Program Transformation to Identify List-Based Parallel Skeletons

Venkatesh Kannan G. W. Hamilton

School of Computing, Dublin City University, Ireland

{vkannan, hamilton}@computing.dcu.ie

Algorithmic skeletons are used as building-blocks to ease the task of parallel programming by abstracting the details of parallel implementation from the developer. Most existing libraries provide implementations of skeletons that are defined over flat data types such as lists or arrays. However, skeleton-based parallel programming is still very challenging as it requires intricate analysis of the underlying algorithm and often uses inefficient intermediate data structures. Further, the algorithmic structure of a given program may not match those of list-based skeletons. In this paper, we present a method to automatically transform any given program to one that is defined over a list and is more likely to contain instances of list-based skeletons. This facilitates the parallel execution of a transformed program using existing implementations of list-based parallel skeletons. Further, by using an existing transformation called *distillation* in conjunction with our method, we produce transformed programs that contain fewer inefficient intermediate data structures.

1 Introduction

In today's computing systems, parallel hardware architectures that use multi-core CPUs and GPUs (Graphics Processor Units) are ubiquitous. On such hardware, it is essential that the programs developed be executed in parallel in order to effectively utilise the computing power that is available. To enable this, the parallelism that is inherent in a given program needs to be identified and exploited. However, parallel programming is tedious and error-prone when done by hand and is very difficult for a compiler to do automatically to the desired level.

To ease the task of parallel programming, a collection of algorithmic skeletons [1, 2] are often used for program development to abstract away from the complexity of implementing the parallelism. In particular, *map*, *reduce* and *zipWith* are primitive parallel skeletons that are often used for parallel programming [3, 4]. Most libraries such as Eden [5], SkeTo [6], Data Parallel Haskell (DPH) [7], and Accelerate [8] provide parallel implementations for these skeletons defined over flat data types such as lists or arrays. However, there are two main challenges in skeleton-based programming:

1. Using multiple skeletons in a program often introduces inefficient intermediate data structures [9, 10].
2. There may be a mismatch in data structures and algorithms used by the skeletons and the program [11, 12].

For example, consider the matrix multiplication program shown in Example 1.1, where *mMul* computes the product of two matrices *xss* and *yss*. The function *map* is used to compute the dot-product ($dotp$) of each row in *xss* and those in the transpose of *yss*, which is computed by the function *transpose*. Note that this definition uses multiple intermediate data structures, which is inefficient.

Submitted to:
VPT 2016

Example 1.1 (Matrix Multiplication)

```
mMul :: [[a]] → [[a]] → [[a]]
mMul xss yss
where
mMul [] yss              = []
mMul (xs : xss) yss      = (map (transpose yss []) (dotp xs)) : (mMul xss yss)
map [] f                 = []
map (x : xs) f           = (f x) : (map xs f)
dotp [] ys               = 0
dotp (x : xs) []         = 0
dotp (x : xs) (y : ys)   = (x ∗ y) + (dotp xs ys)
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) = (append ys [x]) : (rotate xs yss)
append [] ys             = ys
append (x : xs) ys       = x : (append xs ys)
```

A version of this program defined using the built-in *map*, *reduce* and *zipWith* skeletons is shown in Example 1.2.

Example 1.2 (Hand-Parallelised Matrix Multiplication)

```
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 [] yss         = yss
transpose (xs : xss) yss = transpose xss (rotate xs yss)
rotate xs yss            = zipWith (λx.λys.append ys [x]) xs yss
append [] ys             = ys
append (x : xs) ys       = x : (append xs ys)
```

As we can observe, though defined using parallel skeletons, this implementation still employs multiple intermediate data structures. For instance, the matrix constructed by the *transpose* function is subsequently decomposed by *map*. It is challenging to obtain a program that uses skeletons for parallel evaluation and contains very few intermediate data structures.

Therefore, it is desirable to have a method to automatically identify potential parallel computations in a given program, transform them to operate over flat data types to facilitate their execution using parallel skeletons provided in existing libraries, and reduce the number of inefficient intermediate data structures used.

In this paper, we present a transformation method with the following aspects:

1. Reduces inefficient intermediate data structures in a given program using an existing transformation technique called *distillation* [13]. (Section 3)
2. Automatically transforms the distilled program by *encoding* its inputs into a single *cons*-list, referred to as the *encoded list*. (Section 4)
3. Allows for parallel execution of the encoded program using efficient implementations of map and map-reduce skeletons that operate over lists. (Section 5)

In Section 6, we present concluding remarks on possible improvements to our transformation method and discuss related work.

2 Language

We focus on the automated parallelisation of functional programs because pure functional programs are free of side-effects, which makes them easier to analyse, reason about, and manipulate using program transformation techniques. This facilitates parallel evaluation of independent sub-expressions in a program. The higher-order functional language used in this work is shown in Definition 2.1.

Definition 2.1 (Language Grammar)

$$\begin{array}{lll}
e ::= & x & \text{Variable}\\
\mid & c\; e_1 \ldots e_N & \text{Constructor Application}\\
\mid & e_0 & \text{Function Definition}\\
 & \textbf{where} & \\
 & f\; p^1_1 \ldots p^1_M\; x^1_{(M+1)} \ldots x^1_N = e_1 \;\ldots\; f\; p^K_1 \ldots p^K_M\; x^K_{(M+1)} \ldots x^K_N = e_K & \\
\mid & f & \text{Function Call}\\
\mid & e_0\; e_1 & \text{Application}\\
\mid & \textbf{let}\; x_1 = e_1 \;\ldots\; x_N = e_N\; \textbf{in}\; e_0 & \textbf{let}\text{–expression}\\
\mid & \lambda x.e & \lambda\text{–Abstraction}\\
p ::= & x \mid c\; p_1 \ldots p_N & \text{Pattern}
\end{array}$$

A program in this language is an expression, which can be a variable, constructor application, function definition, function call, application, **let**-expression or λ-expression. Variables introduced in a λ-expression, **let**-expression, or function definition are *bound*, while all other variables are *free*. 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. Patterns in a function definition header are grouped into two – $p^k_1 \ldots p^k_M$ are inputs that are pattern-matched, and $x^k_{(M+1)} \ldots x^k_N$ are inputs that are not pattern-matched. The series of patterns $p^1_1 \ldots p^1_M, \ldots, p^K_1 \ldots p^K_M$ in a function definition must be non-overlapping and exhaustive. We use [] and (:) as short notations for the *Nil* and *Cons* constructors of a *cons*-list and $++$ for list concatenation. The set of free variables in an expression e is denoted as $fv(e)$.

Definition 2.2 (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 $e_1, \ldots, e_N$.

The call-by-name operational semantics of our language is defined using an evaluation relation as shown in Definition 2.3.

