Transformation of Functional Programs for Identification of Parallel Skeletons

submitted by

Venkatesh Kannan, M.Sc., B.E.

A dissertation submitted in fulfilment of the requirements for the award of
Doctor of Philosophy (Ph.D.)
at the School of Computing, Dublin City University, Ireland.

supervised by

Dr. Geoffrey W. Hamilton

January 2017

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy, is entirely my own work, that I have exercised reasonable care to ensure that the work is original and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed _____ (Venkatesh Kannan)

I.D. Number _____

Date _____

Acknowledgements

I owe everything that I have learnt and experienced to my mother (Bama) and father (Kannan). Their belief drives me towards the best I can be. My brother's encouragement kept me going during times of self-doubt.

My deepest gratitude goes to my supervisor, Dr. Geoff Hamilton, who has been integral in my growth while working on this thesis. I have learnt from and continue to admire his patience, encouragement, cheerfulness and continued support over the last four years. His one-liner "In Ph.D., even too much is never enough." set me on the right path from day one. I am forever indebted to his mentorship.

I would like to thank the DCU School of Computing and Lero : The Irish Software Research Centre for providing an enjoyable working environment. Michael Dever was instrumental in my settling down at DCU and understanding a lot of related work.

I am grateful to all my friends who have been interested in and encouraged my progress throughout the four years of my thesis work.

Contents

List of Figures

Abstract

Transformation of Functional Programs for Identification of Parallel Skeletons

Venkatesh Kannan

Hardware is becoming increasingly parallel. Thus, it is essential to identify and exploit inherent parallelism in a given program to effectively utilise the computing power available. 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. One possible approach to parallel programming is to use transformation techniques to automatically identify and explicitly specify parallel computations in a given program using parallelisable algorithmic skeletons.

Current existing methods for systematic derivation of parallel programs or parallel skeleton identification allow automation. However, they place constraints on the programs to which they are applicable, require manual derivation of operators with specific properties for parallel execution, or allow the use of inefficient intermediate data structures in the parallel programs.

In this thesis, we present a program transformation method that addresses these issues and has the following attributes: (1) Reduces the number of inefficient data structures used in the parallel program; (2) Transforms a program into a form that is more suited to identifying parallel skeletons; (3) Automatically identifies skeletons that can be efficiently executed using their parallel implementations. Our transformation method does not place restrictions on the program to be parallelised, and allows automatic verification of skeleton operator properties to allow parallel execution.

To evaluate the performance of our transformation method, we use a set of benchmark programs. The parallel version of each program produced by our method is compared with other versions of the program, including parallel versions that are derived by hand. Consequently, we have been able to evaluate the strengths and weaknesses of the proposed transformation method. The results demonstrate improvements in the efficiency of parallel programs produced in some examples, and also highlight the role of some intermediate data structures required for parallelisation in other examples.

Chapter 1

Introduction

1.1 Motivation

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 available. To enable this, the parallelism inherent in a given program needs to be identified and exploited. For a program that contains potential parallel computations, this can be accomplished through *implicit* parallel programming, *explicit* parallel programming or a combination of both [41].

In implicit parallel programming, the developer is tasked with simply implementing a program for a given algorithm. The responsibility of identifying and executing parallel computations efficiently is handled under-the-hood by the compiler and/or the run-time system. Some programming languages that support this approach are HPF (High Performance Fortran), NESL [8] and SAC (Single Assignment C). The key aspect of this approach is that the programmer does not explicitly specify any information about parallelism that may be present. As a result, the developer has no visibility and control of a program's parallelisation and its execution. Furthermore, it is very difficult for a compiler to automatically parallelise a program to a sufficient and desired level.

In explicit parallel programming, the developer has to analyse and identify inherent parallelism in a given algorithm. This is then explicitly specified during program development. As a result, the developer has complete and fine-grained control over the parallel execution of the program on hardware. Some programming languages that support this approach are Erlang, Ada and Java. In this approach, it can be tedious for

the developer to analyse, identify and efficiently specify all aspects that are associated with the efficient parallelisation of a program on a given hardware. Additionally, this increases the potential for run-time errors such as deadlocks while making debugging much more difficult.

In the spectrum that lies between implicit and explicit parallelisation approaches, there exist a number of methods, such as evaluation strategies [84], that allow a developer to use some form of annotation to specify the computations in a program that can be executed in parallel. The primary benefit of this approach is that, while it allows the developer a certain degree of flexibility to specify how a given program can be executed in parallel, it abstracts away from fine-grained details related to the target hardware architecture and run-time system, which may be too tedious for the developer to deal with. Glasgow Parallel Haskell (GpH) [84] is one such approach that allows speculative parallelisation of expressions, which are annotated for parallelisation, by the runtime system.

Each of these approaches offers different advantages and degrees of control to a developer. However, parallel programming is tedious and error-prone when done by hand (explicit parallel programming), and is very difficult for a compiler to do automatically to the desired level (implicit parallel programming). Therefore, a desirable approach to parallel programming is to transform programs into a form which makes it easier to identify parallelism. This can lead to automatic identification of parallel computations in a given program. The resulting transformed program can then be executed using efficient implementations for the parallel computations on given parallel hardware.

To design such a technique, which can transform a given sequential program into an equivalent parallel version, the following challenges need to be addressed.

1. A given program may be defined in a form that does not obviously exhibit potential parallelism.

2. The program may be defined over an arbitrary number of inputs, each of which may be of any data type.

Consequently, manual analysis of a program to identify features that allow parallel execution is often not straightforward for many problems.

1.2 Research Hypothesis and Research Questions

Based on this background and a detailed study of related techniques presented in Chapter 2, the research hypothesis of this thesis is that

"*Program transformation can be used to automatically identify parallel computations in a given program, potentially leading to its efficient parallel execution*".

To evaluate this hypothesis, the following research questions have been identified and need to be answered.

1. *How can potential parallel computations in a program be automatically identified?*
   As described earlier, the task of identifying potential parallel computations in a given program is tedious. This can be eased using a technique to automatically analyse, identify and rewrite potential parallel computations in the program into a form that allows their parallel execution.

2. *How can the transformed program be efficiently executed in parallel?*
   Given a transformed program produced by the technique proposed in this thesis, it is essential to efficiently execute it on a given parallel hardware. For this, efficient implementations for the parallel computations that are identified are required. Also, it is important to evaluate the performance of the parallel program produced against the sequential and hand-parallel version(s).

3. *How can a given program be transformed to aid identification of parallel computations?*
   The objective of the techniques presented in this thesis is to apply them to a given program that may be defined over any number of inputs of any data type without restrictions. It is challenging to design such a method to address the vast range of data structures and algorithmic forms. Therefore, it is essential to transform the given program to be more suited for automatic identification of parallel computations.

The remainder of this chapter presents an introduction to the language used in this thesis (Section 1.3), an overview of the proposed solution to evaluate the research hypothesis and answer the research questions (Section 1.4), the major contributions and outcomes from this work (Section 1.5), and the structure of the thesis (Section 1.6).

1.3 Functional Programming

The work presented in this thesis uses functional programming, which is based on λ–calculus [20, 21], where computations correspond to the evaluation of mathematical functions. Since pure functional programs are free from side-effects, the result of evaluating a pure function depends only on its input arguments. Functional programs are therefore relatively easy to analyse, reason about and manipulate using program transformation techniques (discussed in Section 2.1). The higher-order functional language used in this work is shown in Definition 1.1.

Definition 1.1 (Language Grammar):

$$
\begin{array}{rll}
e & ::= & x \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Variable} \\
& | & c\ e_1 \ldots e_N \qquad\qquad\qquad\qquad\quad \text{Constructor Application} \\
& | & e_0 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{where}\text{-expression} \\
& & \textbf{where} \\
& & d_1\ \ldots\ d_J \\
& | & f \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{Function Call} \\
& | & e_0\ e_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{Application} \\
& | & \textbf{let}\ x = e_1\ \textbf{in}\ e_0 \qquad\qquad\qquad\qquad\quad \textbf{let}\text{–expression} \\
& | & \lambda x.e \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \lambda\text{–abstraction} \\
d & ::= & f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = e_1 \qquad\qquad\quad \text{Function Definition} \\
& & \vdots \\
& & f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = e_K \\
p & ::= & x \mid c\ p_1 \ldots p_N \qquad\qquad\qquad\qquad\qquad\quad \text{Pattern}
\end{array}
$$

A program in this language is an expression which can be a variable, constructor application, **where**-expression, 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*. 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 exhaustive.

We use [] and (:) as shorthand notations for the *Nil* and *Cons* constructors of a *cons*-list and $+\!\!+$ for list concatenation. The expression $(e_1, \ldots, e_N)$ denotes a tuple of N elements. Function composition is denoted using $\circ$. The set of free and bound variables in an expression e are denoted as $fv(e)$ and $bv(e)$, respectively. We use

$$\textbf{let}\ x_1 = e_1 \ldots x_N = e_N\ \textbf{in}\ e_0$$

as a shorthand notation for the nested **let**-expression

$$\begin{array}{l}\textbf{let } x_1 = e_1 \\ \quad \ldots \\ \quad \textbf{in let } x_N = e_N \\ \quad\quad \textbf{in } e_0\end{array}$$

A program can also contain data type declarations of the form shown in Definition 1.2. 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 $e :: T$.

Definition 1.2 (Data Type Declaration):

$d ::= \textbf{data}\ T\ \alpha_1 \ldots \alpha_M\ =\ c_1\ t_1^1 \ldots t_N^1 \mid \ldots \mid c_K\ t_1^K \ldots t_N^K$ Data Type Declaration

$t ::= \alpha_m \mid T\ t_1 \ldots t_M$ Type Component

Definition 1.3 (Expression 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 $e_1, \ldots, e_N$, respectively.

Definition 1.4 (Expression Substitution):

$e\{x_1 \mapsto e_1,\ \ldots,\ x_N \mapsto e_N\}$ denotes the simultaneous substitution of the expressions $e_1, \ldots, e_N$ for the variables $x_1, \ldots, x_N$ in expression e.

Definition 1.5 (Expression Instance):

Expression e is an instance of expression e' if $\exists\theta \cdot e \equiv e'\theta$ where θ is an expression substitution, where $\equiv$ is α-equivalence.

The call-by-name operational semantics of our language is defined using an evaluation relation as shown in Definition 1.6.

Definition 1.6 (Evaluation Relation):

$e \overset{r}{\rightsquigarrow}$, iff $\exists e'.e \overset{r}{\rightsquigarrow} e'$ $\qquad$ $e \Downarrow$, iff $\exists v.e \Downarrow v$

$e \Downarrow v$, iff $e \overset{r^*}{\rightsquigarrow} v \wedge \neg(v \overset{r}{\rightsquigarrow})$ $\qquad$ $e \Uparrow$, iff $\forall e'.e \overset{r^*}{\rightsquigarrow} e' \Rightarrow e' \overset{r}{\rightsquigarrow}$

Here, $\Downarrow$ is an evaluation relation between closed expressions (those that can be evaluated in a finite number of steps) 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 1.7. The reduction can be f (unfolding of function f) or β (β-substitution). The transitive closure of the reduction relation is denoted by $\overset{r^*}{\rightsquigarrow}$.

Definition 1.7 (One-Step Reduction Relation):

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

$$\frac{\big(f\ p_1 \ldots p_N = e\big) \wedge \big(f\ e_1 \ldots e_N = (f\ p_1 \ldots p_N)\theta\big)}{(f\ e_1 \ldots e_N) \overset{f}{\rightsquigarrow} e\theta}$$

$$\big(\textbf{let}\ x = e_1\ \textbf{in}\ e_0\big) \overset{\beta}{\rightsquigarrow} \big(e_0\{x \mapsto e_1\}\big)$$

1.3.1 Intermediate Data Structures

A common source of inefficiency in functional programming is the use of intermediate data structures. Functional programs are usually written using multiple functions, each of which may compute an intermediate result that is fed to another function to perform further computations according to the algorithm being implemented [47]. This style of programming employs multiple intermediate data structures, which are the intermediate results constructed by one function and passed to another where they are decomposed. For example, consider the matrix multiplication program shown in Definition 1.1.

Example 1.1 (Matrix Multiplication):

$mMul :: [[a]] \to [[a]] \to [[a]]$
$mMul\ xss\ yss$
where

$$\begin{aligned}
mMul\ []\ yss &= [] \\
mMul\ (xs : xss)\ yss &= (map\ (dotp\ xs)\ (transpose\ yss)) : (mMul\ xss\ yss) \\
dotp\ xs\ ys &= foldr\ (+)\ 0\ (zipWith\ (*)\ xs\ ys) \\
transpose\ yss &= transpose'\ yss\ [] \\
transpose'\ []\ yss &= yss \\
transpose'\ (xs : xss)\ yss &= transpose'\ xss\ (rotate\ xs\ yss) \\
rotate\ []\ yss &= yss \\
rotate\ (x : xs)\ [] &= [x] : (rotate\ xs\ yss) \\
rotate\ (x : xs)\ (ys : yss) &= (ys +\!\!+ [x]) : (rotate\ xs\ yss)
\end{aligned}$$

Here, $mMul$ computes the product of two matrices xss and yss. The built-in function map is used to compute the dot-product ($dotp$) of each row in matrix xss and those in the $transpose$ of matrix yss using built-in functions $foldr$ and $zipWith$. Here, in the definition of $mMul$, function $transpose$ constructs a list that is subsequently deconstructed and consumed by map, which is an example of an inefficient intermediate data structure.

A number of program transformation methods to remove inefficiencies exist in literature [14, 87, 85, 38, 45], and the ones that are most relevant to this thesis are discussed in detail in Section 2.1.

1.3.2 Algorithmic Skeletons

Based upon the convention of developing a program using multiple functions, parallel functional programming is often done by hand using *skeletons* [22, 34]. Skeletons are algorithmic forms of computations that occur in a wide range of problems. Parallel programming also has constructs that encapsulate well-known and commonly occurring computations that can be evaluated in parallel. These constructs are abstracted and defined as higher-order functions called *parallel skeletons* such as *map*, *reduce*, *scan*, *zipWith* and *divide-and-conquer*. Such parallel skeletons, which can have efficient parallel implementations under-the-hood, are often used by developers as building blocks to create parallel programs. A detailed discussion of parallel skeletons and existing libraries that offer their implementations is presented in Section 2.2.3. A parallel version of the matrix multiplication program in Example 1.1 is presented in Example 1.2.

Example 1.2 (Parallel Matrix Multiplication Using Skeletons):

$mMul\ xss\ yss$
where

$$
\begin{array}{lll}
mMul\ xss\ yss & = & farmB\ noPe\ f\ xss \\
 & & \textbf{where} \\
 & & f\ xs\ =\ farmB\ noPe\ (dotp\ xs)\ (transpose\ yss) \\
dotp\ xs\ ys & = & parRedr\ (\lambda xs.(length\ xs) < noPe)\ (+)\ 0\ (zipWith\ xs\ ys) \\
transpose\ yss & = & transpose'\ yss\ [] \\
transpose'\ []\ yss & = & yss \\
transpose'\ (xs : xss)\ yss & = & transpose'\ xss\ (rotate\ xs\ yss) \\
rotate\ []\ yss & = & yss \\
rotate\ (x : xs)\ [] & = & [x] : (rotate\ xs\ yss) \\
rotate\ (x : xs)\ (ys : yss) & = & (ys +\!\!+ [x]) : (rotate\ xs\ yss)
\end{array}
$$

Here, we observe that even though defined using *map*, *reduce* and *zipWith* skeletons that can be parallelised, the program still uses the intermediate data structures that were originally present in the program. There exist techniques, such as skeleton fusion [65], that eliminate intermediate data structures between neighbouring skeletons. However, powerful existing program transformation techniques such as supercompilation [85, 80, 81] and distillation [38] have not been used in such parallelisation approaches. In this thesis, we aim to study the parallelisation of programs on which the existing distillation

technique (discussed in detail in Chapter 3) is applied to eliminate intermediate data structures.

1.4 Proposed Solution

The solution proposed in this thesis to parallelise programs produced by the existing distillation transformation, which reduces intermediate data structures in a program, uses two components that are proposed in this thesis – *data type transformation* and *skeleton identification* – in conjunction with *distillation* as illustrated in Figure 1.1.

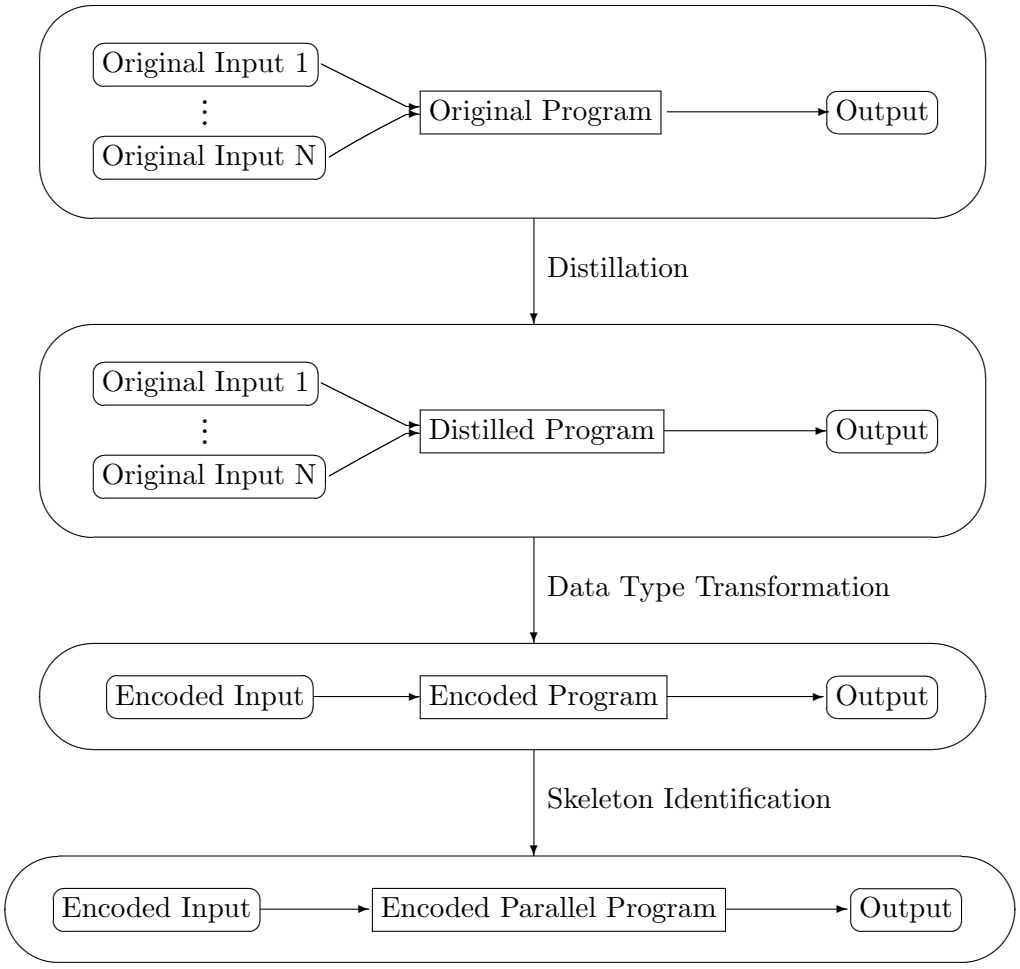


Figure 1.1: Proposed Solution

Here, the *distillation* transformation is first used to reduce the number of intermediate data structures in a given *original program* resulting in a *distilled program*. We perform distillation first since the resultant program is defined in a specific form (referred

to as *distilled form*) on which it is easier to apply the subsequent data type transformation as explained in Chapter 5. It would otherwise be more tedious to design the data type transformation on an arbitrary program if distillation were not performed first. Thus, following distillation, the *data type transformation* is used to combine all inputs of the distilled program into a single input. This yields an *encoded program* that is defined over a single input. Finally, in *skeleton identification*, the encoded program is analysed to identify parallel computations that may be present. This step is guided using parallel skeletons, which are algorithmic forms of parallel computations that we wish to identify automatically. It is then possible to execute the resulting *encoded parallel program* on parallel hardware by making use of efficient implementations of the parallel skeletons that are identified.

