMul_2}\ (z : zs) = [c_5] +\!\!+ (encode_{mMul_2}\ xs\ yss\ zs) \\
encode_{mMul_3}\ []\ yss = [c_6] \\
encode_{mMul_3}\ (x : xs)\ [] = [c_7] \\
encode_{mMul_3}\ (x : xs)\ (ys : yss) = [c_8\ x\ ys] +\!\!+ (encode_{mMul_3}\ xs\ yss) \\
mMul\ xss\ yss \\
\textbf{where} \\
mMul\ xss\ yss = mMul''_1\ (encode_{mMul_1}\ xss\ yss)\ yss \\
mMul'_1\ (c_1 : w)\ yss = [] \\
mMul'_1\ (c_2 : w)\ yss = [] \\
mMul'_1\ \big((c_3\ xs\ zs) : w\big)\ yss = \textbf{let}\ v = \lambda xs.g\ xs \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad g\ [] = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad g\ (x : xs) = x \\
\qquad\qquad\qquad\qquad \textbf{in}\ (mMul'_2\ (encode_{mMul_2}\ zs)\ xs\ yss\ v) : (mMul'_1\ w\ yss)
\end{array}$$

$$
\begin{array}{lcl}
mMul'_2\ (c_4 : w)\ xs\ yss\ v & = & [] \\
mMul'_2\ (c_5 : w)\ xs\ yss\ v & = & \textbf{let}\ v' = \lambda xs.g\ xs \\
 & & \qquad\qquad\qquad\qquad\qquad\quad \textbf{where} \\
 & & \qquad\qquad\qquad\qquad\qquad\quad g\ [] \quad\ = 0 \\
 & & \qquad\qquad\qquad\qquad\qquad\quad g\ (x : xs) \ = v\ xs \\
 & & \quad \textbf{in}\ (mMul'_3\ (encode_{mMul_3}\ xs\ yss)\ v) : (mMul'_2\ w\ xs\ yss\ v') \\
mMul'_3\ (c_6 : w)\ v & = & 0 \\
mMul'_3\ (c_7 : w)\ v & = & 0 \\
mMul'_3\ \big((c_8\ x\ ys) : w\big)\ v & = & (x * (v\ ys)) + (mMul'_3\ w\ v)
\end{array}
$$

3.4 Parallelisation Using Skeletons

In our work, we are interested in identifying parallel computations that can be modelled using map- and reduce-based skeletons as they are versatile and widely applicable in parallel programming. In this section, we discuss the definitions and parallelisation of list-based and polytypic skeletons that are most relevant to the examples used in this paper.

Using the sequential definitions of these skeletons, we automatically identify their instances in an encoded program and replace them with suitable calls to their parallel counterparts. The rules to achieve this are available in our earlier related work [7], which also discusses in detail the properties of the transformation to encode inputs into a list.

3.4.1 List-Based Skeletons

In this paper, the sequential definitions of list-based map and map-reduce skeletons whose instances we identify in an encoded program are as follows:

$$
\begin{array}{l}
map \qquad\qquad :: [a] \to (a \to b) \to [b] \\
map\ []\ f \qquad = [] \\
map\ (x : xs)\ f = (f\ x) : (map\ xs\ f)
\end{array}
$$

$$
\begin{array}{l}
mapRedr \qquad\qquad\quad :: [a] \to (b \to b \to b) \to b \to (a \to b) \to b \\
mapRedr\ []\ g\ v\ f \qquad = v \\
mapRedr\ (x : xs)\ g\ v\ f = g\ (f\ x)\ (mapRedr\ xs\ g\ v\ f)
\end{array}
$$

We identify instances of these definitions and replace them with suitable calls to their parallel counterparts in the Eden library [8] that provides implementations of parallel map and map-reduce skeletons in the following forms:

$$
\begin{array}{ll}
farmB & :: (Trans\ a, Trans\ b) \Rightarrow Int \to (a \to b) \to [a] \to [b] \\
parMapRedr & :: (Trans\ a, Trans\ b) \Rightarrow (b \to b \to b) \to b \to (a \to b) \to [a] \to b \\
parMapRedr1 & :: (Trans\ a, Trans\ b) \Rightarrow (b \to b \to b) \to (a \to b) \to [a] \to b
\end{array}
$$

3.4.2 Polytypic Skeletons

For the discussions in this paper, we present polytypic reduce and map-reduce skeletons in Examples 8 and 9, respectively, defined over a data type T_f of the form shown in Definition 5.