Definition 2.3 (Evaluation Relation)

$$\begin{array}{ll}
e \overset{r}{\rightsquigarrow}, \text{ iff } \exists e'.e \overset{r}{\rightsquigarrow} e' & e \Downarrow, \text{ iff } \exists v.e \Downarrow v\\
e \Downarrow v, \text{ iff } e \overset{r^*}{\rightsquigarrow} v \wedge \neg(v \overset{r}{\rightsquigarrow}) & e \Uparrow, \text{ iff } \forall e'.e \overset{r^*}{\rightsquigarrow} e' \Rightarrow e' \overset{r}{\rightsquigarrow}
\end{array}$$

Here, $\Downarrow$ is an evaluation relation between closed expressions and *values*, where values are expressions in weak head normal form (constructor applications and λ–abstractions). $e \Downarrow$ denotes that e converges, $e \Downarrow v$ denotes that e evaluates to the value v, and $e \Uparrow$ denotes that e diverges. $e \overset{r}{\rightsquigarrow}$ denotes reduction of expression e using the one-step reduction relation shown in Definition 2.4. The reduction can be f (unfolding of function f) or β (*beta*-substitution). The transitive closure of the reduction relation is denoted by $\overset{r^*}{\rightsquigarrow}$.

Definition 2.4 (One-Step Reduction Relation)

$$\big((\lambda x.e_0)\ e_1\big) \overset{\beta}{\rightsquigarrow} \big(e_0\{x \mapsto e_1\}\big) \qquad \frac{e_0 \overset{r}{\rightsquigarrow} e_0'}{(e_0\ e_1) \overset{r}{\rightsquigarrow} (e_0'\ e_1)} \qquad \frac{e_1 \overset{r}{\rightsquigarrow} e_1'}{(e_0\ e_1) \overset{r}{\rightsquigarrow} (e_0\ e_1')}$$

$$\frac{\big(f\ p_1 \ldots p_N = e\big) \wedge \big(\exists \theta \cdot \forall n \in \{1,\ldots,N\} \cdot e_n = p_n\theta\big)}{(f\ e_1 \ldots e_N) \overset{f}{\rightsquigarrow} e\theta}$$

$$\big(\mathbf{let}\ x_1 = e_1\ \ldots\ x_N = e_N\ \mathbf{in}\ e_0\big) \overset{\beta}{\rightsquigarrow} \big(e_0\{x \mapsto e_1,\ldots,x_N \mapsto e_N\}\big)$$

A program can also contain data type declarations of the form shown in Definition 2.5. Here, T is the name of the data type, which can be polymorphic, with type variables $\alpha_1,\ldots,\alpha_M$. A data constructor c_k may have zero or more components, each of which may be a type variable or a type application.

Definition 2.5 (Data Type Declaration)

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

3 Distillation

Objective: A given program may contain a number of inefficient intermediate data structures. In order to reduce them, we use an existing transformation technique called *distillation*.

Distillation [13] is a technique that transforms a program to remove intermediate data structures and yields a *distilled program*. It is an unfold/fold-based transformation that makes use of well-known transformation steps – unfold, generalise and fold [14] – and can potentially provide super-linear speedups to programs. The syntax of a distilled program $de^{\{\}}$ is shown in Definition 3.1. Here, ρ is the set of variables introduced by **let**–expressions; these are not decomposed by pattern-matching. Consequently, $de^{\{\}}$ is an expression that has fewer intermediate data structures.

Definition 3.1 (Distilled Form Grammar)

$$\begin{array}{rcll} de^\rho & ::= & x\ de_1^\rho \ldots de_N^\rho & \text{Variable Application} \\ & | & c\ de_1^\rho \ldots de_N^\rho & \text{Constructor Application} \\ & | & de_0^\rho & \text{Function Definition} \\ & & \mathbf{where} & \\ & & f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = de_1^\rho\ \ldots\ f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = de_K^\rho & \\ & | & f\ x_1 \ldots x_N & \text{Function Application} \\ & & \text{where}\ f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = de_1^\rho\ \ldots\ f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = de_K^\rho & \\ & & \qquad \forall n \in \{1,\ldots,N\} \cdot \big(x_n \in \rho\ \Rightarrow\ \forall k \in \{1,\ldots,K\} \cdot p_n^k = x_n^k\big) & \\ & | & \mathbf{let}\ x_1 = de_1^\rho\ \ldots\ x_N = de_N^\rho\ \mathbf{in}\ de_1^{\rho\ \cup\ \{x\}} & \mathbf{let}\text{–expression} \\ & | & \lambda x.de^\rho & \lambda\text{–Abstraction} \\ p & ::= & x \mid c\ p_1 \ldots p_N & \text{Pattern} \end{array}$$

Example 3.1 shows the distilled form of the example matrix multiplication program in Example 1.1. Here, we have lifted the definitions of functions $mMul_2$ and $mMul_3$ to the top level using lambda lifting for ease of presentation.

Example 3.1 (Distilled Matrix Multiplication)

```
$mMul\ xss\ yss$
where
$mMul\ xss\ yss$                        $= mMul_1\ xss\ yss\ yss$
$mMul_1\ []\ zss\ yss$                  $= []$
$mMul_1\ xss\ []\ yss$                  $= []$
$mMul_1\ (xs : xss)\ (zs : zss)\ yss$   $=$ let $v = \lambda xs.g\ xs$
                                              where
                                              $g\ []$         $= 0$
                                              $g\ (x : xs)$   $= x$
                                          in $(mMul_2\ zs\ xs\ yss\ v) : (mMul_1\ xss\ zss\ yss)$
$mMul_2\ []\ xs\ yss\ v$                $= []$
$mMul_2\ (z : zs)\ xs\ yss\ v$          $=$ let $v' = \lambda xs.g\ xs$
                                               where
                                               $g\ []$         $= 0$
                                               $g\ (x : xs)$   $= v\ xs$
                                          in $(mMul_3\ xs\ yss\ v) : (mMul_2\ zs\ xs\ yss\ v')$
$mMul_3\ []\ yss\ v$                    $= 0$
$mMul_3\ (x : xs)\ []\ v$               $= 0$
$mMul_3\ (x : xs)\ (ys : yss)\ v$       $= (x + (v\ ys)) + (mMul_3\ xs\ yss\ v)$
```

In this distilled program, function $mMul_1$ computes the product of matrices xss and yss, and functions $mMul_2$ and $mMul_3$ compute the dot-product of a row in xss and those in the transpose of yss. This version of matrix multiplication is free from intermediate data structures. In particular, distillation removes data structures that are constructed and subsequently decomposed as a part of the algorithm that is implemented in a given program.

Consequence: Using the distillation transformation, we obtain a semantically equivalent version of the original program that has fewer intermediate data structures.