The objective of this approach is to allow a developer to write sequential programs, which can then be automatically analysed for potential parallelism. Consequently, the proposed solution has the following attributes:

1. Using the distillation transformation, reduces the number of inefficient data structures that are commonly used in functional programs.

2. Transforms a given program into a form that is more suited to identifying parallel computations by encoding all inputs into a single input whose type matches the algorithmic structure of the program.

3. Automatically identifies parallel computations in the encoded program as algorithmic skeletons that can be efficiently executed using their parallel implementations.

Based on the first step of the proposed solution, Example 1.3 presents the distilled form of the matrix multiplication program in Example 1.1. Here, 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.

Example 1.3 (Distilled Matrix Multiplication):

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

For the second step of the proposed solution, the result of applying the data type transformation to the distilled matrix multiplication program is presented in Example 1.4. Here, T'_{mMul_1}, T'_{mMul_2} and T'_{mMul_3} are new data types created by combining the inputs of functions $mMul_1$, $mMul_2$ and $mMul_3$, respectively, using the data type transformation proposed in this thesis. These functions are also transformed into $mMul'_1$, $mMul'_2$ and $mMul'_3$ to operate over the transformed inputs that are computed using the functions $encode_{mMul_1}$, $encode_{mMul_2}$ and $encode_{mMul_3}$, respectively, which are also defined using the data type transformation.

Example 1.4 (Transformed Matrix Multiplication Program):

$$
\begin{array}{l}
\textbf{data}\ T'_{mMul_1}\ a = c_1 \mid c_2 \mid c_3\ [a]\ [a] \\
\textbf{data}\ T'_{mMul_2}\ a = c_4 \mid c_5 \\
\textbf{data}\ T'_{mMul_3}\ a = c_6 \mid c_7 \mid c_8\ a\ [a] \\
encode_{mMul_1}\ []\ zss = [c_1] \\
encode_{mMul_1}\ xss\ [] = [c_2] \\
encode_{mMul_1}\ (xs : xss)\ (zs : zss) = [c_3\ xs\ zs] +\!\!+ (encode_{mMul_1}\ xss\ zss) \\
encode_{mMul_2}\ [] = [c_4] \\
encode_{mMul_2}\ (z : zs) = [c_5] +\!\!+ (encode_{mMul_2}\ xs\ yss\ zs) \\
encode_{mMul_3}\ []\ yss = [c_6] \\
encode_{mMul_3}\ (x : xs)\ [] = [c_7] \\
encode_{mMul_3}\ (x : xs)\ (ys : yss) = [c_8\ x\ ys] +\!\!+ (encode_{mMul_3}\ xs\ yss)
\end{array}
$$

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

For the third step of the proposed solution, by identifying parallel skeletons, this transformed matrix multiplication program is then defined using parallel skeletons as shown in Example 1.5. Here, the function $mMul'_1$ and $mMul'_2$ from Example 1.4 are identified as instances of the parallelisable *map* and *mapReduce1* skeletons skeletons, respectively.

Example 1.5 (Parallel Matrix Multiplication Program):

$$
\begin{array}{l}
mMul\ xss\ yss \\
\textbf{where} \\
mMul\ xss\ yss\ =\ mMul''_1\ (encode_{mMul_1}\ xss\ yss)\ yss \\
\begin{array}{lcl}
mMul''_1\ w\ yss & = & map\ f\ w \\
 & & \textbf{where} \\
 & & \begin{array}{lcl}
 f\ c_1 & = & [] \\
 f\ c_2 & = & [] \\
 f\ (c_3\ xs\ zs) & = & \textbf{let}\ v = \lambda xs.g\ xs \\
 & & \qquad\qquad \textbf{where} \\
 & & \qquad\qquad g\ [] \quad\quad\ = 0 \\
 & & \qquad\qquad g\ (x : xs) = x \\
 & & \textbf{in}\ mMul''_2\ zs\ xs\ yss\ v
 \end{array}
\end{array} \\
\begin{array}{lcl}
mMul''_2\ []\ xs\ yss\ v & = & [] \\
mMul''_2\ (z : zs)\ xs\ yss\ v & = & \textbf{let}\ v' = \lambda xs.g\ xs \\
 & & \qquad\qquad \textbf{where} \\
 & & \qquad\qquad g\ [] \quad\quad\ = 0 \\
 & & \qquad\qquad g\ (x : xs) = v\ xs \\
 & & \textbf{in}\ (mMul''_3\ (encode_{mMul_3}\ xs\ yss)\ v) : (mMul''_2\ zs\ xs\ yss\ v')
\end{array}
\end{array}
$$

$$
\begin{array}{lll}
mMul''_3\ w\ v & = & mapReduce1\ g\ f\ w \\
& & \textbf{where} \\
& & g \qquad\quad = (+) \\
& & f\ c_6 \quad = 0 \\
& & f\ c_7 \quad = 0 \\
& & f\ (c_8\ x\ ys) = x * (v\ ys)
\end{array}
$$

1.5 Contributions

The major contributions of the proposed parallelisation method are as follows:

1. The parallelisation transformation can be fully automated without restrictions on the programs that can be transformed.

2. The parallel programs produced contain fewer intermediate data structures.

3. We gain insight into the role of intermediate data structures in program parallelisation from evaluating the benchmark programs.

4. The input data types are transformed into a structure that matches the algorithmic structure of the program.

Though the proposed parallelisation method can be fully automated, a complete implementation is not available yet. Also, the work presented in this thesis has resulted in the following publications:

1. Venkatesh Kannan and G.W. Hamilton, "Extracting Data Parallel Computations from Distilled Programs", *4th International Valentin Turchin Workshop on Metacomputation (META)*, 2014.

2. Venkatesh Kannan and G.W. Hamilton, "Program Transformation To Identify Parallel Skeletons", *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), revised version under review in International Journal of Parallel Programming (IJPP)*, 2016.

3. Venkatesh Kannan and G.W. Hamilton, "Program Transformation To Identify List-Based Parallel Skeletons", *4th International Workshop on Verification and Program Transformation (VPT), revised version in Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 2016.

4. Venkatesh Kannan and G.W. Hamilton, "Functional Program Transformation for Parallelisation using Skeletons", *9th International Symposium on High-Level Parallel Programming and Applications (HLPP), revised version in International Journal of Parallel Programming (IJPP)*, 2016.

5. Venkatesh Kannan and G.W. Hamilton, "Distilling New Data Types", *5th International Valentin Turchin Workshop on Metacomputation (META)*, 2016.

1.6 Structure of Thesis

The rest of this thesis is structured as follows:

Chapter 2 presents the relevant literature by outlining existing work in the areas of program transformation, development of parallel programs using program transformation, techniques and libraries that support parallel programs using algorithmic skeletons, and the use of tree contraction methods to efficiently process tree data structures in parallel.

Chapter 3 describes in detail *labelled transitions systems* (LTS) and how the existing *distillation* transformation is specified using the LTS framework. We use distillation in our parallelisation transformation to reduce the use of intermediate data structures. Chapter 4 presents our technique to identify potential parallel computations in a given program. We also present the parallel skeletons that we employ to define the identified parallel computations and existing libraries that allow their efficient execution on parallel hardware. Chapter 5 presents our data type transformation technique that allows transformation of a given program defined on inputs of any data type into a form that is more suitable for identifying skeletons using the method presented in Chapter 4.

Chapter 6 introduces the set of benchmark programs used to evaluate our transformation method and presents the results of applying our method to the programs. It also includes a detailed analysis of the parallel benchmark programs produced by our transformation method. Chapter 7 presents a summary of the results obtained from this research with ideas about possible future work to enhance the presented techniques and fill the gaps that remain.

Chapter 2

Related Work

This chapter presents an overview of existing program transformation techniques that are well-known for eliminating intermediate data structures (Section 2.1) and for systematically deriving parallel programs (Sections 2.2.1 and 2.2.2). Also discussed are some of the existing embedded languages and libraries that are used for parallel programming using skeletons (Section 2.2.3) and tree contraction algorithms that have been developed for efficient evaluation of tree data structures (Section 2.2.4). In Section 2.3, we discuss the use of refactoring techniques to derive parallel programs from their sequential definitions.

2.1 Transformation of Functional Programs

The process of systematically analysing and manipulating a given program is called program transformation. The basic idea is to take a given initial program P_0 and transform it to a final program P_N such that $Sem(P_0) = Sem(P_N)$ for a given semantic function Sem, such as the operational semantics defined in Section 1.3. This may be done by constructing a sequence of programs $\langle P_0, \ldots, P_N \rangle$, such that $Sem(P_n) = Sem(P_{n+1})$ for $0 \leq n < N$. This ensures that the semantic equivalence between the initial program and the transformed programs is maintained.

Program transformation is commonly used to improve programs by removing inefficiencies such as intermediate data structures. Based on our study of program transformation techniques, they can be broadly classified into two classes: techniques based on unfold/fold rules and those based on program calculation.

2.1.1 Unfold/Fold-Based Program Transformation

The unfold/fold-based transformation techniques make use of the following rules first proposed by Burstall & Darlington [14] as a method for developing recursive programs:

- *Definition Rule*: Introduce a new function definition in the program as follows, where f is the function name and $e_1, \ldots, e_K$ are the function bodies.

$$f\ p_1^1 \ldots p_N^1 = e_1\ \ldots\ f\ p_1^K \ldots p_N^K = e_K$$

- *Unfold Rule*: Given a function definition $f\ p_1^1 \ldots p_N^1 = e_1\ \ldots\ f\ p_1^K \ldots p_N^K = e_K$, replace all calls to function f with the corresponding instance of the function body. For example, in the definition of $mMul$ from Example 1.1, the function call $(transpose\ yss)$ in the expression

$$(map\ (dotp\ xs)\ (transpose\ yss)) : (mMul\ xss\ yss)$$

  can be unfolded using the function definition $transpose\ yss = transpose'\ yss\ []$ as follows:

$$\big(map\ (dotp\ xs)\ (transpose'\ yss\ [])\big) : (mMul\ xss'\ yss)$$

- *Fold Rule*: Given a function definition $f\ p_1^1 \ldots p_N^1 = e_1\ \ldots\ f\ p_1^K \ldots p_N^K = e_K$ and an expression e' such that e' is an instance of e_k, replace e' with a corresponding call to function f.
  For example, given the function definition $transpose\ yss = transpose'\ yss\ []$, the expression

$$\big(map\ (dotp\ xs)\ (transpose'\ yss\ [])\big) : (mMul\ xss'\ yss)$$

  can be rewritten as shown below by folding w.r.t. function $transpose$.

$$\big(map\ (dotp\ xs)\ (transpose\ yss)\big) : (mMul\ xss'\ yss)$$

- **let**-*abstraction*: Given an expression e with a sub-expression e', abstract e' by introducing a **let** variable x and rewrite expression e as **let** $x = e'$ **in** $e[x/e']$. Here, $e[x/e']$ denotes the expression e with sub-expression e' replaced by variable x.

Using these rules, Pettorossi et al. [74] proposed a transformation system that includes the following transformation strategies:

- *Composition*: If the sub-expression $f\ (g\ x)$ occurs in an expression e, then
  - introduce a new function h such that $h\ x = f\ (g\ x)$
  - obtain a definition for h without calls to f and g.
  - using the folding rule, replace all occurrences of $f\ (g\ x)$ by $h\ x$.

  The purpose of the composition strategy is to avoid intermediate data structures. In the above example, let g generate an intermediate data structure as input for f. If f computes a result incrementally using each output generated by g, then f can avoid waiting for the entire intermediate data structure to be generated by g. By composing f and g, the expression can be rewritten so that f is able to generate the 'next item' by using the finite intermediate output of g.

  An example of using the composition strategy is presented in Example 2.1.

Example 2.1 (Composition):

$sum\ (evens\ xs)$
where
$sum\ [] = 0$
$sum\ (x : xs) = x + (sum\ xs)$
$evens\ [] = []$
$evens\ (x : xs) = h\ (isEven\ x)$
where
$h\ True = x : (evens\ xs)$
$h\ False = evens\ xs$

$sumEvens\ xs$
where
$sumEvens\ [] = 0$
$sumEvens\ (x : xs) = h\ (isEven\ x)$
where
$h\ True = x + (sumEvens\ xs)$
$h\ False = sumEvens\ xs$

Here, the function *evens* filters all even elements in list xs and the resulting list is then used by function *sum* to add the values of all even elements. Using the composition strategy, the intermediate data structure between *evens* and *sum* (a list of elements with even values) can be eliminated by deriving the function *sumEvens*.

- *Tupling*: Given an expression $E[f_1\ e_1,\ \ldots,\ f_M\ e_M]$ where E is an expression context and x is a variable shared by $e_1, \ldots, e_M$,

  - introduce a function $g\ x\ y_1 \ldots y_J = (f_1\ e_1,\ \ldots,\ f_M\ e_M)$, where $x, y_1, \ldots, y_J$ are the free variables in $e_1, \ldots, e_M$.
  - obtain a definition for g without calls to $f_1, \ldots, f_M$.
  - using the **let**-abstraction and folding rules, fold w.r.t. function g and replace the expression $E[f_1\ e_1,\ \ldots,\ f_M\ e_M]$ by

$$\textbf{let}\ (u_1, \ldots, u_M) = g\ x\ y_1 \ldots y_M\ \textbf{in}\ E[u_1, \ldots, u_M]$$

  The purpose of the tupling strategy is to avoid multiple accesses to the same data structure and repetitive sub-expression evaluation. In the above example, $e_1, \ldots e_M$ all access the same data structure x. Tupling them together as function g will combine the accesses to x and the memory allocated to x can be released after the evaluation of g. Also, if the expressions $e_1, \ldots, e_M$ require evaluation of a common sub-expression, then tupling these expressions as function g will avoid the repeated evaluation of the common sub-expression for each expression $e_1, \ldots, e_M$.

  An example of using the tupling strategy is presented in Example 2.2.

  Example 2.2 (Tupling):
  $average\ xs = (sum\ xs)/(length\ xs)$
  where
  $sum\ \ = \ foldr\ (+)\ 0$
  $length\ = \ foldr\ (\lambda x.\lambda y.1 + y)\ 0$

  $average'\ xs = \textbf{let}\ (s, l) = sl\ x\ \textbf{in}\ s/l$
  where
  $sl = foldr\ \big(\lambda x.\lambda(s, l).(x + s,\ 1 + l)\big)\ (0, 0)$

  Here, the function to compute the *average* of all elements in a list xs is defined using functions *sum* and *length*. Both *sum* and *length* traverse the list independently using the $foldr$ function, which is inefficient. Using the tupling strategy, we can obtain a definition $average'$ where the values of computations performed by *sum* and *length* are tupled in each iteration, resulting in a single traversal of the list.

- *Generalisation*: Given a recursive function definition $f\ p_1 \ldots p_M\ x_{(M+1)} \ldots x_N = e$ where e' is a sub-expression in e,

- introduce a generalised expression $g\ x\ y_1 \ldots y_J = e[x/e']$, where e' is replaced by variable x in e and $x, y_1, \ldots, y_J$ are the free variables in $e[x/e']$.
- using the unfolding rule w.r.t. f and the folding rule w.r.t. g, obtain a recursive definition for function g without calls to f.
- replace all instances of $e[x/e']$ by corresponding calls to g.

The generalisation strategy is helpful when the folding rule cannot be applied due to mismatch of the arguments of two or more instances of a function call in an expression. Generalisation can replace an expression in its context with a semantically equivalent expression. This creates expressions to which unfolding and folding rules, in conjunction with other transformation strategies, can be applied.

An example of using the generalisation strategy is presented in Example 2.3.

Example 2.3 (Generalisation of Expression to Variable):

$$
\begin{array}{l}
\mathit{height}\ \mathit{xt} \\
\textbf{where} \\
\mathit{height}\ (\mathit{Leaf}\ x) \quad\quad\quad\quad\quad\quad = 0 \\
\mathit{height}\ (\mathit{Branch}\ \mathit{xt}_1\ \mathit{xt}_2) \quad = 1 + ((\mathit{height}\ \mathit{xt}_1) \uparrow (\mathit{height}\ \mathit{xt}_2))
\end{array}
$$

$$
\begin{array}{l}
\mathit{height}'\ \mathit{xt} = \mathit{height}''\ \mathit{xt}\ 0\ 0 \\
\textbf{where} \\
\mathit{height}''\ (\mathit{Leaf}\ x)\ n\ m \quad\quad\quad\quad = n \uparrow m \\
\mathit{height}''\ (\mathit{Branch}\ \mathit{xt}_1\ \mathit{xt}_2)\ n\ m \ = \ \textbf{let}\ m' = \mathit{height}''\ \mathit{xt}_2\ (n+1)\ m \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{height}''\ \mathit{xt}_1\ (n+1)\ m'
\end{array}
$$

Here, the function *height* computes the height of a binary tree where the binary operator $\uparrow$ returns the larger of its two inputs. This definition is memory intensive due to possible heavy use of the stack for the recursive calls in the expression $1 + ((\mathit{height}\ \mathit{xt}_1) \uparrow (\mathit{height}\ \mathit{xt}_2))$. Using the generalisation strategy, we can obtain a tail-recursive definition height', also shown in Example 2.3, that utilises memory more efficiently. Here, the result of the recursive call to compute the height of branch xt_2 is extracted by generalisation to a variable.

The result of generalisation is also similar to λ-lifting [49] where sub-expressions that are abstracted and lifted to the top-level are bound to variables such as in a **let**-expression.

Deforestation

Primarily based on the composition strategy, Wadler [87] proposed *deforestation*, a transformation that automatically eliminates intermediate data structures from first-order programs. Deforestation requires that the initial program contains definitions that are *linear* and expressions are built with *treeless* functions. An expression is linear if no variable in it occurs more than once. For example, w.r.t. a function *append* that appends two given lists, the function application $append\ xs\ ys$ is linear, while $append\ xs\ xs$ is not. Given a set of functions $\overline{f}$, an expression is treeless w.r.t. $\overline{f}$ if it is linear, contains calls to functions only in $\overline{f}$, and all arguments in function applications are variables. Given such a program, the deforestation technique produces an expression that is also treeless, i.e. does not create intermediate data structures.