Example 8 (Reduce skeleton for encoded data type T_f)

$$
\begin{aligned}
&reduce_f\ (c_1\ x_1^1 \ldots x_N^1)\ g_1 \ldots g_K \;\; = \;\; e_1\\
&\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&reduce_f\ (c_K\ x_1^K \ldots x_N^K)\ g_1 \ldots g_K \;\; = \;\; e_K\\
&\text{where } \forall k \in \{1, \ldots, K\} \cdot e_k = g_k\ z_1^k \ldots z_L^k\ (reduce_f\ y_1^k\ g_1 \ldots g_K) \ldots (reduce_f\ y_J^k\ g_1 \ldots g_K)\\
&\qquad\qquad \{y_1^k, \ldots, y_J^k\} = \left\{x \mid (x :: T_f) \in \{x_1^k, \ldots, x_N^k\}\right\}\\
&\qquad\qquad \{z_1^k, \ldots, z_L^k\} = \{x_1^k, \ldots, x_N^k\} \setminus \{y_1^k, \ldots, y_J^k\}
\end{aligned}
$$

Example 9 (Map-reduce skeleton for encoded data type T_f)

$$
\begin{aligned}
&mapReduce_f\ (c_1\ x_1^1 \ldots x_N^1)\ g_1 \ldots g_K\ f_1 \ldots f_M \;\; = \;\; e_1\\
&\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&mapReduce_f\ (c_K\ x_1^K \ldots x_N^K)\ g_1 \ldots g_K\ f_1 \ldots f_M \;\; = \;\; e_K\\
&\text{where } \forall k \in \{1, \ldots, K\} \cdot e_k = g_k\ (f_1^k\ z_1^k) \ldots (f_L^k\ z_L^k)\ (mapReduce_f\ y_1^k\ g_1 \ldots g_K\ f_1 \ldots f_M)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (mapReduce_f\ y_J^k\ g_1 \ldots g_K\ f_1 \ldots f_M)\\
&\qquad\qquad \{y_1^k, \ldots, y_J^k\} = \left\{x \mid (x :: T_f) \in \{x_1^k, \ldots, x_N^k\}\right\}\\
&\qquad\qquad \{z_1^k, \ldots, z_L^k\} = \{x_1^k, \ldots, x_N^k\} \setminus \{y_1^k, \ldots, y_J^k\}\\
&\{f_1, \ldots, f_M\} = \bigcup_{k=1}^{K} \{f_1^k, \ldots, f_L^k\}
\end{aligned}
$$

A simplistic approach to parallelise these polytypic $reduce_f$ and $mapReduce_f$ definitions is by evaluating each recursive call simultaneously as shown in Examples 10 and 11, respectively.

Example 10 (Parallel reduce skeleton for type T_f using GpH)

$$
\begin{aligned}
&reduce_f\ (c_1\ x_1^1 \ldots x_N^1)\ t\ g_1 \ldots g_K \;\; = \;\; e_1\\
&\vdots \qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&reduce_f\ (c_K\ x_1^K \ldots x_N^K)\ t\ g_1 \ldots g_K \;\; = \;\; e_K\\
&\text{where } \forall k \in \{1, \ldots, K\} \cdot e_k = h\ (t \;\leq\; 0)\\
&\qquad\qquad \textbf{where}\\
&\qquad\qquad h\ True \;\; = g_k\ z_1^k \ldots z_L^k\ (reduce_f\ y_1^k\ t\ g_1 \ldots g_K)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (reduce_f\ y_J^k\ t\ g_1 \ldots g_K)\\
&\qquad\qquad h\ False = runEval\ \$\ do\\
&\qquad\qquad\qquad\qquad w_1^k \leftarrow rpar\ (reduce_f\ y_1^k\ (t-1)\ g_1 \ldots g_K)\\
&\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\qquad\qquad w_J^k \leftarrow rseq\ (reduce_f\ y_J^k\ (t-1)\ g_1 \ldots g_K)\\
&\qquad\qquad\qquad\qquad return\ (g_k\ z_1^k \ldots z_L^k\ w_1^k \ldots w_J^k)\\
&\qquad\qquad \{y_1^k, \ldots, y_J^k\} = \left\{x \mid (x :: T_f) \in \{x_1^k, \ldots, x_N^k\}\right\}\\
&\qquad\qquad \{z_1^k, \ldots, z_L^k\} = \{x_1^k, \ldots, x_N^k\} \setminus \{y_1^k, \ldots, y_J^k\}
\end{aligned}
$$