4 Encoding Transformation

Objective: The data types and the algorithm of a distilled program, which we want to parallelise, may not match with those of the skeletons defined over lists. This would inhibit the potential identification of parallel computations that could be encapsulated using the map or map-reduce skeletons. To resolve this, we define a transformation that encodes the inputs of a distilled program into a single *cons*-list. The resulting encoded program is defined in a form that facilitates identification of list-based parallel skeleton instances.

To perform the encoding transformation, we first lift the definitions of all functions in a distilled program to the top-level using lambda lifting. Following this, for each recursive function f defined in the top-level **where**-expression of the distilled program, we encode the inputs $p_1, \ldots, p_M$ that are pattern-matched in the definition of f. Other inputs $x_{(M+1)}, \ldots, x_N$ that are never pattern-matched in the definition of f are not encoded. Further, we perform this encoding only for the recursive functions in a distilled program because they are potential instances of parallel skeletons, which are also defined recursively. The three steps to encode inputs $x_1, \ldots, x_M$ of function f into a *cons*-list, referred to as the encoded list $\bar{x}$, are illustrated in Figure 1 and described below. Here, we encode the pattern-matched inputs $x_1, \ldots, x_M$ into a *cons*-list of type $[T_f]$, where T_f is a new type created to contain the pattern-matched variables in $x_1, \ldots, x_M$.

Consider the definition of a recursive function f, with inputs $x_1, \ldots, x_M, x_{(M+1)}, \ldots, x_N$, of the form shown in Definition 4.1 in a distilled program. Here, for each body e_k corresponding to function header

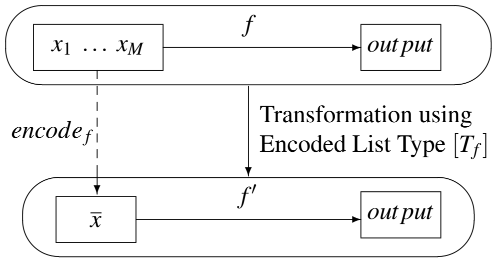

Figure 1: Steps to Encode Inputs of Function f

$f\ p_1^k \ldots p_M^k\ x_{(M+1)}^k \ldots x_N^k$ in the definition of f, we use one of the recursive calls to function f that may appear in e_k. All other recursive calls to f in e_k are a part of the context E_k.

Definition 4.1 (General Form of Recursive Function in Distilled Program)
$f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N$
where

$$\begin{array}{lcl} f\ p_1^1 \ldots p_M^1\ x_{(M+1)} \ldots x_N & = & e_1 \\ \vdots & & \vdots \\ f\ p_1^K \ldots p_M^K\ x_{(M+1)} \ldots x_N & = & e_K \end{array}$$

where $\exists k \in \{1,\ldots,K\} \cdot e_k = E_k \left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k \right]$

The three steps to encode the pattern-matched inputs are as follows:

1. **Declare a new encoded data type T_f :**
   First, we declare a new data type T_f for elements of the encoded list. This new data type corresponds to the data types of the pattern-matched inputs of function f that are encoded. The rules to declare type T_f are shown in Definition 4.2.

   Definition 4.2 (Rules to Declare Encoded Data Type for List)
   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 the type variables of the data types of the pattern-matched inputs
   $\forall k \in \{1,\ldots,K\} \cdot$
   c_k is a fresh constructor for T_f corresponding to $p_1^k \ldots p_M^k$ of the pattern-matched inputs

   $$\langle T_1^k,\ldots,T_L^k \rangle = \begin{cases} \langle T \mid (x :: T) \in (fv(E_k) \setminus \{x_{(M+1)},\ldots,x_N\}) \rangle, & \text{if } e_k = E_k \left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k \right] \\ \langle T \mid (x :: T) \in (fv(e_k) \setminus \{x_{(M+1)},\ldots,x_N\}) \rangle, & \text{otherwise} \end{cases}$$
   where $f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = e_k$

   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 inputs $x_1 \ldots x_M$ of function f that are encoded. As stated above, our objective is to encode the inputs of a recursive function f into a list, where each element contains the pattern-matched variables consumed in an iteration of f. To achieve this, the variables bound by constructor c_k correspond to 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). Consequently, the type components of constructor c_k are given as defined in the sequence $\langle T_1^k,\ldots,T_L^k \rangle$.

2. **Define a function *encode*$_f$:**
   For a recursive function f of the form shown in Definition 4.1, we use the rules in Definition 4.3 to define function $encode_f$ to build the encoded list, in which each element is of type T_f.

Definition 4.3 (Rules to Define Function $encode_f$)

$$
\begin{array}{l}
encode_f\ x_1 \ldots x_M \\
\textbf{where} \\
encode_f\ p_1^1 \ldots p_M^1 \;=\; e_1' \\
\vdots \qquad\qquad\qquad \vdots \\
encode_f\ p_1^K \ldots p_M^K \;=\; e_K' \\
\text{where} \\
\forall k \in \{1,\ldots,K\} \cdot e_k' = \begin{cases} \left[c_k\ z_1^k \ldots z_L^k\right] +\!\!+ (encode_f\ x_1^k \ldots x_M^k), & \text{if } e_k = E_k\left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k\right] \\ \text{where } \{z_1^k,\ldots,z_L^k\} = fv(E_k) \setminus \{x_{(M+1)},\ldots,x_N\} & \\ \left[c_k\ z_1^k \ldots z_L^k\right], & \text{otherwise} \\ \text{where } \{z_1^k,\ldots,z_L^k\} = fv(e_k) \setminus \{x_{(M+1)},\ldots,x_N\} & \end{cases} \\
\qquad\qquad\qquad\quad \text{where } f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = e_k
\end{array}
$$

Here, for each pattern $p_1^k \ldots p_M^k$ of the pattern-matched inputs, the $encode_f$ function creates a list element. This element is composed of a fresh 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 recursive function f, we transform the distilled program using the rules in Definition 4.4 by defining a recursive function f', which operates over the encoded list, corresponding to function f.

Definition 4.4 (Rules to Define Encoded Function Over Encoded List)

$$
\begin{array}{l}
f'\ \overline{x}\ x_{(M+1)} \ldots x_N \\
\textbf{where} \\
f'\ \left((c_1\ z_1^1 \ldots z_L^1) : \overline{x^1}\right)\ x_{(M+1)} \ldots x_N \;=\; e_1' \\
\vdots \qquad\qquad\qquad\qquad\qquad \vdots \\
f'\ \left((c_K\ z_1^K \ldots z_L^K) : \overline{x^K}\right)\ x_{(M+1)} \ldots x_N \;=\; e_K' \\
\text{where} \\
\forall k \in \{1,\ldots,K\} \cdot e_k' = \begin{cases} E_k\left[f'\ \overline{x^k}\ x_{(M+1)}^k \ldots x_N^k\right], & \text{if } e_k = E_k\left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k\right] \\ e_k, & \text{otherwise} \end{cases} \\
\qquad\qquad\qquad\quad \text{where } f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = e_k
\end{array}
$$

Here,

- In each function definition header of f, replace the pattern-matched inputs 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. For instance, a function header $f\ p_1 \ldots p_M\ x_{(M+1)} \ldots x_N$ is transformed to $f'\ \overline{p}\ x_{(M+1)} \ldots x_N$, where $\overline{p}$ is a pattern to match the first element in the encoded list with a pattern of the type T_f.
- In each call to function f, replace the pattern-matched inputs with their encoding. For instance, a call $f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N$ is transformed to $f'\ \overline{x}\ x_{(M+1)} \ldots x_N$, where $\overline{x}$ is the encoding of the pattern-matched inputs $x_1,\ldots,x_M$.