For example, consider the program shown below to append three lists using the function *append* that is defined to append two lists.

$$
\begin{array}{l}
append\ (append\ xs\ ys)\ zs \\
\textbf{where} \\
append\ []\ ys \quad\quad = ys \\
append\ (x : xs)\ ys\ = x : (append\ xs\ ys)
\end{array}
$$

This definition uses the intermediate data structure produced by $append\ xs\ ys$. Deforestation of this program produces a version that is free of this intermediate data structure as shown below, where $append_3$ is a function defined to concatenate three lists and $append_2$ is a function defined to concatenate two lists.

$$
\begin{array}{l}
append_3\ xs\ ys\ zs \\
\textbf{where} \\
append_3\ []\ ys\ zs \quad\quad = append_2\ ys\ zs \\
append_3\ (x : xs)\ ys\ zs\ = x : (append_3\ xs\ ys\ zs) \\
append_2\ []\ ys \quad\quad = ys \\
append_2\ (x : xs)\ ys \quad = x : (append_2\ xs\ ys)
\end{array}
$$

Even though the deforestation transformation is powerful, it is restrictive as it requires that the functions in the input program be treeless. However, this restriction can be alleviated [17, 36, 63] as all functions can be made pseudo-treeless prior to applying the deforestation transformation.

Supercompilation

Turchin introduced the *supercompiler* transformation technique [85] that eliminates intermediate data structures by maintaining and propagating both positive and negative

information during transformation using the techniques *driving* (forced unfolding guided by functional configurations) and folding. Positive information can be regarded, for example, as observations that a variable matches a pattern, and negative information can be regarded as the observation that a variable does not match a pattern.

Based on Turchin's supercompiler, Sørensen et al. [80, 81] introduced *positive supercompilation*. A simplified form of Turchin's supercompiler, the positive supercompiler [80], propagates positive information about pattern-matched variables into the function body expressions, and uses driving to build a potentially infinite tree of states and transitions for a given program. These trees are made finite by a combination of folding, i.e. directing states to their ancestors (if the expression is an instance of the ancestor), and generalising expressions that contain an embedding of previously encountered expressions, potentially leading to foldings.

For example, consider the following program to find a *match* for a pattern *pp* in a given string *ss* where the equality check operator == is considered to be built-in.

$$
\begin{array}{lll}
match\ pp\ ss & = & loop\ pp\ ss\ pp\ ss \\
loop\ []\ ss\ op\ os & = & True \\
loop\ (p:ps)\ []\ op\ os & = & False \\
loop\ (p:ps)\ (s:ss)\ op\ os & = & h\ (p == s) \\
 & & \textbf{where} \\
 & & h\ True\ = loop\ pp\ ss\ op\ os \\
 & & h\ False\ = next\ os\ op \\
next\ []\ op & = & False \\
next\ (o:os)\ op & = & loop\ op\ os\ op\ os
\end{array}
$$

This definition of *match* is inefficient because the result of non-matching characters in the given string is ignored in subsequent iterations to find a match in the rest of the string. This results in repeated matching tests with non-matching characters that can be inferred from the previous iterations. The result of positive supercompilation of the $match_{AAB}$ program is shown below.

$$
\begin{array}{lll}
match_{AAB}\ [] & = & False \\
match_{AAB}\ (s:ss) & = & h\ (A == s) \\
 & & \textbf{where} \\
 & & h\ True\ = match_{AB}\ ss \\
 & & h\ False\ = match_{AAB}\ ss
\end{array}
$$

$$
\begin{array}{lll}
match_{AB}\ [] & = & False \\
match_{AB}\ (s : ss) & = & h\ (A == s) \\
 & & \textbf{where} \\
 & & h\ True\ = match_B\ ss \\
 & & h\ False\ = h'\ (A == s) \\
 & & \quad \textbf{where} \\
 & & \quad h'\ True\ = match_{AB}\ ss \\
 & & \quad h'\ False\ = match_{AAB}\ ss \\
match_B\ [] & = & False \\
match_B\ (s : ss) & = & h\ (B == s) \\
 & & \textbf{where} \\
 & & h\ True\ = True \\
 & & h\ False\ = h'\ (A == s) \\
 & & \quad \textbf{where} \\
 & & \quad h'\ True\ = match_A\ ss \\
 & & \quad h'\ False\ = h''\ (A == s) \\
 & & \quad\quad \textbf{where} \\
 & & \quad\quad h''\ True\ = match_{AB}\ ss \\
 & & \quad\quad h''\ False\ = match_{AAB}\ ss
\end{array}
$$

This version of the pattern matcher is a specialised program $match_{AAB}$ for a given pattern AAB. Here, if the matching test fails on the head of the input pattern, then the head of the input string is used to determine the next matching attempt. Thus positive supercompilation is able to produce an efficient definition by performing positive information propagation, despite the redundant inner tests for $A == s$ in the definitions of the $match_{AB}$ and $match_B$ functions. This resulting program corresponds to the Knuth-Morris-Pratt (KMP) algorithm for pattern-matching [58].

The deforestation transformation presented earlier is not capable of performing such transformations as it does not propagate positive information as done in the supercompilation technique.

Distillation

Distillation [37, 38] builds on positive supercompilation but works on the meta-level. Rather than operating on expressions, distillation works on the results of transforming these expressions, which may be recursive. As a result, many superlinear improvements can be obtained using distillation which are not obtained by using any of the previously described transformation techniques such as deforestation or supercompilation.

An an example, consider the following naïve list reversal program *nrev* that computes the reverse of a given list vs. This definition employs the intermediate data structure

nrev xs that is used by the *app* function to compute the reverse of the input list *xs*. This definition of *nrev* is of quadratic time and space complexity w.r.t. the length of the list *vs*.

$$
\begin{array}{l}
nrev\ vs \\
\textbf{where} \\
nrev\ [] \quad = [] \\
nrev\ (x : xs) \quad = app\ (nrev\ xs)\ [x] \\
app\ []\ ys \quad = ys \\
app\ (x : xs)\ ys \ = x : (app\ xs\ ys)
\end{array}
$$

A more efficient definition that is linear w.r.t. the length of list *vs* is shown below, which is achievable using the distillation transformation. This definition works by computing the resulting list as an accumulating argument.

$$
\begin{array}{l}
arev\ vs \\
\textbf{where} \\
arev\ xs \quad = arev'\ xs\ [] \\
arev'\ []\ ys \quad = ys \\
arev'\ (x : xs)\ ys \ = arev'\ xs\ (x : ys)
\end{array}
$$

The distillation transformation is discussed in further detail in Chapter 3 using examples.

2.1.2 Calculation-Based Program Transformation

Though a number of effective unfold/fold-based transformation techniques exist, a new class of techniques called calculation-based transformation were introduced to provide a concrete framework to define the transformation process. These techniques that transform programs through calculation are based on the theory of Constructive Algorithmics [7, 62, 66, 31]. This involves serially applying a set of calculational laws to a given program to transform it into a version that follows the properties described by the laws.

The calculational approach imposes restrictions on the form of input programs to make their recursion structure explicit, resulting in forms such as catamorphisms, anamorphisms and hylomorphisms [83]. The definitions for catamorphism, anamorphism and hylomorphism over *cons*-lists are presented in Definitions 2.1, 2.2 and 2.3 respectively.

Definition 2.1 (List Catamorphism):
A list catamorphism h is defined as shown below given a binary operator $\oplus :: a \to b \to b$ and a unit expression e.

$$\begin{aligned} h\ [] &= e \\ h\ (x : xs) &= x\ \oplus\ (h\ xs) \end{aligned}$$

This catamorphism is represented as $h = ([e, \oplus])$.

Catamorphisms are generalised *fold* operations that provide a standard way to consume a data structure. They substitute the constructors of a data type with other operations with the same signature. An example of a program that can be expressed as a catamorphism is the *sumList* function that computes the sum of elements in a given list.

$$\begin{aligned} sumList\ [] &= 0 \\ sumList\ (x : xs) &= x + (sumList\ xs) \end{aligned}$$

Definition 2.2 (List Anamorphism):
A list anamorphism h is defined as shown below given a predicate function $p :: b \to Bool$ and a function $g :: b \to (a, b)$.

$$\begin{aligned} &h\ (p\ x) \\ &\textbf{where} \\ &h\ True\ = [] \\ &h\ False\ = \textbf{let}\ (y, x') = g\ x\ \textbf{in}\ y : (h\ x') \end{aligned}$$

This anamorphism is represented as $h = [\![g, p]\!]$.

Dually, anamorphisms are generalised *unfold* operations that offer a standard way of constructing data structures. An example of a program that can be expressed as an anamorphism is the $fibs$ function that builds a list of n fibonacci numbers after 1 (*Succ Zero*).

$$\begin{aligned} fibs\ Zero &= [Succ\ Zero] \\ fibs\ (Succ\ Zero) &= [Succ\ Zero] \\ fibs\ (Succ\ (Succ\ n)) &= \big((head\ (fibs\ n')) + (head\ (fibs\ n''))\big) : (fib\ n') \end{aligned}$$

Definition 2.3 (List Hylomorphism):
A list hylomorphism h is defined as shown below given a binary operator $\oplus :: b \to c \to c$, a predicate function $p :: a \to Bool$, a function $g :: a \to (b, a)$ and a unit expression e.

$$\begin{aligned} &h\ (p\ x) \\ &\textbf{where} \\ &h\ True\ = e \\ &h\ False\ = \textbf{let}\ (y, x') = g\ x\ \textbf{in}\ y \oplus (h\ x') \end{aligned}$$

This hylomorphism is represented as $h = [\![(e, \oplus), (g, p)]\!]$.

A hylomorphism is a composition of an anamorphism with a catamorphism, i.e. $[\![(e, \oplus), (g, p)]\!] = ([e, \oplus]) \circ [\![g, p]\!]$. For example, the composition $sumList \circ fibs$ can be expressed as a hylomorphism.

Writing programs as catamorphisms, anamorphisms or hylomorphisms can be quite tedious as they are quite abstract and require some familiarity with category theory. Therefore, some methods have been proposed to structure recursive functions in programs into these forms [43, 72].

The general procedure to formally transform a program into calculational form is as follows:

1. *Capture program structure in specialised form*: Transform the given program into a more restrictive form for practicality. Even though hylomorphims have the most general calculational properties, they may be too general for a specific optimising transformation.

2. *Design calculational laws*: According to the specialised form of programs, specialised calculational laws are designed. These laws describe desirable properties of programs.

3. *Design calculational algorithm*: Finally, to transform a given program, a calculational algorithm is designed to specify how to convert a given program into the specialised form, and how to apply the newly designed calculational laws in a systematic way.

Loop Fusion

Based on this framework, Hu et al. [45] demonstrate how to formalise the well-known loop fusion transformation [3] in calculational form. Loop fusion is an optimisation technique used in compiler construction to fuse adjacent loops in a program into a single loop. This reduces loop overhead and improves run-time performance. The loop fusion proposed in calculational form transforms programs defined over lists where loops are specified by recursive definitions. The following three cases of loops are addressed in this transformation: (1) two adjacent loops where the result of the first is used by the second; (2) two adjacent loops where the result of the first is not used by the second; and (3) one loop is used inside another.

For example, consider the program shown below to compute the sum of all "bigger" elements in a list. An element is bigger if it is greater than the sum of all elements that appear after this element in the list. Here, the composition $(sum \circ biggers)$ is an example of adjacent loops in case (1) and the call to sum from $biggers$ is an example of one loop used inside another described in case (3).

$$
\begin{array}{lll}
sumBiggers\ xs & = & (sum \circ biggers)\ xs \\
\textbf{where} & & \\
biggers\ [] & = & [] \\
biggers\ (x : xs) & = & h\ (x \geq (sum\ xs)) \\
 & & \textbf{where} \\
 & & h\ True\ = x : (biggers\ xs) \\
 & & h\ False\ = biggers\ xs \\
sum\ [] & = & 0 \\
sum\ (x : xs) & = & x + (sum\ xs)
\end{array}
$$

The calculation-based loop fusion transformation proposed in [45] can eliminate these types of loops from a given program. The transformation uses a list-mutumorphism as a specialised form for the programs. List-mutumorphism, shown in Definition 2.4, is a general form that covers all primitive recursive functions over lists.

Definition 2.4 (List Mutumorphism):
A function f_1 is a list mutumorphism w.r.t. other functions $f_2, \ldots, f_N$ if each f_n $(1 \leq n \leq N)$ is defined in the following form.

$$
\begin{array}{lll}
f_n\ [] & = & e_n \\
f_n\ (x : xs) & = & a \oplus_n (f_1\ xs, \ldots, f_N\ xs)
\end{array}
$$

where e_n $(1 \leq n \leq N)$ are given constants and $\oplus_n$ $(1 \leq n \leq N)$ are given binary functions. Function f_1 is represented as $f_1 = [\![(e_1, \ldots, e_N), (\oplus_1, \ldots, \oplus_N)]\!]$.

To transform programs defined as list-mutumorphisms, the following calculational laws are defined:

1. *Flattening*: This rule is used to flatten nested loops that are expressed as a list-mutumorphism to a list-homomorphism. A list-homomorphism can be defined as shown in Definition 2.5.

   Definition 2.5 (List Homomorphism):
   A list homomorphism can be written as $hom_l = ([e, \oplus])_l$, uniquely described by e and $\oplus$.

$$
\begin{array}{lll}
hom_l\ [] & = & e \\
hom_l\ (x : xs) & = & x \oplus (hom_l\ xs)
\end{array}
$$

A list homomorphism is a special case of list mutumorphism – $([e, \oplus]) = [\![(e), (\oplus)]\!]$ – and represents a single-recursive function defined over a list.

2. *Tupling*: This rule, similar to the tupling strategy in unfold/fold-based transformations, is used to merge two independent loops. It is defined over list-homomorphisms since all mutumorphisms can be transformed into homomorphisms using the flattening rule.

3. *Shortcut Fusion*: This rule is the calculational equivalent of the deforestation transformation. It states that if a function can be defined in terms of a *build* function then it can be fused into a list-homomorphism from its right. The *build* function is a list production function defined as $build\ g = g\ ([], (:))$.

4. *Warm-up*: This rule allows for a *build* function to be derived for a given list-homomorphic definition.

5. *Mutumorphism Promotion*: This rule allows fusing a function with a mutumorphism. Even though mutumorphisms can be transformed into homomorphisms and then fused using the shortcut fusion rule, this rule allows more flexibility in the transformation process.

The result of transforming the *sumBiggers* program is shown below. Here, both the adjacent and nested loops have been eliminated by extracting the recursive call to $sumBiggers'$.

$$
\begin{array}{lcl}
sumBiggers\ xs & = & fst\ (sumBiggers'\ xs) \\
\textbf{where} & & \\
fst\ (a, b) & = & a \\
sumBiggers'\ [] & = & (0, 0) \\
sumBiggers'\ (x : xs) & = & \textbf{let}\ (r, s) = sumBiggers'\ xs \\
 & & \quad \textbf{in}\ \big(h\ (x \geq s) \\
 & & \quad\quad \textbf{where} \\
 & & \quad\quad h\ True\ = (x + r,\ x + s) \\
 & & \quad\quad h\ False\ = (r,\ x + s)\big)
\end{array}
$$

Calculation-based transformations are powerful and their framework allows better automation and derivation of more efficient programs. However, they require that the initial program be defined in specialised forms, which may either be too generic and abstract like hylomorphisms or too specific and restrictive like list-homomorphisms. As a result, it is tedious for developers to define their programs in these forms and even unrealistic in some cases. Programs that are not in these forms need to be first transformed

into such specialised forms before designing the calculational laws and algorithm. This poses significant overhead to the transformation process. Further, many calculational transformations make use of associative operators, whose associativity property and unit values must be known prior to the application of these techniques [19].

2.2 Parallelisation of Functional Programs

The two program transformation approaches – unfold/fold-based and calculation-based – are not only used to eliminate inefficiencies in programs but also to transform them into certain desirable forms. To this end, program transformation has been used to manipulate a given program to make explicit any inherent parallelism, or to systematically derive parallel programs.

2.2.1 Unfold/Fold-Based Program Parallelisation

The transformation rules and strategies described in Section 2.1.1 have been used in different approaches to systematically transform a given program into a form that enables parallel evaluation of sub-expressions.

Parallelisation via Context Preservation

In his initial work, Chin [18] proposed the following 4-stage method to synthesise parallel programs from sequential definitions. For example, consider the program shown below to compute the product of all elements in a given list.

$$\begin{array}{lcl} product\ [] & = & 1 \\ product\ (x : xs) & = & x * (product\ xs) \end{array}$$

To ensure parallelisation, the authors define a heuristic that all operations (for example $*$, *product*) applied to the recursive variables (for example xs) must be either associative or distributive.

The four steps involved in the transformation are:

1. *Obtain two recursive definitions*: Using the heuristic and by unfolding the recursive call in the definition of the function, an additional definition for the same function is obtained. The contexts in these two definitions with holes in place of the non-recursive variables are identical.

For example, the following two recursive definitions can be obtained for the *product* function.

$$\begin{array}{lcl} product\ [] & = & 1 \\ product\ [x] & = & x \\ product\ ([x] +\!\!+\ xs) & = & x * (product\ xs) \end{array}$$

$$\begin{array}{lcl} product\ [] & = & 1 \\ product\ [x] & = & x \\ product\ ([x] +\!\!+\ [x]' +\!\!+\ xs) & = & (x * x') * (product\ xs) \end{array}$$

2. *Second-Order Generalisation*: The two recursive definitions are used to obtain a single parallel definition using second-order generalisation. In this process, one or more unknown functions may be introduced for the mismatched sub-expressions in the two recursive definitions.

   The parallel recursive definition for the *product* function is shown below. Here, a new function f is introduced by second-order generalisation to resolve the mismatch in sub-expressions $x * (product\ xs)$ and $(x * x') * (product\ xs)$ in the two recursive definitions obtained from step (1).

$$\begin{array}{lcl} product\ [] & = & 1 \\ product\ [x] & = & x \\ product\ (xr +\!\!+\ xs) & = & (f\ xr) * (product\ xs) \end{array}$$

3. *Derive unknown functions*: By deciding suitable base-cases and induction steps, the definitions for each of the unknown functions introduced in the parallel definition are synthesised. These auxiliary functions are instrumental in defining the associative *combine* operators that are required to parallelise the recursive function.

   The definition for the unknown function f derived for the *product* function is shown below. This definition is syntactically identical to the definition of the original *product* function.

$$\begin{array}{lcl} f\ [] & = & 1 \\ f\ (x : xs) & = & x * (f\ xs) \end{array}$$

   By unifying this definition with the generalised definition of *product*, the following parallel version for the *product* function is created.

$$\begin{array}{lcl} product\ [] & = & 1 \\ product\ [x] & = & x \\ product\ (xr +\!\!+\ xs) & = & (product\ xr) * (product\ xs) \end{array}$$

4. *Tupling for efficient parallelism*: The parallel definition derived may contain redundant recursive calls. This is inefficient as the same computation may be performed in different sub-expressions. Using the tupling strategy described in Section 2.1.1, such redundant function calls are unified to obtain an efficient parallel definition.

This approach was later formalised by Chin et al. [19] using context preservation. The rules of context preservation allow extraction of sub-expressions that can be evaluated in parallel, and a context preservation theorem states when these rules are applicable to a linear recursive function for parallelisation.

These methods illustrate how unfold/fold-based transformation techniques can be augmented with the generalisation strategy to systematically derive efficient parallel programs from their sequential versions. Their main advantage is that they automatically invent auxiliary functions to define associative operators required for parallel evaluation. However, these methods are designed for first-order programs (those that do not have functions as input arguments or output) and address functions that contain only a single recursive input argument. Furthermore, these methods place constraints over the input types and sizes in certain cases for the parallelisation procedure to be applicable.