Example 11 (Parallel map-reduce skeleton for type T_f using GpH)

$$
\begin{array}{l}
mapReduce_f\ (c_1\ x_1^1 \ldots x_N^1)\ t\ g_1 \ldots g_K\ f_1 \ldots f_M\ =\ e_1 \\
\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
mapReduce_f\ (c_K\ x_1^K \ldots x_N^K)\ t\ g_1 \ldots g_K\ f_1 \ldots f_M\ =\ e_K \\
\text{where} \\
\forall k \in \{1, \ldots, K\} \cdot e_k = h\ (t\ \leq\ 0) \\
\qquad\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\qquad h\ True = g_k\ (f_k\ z_1^k) \ldots (f_L^k\ z_L^k)\ (mapReduce_f\ y_1^k\ t\ g_1 \ldots g_K\ f_1 \ldots f_M) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (mapReduce_f\ y_J^k\ t\ g_1 \ldots g_K\ f_1 \ldots f_M) \\
\qquad\qquad\qquad\qquad h\ False = runEval\ \$\ do \\
\qquad\qquad\qquad\qquad\qquad\qquad w_1^k \leftarrow rpar\ (mapReduce_f\ y_1^k\ (t-1)\ g_1 \ldots g_K\ f_1 \ldots f_M) \\
\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad w_J^k \leftarrow rseq\ (mapReduce_f\ y_J^k\ (t-1)\ g_1 \ldots g_K\ f_1 \ldots f_M) \\
\qquad\qquad\qquad\qquad\qquad\qquad return\ (g_k\ (f_k\ z_1^k) \ldots (f_L^k\ z_L^k)\ w_1^k \ldots w_J^k) \\
\qquad\qquad\qquad\qquad \{y_1^k, \ldots, y_J^k\} = \left\{x \mid (x :: T_f) \in \{x_1^k, \ldots, x_N^k\}\right\} \\
\qquad\qquad\qquad\qquad \{z_1^k, \ldots, z_L^k\} = \{x_1^k, \ldots, x_N^k\} \setminus \{y_1^k, \ldots, y_J^k\} \\
\{f_1, \ldots, f_M\} = \bigcup_{k=1}^{K} \{f_1^k, \ldots, f_L^k\}
\end{array}
$$

Here, t is a threshold value to control the number of parallel threads created by the polytypic skeletons using the *rpar* construct. For an n-ary encoded data type T_f (where $n > 1$), the initial value of t can be determined using the following simple rule-of-thumb where P is the number of processor cores:

$$\texttt{IF}\ n = 1\ \texttt{THEN}\ t = P\ \texttt{ELSE}\ t = \lceil log_n\ P \rceil + 1$$

Based on our transformation steps presented in Sect. 3.3, if $n = 1$, then the inputs will be encoded into a list to identify list-based parallel skeletons.

4 Evaluation

In this paper, we present the evaluation of three programs—matrix multiplication, dot-product of binary trees and maximum prefix sum—to illustrate interesting aspects of our transformation. The programs are evaluated on a 12-core Intel Xeon E5 processor each clocked at 2.7 GHz and 64 GB of main memory. GHC version 7.10.2 is used for the sequential versions of the benchmark programs, the Eden language compiler based on GHC 7.8.2 is used for list-based map and map-reduce skeletons, and the polytypic skeletons are parallelised using the Glasgow Parallel Haskell extension for GHC 7.10.2.

For all parallel versions of a benchmark program, only those skeletons that are present in the top level are executed using their parallel implementations and nested skeletons are executed using their sequential versions. The reason for this approach is to avoid uncontrolled creation of too many threads which we observe to result in inefficient parallel execution where the cost of thread creation and management is greater than the cost of parallel execution.

4.1 Example: Matrix Multiplication

The original sequential, hand-parallel, distilled and encoded versions of the matrix multiplication program were presented in Examples 1, 3, 6 and 7, respectively. Following this, we parallelise the encoded program using the list-based *map* and *mapRedr* skeletons presented in Sect. 3.4.1 and obtain the encoded parallel program shown in Example 12. Here, we observe that the encoded $mMul'_1$ and $mMul'_3$ functions are instances of the *map* and *mapRedr* skeletons and hence are defined using the *farmB* and *parMapRedr1* skeletons from the Eden library. The *farmB* skeleton requires the degree of parallelism to be specified which is usually given as the number of processors (*noPe*).