The encoded data types, encode functions and encoded program obtained for the distilled matrix multiplication program from Example 3.1 are shown in Example 4.1.

Example 4.1 (Distilled Matrix Multiplication with Inputs Encoded to *cons*-list)

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

$$\begin{array}{lcl}
encode_{mMul_1}\ []\ zss & = & [c_1] \\
encode_{mMul_1}\ xss\ [] & = & [c_2] \\
encode_{mMul_1}\ (xs:xss)\ (zs:zss) & = & [c_3\ xs\ zs] +\!\!+ (encode_{mMul_1}\ xss\ zss) \\
encode_{mMul_2}\ [] & = & [c_4] \\
encode_{mMul_2}\ (z:zs) & = & [c_5] +\!\!+ (encode_{mMul_2}\ xs\ yss\ zs) \\
encode_{mMul_3}\ []\ yss & = & [c_6] \\
encode_{mMul_3}\ (x:xs)\ [] & = & [c_7] \\
encode_{mMul_3}\ (x:xs)\ (ys:yss) & = & [c_8\ x\ ys] +\!\!+ (encode_{mMul_3}\ xs\ yss)
\end{array}$$

$$\begin{array}{lcl}
mMul'\ xss\ yss & & \\
\textbf{where} & & \\
mMul'\ xss\ yss & = & mMul'_1\ (encode_{mMul_1}\ xss\ yss)\ yss \\
mMul'_1\ (c_1:\overline{x})\ yss & = & [] \\
mMul'_1\ (c_2:\overline{x})\ yss & = & [] \\
mMul_1\ \big((c_3\ xs\ zs):\overline{x}\big)\ yss & = & \textbf{let}\ v = \lambda xs.g\ xs \\
 & & \quad\quad \textbf{where} \\
 & & \quad\quad g\ [] \quad\quad = 0 \\
 & & \quad\quad g\ (x:xs) = x \\
 & & \textbf{in}\ (mMul'_2\ (encode_{mMul_2}\ zs)\ xs\ yss\ v) : (mMul'_1\ \overline{x}\ yss) \\
mMul'_2\ (c_4:\overline{x})\ xs\ yss\ v & = & [] \\
mMul'_2\ (c_5:\overline{x})\ xs\ yss\ v & = & \textbf{let}\ v' = \lambda xs.g\ xs \\
 & & \quad\quad \textbf{where} \\
 & & \quad\quad g\ [] \quad\quad = 0 \\
 & & \quad\quad g\ (x:xs) = v\ xs \\
 & & \textbf{in}\ (mMul'_3\ (encode_{mMul_3}\ xs\ yss)\ v) : (mMul'_2\ \overline{x}\ xs\ yss\ v') \\
mMul'_3\ (c_6:\overline{x})\ v & = & 0 \\
mMul'_3\ (c_7:\overline{x})\ v & = & 0 \\
mMul'_3\ \big((c_8\ x\ ys):\overline{x}\big)\ v & = & (x * (v\ ys)) + (mMul'_3\ \overline{x}\ v)
\end{array}$$

4.1 Correctness

The correctness of the encoding transformation can be established by proving that the result computed by each recursive function f in the distilled program is the same as the result computed by the corresponding recursive function f' in the encoded program. That is,

$$\begin{array}{l}
\big(f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N\big) = \big(f'\ \overline{x}\ x_{(M+1)} \ldots x_N\big) \\
\text{where } \overline{x} = encode_f\ x_1 \ldots x_M
\end{array}$$

Proof:

The proof is by structural induction over the encoded list type $[T_f]$.

Base Case:

For the encoded list $\overline{x^k} = \big((c_k\ z_1^k \ldots z_L^k) : []\big)$ computed by $encode_f\ p_1^k \ldots p_M^k$,

1. By Definition 4.1, L.H.S. evaluates to e_k.
2. By Definition 4.4, R.H.S. evaluates to e_k.

Inductive Case:

For the encoded list $\overline{x^k} = \big((c_k\ z_1^k \ldots z_L^k) : \overline{x^k}\big)$ computed by $encode_f\ p_1^k \ldots p_M^k$,

1. By Definition 4.1, L.H.S. evaluates to $E_k\left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k\right]$.

2. By Definition 4.4, R.H.S. evaluates to $E_k\left[f'\ \overline{x^k}\ x^k_{(M+1)} \dots x^k_N\right]$.

3. By inductive hypothesis, $\left(f\ x_1 \dots x_M\ x_{(M+1)} \dots x_N\right) = \left(f'\ \overline{x}\ x_{(M+1)} \dots x_N\right)$. $\square$

Consequence: As a result of the encoding transformation, the pattern-matched inputs of a recursive function are encoded into a *cons*-list by following the recursive structure of the function. Parallelisation of the encoded program produced by this transformation by identifying potential instances of map and map-reduce skeletons is discussed in Section 5.

5 Parallel Execution of Encoded Programs

Objective: An encoded program defined over an encoded list is more likely to contain recursive functions that resemble the structure of *map* or *map-reduce* skeletons. This is because the $encode_f$ function constructs the encoded list in such a way that it reflects the recursive structure of the map and map-reduce skeletons defined over a *cons*-list. Therefore, we look for instances of these skeletons in our encoded program.

In this work, we identify instances of only map and map-reduce skeletons in an encoded program. This is because, as shown in Property 5.1, any function that is an instance of a reduce skeleton in an encoded program that operates over an encoded list cannot be efficiently evaluated in parallel because the reduction operator will not be associative.

Property 5.1 (Non-Associative Reduction Operator for Encoded List)
Given an encoded program defined over an encoded list, the reduction operator $\oplus$ in any instance of a reduce skeleton is not associative, that is $\forall x,y,z \cdot (x \oplus (y \oplus z)) \neq ((x \oplus y) \oplus z)$.

Proof:

1. From Definition 4.4, given an encoded function f',

   $f'\ ::\ [T_f]\ \rightarrow\ T_{(M+1)}\ \dots\ T_N\ \rightarrow\ b$
   where $[T_f]$ is the encoded list data type.
   $T_{(M+1)}, \dots, T_N$ are data types for inputs that are not encoded.
   b is the output data type.