For example, consider the *vprod* function shown below defined on two lists. This program is potentially parallelisable since each point-wise product can be computed independently.

$$\begin{array}{lcl} vprod\ []\ ys & = & [] \\ vprod\ (x:xs)\ [] & = & [] \\ vprod\ (x:xs)\ (y:ys) & = & [x*y] +\!\!+ (vprod\ xs\ ys) \end{array}$$

The parallelised form of *vprod* produced by [19] shown below, where f_{vprod} is an auxiliary function introduced by generalisation, requires that the length of lists xr and yr be equal. Even though this can be proved for this program, such constraints limit complete automation of the parallelisation transformation.

$$\begin{array}{lcl} vprod\ []\ ys & = & [] \\ vprod\ xs\ [] & = & [] \\ vprod\ (xr +\!\!+ xs)\ (yr +\!\!+ ys) & = & (f_{vprod}\ xr\ yr) +\!\!+ (vprod\ xs\ ys) \end{array}$$

An Analytical Method for Parallelisation

Ahn & Han [2] proposed a method to analyse first-order programs given they operate over a single recursive input and zero or more accumulating arguments, and generate a data parallel version. This is achieved by classifying sub-expressions based on their

usage relations to the accumulating parameters and the results of recursive calls using program slicing [89] of their first-order language. To classify sub-expressions in the body of a function definition, results of the recursive calls are classified as *red results* (those that do not use any value of accumulating arguments) and *black results* (those that may use values of accumulating arguments). Following this, the sub-expressions are classified as follows:

- *White expressions*: Those that do not use accumulating parameters and results of recursive calls.
- *Red expressions*: Those that do not use accumulating parameters and are used to evaluate red results.
- *Blue expressions*: Those that do not use accumulating parameters and are not used to evaluate red results.
- *Yellow expressions*: Those that evaluate accumulating parameters.
- *Green expressions*: Those that do not use black results.
- *Black expressions*: All remaining expressions.

Based on this classification, each coloured sub-expression is defined using polytypic data parallel skeletons – *map*, *reduce*, $scan_{up}$, $scan_{down}$ and *zip*. Using the usage relations of the coloured sub-expressions:

- white, blue and green expressions are transformed into the argument functions of *map* skeletons,
- red expressions are transformed into the argument functions of $scan_{up}$ skeletons,
- yellow expressions are transformed into the arguments function of $scan_{down}$ skeletons, and
- black expressions are transformed into the argument functions of *reduce* skeletons.

Each of these polytypic skeletons can potentially be implemented in parallel for efficient execution. Although this analytical method is simple and powerful to define programs using data parallel skeletons, there are a few drawbacks. Firstly, this method

is only capable of transforming programs that are defined in a first-order language. Secondly, the transformation is designed to analyse recursive functions that have only one recursive input argument. Lastly, the resulting parallel program defined using data parallel skeletons makes extensive use of intermediate data structures between the skeletons. These factors potentially limit applicability and efficiency of this transformation.

AutoPar

More recently, Dever [26] proposed a transformation technique called *AutoPar*, which is capable of parallelising a given sequential program and eliminating intermediate data structures. AutoPar is designed to transform higher-order language programs that are defined over a single recursive input of any data type. The objective of this technique is to transform the recursive input of a given program into a well-partitioned join-list defined as follows:

$$\textbf{data}\ \mathit{JoinList}\ a\ =\ \mathit{Empty}\ |\ \mathit{Singleton}\ a\ |\ \mathit{Join}\ (\mathit{JoinList}\ a)\ (\mathit{JoinList}\ a)$$

Following this, the original program is transformed to operate over the join-list input using the *distillation* technique (presented in Chapter 3) to eliminate intermediate data structures. Since the join-list data can be effectively partitioned, the transformed program is parallelisable using an efficient divide-and-conquer approach. Using a parallelisation algorithm to analyse the transformed program, AutoPar explicitly specifies independent sub-expressions for parallel evaluation using the *par* and *pseq* parallel constructs in the Glasgow Haskell Compiler (GHC).

The primary advantages of this transformation technique are that it can parallelise higher-order language programs and can handle functions defined over a recursive input of any data type. However, it cannot be used to parallelise programs that contain functions defined over multiple inputs, and there may also be a mismatch between the join-list structure and the algorithmic structure of a program.

Extracting Data Parallel Computations from Distilled Programs

Based on the approach of *AutoPar* [26], Kannan et al. [54] proposed a parallelisation method where the inputs of a given program are transformed into *cons*-lists using the data type transformation technique in *AutoPar* and the distillation technique. The

resulting programs that operate over *cons*-lists were then parallelised by automatically identifying list-based parallel skeletons such as *map*, *reduce* and *zipWith*.

Though this approach resulted in automatic parallelisation of some programs, other programs that were straight-forward to parallelise by hand were not parallelised by this method. The primary reason for this was identified to be mismatch between the data structures of the transformed *cons*-list inputs and the algorithmic structures of the transformed programs. Thus, this work laid the foundation for the ideas behind the transformation method presented in this thesis.

2.2.2 Calculation-Based Program Parallelisation

As explained in Section 2.1.2, calculation-based transformation provides a well-defined framework to systematically transform programs using three steps: (1) define program in a specialised form; (2) design calculational laws; and (3) design a calculational algorithm. Using this framework, programs can be transformed not only to eliminate intermediate data structures, but also to systematically derive parallel programs.

Homomorphism for Parallelisation

Skillicorn [79] first expressed the usage of Bird Meerten's Formalism (BMF) [7] as a model for program parallelisation. BMF allows the following:

- a polymorphic data type described by its constructors.
- a set of operations defined on the data type, including a *map* operation and a *reduce* operation.
- equations on the operations and on the constructors defining how the operations relate to each other and how to evaluate homomorphisms in terms of the constructors.
- a guarantee of the completeness of the set of equations.
- a guarantee that any function defined over the data type that relies only on the type properties can be expressed in terms of the operations.

Skillicorn's work shows that homomorphisms over a given data type, such as lists, can be evaluated in parallel using the *map* and *reduce* operations defined on the same

data type. This proposal led to a series of works by Cole [24, 23] and Gorlatch [35] for systematically deriving homomorphic algorithms for different problems that operate over lists. The essence of these works resulted in Gibbon's Third Homomorphism Theorem [32] as stated in Theorem 2.1, which uses the leftward and rightward computability properties of a function to prove that it is a homomorphism.

Definition 2.6 (Leftward Functions):
A function h defined over a list can be computed leftward (from left to right) for a binary operator $\oplus$ and given constant e iff for all elements x and lists xs,

$$\begin{aligned} h\ [] &= e \\ h\ ([x] +\!\!+\ xs) &= x \oplus (h\ xs) \end{aligned}$$

Definition 2.7 (Rightward Functions):
A function h defined over a list can be computed rightward (from right to left) for a binary operator $\oplus$ and given constant e iff for all elements x and lists xs,

$$\begin{aligned} h\ [] &= e \\ h\ (xs +\!\!+\ [x]) &= (h\ xs) \oplus x \end{aligned}$$

Theorem 2.1 (Third Homomorphism Theorem):
If a function h is both leftwards and rightwards, then h is a homomorphism.

Consequently, this allows parallel evaluation of a computation defined over a list as a homomorphism if the binary operator in the homomorphism is associative. This is because the homomorphism can then be computed according to any paranthesisation of the elements of the list.

Homomorphism on Trees

Morihata et al. [71] extended the third homormophism theorem over binary trees to allow their parallel evaluation. This was achieved by converting a given binary tree into a *zipper* structure, which is a list of sub-trees that are encountered when walking from the root to a leaf.

For example, the zipper structure that expresses a path from the root of a binary tree to the black leaf is shown in Figure 2.1.

A function $z2t$ to transform a zipper structure into a binary tree is defined as shown below.

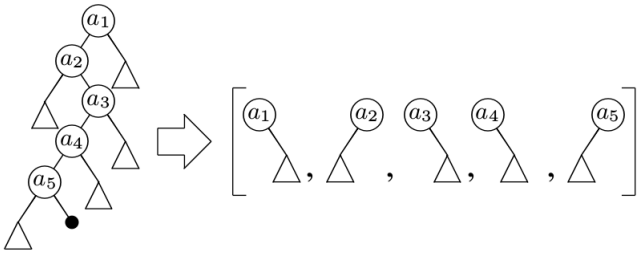


Figure 2.1: Example of a Zipper Structure [71]

$$\begin{aligned}
&\textbf{data}\ Tree\ =\ Leaf \mid Node\ Int\ Tree\ Tree\\
&\textbf{data}\ Zipper\ =\ [(Either\ (Int, Tree)\ (Int, Tree)]\\
&z2t\ [] &&= Leaf\\
&z2t\ ([Left\ (n,l)]\ +\!\!+\ r) &&= Node\ n\ l\ (z2t\ r)\\
&z2t\ ([Right\ (n,r)]\ +\!\!+\ l) &&= Node\ n\ (z2t\ l)\ r
\end{aligned}$$

For the transformation, upward and downward computations over the binary tree are defined in terms of the corresponding zipper structure. Following this, using a divide-and-conquer approach, a decomposition operation for the binary tree is defined as a list-homomorphism on the zipper structure. This allows parallel evaluation of the zipper and hence the binary tree.

For example, consider the function *sumTree* shown below, which computes the total of all values stored in the nodes of a binary tree.

$$\begin{aligned}
&sumTree\ Leaf &&= 0\\
&sumTree\ (Node\ n\ l\ r) &&= n + (sumTree\ l) + (sumTree\ r)
\end{aligned}$$

The function $sumTree_{\downarrow}$ shown below is defined over the zipper and performs the same computation downward from the root of the binary tree to a terminal leaf.

$$\begin{aligned}
&sumTree_{\downarrow}\ [] &&= 0\\
&sumTree_{\downarrow}\ (x\ +\!\!+\ [Left\ (n,l)]) &&= (sumTree_{\downarrow}\ x) + n + (sumTree\ l)\\
&sumTree_{\downarrow}\ (x\ +\!\!+\ [Right\ (n,r)]) &&= (sumTree_{\downarrow}\ x) + n + (sumTree\ r)
\end{aligned}$$

The function $sumTree_{\uparrow}$ shown below is defined over the zipper and performs the same computation from a terminal leaf upward to the root node.

$$\begin{aligned}
&sumTree_{\uparrow}\ [] &&= 0\\
&sumTree_{\uparrow}\ ([Left\ (n,l)]\ +\!\!+\ r) &&= n + (sumTree\ l) + (sumTree_{\uparrow}\ r)\\
&sumTree_{\uparrow}\ ([Right\ (n,r)]\ +\!\!+\ l) &&= n + (sumTree_{\uparrow}\ l) + (sumTree\ r)
\end{aligned}$$

In $sumTree_{\downarrow}$ and $sumTree_{\uparrow}$, the computations to sum the trees in the zipper are performed using the *sumTree* function. Using these versions for *sumTree* and the third homomorphism theorem over binary trees, a parallel version *sumPara* is derived. The definition for *sumPara* is shown below, in which a zipper can be decomposed using the $+\!\!+$ operator and processed in parallel.

$$
\begin{array}{lcl}
sumPara\ [] & = & 0 \\
sumPara\ ([Left\ (n,l)]) & = & n + (sumTree\ l) \\
sumPara\ ([Right\ (n,r)]) & = & n + (sumTree\ r) \\
sumPara\ (xs +\!\!+ ys) & = & (sumPara\ xs) + (sumPara\ ys)
\end{array}
$$

Parallelisation Using J-Homomorphisms

Based on the theory of list homormophisms, Hu et al. [45] proposed a calculational transformation to parallelise programs defined on lists. Since *cons*-lists are inherently sequential, the specialised form proposed in this transformation is a J-homomorphism on *join*-lists as shown in Definition 2.8.

Definition 2.8 (J-Homomorphism):
A function h is a J-homomorphism when defined over finite lists using list concatenation for which there exists an associative binary operator $\oplus$ such that, for the concatenation operator $+\!\!+$ and all finite lists xs and ys,

$$
\begin{array}{lcl}
h\ [] & = & v \\
h\ (xs +\!\!+ ys) & = & (h\ xs) \oplus (h\ ys)
\end{array}
$$

This form allows parallel evaluation of $h\ xs$ and $h\ ys$ using the associativity of $\oplus$. However, list-based programs may not always be defined on join-lists. Therefore, the main calculational rule allows conversion of a *cons*-list homomorphism to a J-homomorphism. This allows transformation of programs defined over *cons*-lists to be defined over *join*-lists.

For example, consider the *mss* program shown below that computes the maximum segment sum in a given *cons*-list. Here the binary operator $\uparrow$ returns the larger of two input arguments and is assumed as built-in.

$$
\begin{array}{lcl}
mss\ [] & = & 0 \\
mss\ (x:xs) & = & x\ \uparrow\ (x + (mis\ xs))\ \uparrow\ (mss\ xs) \\
mis\ [] & = & 0 \\
mis\ (x:xs) & = & x\ \uparrow\ (x + (mis\ xs))
\end{array}
$$

The *mis* function can be redefined using a J-homomorphism function h as shown below.

$$
\begin{array}{lcl}
mis\ [] & = & 0 \\
mis\ (x:xs) & = & \textbf{let}\ (x_1, x_2) = h\ xs\ \textbf{in}\ x_1\ \uparrow\ x_2 \\
\textbf{where} & & \\
h\ [x] & = & (x, x) \\
h\ (xs +\!\!+ ys) & = & (h\ xs) \otimes (h\ ys) \\
(x_1, x_2) \otimes (y_1, y_2) & = & (x_1 \uparrow (x_2 + y_1),\ x_2 + y_2)
\end{array}
$$

Subsequently, the calculation algorithm for this transformation consists of two steps: (1) application of the loop-fusion calculation (described in Section 2.1.2) to a given program to obtain a compact program defined in terms of homomorphisms; and (2) application of the calculational rule to transform *cons*-list homomorphisms to J-homomorphisms.

For the maximum segment sum example, this transformation produces the *mss_mis* function shown below, which is defined over a join-list. Here the J-homomorphism function h can be evaluated in parallel using the associativity of the binary operator $\oplus$.

$$
\begin{array}{l}
mss\_mis\ xs\ =\ \textbf{let}\ (x_1,x_2,x_3,x_4,x_5) = h\ xs\ \textbf{in}\ f\ x_1\ x_2\ x_3\ x_4\ x_5\ (0,0) \\
\textbf{where} \\
h\ [x] \qquad\quad = (x,x,0,x,x) \\
h\ (xs +\!\!+ ys)\ =\ (h\ xs) \oplus (h\ ys) \\
(x_1,x_2,x_3,x_4,x_5) \oplus (y_1,y_2,y_3,y_4,y_5) = (x_1 \uparrow (x_2+y_4) \uparrow (x_3+y_1), (x_2+y_5) \uparrow (x_3+y_2), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x_3+y_3,\ x_4 \uparrow (x_5+y_4),\ x_5+y_5) \\
f\ x_1\ x_2\ x_3\ x_4\ x_5\ (s,i) = (x_1 \uparrow (x_2+i) \uparrow (x_3+s),\ x_4 \uparrow (x_5+i))
\end{array}
$$

Diffusion Transformation

Hu et al. [44] proposed another calculation-based transformation called *diffusion*, which uses the specialised form shown in Definition 2.9 for the programs to be parallelised. Here, function h has two parameters: the first is of type *cons*-list and the second is an accumulating argument. This form is an extension of a *map-reduce* skeleton that also encompasses computation of an accumulating parameter.

Definition 2.9 (Specialised Form for Duffusion):

$$
\begin{array}{l}
h\ []\ v \qquad\quad = g_1\ v \\
h\ (x:xs)\ v\ =\ (g_2\ x\ v)\ \oplus\ (h\ xs\ (v\ \otimes\ (g_3\ x)))
\end{array}
$$

To illustrate the diffusion transformation, consider the following program where function *bm* solves the bracket matching problem. It takes two inputs: (1) a list of tokens xs in a given expression, and (2) a stack s. It is assumed that the stack operations *push*, *pop*, *notEmpty*, *top* and the token operations *match*, *isClose*, *isOpen* are built-in, and hence are not defined here. The *match*, *isClose* and *isOpen* operations return *true* if two given tokens are the same, a given token is a close parenthesis and a given token is an open parenthesis, respectively, and false otherwise.

$$
\begin{array}{lll}
bm\ []\ s & = isEmpty\ s & \\
bm\ (x:xs)\ s & = f\ (isOpen\ x) & \\
 & \quad \textbf{where} & \\
 & \quad f\ True\ = bm\ xs\ (push\ x\ s) & \\
 & \quad f\ False = f'\ (isClose\ x) & \\
 & \qquad \textbf{where} & \\
 & \qquad f'\ True\ = (notEmpty\ s) \wedge (match\ x\ (top\ s)) \wedge & \\
 & \qquad\qquad (bm\ xs\ (pop\ s)) & \\
 & \qquad f'\ False = bm\ xs\ s &
\end{array}
$$

The steps of the calculational algorithm, which includes the calculational laws, designed for the diffusion transformation are as follows:

1. *Linearise Recursive Calls*: If there are multiple recursive calls in a given recursive function, then they are merged into a single one. This is because diffusion requires that a recursive function contain only one recursive call.

   The result of transforming the bracket matching program to combine the three different recursive calls to function *bm* is shown below. This definition contains only one recursive call to function *bm* and is called a *linear recursive definition*.

$$
\begin{array}{lll}
bm\ [] & = isEmpty\ s & \\
bm\ (x:xs) & = (f_2\ (x,s)) \wedge (bm\ xs\ (f_3\ x\ s)) & \\
f_2\ (x,s) & = f_2'\ (isOpen\ x) & \\
 & \quad \textbf{where} & \\
 & \quad f_2'\ True\ = True & \\
 & \quad f_2'\ False\ = f_2''\ (isClose\ x) & \\
 & \qquad \textbf{where} & \\
 & \qquad f_2''\ True\ = (notEmpty\ s) \wedge (match\ x\ (top\ s)) & \\
 & \qquad f_2''\ False\ = True & \\
f_3\ x\ s & = f_3'\ (isOpen\ x) & \\
 & \quad \textbf{where} & \\
 & \quad f_3'\ True\ = push\ x\ s & \\
 & \quad f_3'\ False\ = f_3''\ (isClose\ x) & \\
 & \qquad \textbf{where} & \\
 & \qquad f_3''\ True\ = pop\ s & \\
 & \qquad f_3''\ False\ = s &
\end{array}
$$

2. *Identify Associative Operators*: A function defined in the specialised form (Definition 2.9) can be executed in parallel if the binary operator $\oplus$ is associative over the output type of function h and the operator $\otimes$ is associative over the type of the accumulating parameter v. Therefore, these operators have to be derived manually from the program to be transformed.