Example 12 (Matrix multplication—encoded parallel program)

$$
\begin{array}{l}
mMul''_1\ w\ yss\ =\ farmB\ noPe\ f\ w \\
\quad \textbf{where} \\
\quad f\ c_1 \quad\quad\quad = [] \\
\quad f\ c_2 \quad\quad\quad = [] \\
\quad f\ (c_3\ xs\ zs) \ = \textbf{let}\ v = \lambda xs.g\ xs \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad g\ [] \quad\quad = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad g\ (x : xs) \ = x \\
\qquad\qquad\qquad\quad \textbf{in}\ mMul''_2\ (encode_{mMul_2}\ zs)\ xs\ yss\ v \\
mMul'_2\ (c_4 : w)\ xs\ yss\ v\ =\ [] \\
mMul'_2\ (c_5 : w)\ xs\ yss\ v\ =\ \textbf{let}\ v' = \lambda xs.g\ xs \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad g\ [] \quad\quad = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad g\ (x : xs) \ = v\ xs \\
\qquad\qquad\qquad\quad \textbf{in}\ (mMul'_3\ (encode_{mMul_3}\ xs\ yss)\ v) : (mMul'_2\ w\ xs\ yss\ v') \\
mMul''_3\ w\ v\ =\ parMapRedr1\ g\ f\ w \\
\quad \textbf{where} \\
\quad g \quad\quad\quad = (+) \\
\quad f\ c_6 \quad\quad = 0 \\
\quad f\ c_7 \quad\quad = 0 \\
\quad f\ (c_8\ x\ ys) \ = x * (v\ ys)
\end{array}
$$

Figure 1 presents the speedups achieved by the encoded parallel program in comparison with the original, distilled and hand-parallel versions. An input of size `NxM` denotes the multiplication of matrices of sizes `NxM` and `MxN`. When compared to the original program, we observe that the encoded parallel version achieves a positive speedup for all input sizes. For input size `100x1000`, the speedup achieved is `6x-25x` more than the speedups achieved for the other input sizes. This is due to the intermediate data structure *transpose yss*, which is of the order of 1000 elements for input size `100x1000` and of the order of 100 elements for the other inputs, that is absent in the encoded parallel program. This can be verified from the comparison with the distilled version, which is also free of intermediate data structures. Hence, the encoded parallel program has a similar speedup for all input sizes when compared to the distilled version.

We also observe that even though the hand-parallel and encoded parallel versions parallelise the equivalent computations in the same fashion as can be seen from Examples 3 and 12, the encoded parallel version is marginally faster than for input sizes

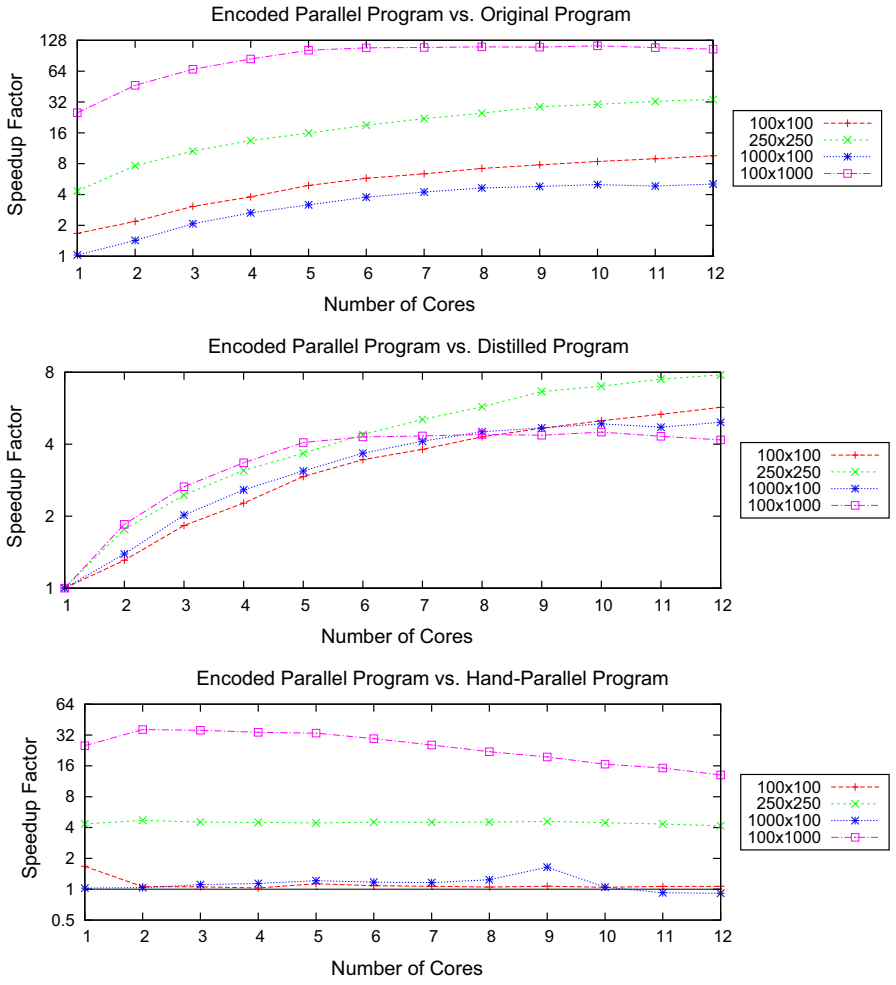

Fig. 1 Speedup of matrix multiplication

`100x100` and `1000x100`, `4x-4.5x` faster for input size `250x250` and `15x-32x` faster for input size `100x1000`. This is due to the use of intermediate data structures in the hand-parallel version, which has been removed in the encoded parallel version. Further, the hand-parallel version scales better with a higher number of cores the input size `100x1000`. This is because the encoded parallel version achieves better speedup even with fewer cores due to the elimination of intermediate data structures, and hence does not scale as impressively as the hand-parallel version.