2. If f' is an instance of a reduce skeleton, then the type of the binary reduction operator is given by $\oplus :: T_f \rightarrow b \rightarrow b$.

3. Given that T_f is a newly created data type, it follows from (2) that the binary operator $\oplus$ is not associative because the two input data types T_f and b cannot not be equal. $\square$

5.1 Identification of Skeletons

To identify skeleton instances in a given program, we use a framework of *labelled transition systems (LTSs)*, presented in Definition 5.1, to represent and analyse the encoded programs and skeletons. This is because LTS representations enable matching the recursive structure of the encoded program with that of the skeletons rather than finding instances by matching expressions.

Definition 5.1 (Labelled Transition System (LTS))
A LTS for a given program is given by $l = (\mathscr{S},\ s_0,\ Act,\ \rightarrow)$ where:

- $\mathscr{S}$ is the set of *states* of the LTS, where each state has a unique label s.
- $s_0 \in \mathscr{S}$ is the start state denoted by *start(l)*.
- Act is one of the following actions:
  - x, a free variable or **let**-expression variable,
  - c, a constructor in an application,
  - λ, a λ-abstraction,
  - $@$, an expression application,
  - $\#i$, the i^{th} argument in an application,
  - p, the set of patterns in a function definition header,
  - **let**, a **let**-expression body.
- $\rightarrow\ \subseteq \mathscr{S} \times Act \times \mathscr{S}$ relates pairs of states by actions in Act such that if $s \in \mathscr{S}$ and $s \xrightarrow{\alpha} s'$ then $s' \in \mathscr{S}$ where $\alpha \in Act$.

The LTS corresponding to a given program e can be constructed by $\mathscr{L}[\![e]\!]\ s_0\ \emptyset\ \emptyset$ using the rules $\mathscr{L}$ shown in Definition 5.2. Here, s_0 is the start state, ϕ is the set of previously encountered function calls mapped to their corresponding states, and Δ is the set of function definitions. A LTS built using these rules is always finite because if a function call is re-encountered, then the corresponding state is reused.

Definition 5.2 (LTS Representation of Program)

$$
\begin{array}{ll}
\mathscr{L}[\![x]\!]\ s\ \phi\ \Delta & = s \rightarrow (x, \mathbf{0}) \\
\mathscr{L}[\![c\ e_1 \ldots e_N]\!]\ s\ \phi\ \Delta & = s \rightarrow (c, \mathbf{0}), (\#1, \mathscr{L}[\![e_1]\!]\ s_1\ \phi\ \Delta), \ldots, (\#N, \mathscr{L}[\![e_N]\!]\ s_N\ \phi\ \Delta) \\
\mathscr{L}[\![e_0\ \mathbf{where}\ \delta_1 \ldots \delta_J]\!]\ s\ \phi\ \Delta & = \mathscr{L}[\![e_0]\!]\ s\ \phi\ \left(\Delta \cup \{f_1 \mapsto \delta_1, \ldots, f_J \mapsto \delta_J\}\right) \\
\multicolumn{2}{r}{\text{where } \forall j\{1,\ldots,J\} \cdot \delta_j = f_j\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = e_1\ \ldots\ f_j\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = e_K} \\
\mathscr{L}[\![f]\!]\ s\ \phi\ \Delta & = \begin{cases} l \quad \text{where } \phi(f) = start(l), & \text{if } f \in dom(\phi) \\ \mathscr{L}[\![\Delta(f)]\!]\ s\ (\phi \cup \{f \mapsto s\})\ \Delta, & \text{otherwise} \end{cases} \\
\mathscr{L}\left[\!\!\left[\begin{array}{lcl} f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 & = & e_1 \\ \vdots & & \vdots \\ f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K & = & e_K \end{array}\right]\!\!\right] s\ \phi\ \Delta & = \begin{cases} s \rightarrow (p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1, l_1),\ \ldots, \\ \qquad (p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K, l_K) \\ \text{where } \forall k \in \{1,\ldots,K\} \cdot l_k = \left(\mathscr{L}[\![e_k]\!]\ s_k\ \phi\ \Delta\right) \end{cases} \\
\mathscr{L}[\![e_0\ e_1]\!]\ s\ \phi\ \Delta & = s \rightarrow (@, \mathscr{L}[\![e_0]\!]\ s_0\ \phi\ \Delta), (\#1, \mathscr{L}[\![e_1]\!]\ s_1\ \phi\ \Delta) \\
\mathscr{L}[\![\mathbf{let}\ x_1 = e_1\ \ldots\ x_N = e_N\ \mathbf{in}\ e_0]\!]\ s\ \phi\ \Delta & = s \rightarrow \big(\mathbf{let}, \mathscr{L}[\![e_0]\!]\ s_0\ \phi\ \Delta), \\
 & \qquad (x_1, \mathscr{L}[\![e_1]\!]\ s_1\ \phi\ \Delta), \ldots, (x_N, \mathscr{L}[\![e_N]\!]\ s_N\ \phi\ \Delta) \\
\mathscr{L}[\![\lambda x.e]\!]\ s\ \phi\ \Delta & = s \rightarrow (\lambda, \mathscr{L}[\![e]\!]\ s_1\ \phi\ \Delta)
\end{array}
$$

Definition 5.3 (LTS Substitution)
A substitution is denoted by $\theta = \{x_1 \mapsto l_1, \ldots, x_N \mapsto l_N\}$. If l is an LTS, then $l\theta = l\{x_1 \mapsto l_1, \ldots, x_N \mapsto l_N\}$ is the result of simultaneously replacing the LTSs $s_n \rightarrow (x_n, \mathbf{0})$ with the corresponding LTS l_n in the LTS l while ensuring that bound variables are renamed appropriately to avoid name capture.

Potential instances of skeleton LTSs can be identified and replaced with suitable calls to corresponding skeletons in the LTS of an encoded program l by $\mathscr{S}[\![l]\!]\ \emptyset\ \langle\rangle\ \omega$ using the rules presented in Definition 5.4.