For the bracket matching example, the associative operator $\oplus$ is identified as the $\wedge$ operator. Given the *Stack* data type for function *bm* declared as shown below, the $\otimes$ operator is manually derived as follows.

$$
\begin{array}{l}
\textbf{data}\ Stack\ a\ =\ Empty \mid Push\ a\ Stack \mid Pop\ Stack \\
s \otimes Empty \quad\ = s \\
s \otimes (Push\ x\ s') = Push\ x\ (s \otimes s') \\
s \otimes (Pop\ s') \quad = Pop\ (s \otimes s')
\end{array}
$$

Further, the functions that correspond to g_1, g_2 and g_3 in Definition 2.9 are $isEmpty$, function f_2 (in the linear recursive definition of *bm*) and function f_3 (defined below using the associative operator $\otimes$), respectively.

$$
\begin{array}{l}
f_3\ x\ s\ =\ s \otimes (f_3'\ (isOpen\ x)) \\
\qquad\quad \textbf{where} \\
\qquad\quad f_3'\ True \quad\ = Push\ x\ Empty \\
\qquad\quad f_3'\ False\ x\ = f_3''\ (isClose\ x) \\
\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\quad f_3''\ True\ \ = Pop\ Empty \\
\qquad\qquad\qquad\quad f_3''\ False\ = Empty
\end{array}
$$

3. *Apply Diffusion Theorem*: The "diffusion theorem" states that a function h defined in the form shown in Definition 2.9 can be rewritten into the form shown in Definition 2.10 using the *map*, *reduce*, *scan* and *zip* skeletons.

Definition 2.10 (Diffused Form):

$$
\begin{array}{rl}
h\ xs\ c\ = & \textbf{let}\ (cs' +\!\!+ [c']) = map\ (c \otimes)\ (scan\ (\otimes)\ (map\ g_3\ xs)) \\
& \quad\ ac = zip\ xs\ cs' \\
& \textbf{in}\ (reduce\ (\oplus)\ (map\ g_2\ ac))\ \oplus\ (g_1\ c')
\end{array}
$$

Note that computation of the accumulating parameter is inherently sequential in the definition of h in Definition 2.9. In order to have an efficient parallel evaluation, the accumulating parameter is removed from the recursion in the diffused form by precomputing it using the expression $map\ (c \otimes)\ (scan \otimes (map\ g_3\ xs))$ using a *scan* skeleton. Since all skeletons in this diffused form can be implemented efficiently for parallel evaluation, the function h can hence be evaluated in parallel. This diffused form in Definition 2.10 was later formalised as the *accumulate* skeleton, which is explained in Section 2.2.3.

The definition of the bracket matching example program can be transformed using the diffusion theorem into the diffused form as shown below, which is parallelisable.

$$
\begin{aligned}
bm\ xs\ c\ =\ & \textbf{let}\ (cs' +\!\!+ [c']) = map\ (c \otimes)\ (scan\ (\otimes)\ (map\ f_3\ xs)) \\
& \quad ac = zip\ xs\ cs' \\
& \textbf{in}\ (reduce\ (\oplus)\ (map\ f_2\ ac))\ \oplus\ (isEmpty\ c')
\end{aligned}
$$

The diffusion transformation was originally defined for programs that operate over a *cons*-list. This has been generalised over binary trees by defining the *map*, *reduce*, *scan*, *zip* skeletons and the diffusion theorem to operate over binary trees. Although the diffusion transformation can be defined for a higher-order language program over any data type, there are a few practical drawbacks that remain. For instance, defining a given algorithm using the specialised form may not always be straightforward, and the binary operators $\oplus$ and $\otimes$ have to be derived and checked for associativity manually. This requirement of manual derivation of associative operators is prevalent in many calculation-based transformation methods and prevents complete automation of deriving a parallel program from a sequential version.

Similar to the approach of calculating programs from a given specification by refinement using transformation rules, Goldberg et al. [33] proposed a method to expressing high-level architecture-independent specifications and a lower-level architecture-specific implementation. Through successive refinement and interactive experimentation, this approach allows development of parallel algorithms from a specification to various efficient architecture-dependent implementations. The language and tools to support this methodology of deriving parallel programs is packaged into the Proteus system.

2.2.3 Skeletons and Libraries for Parallel Programming

The process of developing and implementing parallel programs has been made easier by the use of parallel skeletons. Their use as building blocks for parallel program development has been widely studied based on the seminal works of Cole [22] and Darlington et al. [25]. Some of the most widely used parallel skeletons [34] are *map*, *reduce*, *scan*, *zipWith*, *pipe*, *farm* and *DC* (divide-and-conquer).

For an instance of a given data type T of the form shown in Definition 1.2 with constructors $c_1, \ldots, c_K$, the *map*, *reduce*, *scan* and *zipWith* skeletons are defined as follows:

- *map*: A classic data parallel skeleton, the *map* skeleton applies a given function f over each element in a data-set and produces an output of the same size as that of the input. For example, a *map* operation over a list is illustrated in Figure 2.2.

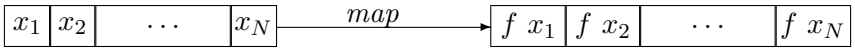


Figure 2.2: *map* Operation over List

The definition of a *map* that operates over a generic type T is shown below.

$$map\ ::\ (T\ a) \to (a \to b) \to (T\ b)$$
$$map\ (c_k\ y\ x_1 \ldots x_N)\ f\ =\ c_k\ (f\ y)\ (map\ x_1\ f) \ldots (map\ x_N\ f)$$

where c_k is a constructor of type T

- *reduce*: Also known as the *fold* skeleton, this skeleton uses a reduction operator $\oplus$ to collapse a data-set into a single value. The reduce skeleton can be implemented in parallel using the divide-and-conquer approach to split and reduce subsets of given data-sets in parallel. The reduction operator must be associative for efficient parallel reduction. For example, a *reduce* operation over a list is illustrated in Figure 2.3.

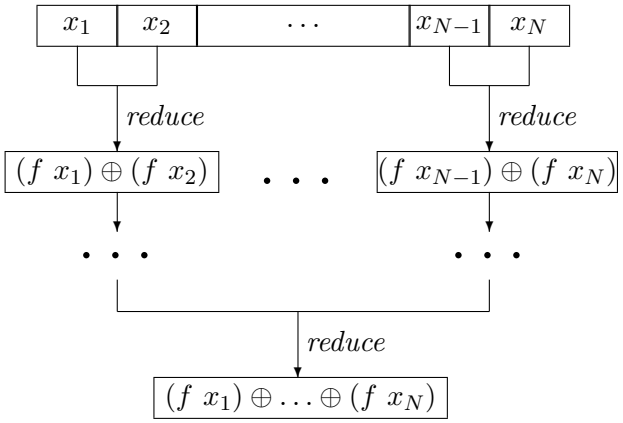


Figure 2.3: *reduce* Operation over List

The definition of a *reduce* that operates over a generic type T is shown below.

$$reduce\ ::\ (T\ a) \to (a \to b) \to (b \to b \to b) \to b$$
$$reduce\ (c_k\ y\ x_1 \ldots x_N)\ f\ (\oplus)\ =\ (f\ y) \oplus (reduce\ x_1\ f\ (\oplus)) \oplus\ \ldots\ \oplus\ (reduce\ x_N\ f\ (\oplus))$$

where c_k is a constructor of type T

- *map-reduce*: A *map-reduce* skeleton unifies the computations performed by a *map* skeleton followed by a *reduce* skeleton on a given data structure. For instance, the definitions of the *map-reduce* skeleton that operates over a *cons*-list xs using a map operator f, a reduction operator g and a unit value v for the reduction operator are shown below. Here, *mapRedr* reduces the list from the right, and *mapRedl* reduces the list from the left using an accumulating argument.

$$
\begin{aligned}
&mapRedr \;::\; [a] \to (b \to b \to b) \to b \to (a \to b) \to b \\
&mapRedr\ []\ g\ v\ f \;=\; v \\
&mapRedr\ (x : xs)\ g\ v\ f \;=\; g\ (f\ x)\ (mapRedr\ xs\ g\ v\ f)
\end{aligned}
$$

$$
\begin{aligned}
&mapRedl \;::\; [a] \to (b \to b \to b) \to b \to (a \to b) \to b \\
&mapRedl\ []\ g\ v\ f \;=\; v \\
&mapRedl\ (x : xs)\ g\ v\ f \;=\; mapRedl\ xs\ g\ (g\ v\ (f\ x))\ f
\end{aligned}
$$

Even though these definitions are sequential, the map-reduce computation can be executed in parallel using the divide-and-conquer paradigm discussed later in this section. For example, consider the map-reduce computation performed on a non-empty list as illustrated in Figure 2.4. Here, the map operation f can be performed over each element in the list in parallel. Following this, the reduction using operator g can be performed by computing the sub-results in parallel.

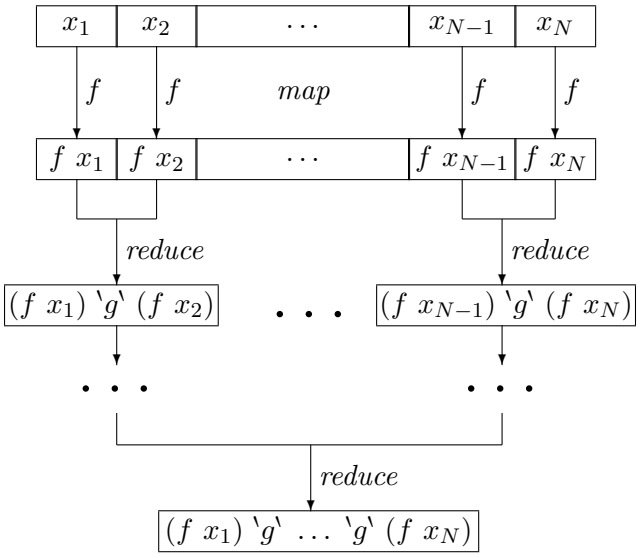


Figure 2.4: *map-reduce* Computation over a List

- *scan*: The *scan* skeleton operates over an input data-set and computes an output data-set of the same size as the input whose elements are the intermediate results of a reduce operation performed on the input. For example, a *scan* operation over a list using the *scan* operator $\oplus$ is illustrated in Figure 2.5.

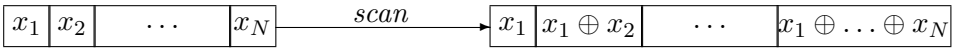


Figure 2.5: *scan* Operation over List

Depending on the data type of the input data-set, different approaches have been proposed to implement the *scan* skeleton in parallel. The *scan* skeleton can be

implemented as $scan_{up}$ or $scan_{down}$ by considering the data structure to be scanned as an n-ary tree as shown below.

$$scan_{up} \;::\; (T\ a) \to (a \to a \to \ldots \to a \to b) \to (T\ b)$$
$$scan_{up}\ (c_k\ y\ x_1 \ldots x_N)\ f \quad = \; c_k\ (f\ y\ (root\ x'_1) \ldots (root\ x'_N))\ x'_1 \ldots x'_N$$

where c_k is a constructor of type T

$$\forall n \in \{1, \ldots, N\} \cdot x'_n = scan_{up}\ x_n\ f$$
$$root\ (c_k\ y\ x_1 \ldots x_N) = y$$

$$scan_{down} \;::\; (T\ a) \to (a \to b \to (b, b, \ldots, b)) \to b \to (T\ b)$$
$$scan_{down}\ (c_k\ y\ x_1 \ldots x_N)\ f\ v \;=\; c_k\ y'\ (scan_{down}\ x_1\ f\ v_1) \ldots (scan_{down}\ x_N\ f\ v_N)$$

where c_k is a constructor of type T

$$(y', v_1, \ldots, v_N) = f\ y\ v$$

- *zipWith*: This data parallel skeleton combines two inputs of the same size pointwise, using a binary operator f, into an output of the same size as the inputs. For example, a *zipWith* operation over a list is illustrated in Figure 2.6.

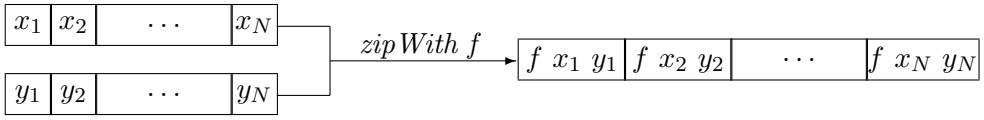


Figure 2.6: *zipWith* Operation over List

The definition of a *zipWith* that operates over a generic type T is shown below.

$$zipWith \;::\; (T\ a) \to (T\ b) \to (a \to b \to c) \to (T\ c)$$
$$zipWith\ (c_k\ z\ x_1 \ldots x_N)\ (c_k\ w\ y_1 \ldots y_N)\ f \;=\; c_k\ (f\ z\ w)\ (zipWith\ x_1\ y_1\ f)$$
$$\vdots$$
$$(zipWith\ x_N\ y_N\ f)$$

where c_k is a constructor of type T

Further, Darlington et al. [25] presented other high-level skeletons including the *pipe* and *DC* (divide-and-conquer) skeletons that are described below:

- *pipe*: This skeleton captures simple linear task parallelism, where a program is composed of a series of functions and the input can be streamed through them.

$$pipe \;::\; [\alpha \to \alpha] \to (\alpha \to \alpha)$$
$$pipe \;=\; foldr1\ (\circ)$$

The *pipe* skeleton defined above takes a list of functions as input and composes them using the $foldr1$ primitive. It can be implemented in parallel by allocating the execution of each function to a different processing unit.

- *divide-and-conquer*: This skeleton is applicable when a given task is divisible into a set of smaller instances of the same task (the *divide* step). These smaller instances are solved and their solutions aggregated to compute the result of the original task (the *conquer* step). An overview of the divide-and-conquer approach is illustrated in Figure 2.7.

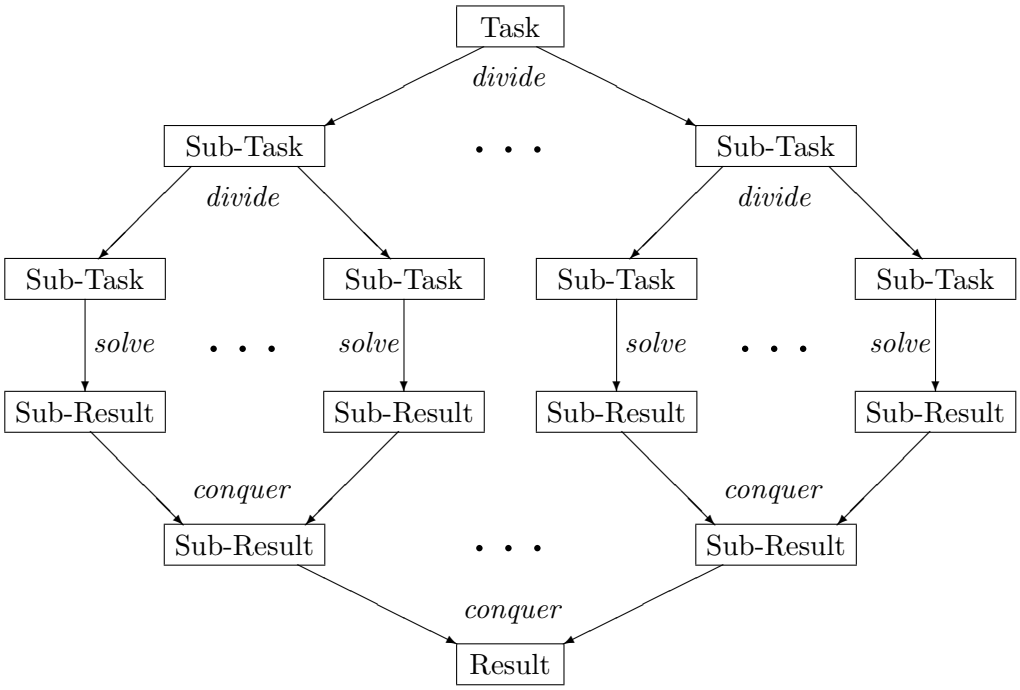


Figure 2.7: Overview of Divide-and-Conquer

Here, the parallelism stems from solving each sub-task independently and conquering the sub-results in parallel. Unlike the *reduce* skeleton, the developer can control the degree of parallelism produced here using a suitable *divide* operation.

In the following definition of the *divide-and-conquer* skeleton, t checks if a given task x is divisible, s computes the solution for x, d divides x into a list of smaller instances and c combines the results of a list of tasks.

$$
\begin{aligned}
&DC \;::\; (\alpha \to Bool) \to (\alpha \to \beta) \to (\alpha \to [\alpha]) \to ([\beta] \to \beta) \to \alpha \to \beta \\
&DC\ t\ s\ d\ c\ x \;=\; test\ (t\ x) \\
&\qquad\qquad \textbf{where} \\
&\qquad\qquad test\ True \;=\; c\ (map\ (DC\ t\ s\ d\ c)\ (d\ x)) \\
&\qquad\qquad test\ False \;=\; s\ x
\end{aligned}
$$

As an example, merge sort can be implemented using the divide-and-conquer skeleton as shown below, where *isSingleton* decides if a list to be sorted can be divided, *split* divides a given list into a list of sub-lists, and *merge* combines two sorted lists into a single sorted list.

$$
\begin{array}{lcl}
mergesort & :: & [\alpha] \to (\alpha \to \alpha \to Bool) \to [\alpha] \\
mergesort\ [] & = & [] \\
mergesort\ xs & = & ((DC\ isSingleton\ id\ split) \circ foldr \circ merge)\ xs \\
 & & \textbf{where}\ isSingleton\ xs = length\ xs \leq 1 \\
split & :: & [\alpha] \to [[\alpha]] \\
merge & :: & [\alpha] \to [\alpha] \to (\alpha \to \alpha \to Bool) \to [\alpha]
\end{array}
$$

A number of skeleton libraries have been created to aid practical parallel program development.

Accumulate Skeleton

As mentioned in Section 2.2.2, the diffused form shown in Definition 2.10 was formalised as the *accumulate* skeleton [48], which uses the *map* and *reduce* skeletons over lists, and the *scan* and *zip* skeletons to parallelise the computation of an accumulating parameter in a recursive function. As a result, any program that is defined using the *accumulate* skeleton can be evaluated in parallel.

However, the *accumulate* skeleton in the form shown in Definition 2.10 makes use of intermediate data structures. For example, the result of $map\ g_3\ xs$ is immediately consumed by *scan* during the precomputation of the accumulating parameter. Using the fusion transformation (a.k.a. shortcut deforestation), the authors of diffusion eliminate the intermediate data structures and use an efficient definition of the *accumulate* skeleton shown in Definition 2.11, which is implemented using MPI. Here, the function $reducel2$ collapses two lists simultaneously from left to right into a single value using an inital value $i_\oplus$ and a binary operator $\ominus$. The function *scanl* scans a list from left to right with inital value c and binary operator $\oslash$. The parallelism of the accumulate skeleton stems from parallel implementations of the $reducel2$ and *scanl* skeletons.

Definition 2.11 (Efficient *accumulate* Skeleton for Implementation):

$$
\begin{array}{rcl}
h\ xs\ c & = & \textbf{let}\ (cs' +\!\!+ [c']) = scanl\ (\oslash)\ c\ xs \\
 & & \quad u \ominus (v, w) = u\ \oplus\ (f\ (v, w)) \\
 & & \quad s \oslash t = s\ \otimes\ (g_2\ t) \\
 & & \textbf{in}\ \big(reducel2\ (\ominus)\ i_\oplus\ xs\ (cs' \oplus (g_2\ c'))\big)
\end{array}
$$

SkeTo Library

Matsuzaki et al. [64] created the SkeTo library to develop a library of skeletons that provide programmers with a set of parallel skeletons that have efficient implementations

under-the-hood. This library contains the *map*, *reduce*, *scan* and *hom* (a.k.a. *map-reduce*) skeletons for three data types: parallel list, parallel matrix and binary trees. These skeletons are packaged into a C++ library and can be used in standard sequential programs with implementations in standard C++ and MPI, and optimisation mechanisms in OpenC++, a meta-language for C++.