4.2 Example: Dot-Product of Binary Trees

The original program to compute the dot-product of binary trees was presented in Example 2. The distilled version of this program remains the same as there are no

intermediate data structures. A hand-parallel version of this program using GpH was shown in Example 5. By following the steps in Sect. 3.3, we observe that the encoded data type $T_{dot\,P}$ created for function $dot\,P$ is binary (i.e. $n = 2$). Hence, we transform the dot product program to obtain an encoded program that operates over the encoded data type $T_{dot\,P}$. Following this, we can obtain an encoded parallel program using the $reduce_{dot\,P}$ skeleton defined over the encoded type $T_{dot\,P}$ as shown in Example 13.

Example 13 (Dot product of binary trees—encoded parallel program)

$$
\begin{array}{l}
\textbf{data}\ T_{dot\,P}\ a\ ::=\ c_1 \mid c_2 \mid c_3\ a\ a\ (T_{dot\,P}\ a)\ (T_{dot\,P}\ a) \\
encode_{dot\,P}\ E\ yt\ = c_1 \\
encode_{dot\,P}\ (B\ x\ xt_1\ xt_2)\ E\ = c_2 \\
encode_{dot\,P}\ (B\ x\ xt_1\ xt_2)\ (B\ y\ yt_1\ yt_2) = c_3\ x\ y\ (encode_{dot\,P}\ xt_1\ yt_1)\ (encode_{dot\,P}\ xt_2\ yt_2) \\
reduce_{dot\,P}\ c_1\ t\ g_1\ g_2\ g_3\ = g_1 \\
reduce_{dot\,P}\ c_2\ t\ g_1\ g_2\ g_3\ = g_2 \\
reduce_{dot\,P}\ (c_3\ x\ y\ lt\ rt)\ t\ g_1\ g_2\ g_3\ = \\
\quad \left\{ \begin{array}{l}
h\ (t \leq 0) \\
\textbf{where} \\
h\ True\ = g_3\ x\ y\ (reduce_{dot\,P}\ lt\ t\ g_1\ g_2\ g_3)\ (reduce_{dot\,P}\ rt\ t\ g_1\ g_2\ g_3) \\
h\ False = runEval\ \$\ do \\
\qquad\qquad x' \leftarrow rpar\ (reduce_{dot\,P}\ lt\ (t-1)\ g_1\ g_2\ g_3) \\
\qquad\qquad y' \leftarrow rseq\ (reduce_{dot\,P}\ rt\ (t-1)\ g_1\ g_2\ g_3) \\
\qquad\qquad return\ (g_3\ x\ y\ x'\ y')
\end{array} \right. \\
dot\,P''\ (encode_{dot\,P}\ xt\ yt)\ t \\
\textbf{where} \\
dot\,P''\ w\ t\ =\ reduce_{dot\,P}\ w\ t\ g_1\ g_2\ g_3 \\
\qquad \textbf{where} \\
\qquad g_1\ = 0 \\
\qquad g_2\ = 0 \\
\qquad g_3\ x\ y\ x'\ y'\ =\ (x * y) + x' + y'
\end{array}
$$

Figure 2 presents the speedups of the encoded parallel program compared to the original and the hand-parallel programs. An input size indicated by `N` denotes the dot-product of two identical balanced balanced binary trees with `N` nodes each. When compared to the original program we observe that the encoded parallel version achieves a positive speedup of `1.5x-2.6x` for all input sizes. The speedup achieved for the different input sizes scales equally for varying numbers of cores. When compared to the hand-parallel version, we observe that both programs achieve similar speedups for all input sizes. This is primarily because both versions essentially parallelise the dot-product computation in a similar fashion as can be observed from their definitions.

Further, we also note from our experiments that if the input trees are not well-balanced, then speedups achieved by both the encoded parallel and hand-parallel versions are reduced. This is because, from the definitions of these parallel versions, it is evident that their parallelisation and workloads of the parallel threads are dictated by the structure of the input tree. As discussed in other existing work [9], better implementations for n-ary tree skeletons can be obtained using tree-contraction [10, 11] approaches that can efficiently operate over unbalanced trees.