Definition 5.4 (Extraction of Program from LTS with Skeletons)

$$\mathscr{S}[\![l]\!]\ \rho\ \sigma\ \omega\ =\ \begin{cases} \left.\begin{array}{l} f\ e_1 \ldots e_N, \qquad \text{if } \exists (f\ x_1 \ldots x_N, l') \in \omega, \theta \cdot l'\theta = l \\ \text{where} \\ \theta = \{x_1 \mapsto l_1, \ldots, x_N \mapsto l_N\} \\ \forall n \in \{1, \ldots, N\} \cdot e_n = (\mathscr{S}[\![l_n]\!]\ \rho\ \sigma\ \omega) \\ f\ e_1 \ldots e_N, \qquad \text{if } \exists (s, f) \in \rho \cdot start(l) = s \\ \text{where } \sigma = \langle e_1, \ldots, e_N \rangle \\ \mathscr{S}'[\![l]\!]\ \rho\ \sigma\ \omega, \ \text{otherwise} \end{array}\right\}, & \text{if } \exists s \in states(l), \alpha \cdot s \xrightarrow{\alpha} start(l) \\ \mathscr{S}'[\![l]\!]\ \rho\ \sigma\ \omega, & \text{otherwise} \end{cases}$$

$$\mathscr{S}'[\![s \to (x, \mathbf{0})]\!]\ \rho\ \sigma\ \omega \qquad = x\ e_1 \ldots e_N \quad \text{where } \sigma = \langle e_1, \ldots, e_N \rangle$$

$$\mathscr{S}'[\![s \to (c, \mathbf{0}), (\#1, l_1), \ldots, (\#N, l_N)]\!]\ \rho\ \sigma\ \omega \qquad = c\ (\mathscr{S}[\![l_1]\!]\ \rho\ \sigma\ \omega) \ldots (\mathscr{S}[\![l_N]\!]\ \rho\ \sigma\ \omega)$$

$$\mathscr{S}'\left[\!\!\left[\begin{array}{l} s \to (p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1, l_1),\ \ldots, \\ \quad (p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K, l_K) \end{array}\right]\!\!\right]\ \rho\ \sigma\ \omega = \begin{cases} f\ e_1 \ldots e_N \\ \textbf{where} \\ f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 \ = \ e'_1 \\ \vdots \qquad\qquad\qquad\qquad \vdots \\ f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K \ = \ e'_K \\ \text{where } f \text{ is fresh, } \sigma = \langle e_1, \ldots, e_N \rangle \\ \qquad \forall k \in \{1, \ldots, K\} \cdot e'_k = \left(\mathscr{S}[\![l_k]\!]\ \rho'\ \langle\rangle\ \omega\right) \\ \qquad \rho' = \rho \cup \{(s, f)\} \end{cases}$$

$$\mathscr{S}'[\![s \to (@, l_0), (\#1, l_1)]\!]\ \rho\ \sigma\ \omega \qquad = \mathscr{S}[\![l_0]\!]\ \rho\ \langle (\mathscr{S}[\![l_1]\!]\ \rho\ \omega\ \langle\rangle) : \sigma \rangle\ \omega$$

$$\mathscr{S}'[\![s \to (\mathbf{let}, l_0), (x_1, l_1), \ldots, (x_N, l_N)]\!]\ \rho\ \sigma\ \omega \qquad = \mathbf{let}\ x_1 = (\mathscr{S}[\![l_1]\!]\ \rho\ \sigma\ \omega)\ \ldots\ x_N = (\mathscr{S}[\![l_N]\!]\ \rho\ \sigma\ \omega)\ \ \mathbf{in}\ (\mathscr{S}[\![l_0]\!]\ \rho\ \sigma\ \omega)$$

$$\mathscr{S}'[\![s \to (\lambda, l)]\!]\ \rho\ \sigma\ \omega \qquad = \lambda x.(\mathscr{S}[\![l]\!]\ \rho\ \sigma\ \omega) \quad \text{where } x \text{ is fresh}$$

Here, the parameter ρ contains the set of new functions that are created and associates them with their corresponding states in the LTS. The parameter σ contains the sequence of arguments of an application expression. The set ω is initialised with pairs of application expression and corresponding LTS representation of each parallel skeleton to be identified in a given LTS; for example, $(map\ xs\ f, l)$ is a pair in ω where *map xs f* is the application expression for *map* and l is its LTS representation.

The definitions of list-based map and map-reduce skeletons whose instances we identify in an encoded program are as follows:

$$\begin{array}{lcl} map & :: & [a] \to (a \to b) \to [b] \\ map\ []\ f & = & [] \\ map\ (x : xs)\ f & = & (f\ x) : (map\ xs\ f) \\ mapReduce & :: & [a] \to (b \to b \to b) \to b \to (a \to b) \to b \\ mapReduce\ []\ g\ v\ f & = & v \\ mapReduce\ (x : xs)\ g\ v\ f & = & g\ (f\ x)\ (mapReduce\ xs\ g\ v\ f) \end{array}$$

Property 5.2 (Non-Empty Encoded List)
Given rules in Definition 4.3 to encode inputs into a list, $\forall f, x_1, \ldots, x_M \cdot \left(encode_f\ x_1 \ldots x_M\right) \neq []$.
Proof:
From Definition 4.3, $\exists k \in \{1, \ldots, K\} \cdot p_1^k \ldots p_M^k$ that matches inputs $x_1 \ldots x_M$.
Consequently, $\left(encode_f\ x_1 \ldots x_M\right) = [c_k\ z_1^k \ldots z_L^k] +\!\!+ \left(encode_f\ x_1^k \ldots x_M^k\right)$. Therefore, the list computed by $encode_f\ x_1 \ldots x_M$ is at least a singleton. □

From Property 5.2, it is evident that the encoded programs produced by our transformation will always be defined over non-empty encoded list inputs. Consequently, to identify instances of *map* and *mapReduce* skeletons in an encoded program, we represent only the patterns corresponding to non-empty inputs, i.e. $(x : xs)$, in the LTSs built for the skeletons.

As an example, the LTSs built for the map skeleton and the $mMul'_1$ function in the encoded program for matrix multiplication in Example 4.1 are illustrated in Figures 2 and 3, respectively. Here, we observe that the LTS of $mMul'_1$ is an instance of the LTS of *map* skeleton. Similarly, the LTS of $mMul'_3$ is an instance of the LTS of *mapReduce* skeleton.

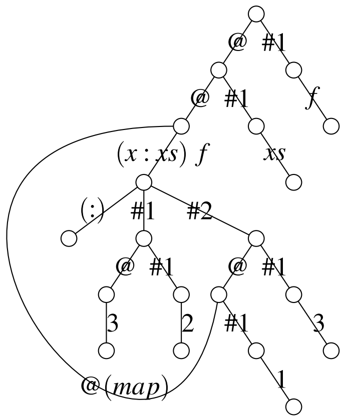

Figure 2: LTS for *map* Skeleton.