Libraries for GPU

On the other hand, a number of libraries have been created for parallel program development targeting GPUs. The Data Parallel Haskell [76, 15] and Accelerate [16] libraries are a few prominent ones. Both these libraries offer parallel skeletons defined over special parallel data types. Operations in the DPH library are defined over a parallel array data type where the elements are indexed by values of integer type, with the parallel operations being $mapP$, $filterP$, $zipWithP$, and so forth. The Accelerate library is defined as an embedded language in Haskell, where the operations are defined over a customised multi-dimensional array data type. The core Accelerate library operations include the *map*, $zipWith$, $fold$ and *scan* skeletons. Upon compilation, the calls to these skeletons are transformed into CUDA kernels using parameterisable implementations for the skeletons that serve as templates. The instantiated CUDA kernels generated for the skeletons used in a program can subsequently be executed on a GPU.

The SkePU [30] and SkelCL [82] provide skeletons that operate over flat data types such as arrays, lists or vectors. The SkePU library offers implementations of the *map*, *reduce*, *map-reduce* and *scan* skeletons among others. While the *map* skeleton can operate over one, two or three vector or matrix inputs, the other skeletons operate over a single input. The SkelCL library offers implementations of the *map*, *reduce*, *zip* and *scan* skeletons. While the *map*, *reduce* and *scan* skeletons operate over a single vector input, the *zip* skeleton operated over two inputs.

Additionally, frameworks such as FastFlow [5] and Skel [57] offer libraries that are composed of parallel patterns and skeletons for stream, task and data parallelism with efficient implementations for execution on heterogeneous platforms.

While there has been extensive work on skeleton-based parallel programming, the following issues still need to be addressed:

- The libraries available for practical use in programs contain operations that are defined over limited data types such as lists, arrays and binary trees. There are

practically no libraries that address generic (polytypic) data types such as n-ary trees to address irregular data structures, which could still be processed in parallel.

- Tree contraction, explained in Section 2.2.4, provides algorithms that can potentially lead to parallel evaluation of n-ary trees. However, their applicability to obtain parallel implementations for n-ary trees is yet to be extensively adopted.

A Parallel Compiler for SML

SkelML [78] is a compiler that parallelises higher-order functional programs. The compiler automatically extracts and exploits *map* and *fold* computations for execution on processor farms and processor trees, respectively. This is achieved by nesting the parallel skeletons in a processor topology that matches the structure of the Standard ML source. This work by Scaife et al. describes the analysis leading from a Standard ML input program through higher-order functions to an executable parallel program. Also described are the related runtime system and the execution model.

NESL

A nested data parallel language (NESL [8]) proposed by Blelloch provides a set of data parallel constructs (skeletons) that operate over sequences (ordered sets). NESL supports nested sequences and nested parallelism that allow multiple applications of a parallel function in parallel. This allows dealing with dynamically changing data structures such as graphs and sparse matrices. NESL also includes a cost model that allows calculating asymptotic running time for programs on various parallel machine models.

LVish

LVish [60] is a library that implements LVars [59] which allows creation of parallel programs that are deterministic by construction. This is required to ensure the correctness of the results that are computed by the parallel operations. LVars achieves this using monotonic data structures that allow data only to be added and never removed. Using a *par* monad, LVish allows encapsulation of determinism-preserving effects while allowing flexible communication among parallel tasks. As a result, LVish guarantees a deterministic behaviour of the parallel programs.

Glasgow Parallel Haskell

An extension of the Haskell language is Glasgow Parallel Haskell (GpH) [84] which allows thread-based semi-explicit parallelisation of a program. Using the constructs *par* and *pseq* to explicitly specify and control the parallelisable computations in a program, the threads are then automatically managed by the run-time system.

The *par* construct is used to specify the parallel evaluation of two expressions. For example, x '*par*' y indicates that x may be evaluated in parallel with y. The *pseq* construct is used to enforce the evaluation order between two expressions. For example, x '*pseq*' y causes x to be evaluated (to weak head normal form) before the evaluation of y. Further, using the *Eval* monad in Haskell, the *par* and *pseq* constructs are lifted to the *rpar* and *rseq* constructs, respectively. The following example illustrates how the *rpar* and *rseq* constructs can be used to parallelise a program to compute the Fibonacci series sum.

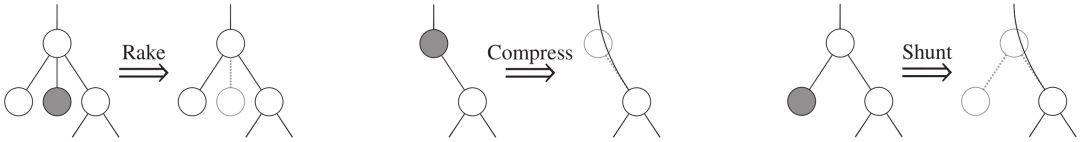
$$
\begin{array}{l}
\textbf{data}\ IntValue\ =\ Zero \mid Succ\ IntValue \\
fibSum\ ::\ IntValue \rightarrow Int \\
fibSum\ x \\
\textbf{where} \\
fibs\ Zero\ =\ 1 \\
fibs\ (Succ\ Zero)\ =\ 1 \\
fibs\ (Succ\ (Succ\ x))\ =\ runEval\ \$\ do \\
\qquad\qquad\qquad\qquad\qquad\qquad x' \leftarrow rpar\ (fibs\ x) \\
\qquad\qquad\qquad\qquad\qquad\qquad y' \leftarrow rseq\ (fibs\ (Succ\ x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad return\ (x' + y')
\end{array}
$$

Here, we observe that the evaluation of $fibs\ x$ may be performed in parallel with that of $fibs\ (Succ\ x)$ whose evaluation must be completed before $x' + y'$ is evaluated.

2.2.4 Tree Contraction for Program Parallelisation

Given a rooted tree, the objective of some tree computation algorithms is to reduce the given tree to a single node or value. A divide-and-conquer approach starts at the root and divides the tree into a number of sub-trees of nearly equal size for parallel computation on each. This is especially challenging when the given tree is unbalanced. On the other hand, tree contraction starts from the leaf nodes in the tree and performs the computation by propagating values of sub-computations to their parents, until they reach the root. This approach caters more naturally to the problem of efficiently parallelising the computation on unbalanced trees.

For instance, consider computations on binary trees. In the tree contraction based approach, an atomic operation called *shunt* is responsible for propagating the values of the sub-computations from the leaf nodes to their immediate parents. Each *shunt* operation is composed of a *rake* and a *compress* operation introduced by Miller & Reif [67], which can be applied in parallel to disjoint parts of the tree. The *rake* operation, shown in Figure 2.8a, removes the given leaf node, and the compress operation, shown in Figure 2.8b, is responsible for reducing the chain of nodes thereby produced by *rake*.



(a) Rake Operation (b) Compress Operation (c) Shunt Operation

Figure 2.8: Tree Contraction Operations – Rake, Compress and Shunt [70]

Consider a skewed binary tree, where each internal vertex has one child that is a leaf and one that is not. Here, after a parallel application of rake on all leaves, the result becomes a linear list. This is followed by a series of compress operations, which is not work-optimal. This can be overcome if each rake operation on a leaf is followed by a compress operation of its siblings. This combination of rake and compress is called a *shunt* operation. As shown in Figure 2.8c, when a shunt is performed on a leaf node, its parent is compressed and disconnected, resulting in its sibling taking its place.

However, since shunt disconnects the parent of the node it is applied to, there are some conditions under which unchecked parallel application of shunt could produce inconsistent results. Some of these conditions are: (1) shunt for children of the root; (2) shunt on two siblings simultaneously; (3) shunt on two adjacent leaves in left-to-right ordering; and (4) shunt on consecutive left and right odd-numbered leaves (when leaves are numbered from left to right). However, these can be resolved by ordering the leaves of the tree (for example, numbering them from left to right in depth-first order), and then systematically applying shunt in parallel. Abrahamson et al. [1] proposed the following procedure to safely perform shunt contraction of binary trees.

1. Number all leaves from left to right.
2. Perform shunt for all odd-numbered left leaves.
3. Perform shunt for all odd-numbered right leaves.

4. Halve all number of leaves.

5. Repeat step (2) until the tree consists of only one node.

This procedure was extended to n-ary trees by Morihata et al. [4] by defining an M-shunt operation, where each leaf in the tree is *marked*. A shunt contraction with marks (M-shunt) is an operation applied to an internal node whose children are unmarked nodes and the other marked leaves. An M-shunt operation removes the internal node and all its marked children and connects the unmarked child to the parent of the internal node. Figure 2.9 illustrates the behaviour of an M-shunt operation.

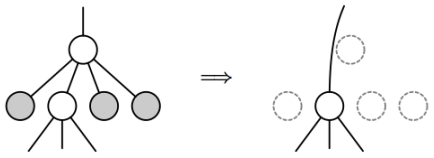


Figure 2.9: *M-shunt* Operation [4]

Using this M-shunt operation, the procedure for parallel contraction of n-ary trees is defined as follows:

1. Number all leaves from left to right.

2. Mark all odd-numbered leaves that have an unmarked right sibling, and apply the M-shunt operation to all possible nodes.

3. Mark all odd-numbered unmarked leaves, and apply the M-shunt operation to all possible nodes.

4. Remove the numbers of the marked leaves and halve those of the unmarked leaves.

5. Repeat step (2) until the tree consists of only one node.

This procedure was later formalised in [70] as the Rake-Shunt Contraction algorithm that is applicable to n-ary trees. It was also shown how this contraction algorithm can be used to implement a parallelisable tree reduction skeleton shown below, where *reduce* is applied to an n-ary tree where each *Node* is composed of a value v and a list of sub-trees.

$$\begin{aligned} reduce\ (Node\ v\ [])\ &=\ v \\ reduce\ (Node\ v\ (t:ts))\ &=\ v\ \otimes\ (foldl1\ (\oplus)\ (t:ts)) \end{aligned}$$

This *reduce* skeleton uses two binary operators – $\oplus$ to merge the values of two siblings, and $\otimes$ to calculate the contribution of the children (sub-trees). Assuming $\oplus$ to be associative, it can be proved that the implementation of *reduce* is parallelisable by performing the rake and compress computations on independent sub-trees simultaneously. Given a tree with N nodes and an EREW-PRAM (Exclusive-Read Exclusive-Write Parallel Random Access Machine) with P processors, [70] shows that the *reduce* skeleton finishes in $O(N/P + logN)$ time.

2.3 Parallelisation by Refactoring

The initial work on *paraforming* [13] proposed a framework that allows parallelisation of a functional program by guiding the developer through a number of code refactoring steps. This aims at enabling the programmer to parallelise a program without having to understand the exact syntax of the underlying programming language. The refactoring steps include *Introduce Data Parallelism* and *Introduce Task Parallelism* among others. The framework is capable of producing parallel programs defined using Glasgow Parallel Haskell and can be extended to use other variants such as Eden.

For example, consider the sequential definition *sumEuler* which computes the Euler totient of a given number, where function *mkList* for input n builds a list $[1 \ldots n]$.

$$sumEuler\ n\ =\ sum\ \circ\ map\ euler\ \circ\ mkList$$

Here, the function *euler* is mapped to each element in the list, the result of which is summed up using *sum* to compute the Euler totient. By applying the *Introduce Data Parallelism* and *Introduce Clustering* refactorings, paraforming can produce a parallel version of *sumEuler* as shown below.

$$\begin{aligned} sumEulerParListChunk\ c\ n\ =\ sum\ (map\ euler\ (mkList\ n) \\ `using` \\ parListChunk\ n\ rdeepseq) \end{aligned}$$

This definition performs parallel computations on chunks of elements in the input list. Here, for example, the existing *parListChunk* strategy is used by the *Introduce Clustering* refactoring to create chunks of size c, on each of which the *euler* function is mapped in parallel.

This work on paraforming was further extended to a language-independent parallel refactoring framework in [11] that can be specialised to work with languages including

Erlang, C and C++. This was achieved by providing the grammar and refactoring rules for the target language. By starting with a sequential program in one of the languages, an abstract syntax tree is built on which the user-driven refactoring rules are applied. The result is then pretty printed into a program in the target language. As discussed by the authors, while language-independence is powerful, there are a number of pitfalls. For instance, the refactoring rules need to encompass and address the semantics of each language that is targeted.

The paraforming approach forms the core of the *paraphrase* refactoring tool [12, 40] which aims to produce parallel programs using rewrite rules that are based on computation patterns. Computation patterns in the *paraphrase* project include *pipe*, *farm*, *map* and *reduce* among others. The workflow of refactoring a sequential program into a parallel version in paraphrase is as follows:

1. The initial program is parsed into an abstract syntax tree (AST) which includes static semantics such as use- and bind-locations for variables, and the types of variables and functions.

2. This AST is transformed into a *unified* AST to facilitate refactoring into different programming paradigms and languages supported by paraphrase.

3. The *unified* AST is then transformed into a *component* AST that allows expressing parameterised parallel computation patterns.

4. The *component* AST is refactored using the rewriting rules for the parallel computation patterns.

5. Finally, the resulting AST is then pretty printed into the source language which is the refactored program.

The computation patterns are expressed as high-level abstractions which allows their extension by the user. Also, the refactoring rules can be extended by adding rewriting rules in a general syntax along with pre- and post-conditions.

Refactoring programs is a technique that is powerful and extensively used. However, the workflow requires user guidance for good reason, which prevents complete automation of the program parallelisation process.

2.4 Summary

A detailed study of the existing program parallelisation approaches using program transformation techniques and skeleton-based approaches highlights a few gaps that exist.

- **Parallelisation using Program Transformation**: Even though some of the existing transformation techniques are powerful in producing parallel programs from a given sequential higher-order program, most of them address programs that operate over only one recursive input. Some of these techniques are designed for programs that are defined over certain data types such as lists or binary trees. Furthermore, they often require user input, such as manual derivation of operators that satisfy certain algebraic properties to allow parallel evaluation of the transformed programs. This prevents complete automation of the parallelisation process.

- **Parallelisation using Skeletons**: Parallel skeletons abstract away the complexity of parallel implementation from the developer by providing the skeletons and their efficient implementations through libraries. Even though skeleton-based parallel programming is extensively used, it still requires a huge amount of manual effort to identify potential parallel computations in a given program and then explicitly specify them using skeletons. Furthermore, most programs defined using skeletons often use intermediate data structures that are inefficient. For example, a hand-parallelised version of the matrix multiplication program from Example 1.1 is shown in Example 2.4. This parallel version uses the *map*, *reduce* and *zipWith* skeletons.

Example 2.4 (Hand-Parallelised Matrix Multiplication):

$$
\begin{array}{l}
mMul\ xss\ yss \\
\textbf{where} \\
\begin{array}{lcl}
mMul\ xss\ yss & = & map\ f\ xss \\
 & & \quad \textbf{where} \\
 & & \quad f\ xs = map\ (dotp\ xs)\ (transpose\ yss) \\
dotp\ xs\ ys & = & reduce\ (+)\ 0\ (zipWith\ (*)\ xs\ ys) \\
transpose\ xss & = & transpose'\ xss\ [] \\
transpose'\ []\ yss & = & yss \\
transpose'\ (xs : xss)\ yss & = & transpose'\ xss\ (rotate\ xs\ yss) \\
rotate\ xs\ yss & = & zipWith\ (\lambda x.\lambda ys.(ys +\!\!+ [x]))\ xs\ yss
\end{array}
\end{array}
$$

As we can observe, though defined using parallel skeletons, this definition 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 is free of intermediate data structures.

- **Parallelisation by Refactoring**: Refactoring is a powerful and flexible approach that is widely used for program restructuring and parallelisation. Through refactoring steps, which are often abstract, a number of programming languages are addressed allowing for transformation between languages of different paradigms. However, as discussed earlier, refactoring often requires user guidance as a part of the refactoring steps thus preventing complete automation.

Our Parallelisation Approach. Taking into account the above-mentioned limitations of both transformation-based and skeleton-based approaches to program parallelisation, we aim to create a transformation method that has the following attributes:

1. Reduces the use of intermediate data structures in a given program using an existing transformation technique called *distillation*.

2. Automatically transforms the distilled program by encoding all inputs into a single input whose structure reflects the algorithmic structure of the program. The resulting *encoded program* defined on the encoded input is more likely to contain instances of parallel skeletons.

3. Allows for the automatic identification of skeleton instances in the *encoded program* defined on encoded input.

In our approach, we require that the parallel programs that are produced by our transformation method also preserve the correctness of the results. For this, we consider strict evaluation of the parallel computations that are identified by our method as skeletons and their inputs. This will guarantee that the results of the transformed programs are the same as the results of the orignial programs.

In Chapter 3, we discuss the distillation transformation in detail that is used to reduce the intermediate data structures in a given program. In Chapter 4, we present the skeletons that are of interest to us in this research that can be parallelised and

a technique to automatically identify instances of these skeletons in a given program. In Chapter 5, we present our technique to transform a given program by encoding its inputs to facilitate identification of parallel skeletons. In Chapter 6, we then evaluate our transformation method by comparing our transformed programs and their efficiency against their hand-parallelised counterparts.

Chapter 3

Distillation

3.1 Introduction

The transformation technique presented in this thesis uses an existing transformation called *distillation* [38, 37] to reduce the number of intermediate data structures used in a given program. This is the first stage of the proposed solution illustrated in Figure 1.1. In this chapter, we present an overview of the distillation transformation in [38] based on more recent improvements proposed by its authors.

In Section 3.2, we present the language over which the existing distillation transformation is defined. In Section 3.3, we introduce the framework of labelled transition systems (LTS) used by distillation, followed by the transformation rules in Section 3.4. Further, the use of distillation in theorem proving [51] is discussed in Section 3.5.

3.2 Language for Distillation

The higher-order functional language used by the distillation transformation is presented in Definition 3.1. This language differs from the one used in the rest of this thesis in how pattern-matching is performed using **case**-expressions and functions are defined by assigning function bodies to function names in **where** expressions.

Definition 3.1 (Language Grammar for Distillation):

e	$::= x$	Variable
	$\mid c\ e_1 \ldots e_N$	Constructor Application
	$\mid e_0$ **where** $f_1 = e_1\ \ldots f_N = e_N$	**where**-expression
	$\mid f$	Function Call
	$\mid$ **case** e_0 **of** $p_1 \to e_1 \mid \ldots \mid p_K \to e_K$	**case** expression
	$\mid e_0\ e_1$	Application
	$\mid$ **let** $x = e_1$ **in** e_0	**let**–expression
	$\mid \lambda x.e$	λ–abstraction
p	$::= c\ x_1 \ldots x_N$	Pattern

A program in this language is an expression which can be a variable, constructor application, λ-abstraction, function call, application, **case**, **let** or **where**. Variables introduced by λ-abstractions, **case** patterns and **let**-expressions are *bound*, while all other variables are *free*. The patterns in **case**-expressions may not be nested and no variable may appear more than once within a pattern. The patterns in a **case**-expression must be non-overlapping and exhaustive.