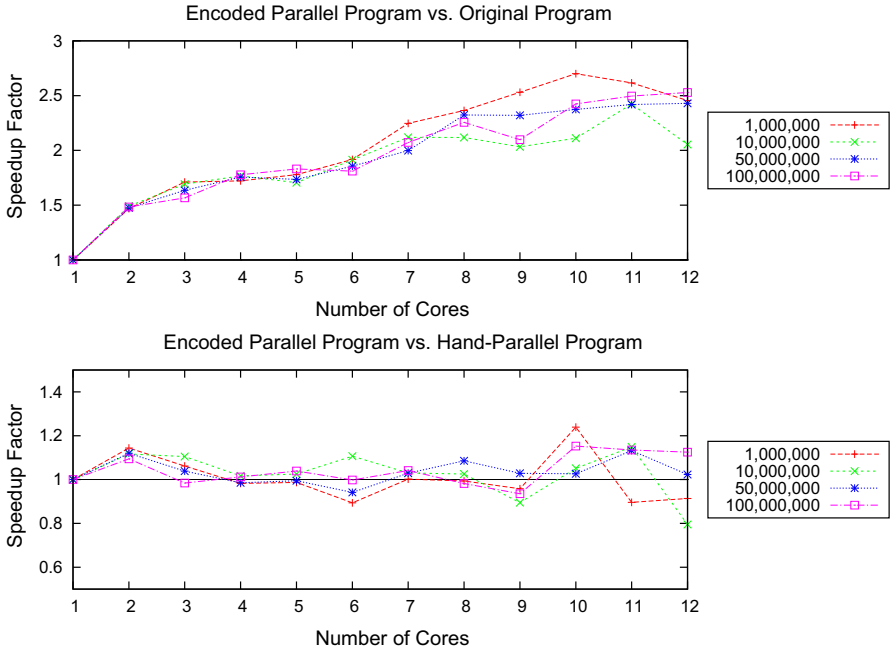

Fig. 2 Speedup of dot-product of binary trees

4.3 Example: Maximum Prefix Sum

The original program to compute the maximum prefix sum of a given list is presented in Example 14, where mps_1 computes the maximum prefix sum of list xs using the recursive function mps_2. Here, we consider max to be a built-in operator.

Example 14 (Maximum prefix sum—original program)

$$
\begin{array}{l}
mps_1 \;::\; [Int] \to Int \\
mps_1\ xs \\
\textbf{where} \\
\begin{array}{lcl}
mps_1\ [] & = & 0 \\
mps_1\ (x : xs) & = & mps_2\ xs\ x \\
mps_2\ []\ v & = & v \\
mps_2\ (x : xs)\ v & = & \textbf{let}\ v' = x + v \\
 & & \textbf{in}\ max\ v\ \big(max\ v'\ (mps_2\ xs\ v')\big)
\end{array}
\end{array}
$$

By applying the parallelisation transformation, we encode the input xs of the recursive function mps_2 into a list of type $[T'_{mps_2}]$, and the resulting encoded parallel program is presented in Example 15. Here, mps''_2 is found to be an instance of the list-based *accumulate* skeleton [1] and is defined using a call to its parallel implementation (*parAccumulate*).

Example 15 (Maximum prefix sum—encoded parallel program)

$$
\begin{array}{l}
\textbf{data}\ T'_{mps_2}\ a\ =\ c_1 \mid c_2\ a \\
encode_{mps_2}\ []\ \ \ \ \ \ \ \ =\ [c_1] \\
encode_{mps_2}\ (x : xs)\ =\ [c_2\ x] +\!\!+ (encode_{mps_2}\ xs) \\
mps_1\ xs \\
\textbf{where} \\
mps_1\ xs\ \ \ \ \ \ \ \ \ \ \ \ =\ mps''_2\ (encode_{mps_2}\ xs)\ 0 \\
mps''_2\ w\ v\ =\ parAccumulate\ p\ \oplus\ q\ \otimes\ w\ v \\
\qquad\qquad \textbf{where} \\
\qquad\qquad p\ c_1\ v\ \ \ \ \ \ =\ v \\
\qquad\qquad p\ (c_2\ x)\ v\ =\ v \\
\qquad\qquad \oplus\ \ \ \ \ \ \ \ =\ max \\
\qquad\qquad q\ c_1\ \ \ \ \ \ =\ v \\
\qquad\qquad q\ (c_2\ x)\ \ \ =\ x \\
\qquad\qquad \otimes\ \ \ \ \ \ \ \ =\ (+)
\end{array}
$$

Example 16 presents the hand-parallel version of the maximum prefix sum program where the recursive function mps_2 is identified as an instance of the *accumulate* skeleton and is defined using a call to its parallel implementation.