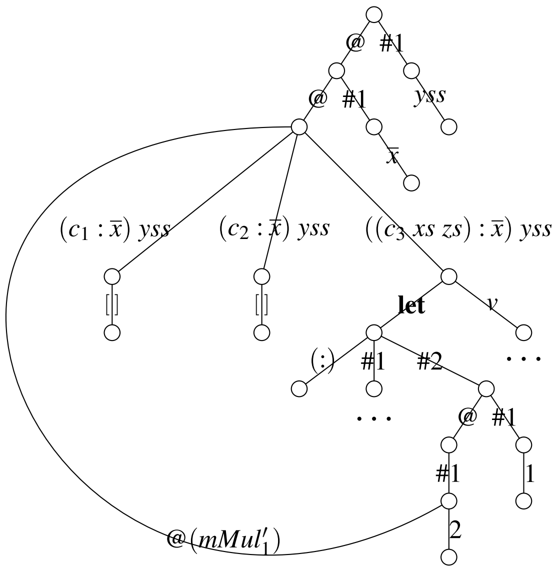

Figure 3: LTS for $mMul'_1$ Function.

5.2 Parallel Implementation of Skeletons

In order to evaluate the parallel programs obtained by our method presented in this chapter, we require efficient parallel implementations of the map and map-reduce skeletons. For the work presented in this paper, we use the Eden library [5] that provides parallel implementations of the map and map-reduce skeletons in the following forms:

$$\begin{array}{lll} parMap & :: & (Trans\ a, Trans\ b)\ \Rightarrow (a \to b) \to [a] \to [b] \\ parMapRedr & :: & (Trans\ a, Trans\ b)\ \Rightarrow (b \to b \to b) \to b \to (a \to b) \to [a] \to b \\ parMapRedl & :: & (Trans\ a, Trans\ b)\ \Rightarrow (b \to b \to b) \to b \to (a \to b) \to [a] \to b \end{array}$$

The *parMap* skeleton implemented in Eden creates a separate process for each application of the map operation, i.e. as many processes as the list elements. The parallel map-reduce skeletons, *parMapRedr* and *parMapRedr*, are implemented using the *parMap* skeleton described above. The result of *parMap* is reduced sequentially using the conventional *foldr* and *foldl* functions, respectively.

Currently, the map-reduce skeletons in the Eden library are defined using the *foldr* and *foldl* functions that require a unit value for the reduction/fold operator to be provided as an input. However, it is evident from Property 5.2 that the skeletons that are potentially identified will always be applied on non-empty lists. Therefore, we augment the skeletons provided in Eden by adding the following parallel map-reduce skeletons that are defined using the *foldr1* and *foldl1* functions, which are defined for non-empty lists, thereby avoiding the need to obtain a unit value for the reduction operator.

$$\begin{array}{lll} parMapRedr1 & :: & (Trans\ a, Trans\ b)\ \Rightarrow (b \to b \to b) \to (a \to b) \to [a] \to b \\ parMapRedl1 & :: & (Trans\ a, Trans\ b)\ \Rightarrow (b \to b \to b) \to (a \to b) \to [a] \to b \end{array}$$

To execute the encoded program produced by our transformation in parallel, we replace the identified skeleton instances with suitable calls to the corresponding skeletons in the Eden library. For example, by replacing functions $mMul'_1$ and $mMul'_3$, which are instances of *map* and *mapReduce* skeletons respectively, with suitable calls to *parMap* and *parMapRedr1*, we obtain the transformed matrix multiplication program $mMul''$ shown in Example 5.1.

Example 5.1 (Encoded Matrix Multiplication Defined Using Skeletons)

```
mMul'' xss yss
where
mMul''_1 xss yss  =  mMul''_1 (encode_mMul1 xss yss) yss
mMul''_1 x̄ yss    =  parMap f x̄
                     where
                     f c1          =  []
                     f c2          =  []
                     f (c3 xs zs)  =  let v = λxs.g xs
                                              where
                                              g []        =  0
                                              g (x : xs)  =  x
                                         in mMul''_2 (encode_mMul2 zs) xs yss v
mMul''_2 (c4 : x̄) xs yss v  =  []
mMul''_2 (c5 : x̄) xs yss v  =  let v' = λxs.g xs
                                         where
                                         g []        =  0
                                         g (x : xs)  =  v xs
                               in (mMul''_3 (encode_mMul3 xs yss) v) : (mMul''_2 x̄ xs yss v')
mMul''_3 x̄ v  =  parMapRedr1 g f x̄
                 where
                 g x y         =  x+y
                 f c6          =  0
                 f c7          =  0
                 f (c8 x ys)   =  x∗(v ys)
```

Consequence: By automatically identifying instances of list-based map and map-reduce skeletons, we produce a program that is defined using these parallelisable skeletons. Using parallel implementations for these skeletons that are available in existing libraries such as Eden, it is possible to execute the transformed program on parallel hardware.

6 Conclusion

6.1 Summary

We have presented a transformation method to automatically identify parallel computations in a given program that can be encapsulated using parallelisable map or map-reduce skeletons defined over lists. This is achieved by encoding the inputs of each recursive function in the given program into a *cons*-list by using the algorithmic structure of the function. By transforming recursive functions to operate over lists, we facilitate the identification of skeleton instances, in particular map and map-reduce skeletons, that are defined over lists. By using the *distillation* transformation in conjunction with our method, we also reduce inefficient intermediate data structures that are present in the original program. As a result, we produce encoded programs that are potentially defined using calls to parallelisable map and map-reduce skeletons and contain fewer inefficient intermediate data structures, which is difficult to achieve for non-trivial problems. Importantly, the language on which we propose our transformation does not impose any restrictions on the program to be transformed or its inputs. Further, we use the parallel implementations of the skeletons provided in the Eden library for efficient execution of our transformed programs on parallel hardware. It is evident from our detailed evaluations that it is possible to identify instances of other versatile skeletons such as the *accumulate* skeleton [15], which allows parallelisation of computations that build results using accumulating parameters.

6.2 Related Work

Previously, following the seminal works by Cole [1] and Darlington et. al. [2] on skeleton-based program development, a majority of the work that followed [3, 9, 16, 8] catered to manual parallel programming. To address the difficulties in choosing appropriate skeletons for a given algorithm, Hu et. al. proposed the *diffusion* transformation [17], which is capable of decomposing recursive functions of a certain form into several functions, each of which can be described by a skeleton. Even though diffusion can transform a wider range of functions to the required form, this method is only applicable to functions with one recursive input. Further they proposed the accumulate skeleton [15] that encapsulates the computational forms of map and reduce skeletons that use an accumulating parameter to build the result. However, the associative property of the reduce and scan operators used in the accumulate skeleton have to be verified and their unit values derived manually.

The calculational approaches to program parallelisation are based on list-homomorphisms [18] and propose systematic ways to derive parallel programs. However, most methods are restricted to programs that are defined over lists [19, 20, 21, 22]. Further, they require manual derivation of operators or their verification for certain algebraic properties to enable parallel evaluation of the programs obtained. Morihata et. al. [23] extended this approach for trees by decomposing a binary tree into a list of sub-trees called *zipper*, and defining upward and downward computations on the zipper structure. However, such calculational methods are often limited by the range of programs and data types they can transform. Also, a common aspect of these calculational approaches is the need to manually derive operators that satisfy certain properties, such as associativity to guarantee parallel evaluation. To address this, Chin et. al. [24] proposed a method that systematically derives parallel programs from sequential definitions and automatically creates auxiliary functions that can be used to define associative operators needed for parallel evaluation. However, their method is restricted to a first-order language and applicable to functions defined over a single recursive linear data type, such as lists, that has an associative decomposition operator, such as $+\!\!+$.