We use this language in this chapter to present the distillation transformation as it is used by the authors of distillation to present their framework and transformation rules. Redefining the framework and transformation rules of distillation over the language used in this thesis requires a significant amount of complicated work. However, using the *pattern-matching compiler* proposed by Wadler [86], we can transform a program defined in the language used in the rest of this thesis (Definition 3.2) into the language used by distillation (Definition 3.1).

Definition 3.2 (Language Grammar):

e	$::= x$	Variable
	$\mid c\ e_1 \ldots e_N$	Constructor Application
	$\mid e_0$ **where** $d_1\ \ldots\ d_J$	**where**-expression
	$\mid f$	Function Call
	$\mid e_0\ e_1$	Application
	$\mid$ **let** $x = e_1$ **in** e_0	**let**–expression
	$\mid \lambda x.e$	λ–abstraction
d	$::= f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = e_1$	Function Definition
	$\vdots$	
	$f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = e_K$	
p	$::= x \mid c\ p_1 \ldots p_N$	Pattern

Consider the naïve reverse program *nrev* shown below which is defined in the language in Definition 1.1 used in this thesis.

$$
\begin{array}{l}
nrev\ xs \\
\textbf{where} \\
nrev\ [] \quad = [] \\
nrev\ (x : xs) \quad = app\ (nrev\ xs)\ [x] \\
app\ []\ ys \quad = ys \\
app\ (x : xs)\ ys \ = x : (app\ xs\ ys)
\end{array}
$$

This can be transformed by the *pattern-matching compiler* to the language in Definition 3.1 as follows:

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

3.3 Labelled Transition Systems

The distillation transformation uses the labelled transition system (LTS) framework, shown in Definition 3.3, to represent programs and define the rules for transformation.

Definition 3.3 (Labelled Transition System (LTS)):
A LTS for a given program is given by a 4-tuple $t = (\mathcal{E},\ e_0,\ Act,\ \rightarrow)$ where:

- $\mathcal{E}$ is the set of *states* of the LTS, each of which is an expression.
- $e_0 \in \mathcal{E}$ is the start state denoted by *root(t)*.
- *Act* is one of the following actions:
  - x, a free variable,
  - i, a bound variable with De Bruijn index i,
  - c, a constructor in an application or **case** expression pattern,
  - λ, a λ-abstraction,
  - @, the function in an application,
  - $\#i$, the i^{th} argument in an application,
  - **case**, a case selector,

  - **let**, a **let** variable,
  - **in**, a **let** body.

- $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is a transition relation, where $e \xrightarrow{\alpha} e'$ denotes a transition from state e to state e' via action α.

A LTS with no transitions is denoted by $\mathbf{0}$. We assume that bound variables are represented using De Bruijn indices. $fv(t)$ and $bv(t)$ are used to denote the free and bound variables respectively of LTS t. $e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_N, t_N)$ denotes a a LTS with root state e where $t_1, \ldots, t_N$ are the LTSs obtained by following the transitions labelled $\alpha_1, \ldots, \alpha_N$, respectively, from e.

Definition 3.4 (LTS Renaming):
$\sigma = \{x_1 \mapsto x'_1, \ldots, x_N \mapsto x'_N\}$ denotes a LTS renaming. Given an LTS t, $t\sigma = t\{x_1 \mapsto x'_1, \ldots, x_N \mapsto x'_N\}$ is the result of simultaneously replacing the free variables $x_1, \ldots, x_N$ with the corresponding variables $x'_1, \ldots, x'_N$, respectively, in the LTS t.

Definition 3.5 (LTS Substitution):
$\theta = \{x_1 \mapsto t_1, \ldots, x_N \mapsto t_N\}$ denotes a LTS renaming. Given an LTS t, $t\theta = t\{x_1 \mapsto t_1, \ldots, x_N \mapsto t_N\}$ is the result of simultaneously replacing the LTSs $x_1 \rightarrow (x_1, \mathbf{0}), \ldots x_N \rightarrow (x_N, \mathbf{0})$ with the corresponding variables $t_1, \ldots, t_N$, respectively, in the LTS t.

Definition 3.6 (Decorated LTS):
A decorated LTS is one in which decorations of the form $f : t$ or $x : t$ denote a function f or a variable x, respectively. A LTS with decoration f is denoted using $lts(f)$.

The LTS corresponding to a program e can be constructed by $\mathcal{L}[\![e]\!]$ using the rules $\mathcal{L}$ shown in Definition 3.7. Here, ρ is the set of previously encountered functions, and ϕ is the set of function definitions in scope. The first time a function is encountered, a LTS is generated using its definition decorated with the function name. If the same function is re-encountered, a transition is created to the LTS for this previous definition. Consequently, the LTS will always be a finite representation of the program.

Definition 3.7 (LTS Representation of Program):

$$\mathcal{L}[\![e]\!] = \mathcal{L}'[\![e]\!]\ \emptyset\ \emptyset$$

$$\begin{aligned}
\mathcal{L}'[\![e = x]\!]\ \rho\ \phi &= e \to (x, \mathbf{0}) \\
\mathcal{L}'[\![e = c\ e_1 \ldots e_N]\!]\ \rho\ \phi &= e \to (c, \mathbf{0}), (\#1, \mathcal{L}'[\![e_1]\!]\ \rho\ \phi), \ldots, (\#N, \mathcal{L}'[\![e_N]\!]\ \rho\ \phi) \\
\mathcal{L}'[\![e = \lambda x.e']\!]\ \rho\ \phi &= e \to (\lambda, \mathcal{L}'[\![e']\!]\ \rho\ \phi) \\
\mathcal{L}'[\![e = f]\!]\ \rho\ \phi &= \begin{cases} lts(f) & \text{, if } f \in \rho \\ f : \mathcal{L}'[\![\phi(f)]\!]\ (\rho \cup \{f\})\ \phi & \text{, otherwise} \end{cases} \\
\mathcal{L}'[\![e \textbf{ where } f_1 = e_1\ \ldots\ f_N = e_N]\!]\ \rho\ \phi & \\
\quad = \mathcal{L}'[\![e]\!]\ \rho\ (\phi \cup \{f_1 \mapsto e_1, \ldots f_N \mapsto e_N\}) & \\
\mathcal{L}'[\![e = \textbf{case } e_0 \textbf{ of } p_1 \to e_1 \mid \ldots \mid p_K \to e_K & \\
\quad = e \to (\textbf{case}, \mathcal{L}'[\![e_0]\!]\ \rho\ \phi), (c_1, \mathcal{L}'[\![e_1]\!]\ \rho\ \phi), \ldots, (c_K, \mathcal{L}'[\![e_K]\!]\ \rho\ \phi) & \\
\quad \text{where } p_k = c_k\ x_1 \ldots x_N & \\
\mathcal{L}'[\![e = e_0\ e_1]\!]\ \rho\ \phi &= e \to (@, \mathcal{L}'[\![e_0]\!]\ \rho\ \phi), (\#1, \mathcal{L}'[\![e_1]\!]\ \rho\ \phi) \\
\mathcal{L}'[\![\textbf{let } x = e_1 \textbf{ in } e_0]\!]\ \rho\ \phi &= \mathcal{L}'[\![e_0\{x \mapsto e_1\}]\!]\ \rho\ \phi
\end{aligned}$$

Conversely, a residual program can be extracted from a given LTS by $\mathcal{R}[\![l]\!]\ \emptyset$ using the rules $\mathcal{R}$ shown in Definition 3.8. Here, the parameter ρ contains the set of functions previously defined in a **where** expression. Upon re-encountering one of these functions, a corresponding function call is used in the residual program.

Definition 3.8 (Extraction of Residual Program from LTS):

$$\begin{aligned}
\mathcal{R}[\![f : t]\!]\ \rho &= \begin{cases} f\ x_1 \ldots x_N & \text{, if } f \in \rho \\ f\ x_1 \ldots x_N\ \textbf{where}\ f = \lambda x_1 \ldots \lambda x_N.(\mathcal{R}[\![t]\!]\ (\rho \cup \{f\})) & \text{, otherwise} \end{cases} \\
&\quad \text{where } \{x_1, \ldots, x_N\} = fv(t) \\
\mathcal{R}[\![x : t]\!]\ \rho &= x\ x_1 \ldots x_N \\
&\quad \text{where } \{x_1, \ldots, x_N\} = fv(t) \\
\mathcal{R}[\![e \to (x, \mathbf{0})]\!]\ \rho &= x \\
\mathcal{R}[\![e \to (i, \mathbf{0})]\!]\ \rho &= i \\
\mathcal{R}[\![e \to (c, \mathbf{0}), (\#1, t_1), \ldots, (\#N, t_N)]\!]\ \rho & \\
&\quad = c\ (\mathcal{R}[\![t_1]\!]\ \rho) \ldots (\mathcal{R}[\![t_N]\!]\ \rho) \\
\mathcal{R}[\![e \to (\lambda, t)]\!]\ \rho &= \lambda x.(\mathcal{R}[\![t]\!]\ \rho) \\
&\quad \text{where } x \text{ is fresh} \\
\mathcal{R}[\![e \to (@, t_0), (\#1, t_1)]\!]\ \rho &= (\mathcal{R}[\![t_0]\!]\ \rho)\ (\mathcal{R}[\![t_1]\!]\ \rho) \\
\mathcal{R}[\![e \to (\textbf{let}, x : t_0), (\textbf{in}, t_1)]\!]\ \rho &= \textbf{let } x = \lambda x_1 \ldots x_N.(\mathcal{R}[\![t_0]\!]\ \rho) \textbf{ in } (\mathcal{R}[\![t_1]\!]\ \rho) \\
&\quad \text{where } \{x_1, \ldots, x_N\} = fv(t_0) \\
\mathcal{R}[\![e \to (\textbf{case}, t_0), (c_1, t_1), \ldots, (c_K, t_K)]\!]\ \rho & \\
&\quad = \textbf{case } (\mathcal{R}[\![t_0]\!]\ \rho) \textbf{ of } p_1 \to (\mathcal{R}[\![t_1]\!]\ \rho) \mid \ldots \mid p_K \to (\mathcal{R}[\![t_K]\!]\ \rho) \\
&\quad \text{where } p_k = c_k\ x_1 \ldots x_N \\
&\qquad\quad c_k \text{ is of arity } N,\ x_1, \ldots, x_N \text{ are fresh.}
\end{aligned}$$

The LTS representation of the naïve reverse program *nrev xs* is illustrated in Figure 3.1.

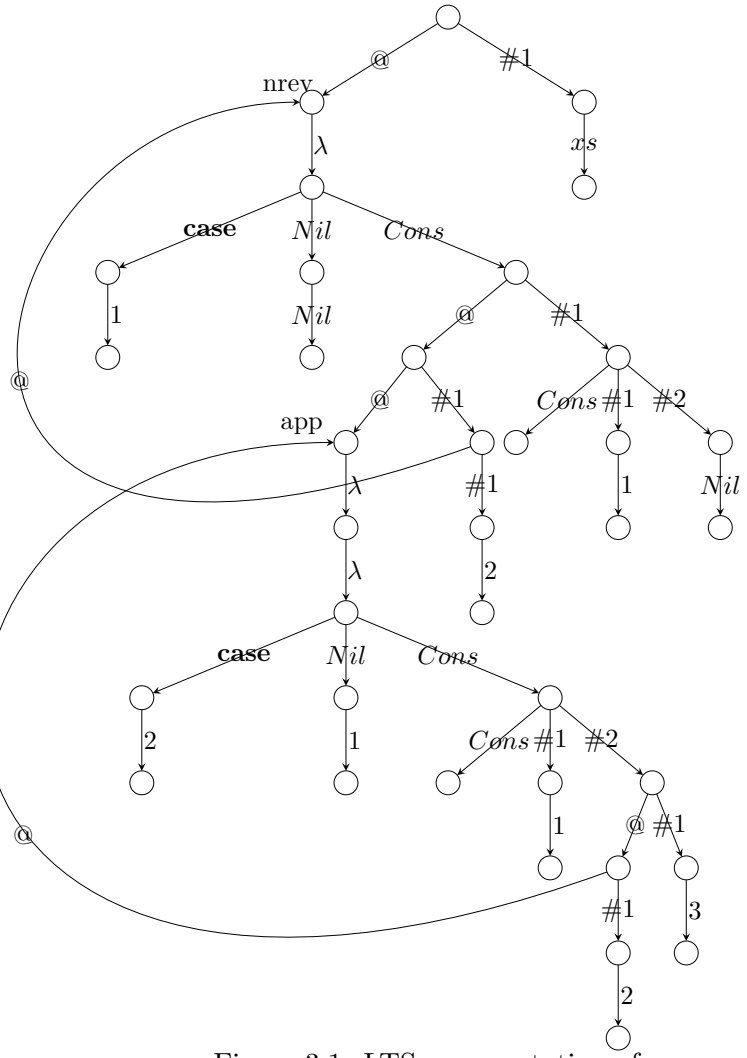


Figure 3.1: LTS representation of *nrev xs*

Definition 3.9 (Simulation):

A binary relation $\mathcal{R}' \subseteq \mathcal{E} \times \mathcal{E}'$ is a *simulation* of labelled transition system $t = (\mathcal{E}, e_0, Act, \rho)$ by $t' = (\mathcal{E}', e'_0, Act', \rho')$ if $(e_0, e'_0) \in \mathcal{R}'$, and for every pair $(e_i, e'_i) \in \mathcal{R}'$ the following holds:

$$\forall e_j \in \mathcal{E} \text{ s.t. } (e_i \xrightarrow{\alpha} e_j) \in \rho \cdot (\exists e'_j \in \mathcal{E}' \text{ s.t. } (e'_i \xrightarrow{\alpha} e'_j) \in \rho' \cdot (e_j, e'_j) \in \mathcal{R}')$$

Definition 3.10 (Bisimulation):

A *bisimulation* $\sim$ is a binary relation $\mathcal{R}'$ such that both $\mathcal{R}'$ and its inverse $\mathcal{R}'^{-1}$ are simulations.

Definition 3.11 (Embedding):

A binary relation $\mathcal{R}' \subseteq \mathcal{E} \times \mathcal{E}'$ is an *embedding* of labelled transition system $t = (\mathcal{E}, e_0, Act, \rho)$ in $t' = (\mathcal{E}', e'_0, Act', \rho')$ if $(e_0, e'_0) \in \mathcal{R}'$, and for every pair $(e_i, e'_i) \in \mathcal{R}'$ one of the following holds:

1. $\forall e_j \in \mathcal{E}$ s.t. $(e_i \xrightarrow{\alpha} e_j) \in \rho \cdot \big(\exists e'_j \in \mathcal{E}'$ s.t. $(e'_i \xrightarrow{\alpha} e'_j) \in \rho' \cdot (e_j, e'_j) \in \mathcal{R}'\big)$

2. $\exists e'_j \in \mathcal{E}'$ s.t. $(e'_i \xrightarrow{\alpha} e'_j) \in \rho' \cdot (e_i, e'_j) \in \mathcal{R}'$

The first rule here is a *coupling* rule, while the second one is a *diving* rule. Two states are related by coupling if the same transitions are possible from each of them and the resulting states are also related by the embedding relation. Two states are related by diving if a transition can be followed in the embedding LTS and the resulting state is related to the embedded LTS state by the embedding relation. We write $t \lesssim t'$ if LTS t is coupled with LTS t'.

Definition 3.12 (Generalisation of LTSs):

$$t \sqcap t' = \begin{cases} \Big(e \to (\alpha_1, t''_1), \ldots, (\alpha_N, t''_N),\ \bigcup_{n=1}^{N} \theta_n,\ \bigcup_{n=1}^{N} \theta'_n\Big), & \text{if } t \lesssim t' \\ \text{where } t = e \to (\alpha_1, t_1), \ldots, (\alpha_N, t_N) & \\ \qquad t' = e' \to (\alpha_1, t'_1), \ldots, (\alpha_N, t'_N) & \\ \qquad \forall n \in \{1 \ldots N\} \cdot t_n \sqcap t'_n = (t''_n, \theta_n, \theta'_n) & \\ (x, \{x \mapsto t\}, \{x \mapsto t'\}), & \text{otherwise} \\ \text{where } x \text{ is fresh} & \end{cases}$$

The result of generalising two LTSs t and t' is a triple (t'', θ, θ'') where t'' is the generalised LTS, and θ and θ' are substitutions such that $t''\theta \sim t$ and $t''\theta' \sim t'$. Within these rules, if both LTSs have the same transitions at the top-level, then these will be the transitions at the top-level of the resulting generalised LTS, and the corresponding LTS components which are the targets of these transitions are further generalised. Unmatched LTS components are generalised by introducing a new generalisation variable x where the value of this variable is the unmatched LTS component.

The result of generalisation is made into a nested **let** using $\theta \lhd t$ for substitution θ and LTS t, which is defined as follows:

$$\begin{aligned} \emptyset \lhd t &= t \\ (\{x \mapsto t'\} \cup \theta) \lhd t &= root(t) \to (\mathbf{let}, x : t'), (\mathbf{in}, \theta \lhd t) \end{aligned}$$

Definition 3.13 (Shallow Reduction Context):

A shallow reduction context $\mathcal{R}'$ is a LTS containing a single hole $\bullet$ in the place of the redex, which can have one of the two following possible forms:

$$\mathcal{R}' ::= e \to (@, \bullet), (\#1, t) \mid e \to (\mathbf{case}, \bullet), (c_1, t_1), \ldots, (c_K, t_K)$$

Definition 3.14 (Evaluation Context):

An evaluation context $\mathcal{E}$ is represented as a sequence of shallow reduction contexts

(known as a zipper [46]), representing the nesting of these contexts from innermost to outermost within which the redex is contained. An evaluation context can therefore have one of the two following possible forms: $\mathcal{E} ::= \langle\rangle \mid \langle \mathcal{R}' : \mathcal{E} \rangle$

Definition 3.15 (Insertion into Evaluation Context):
The insertion of a LTS t into an evaluation context κ, denoted by $\kappa \bullet t$, is defined as follows:

$$\begin{array}{ll} \langle\rangle \bullet t & = t \\ \langle (e \rightarrow (@, \bullet), (\#1, t')) : \kappa \rangle \bullet t & = \kappa \bullet (e \rightarrow (@, t), (\#1, t')) \\ \langle (e \rightarrow (\mathbf{case}, \bullet), (c_1, t_1), \ldots, (c_K, t_K)) : \kappa \rangle \bullet t & = \kappa \bullet (e \rightarrow (\mathbf{case}, t), (c_1, t_1), \ldots, (c_K, t_K)) \end{array}$$

Free variables within the LTS t may become bound within $\kappa \bullet t$; it is assumed that the De Bruijn indices of all bound variables are updated accordingly.

3.4 The Transformation

In this section, we present the distillation algorithm within the labelled transition system framework. Distillation takes as its input the LTS representation of the original program and produces as its output a transformed LTS which can be residualised into a distilled program. The LTS resulting from the transformation of the LTS representation t of a program is given by $\mathcal{D}[\![t]\!]\ \langle\rangle\ \emptyset\ \emptyset$. Here, $\mathcal{D}$ are the distillation rules shown in Definition 3.16 that are defined on a LTS and its surrounding context denoted by κ. The parameter ρ contains the LTS representations of the terms previously extracted by the generalisation steps and ϕ contains the LTS representations of the memoised functions.