Example 16 (Maximum prefix sum—hand-parallelised program (HPP))

$$
\begin{array}{l}
mps_1\ xs \\
\textbf{where} \\
mps_1\ xs\ \ \ =\ mps_2\ xs\ 0 \\
mps_2\ xs\ v\ =\ parAccumulate\ p\ \oplus\ q\ \otimes\ xs\ v \\
\qquad\qquad \textbf{where} \\
\qquad\qquad p\ x\ v\ =\ max\ v\ (x + v) \\
\qquad\qquad \oplus\ \ \ \ =\ max \\
\qquad\qquad q\ \ \ \ \ =\ \lambda x.x \\
\qquad\qquad \otimes\ \ \ \ =\ (+)
\end{array}
$$

Figure 3 presents the speedups of the encoded parallel program compared to the original and hand-parallel program. An input size indicated by `N` denotes the computation of the maximum prefix sum for a list of size `N`. In comparison to the original program, the encoded parallel version achieves a positive speedup ranging between `2.2x-4.5x`. The speedup achieved also increases, though marginally, for increasingly large input sizes, which shows that the encoded parallel version scales well for different input sizes. However, the overall speedup gained does not scale linearly with the number of cores used due to the significant cost of encoding which is a sequential computation.

Further, the encoded parallel and hand-parallel versions parallelise the maximum prefix sum program in the same way using the *accumulate* skeleton, with the hand-parallel version defined over the original input list and the encoded parallel version defined over the encoded list. We observe that the hand-parallel program is marginally faster than the encoded parallel program for all input sizes due to the added cost of encoding inputs in the encoded program. This is an example where encoding inputs may be an overhead when the original input structures already match the algorithmic structure.

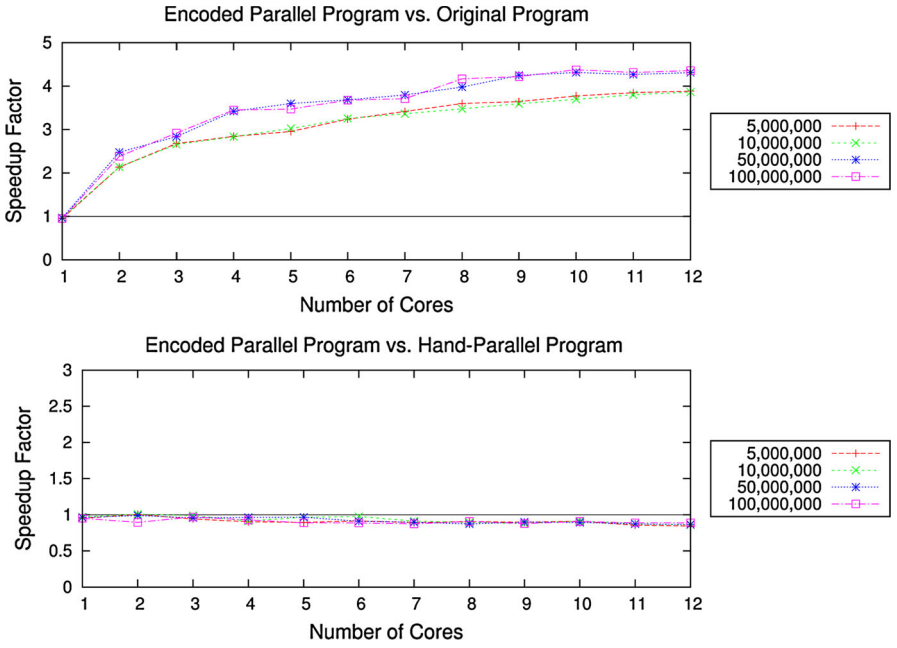

Fig. 3 Speedup of maximum prefix sum

5 Conclusion

5.1 Summary

The proposed transformation facilitates automatic identification of polytypic and list-based skeletons in a given program using the existing distillation transformation and a newly-proposed encoding transformation. Distillation provides efficiency by reducing the number of intermediate data structures used. Thus, the speedup from distillation is directly proportional to the sizes of intermediate data structures eliminated. The encoding transformation targets and benefits programs defined over inputs whose structures does not match that of the algorithm of the program. For instance, the speedups for matrix multiplication are impressive and the ones for dot product of binary trees and maximum prefix sum are moderate. This is because the structures of the inputs of the latter programs match the algorithmic structures of the programs, while it is not the case for the matrix multiplication example used here.