As an alternative to calculational approaches, Ahn et. al. [25] proposed an analytical method to transform general recursive functions into a composition of polytypic data parallel skeletons. Even though their method is applicable to a wider range of problems and does not need associative operators, the transformed programs are defined by composing skeletons and employ multiple intermediate data structures.

Previously, the authors proposed a method to transform the input of a given program into a *cons*-list based on the recursive structure of the input [26]. Since this method does not use the recursive structure of the program to build the *cons*-list, the transformed programs do not lend themselves to be defined using list-based parallel skeletons. This observation led to creating a new encoded data type that matches the algorithmic structure of the program and hence enables identification of polytypic parallel map and reduce skeletons [27]. The new encoded data type is created by pattern-matching and recursively consuming inputs, where a recursive components is created in the new encoded input for each recursive call that occurs in a function body using the input arguments of the recursive call. Consequently, the data structure of the new encoded input reflects the recursive structure of the program. Even though this method leads to better identification of polytypic skeletons, it is not easy to evaluate the performance of the transformed programs defined using these skeletons because existing libraries do not offer implementations of skeletons that are defined over a generic data type. Consequently, the proposed method of encoding the inputs into a list respects the recursive structures of programs and allows evaluation of the transformed programs using existing implementations of list-based parallel skeletons.

Acknowledgment

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).

References

[1] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA, USA, 1991.

[2] Darlington, John and Field, A. J. and Harrison, Peter G. and Kelly, Paul and Sharp, D. W. N. and Wu, Qiang and While, R. Lyndon. *Parallel Programming Using Skeleton Functions.* Lecture Notes in Computer Science, 5th International PARLE Conference on Parallel Architectures and Languages Europe, 1993.

[3] K. Matsuzaki and H. Iwasaki and K. Emoto and Z. Hu, *A Library of Constructive Skeletons for Sequential Style of Parallel Programming*. Proceedings of the 1st ACM International Conference on Scalable Information Systems, InfoScale, 2006.

[4] González-Vélez, Horacio and Leyton, Mario *A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers.* Software – Practice and Experience, 2010.

[5] Loogen, Rita. *Eden Parallel Functional Programming with Haskell.* Lecture Notes in Computer Science, Central European Functional Programming School, Springer Berlin Heidelberg, 2012.

[6] Matsuzaki, Kiminori and Iwasaki, Hideya and Emoto, Kento and Hu, Zhenjiang. *A Library of Constructive Skeletons for Sequential Style of Parallel Programming.* Proceedings of the 1st International Conference on Scalable Information Systems, 2006.

[7] Chakravarty, Manuel M. T. and Leshchinskiy, Roman and Peyton Jones, Simon and Keller, Gabriele and Marlow, Simon. *Data Parallel Haskell: A Status Report.* Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP), 2007.

[8] Chakravarty, Manuel M. T. and Keller, Gabriele and Lee, Sean and McDonell, Trevor L. and Grover, Vinod. *Accelerating Haskell Array Codes with Multicore GPUs.* Proceedings of the Sixth ACM Workshop on Declarative Aspects of Multicore Programming, 2011.

[9] K. Matsuzaki and K. Kakehi and H. Iwasaki and Z. Hu and Y. Akashi, *A Fusion-Embedded Skeleton Library*. Euro-Par 2004 Parallel Processing, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004.

[10] McDonell, Trevor L. and Chakravarty, Manuel M.T. and Keller, Gabriele and Lippmeier, Ben. *Optimising Purely Functional GPU Programs.* ACM SIGPLAN Notices, 2013.

[11] D. B. Skillicorn and D. Talia, Domenico, *Models and Languages for Parallel Computation*. ACM Computing Surveys, June 1998.

[12] Hu, Zhenjiang and Takeichi, Masato and Chin, Wei-Ngan, *Parallelization in Calculational Forms*. Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1998.

[13] G. W. Hamilton and Neil D. Jones. *Distillation with Labelled Transition Systems.* Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, 2012.

[14] Pettorossi, Alberto and Proietti, Maurizio. *Rules and Strategies for Transforming Functional and Logic Programs.* ACM Computing Surveys, June 1996.

[15] Iwasaki, Hideya and Hu, Zhenjiang. *A New Parallel Skeleton for General Accumulative Computations.* International Journal of Parallel Programming, Kluwer Academic Publishers, 2004.

[16] K. Matsuzaki and Z. Hu and M. Takeichi. *Parallel Skeletons for Manipulating General Trees.* Parallel Computin, September 2006.

[17] Zhenjiang Hu and Masato Takeichi and Hideya Iwasaki. *Diffusion: Calculating Efficient Parallel Programs.* ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), 1999.

[18] D. B. Skillicorn *The Bird-Meertens Formalism as a Parallel Model.* Software for Parallel Computation, NATO ASI Series F, Springer-Verlag, 1993.

[19] Sergei Gorlatch. *Constructing List Homomorphisms for Parallelism.* Fakultät für Mathematik und Informatik: MIP, 1995.

[20] Jeremy Gibbons. *The Third Homomorphism Theorem.* Journal of Functional Programming Vol. 6, No. 3, 1996.

[21] Sergei Gorlatch. *Extracting and Implementing List Homomorphisms In Parallel Program Development.* Science of Computer Programming, 1999.

[22] Zhenjiang Hu and Tetsuo Yokoyama and Masato Takeichi. *Program Optimizations and Transformations an Calculation Form.* GTTSE. 2005.

[23] Akimasa Morihata and Kiminori Matsuzaki and Zhenjiang Hu and Masato Takeichi. *The Third Homomorphism Theorem on Trees: Downward & Upward Lead to Divide-and-Conquer.* POPL, 2009.

[24] Wei-Ngan Chin and Takano, A. and Zhenjiang Hu. *Parallelization via Context Preservation.* International Conference on Computer Languages, 1998.

[25] Joonseon Ahn and Taisook Han. *An Analytical Method for Parallelisation of Recursive Functions.* Parallel Processing Letters, 2001.

[26] Venkatesh Kannan and G. W. Hamilton. *Extracting Data Parallel Computations from Distilled Programs.* Fourth International Valentin Turchin Workshop on Metacomputation (META), 2014.

[27] Venkatesh Kannan and G. W. Hamilton. *Program Transformation To Identify Parallel Skeletons.* 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2016.