Thus, the efficiency achieved for matrix multiplication is partly due to the mismatch in the list-of-lists representation of a matrix compared to the algorithmic structure of the program. Such a representation may be considered unnatural given that there exist methods to match the data type structure with the algorithmic structure [12] and optimised representations of matrices in languages such as Haskell's Repa and NESL. Therefore, we need to further evaluate the encoding transformation on its role in the efficiency achieved, in addition to facilitating the identification of skeleton instances.

5.2 Related Work

A seminal work to derive parallel programs using fusion and tupling methods is presented in [3], where the need to extend them to polytypic and nested data structures is highlighted. In [13], the authors propose a method to systematically derive parallel programs from sequential definitions and automatically create auxiliary functions to define associative operators required for parallel evaluation. However, this method is restricted to a first-order language and functions defined over a single recursive linear data type that has an associative decomposition operator such as $+\!\!+$. To address this, [9] presents the *diffusion* transformation to decompose recursive functions into functions that can be described using the *accumulate* skeleton. While diffusion can transform a wide range of functions and can be extended using polytypic *accumulate* skeleton, it is only applicable to functions with one recursive input, and the properties and unit values of skeleton operators are verified and derived manually.

Since calculational approaches to program parallelisation were based on homomorphisms and the third homomorphism theorem on lists [14], this was extended to trees in [15] by decomposing a binary tree into a list of sub-trees called *zipper*, and deriving parallel computations on the zipper structure. Though it addresses tree-based programs, this method requires multiple sequential versions of the program to derive the parallel version. To address polytypic parallel skeletons, [16] presents parallel *map*, *reduce* and *accumulate* skeletons for n-ary trees which are implicitly converted to binary trees and uses parallel skeletons for binary trees. Despite introducing polytypic parallel skeletons, this approach requires manual check and derivation of the operator associativity and unit values, respectively. As an alternative, [17] proposes an analytical method to transform general recursive functions into a composition of polytypic data parallel skeletons. Even though this method is applicable to a wider range of problems, the transformed programs are defined by composing skeletons and employ multiple intermediate data structures.

References

1. Iwasaki, H., Hu, Z.: A new parallel skeleton for general accumulative computations. Int. J. Parallel Program. **32**, 389–414 (2004)
2. Skillicorn, D.B., Talia, D.: Models and languages for parallel computation. ACM Comput. Surv. **30**, 123–169 (1998)
3. Hu, Z., Takeichi, M., Chin, W.-N.: Parallelization in calculational forms. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (1998)
4. Matsuzaki, K., Kakehi, K., Iwasaki, H., Hu, Z., Akashi, Y.: A fusion-embedded skeleton library. In: International Conference on Parallel and Distributed Computing (EuroPar), Lecture Notes in Computer Science, vol. 3149, pp. 644–653. Springer-Verlag (2004)
5. Hamilton, G.W., Jones, N.D.: Distillation with labelled transition systems. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. (2012)
6. Kannan, V., Hamilton, G.W.: Program transformation to identify parallel skeletons. In: Proceedings of the 24th Euromicro International Conference on Parallel, Distributed and Network-Based Processing. (2016)
7. Kannan, V., Hamilton, G.W.: Program transformation to identify list-based parallel skeletons. In: International Workshop on Verification and Program Transformation, Electronic Proceedings in Theoretical Computer Science. (2016)

8. Loogen, R.: Eden parallel functional programming with Haskell. In: Zsók, V., Horváth, Z., Plasmeijer, R. (eds.) Lecture Notes in Computer Science, Central European Functional Programming School, pp. 142–206. Springer, Berlin (2012)
9. Hu, Z., Takeichi, M., Iwasaki, H.: Diffusion: calculating efficient parallel programs. In: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. (1999)
10. Miller, G.L., Reif, J.H.: Parallel tree contraction and its application. In: Proceedings of the 26th Annual Symposium on Foundations of Computer Science. (1985)
11. Morihata, A., Matsuzaki, K.: A practical tree contraction algorithm for parallel skeletons on trees of unbounded degree. In: Proceedings of the International Conference on Computational Science (ICCS). (2011)
12. Backus, J.: Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. Commun. ACM **21**, 613–641 (1978)
13. Chin, W.-N., Takano, A., Hu, Z.: Parallelization via context preservation. In: International Conference on Computer Languages. (1998)
14. Gibbons, J.: The third homomorphism theorem. J. Funct. Program. **6**(4), 657–665 (1996)
15. Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. ACM SIGPLAN SIGACT Symp. Princ. Program. Lang. **44**, 177–185 (2009)
16. Matsuzaki, K., Hu, Z., Takeichi, M.: Parallel skeletons for manipulating general trees. J. Parallel Comput. **32**, 590–603 (2006)
17. Ahn, J., Han, T.: An analytical method for parallelisation of recursive functions. Parallel Process. Lett. **10**, 87–98 (2001)