The distillation rules $\mathcal{D}$ presented in Definition 3.16 perform a normal-order reduction on the LTS representation of the input program. The rules $\mathcal{D}'$ are applied when this normal-order reduction becomes 'stuck' as a result of encountering a variable in a redex position. In this case, the context surrounding the redex is further transformed using the distillation rules $\mathcal{D}$.

The distillation transformation memoises previously encountered functions by adding them to ϕ. If the LTS representation of the current function is a renaming of a memoised one, then a transition is created back to the previous one. If the LTS representation of the current function is an embedding of a memoised one, then generalisation is performed. The resulting generalised LTS is then transformed separately before substituting the extracted components back into it and further transforming. The components extracted by generalisation are also memoised by adding them to ρ. If an extracted component

is a renaming of a memoised one, then a transition is created back to the previous one. If the extracted component is an embedding of a memoised one, then generalisation is performed and further components are extracted. The transformation continues until either a renaming is found of a previously encountered function, or no new extracted components have been memoised since the previous occurrence of a function embedding.

Definition 3.16 (Distillation Rules):

$$\mathcal{D}[\![f : t]\!]\ \kappa\ \rho\ \phi = \begin{cases} \sigma \lhd lts(f') & \text{, if } \exists (f' : t', \rho') \in \phi, \sigma \cdot (t'\sigma) \sim (\kappa \bullet t) \\ \mathcal{D}[\![\rho \lhd t'']\!]\ \langle\rangle\ \rho\ \phi \text{ where } (\kappa \bullet t) \sqcap t' = (t'', \theta, \theta') & \\ & \text{, if } \exists (f' : t', \rho') \in \phi, \sigma \cdot (t'\sigma) \lesssim (\kappa \bullet t) \wedge \rho \neq \rho' \\ \kappa \bullet (f : t) & \text{, if } \exists (f' : t', \rho') \in \phi, \sigma \cdot (t'\sigma) \lesssim (\kappa \bullet t) \wedge \rho = \rho' \\ f' : \mathcal{D}[\![t]\!]\ \kappa\ \rho\ (\phi \cup \{f' : \kappa \bullet t\}) \text{ where } f' \text{ is fresh} & \\ & \text{, otherwise} \end{cases}$$

$$\mathcal{D}[\![x : t]\!]\ \langle\rangle\ \rho\ \phi = x : t$$

$$\mathcal{D}[\![x : t]\!]\ \kappa\ \rho\ \phi = \mathcal{D}[\![t]\!]\ \kappa\ \rho\ \phi$$

$$\mathcal{D}[\![e \to (x, \mathbf{0})]\!]\ \langle\rangle\ \rho\ \phi = e \to (x, \mathbf{0})$$

$$\mathcal{D}[\![e \to (x, \mathbf{0})]\!]\ \langle (e' \to (\mathbf{case}, \bullet), (c_1, t_1), \ldots, (c_K, t_K)) : \kappa \rangle\ \rho\ \phi$$
$$= e' \to (\mathbf{case}, e \to (x, \mathbf{0})), (c_1, t'_1), \ldots, (c_K, t'_K)$$
$$\text{where } c_k \text{ is of arity } k$$
$$t'_k = \mathcal{D}[\![(\kappa \bullet t_k)\{x \mapsto t''_k\}]\!]\ \kappa\ \rho\ \phi$$
$$t''_k = \mathcal{L}[\![c_k\ 1 \ldots K]\!]$$

$$\mathcal{D}[\![e \to (x, \mathbf{0})]\!]\ \langle ((e' \to (@, \bullet)), (\#1, t)) : \kappa \rangle\ \rho\ \phi$$
$$= \mathcal{D}'[\![e \to (x, \mathbf{0})]\!]\ \langle ((e' \to (@, \bullet)), (\#1, t)) : \kappa \rangle\ \rho\ \phi$$

$$\mathcal{D}[\![e \to (c, \mathbf{0}), (\#1, t_1), \ldots, (\#N, t_N)]\!]\ \langle\rangle\ \rho\ \phi$$
$$= e \to (c, \mathbf{0}), (\#1, \mathcal{D}[\![t_1]\!]\ \langle\rangle\ \rho\ \phi), \ldots, (\#N, \mathcal{N}[\![t_N]\!]\ \langle\rangle\ \rho\ \phi)$$

$$\mathcal{D}[\![e \to (c, \mathbf{0}), (\#1, t_1), \ldots, (\#N, t_N)]\!]\ \langle (e' \to (\mathbf{case}, \bullet), (c_1, t'_1), \ldots, (c_K, t'_K)) : \kappa \rangle\ \rho\ \phi$$
$$= \mathcal{D}[\![t'_k\{1 \mapsto t_1, \ldots, N \mapsto t_N\}]\!]\ \kappa\ \rho\ \phi \text{ where } c = c_k$$

$$\mathcal{D}[\![e \to (\lambda, t)]\!]\ \langle\rangle\ \rho\ \phi = e \to (\lambda, \mathcal{D}[\![t]\!]\ \langle\rangle\ \rho\ \phi)$$

$$\mathcal{D}[\![e \to (\lambda, t)]\!]\ \langle ((e' \to (@, \bullet)), (\#1, t')) : \kappa \rangle\ \rho\ \phi$$
$$= \mathcal{D}[\![t\{1 \mapsto t'\}]\!]\ \kappa\ \rho\ \phi$$

$$\mathcal{D}[\![e \to (@, t), (\#1, t')]\!]\ \kappa\ \rho\ \phi$$
$$= \mathcal{D}[\![t]\!]\ \langle (e \to (@, \bullet), (\#1, t')) : \kappa \rangle\ \rho\ \phi$$

$$\mathcal{D}[\![e \to (\mathbf{case}, t_0), (c_1, t_1), \ldots, (c_K, t_K)]\!]\ \kappa\ \rho\ \phi$$
$$= \mathcal{D}[\![t_0]\!]\ \langle (e \to (\mathbf{case}, \bullet), (c_1, t_1), \ldots, (c_K, t_K)) : \kappa \rangle\ \rho\ \phi$$

$$\mathcal{D}[\![e \to (\mathbf{let}, x : t_0), (\mathbf{in}, t_1)]\!]\ \kappa\ \rho\ \phi$$
$$= \begin{cases} \mathcal{D}[\![t_1\{x \mapsto x' : lts(x')\sigma\}]\!]\ \kappa\ \rho\ \phi & \text{, if } \exists x' : t'_0 \in \rho, \sigma \cdot (t'_0\sigma) \sim t_0 \\ \mathcal{D}[\![\theta \lhd e \to (\mathbf{let}, x : t''_0), (\mathbf{in}, t_1)]\!]\ \kappa\ \rho\ \phi & \\ \text{where } t_0 \sqcap t'_0 = (t''_0, \theta, \theta') & \text{, if } \exists x' : t'_0 \in \rho, \sigma \cdot (t'_0\sigma) \lesssim t_0 \\ e \to (\mathbf{let}, x : t_0), (\mathbf{in}, \mathcal{D}[\![t'_1\{x \mapsto x : t_0\}]\!]\ \langle\rangle\ (\rho \cup \{x : t_0\})\ \phi) & \\ \text{where } t'_1 = \mathcal{D}[\![t_1]\!]\ \kappa\ \rho\ \phi & \text{, otherwise} \end{cases}$$

$$\mathcal{D}'[\![t]\!]\ \langle\rangle\ \rho\ \phi\ =\ t$$
$$\mathcal{D}'[\![t]\!]\ \langle(e \to (@, \bullet), (\#1, t')) : \kappa\rangle\ \rho\ \phi$$
$$\quad =\ \mathcal{D}'[\![e \to (@, t), (\#1, \mathcal{D}[\![t']\!]\ \langle\rangle)]\!]\ \kappa\ \rho\ \phi$$
$$\mathcal{D}'[\![t]\!]\ \langle(e' \to (\mathbf{case}, \bullet), (c_1, t_1), \ldots, (c_K, t_K)) : \kappa\rangle\ \rho\ \phi$$
$$\quad =\ e \to (\mathbf{case}, t), (c_1, \mathcal{D}[\![t_1]\!]\ \kappa\ \rho\ \phi), \ldots, (c_K, \mathcal{D}[\![t_K]\!]\ \kappa\ \rho\ \phi)$$

The distillation transformation is able to achieve improvements better than the positive supercompilation transformation described in Section 2.1.1. This is because while positive supercompilation uses the generalisation strategy to extract components which create mismatches, the resulting generalised terms are transformed separately which may result in intermediate data structures. However, in distillation the resulting generalised terms are transformed separately, before being re-combined with the extracted sub-terms and further transformed. Consequently, subsequent generalisation takes place with respect to transformed terms which will contain fewer intermediate data structures.

3.4.1 Distilled Form

As a result of distillation, all transformed programs produced are in the *distilled form* $de^{\emptyset}$. The grammar of a distilled expression de^{ρ} is shown in Definition 3.17. Here, ρ is the set of variables introduced by **let**-expressions. No **let**-variables appear as function arguments that are pattern-matched in the distilled form.

Definition 3.17 (Distilled Language Grammar):

de^{ρ} ::=	$x\ de_1^{\rho} \ldots de_N^{\rho}$	Variable Application
|	$c\ de_1^{\rho} \ldots de_N^{\rho}$	Constructor Application
|	de_0^{ρ} **where** $f_1 = de_1^{\rho}\ \ldots\ f_N = de_N^{\rho}$	**where**–Expression
|	$f\ x_1 \ldots x_N$	Function Application
|	**case** $(x\ de_1^{\rho} \ldots de_N^{\rho})$ **of** $p_1 \to de_1^{\rho} \mid \ldots \mid p_K \to de_K^{\rho}$, $x \notin \rho$	**case** Expression
|	**let** $x = de_1^{\rho}$ **in** $de_0^{\rho \cup \{x\}}$	**let**–Expression
|	$\lambda x.de^{\rho}$	λ–Abstraction
p ::=	$c\ x_1 \ldots x_N$	Pattern

While supercompilation yields an unchanged program for naïve reverse (*nrev xs* introduced in Section 3.2), the result of distillation is as follows and is free of intermediate data structures:

$$\begin{array}{l} arev\ xs \\ \mathbf{where} \\ arev\ =\ \lambda xs.\mathbf{let}\ ys = []\ \mathbf{in}\ arev'\ xs\ ys \\ arev'\ =\ \lambda xs.\lambda ys.\mathbf{case}\ xs\ \mathbf{of} \\ \qquad\qquad\qquad [] \quad\ \to\ ys \\ \qquad\qquad\qquad (x : xs)\ \to\ \mathbf{let}\ ys' = (x : ys)\ \mathbf{in}\ arev'\ xs\ ys' \end{array}$$

In order to use the distillation transformation in conjunction with the rest of the transformations proposed in this thesis, we need to transform the distilled programs in the language shown in Definition 3.17 into the language shown in Definition 3.18 that fits the language in the rest of this thesis.

Definition 3.18 (Distilled Language Grammar):

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

This transformation is straightforward, and is essentially the reverse of Wadler's pattern-matching compiler, as shown for the distilled naïve reverse program. The definition of the *arev xs* program can be transformed to the language in Definition 3.18 as shown below:

$$
\begin{array}{lcl}
arev\ xs & & \\
\textbf{where} & & \\
arev\ xs & = & \textbf{let } ys = [] \textbf{ in } arev'\ xs\ ys \\
arev'\ []\ ys & = & ys \\
arev'\ (x : xs)\ ys & = & \textbf{let } ys' = (x : ys) \textbf{ in } arev'\ xs\ ys'
\end{array}
$$

Correctness of the distillation transformation can be proved by induction over the rules $\mathcal{D}$ in Definition 3.16 to show that the LTS produced by the distillation rules is observationally equivalent to the LTS of the original program. The proofs of correctness and the potential super-linear speedup that can be achieved by distillation are presented in more detail in [38].

3.5 Theorem Proving

The distillation transformation presented in Section 3.4 can be used to facilitate theorem proving using Poitín [39, 51], an existing theorem prover presented in this section. This is achieved by formulating the conjecture to be proved as a program in the language in Definition 3.1 and the theorem prover returns a truth value of the following data type:

$$\textbf{data}\ \mathit{TruthVal} = \mathit{True} \mid \mathit{False} \mid \mathit{Undefined}$$

The rules of the Poitín theorem prover are presented in Definition 3.19. The result of applying these rules uses a Kleene three-valued logic where the proof rules always return an answer. The proof rules return *True* if the conjecture is proved to be true, *False* if it is proved to be false, and *Undefined* in the case of possible non-termination of the input program. Consequently, the proof rules check for the termination of a given distilled program. These rules attempt to prove that an expression in distilled form constitutes a *cyclic pre-proof* as defined in [9], the evaluation of which is recursively progressing towards termination, thus making it a *cyclic proof*. As shown in [9], this will be the case if, between every recursive call of a function, at least one argument of the function is pattern-matched. If this is not the case, then the proof attempt is abandoned as it could otherwise diverge and result in an undefined answer.

Definition 3.19 (Poitín Proof Rules):

$$\mathcal{P}[\![x\ e_1 \ldots e_N]\!]\ \alpha\ \gamma\ \rho\ \phi = \begin{cases} True & \text{, if } x \in \alpha \\ False & \text{, otherwise} \end{cases}$$

$$\mathcal{P}[\![c\ e_1 \ldots e_N]\!]\ \alpha\ \gamma\ \rho\ \phi = \begin{cases} True & \text{, if } c = True \\ False & \text{, otherwise} \end{cases}$$

$$\mathcal{P}[\![\lambda x.e]\!]\ \alpha\ \gamma\ \rho\ \phi = \mathcal{P}[\![e]\!]\ \alpha\ \gamma\ \rho\ \phi$$

$$\mathcal{P}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_0]\!]\ \alpha\ \gamma\ \rho\ \phi = (\mathcal{P}[\![e_1]\!]\ \alpha\ \gamma\ \rho) \wedge (\mathcal{P}[\![e_0]\!]\ (\alpha \cup \{x\})\ \gamma\ \rho)$$

$$\mathcal{P}[\![e_0\ \textbf{where}\ f_1 = e_1 \ldots f_N = e_N]\!]\ \alpha\ \gamma\ \rho\ \phi = \mathcal{P}[\![e_0]\!]\ \alpha\ \gamma\ \rho\ (\phi \cup \{f_1 \mapsto e_1, \ldots, f_N \mapsto e_N\})$$

$$\mathcal{P}[\![\textbf{case}\ (x\ e_1 \ldots e_N)\ \textbf{of}\ p_1 \to e'_1 \mid \ldots \mid p_K \to e'_K]\!]\ \alpha\ \gamma\ \rho\ \phi = \begin{cases} \bigwedge_{k=1}^{K} (\mathcal{P}[\![e'_k]\!]\ \alpha\ (\gamma \cup \{x\})\ \rho\ \phi) & \text{, if } n = 0 \\ \bigwedge_{k=1}^{K} (\mathcal{P}[\![e'_k]\!]\ \alpha\ \gamma\ \rho\ \phi) & \text{, otherwise} \end{cases}$$

$$\mathcal{P}[\![f\ x_1 \ldots x_N]\!]\ \alpha\ \gamma\ \rho\ \phi = \begin{cases} True & \text{, if } \exists (f\ x'_1 \ldots x'_N) \in \rho, n \in \{1, \ldots, N\} \cdot x'_n \in \gamma \\ Undefined & \text{, if } \exists (f\ x'_1 \ldots x'_N) \in \rho \cdot \forall n \in \{1, \ldots, N\} \cdot x'_n \notin \gamma \\ \mathcal{P}[\![e\{x'_1 \mapsto x_1, \ldots, x'_N \mapsto x_N\}\ \alpha\ \gamma\ (\rho \cup \{(f\ x_1 \ldots x_N)\})\ \phi]\!] & \text{, otherwise} \\ \text{where} & \\ f = \lambda x'_1 \ldots \lambda x'_N.e & \end{cases}$$

Within the rules $\mathcal{P}$, α is the set of **let** variables whose values are known to be *True*, γ is the set of variables which are pattern-matched, ρ is the set of previously encountered function calls and ϕ is the set of function definitions that are in scope. These rules can be explained as follows:

- For a variable, if the variable is contained in α, then it is a **let** variable whose value is known to be *True*. Otherwise, the value *False* is returned.

- For a constructor, if the constructor is *True*, then the value *True* is returned. Otherwise, the value *False* is returned.

- For a λ-abstraction, the proof rules are applied on the λ-body and the parameter is considered to be an unknown variable.

- For a **let**-expression, the proof rules are applied to the extracted expression. If the result is *True*, then the associated **let** variable is added to α in the proof of the **let** body.

- For a **where**-expression, the function definitions are added to ϕ.

- For a function call, if the function was already encountered before and if one of the arguments in the previous call is contained in γ, then it has been pattern-matched, so the function is progressing and the value *True* is returned. If none of the arguments of the previous call are contained in γ, then the value *Undefined* is returned because the function is potentially non-terminating. If a call to the function has not been encountered before, then the proof rules are applied to the function bodies by adding the pattern-matched arguments to γ.

The soundness of the theorem proving rules $\mathcal{P}$ are discussed in [50].

Example 3.1 presents the conjecture to check the associativity of the addition operator *add* formulated as a program. Here, *eqNum* tests the equality of two values of type *IntValue*.

Example 3.1 (Associativity of Addition Operator):

$$
\begin{array}{l}
\textbf{data}\ IntValue\ a = Zero \mid Succ\ a \\
eqNum\ (add\ (add\ x\ y)\ z)\ (add\ x\ (add\ y\ z)) \\
\textbf{where} \\
eqNum\ =\ \lambda x.\lambda y.\textbf{case}\ x\ \textbf{of} \\
\qquad\qquad Zero\ \rightarrow\ \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad Zero\ \rightarrow\ True \\
\qquad\qquad\qquad\qquad Succ\ y\ \rightarrow\ False \\
\qquad\qquad Succ\ x\ \rightarrow\ \textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad Zero\ \rightarrow\ True \\
\qquad\qquad\qquad\qquad Succ\ y\ \rightarrow\ eqNum\ x\ y \\
add\ \ =\ \lambda x.\lambda y.\textbf{case}\ x\ \textbf{of} \\
\qquad\qquad Zero\ \rightarrow\ y \\
\qquad\qquad Succ\ x\ \rightarrow\ Succ\ (add\ x\ y)
\end{array}
$$

Example 3.2 presents the distilled version of the program in Example 3.1.

Example 3.2 (Associativity of Addition Operator - Distilled Program):

$$
\begin{array}{l}
f_1\ x\ y\ z \\
\textbf{where} \\
f_1\ =\ \lambda x.\lambda y.\lambda z.\textbf{case}\ x\ \textbf{of} \\
\qquad\qquad Zero\ \ \ \rightarrow\ f_2\ y\ z \\
\qquad\qquad\qquad\ \ \textbf{where} \\
\qquad\qquad\qquad\ \ f_2\ =\ \lambda y.\lambda z.\textbf{case}\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Zero\ \ \ \rightarrow\ f_3\ z \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ f_3\ =\ \lambda z.\textbf{case}\ z\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Zero\ \ \ \rightarrow\ True \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Succ\ z'\ \rightarrow\ f_3\ z' \\
\qquad\qquad\qquad\qquad\qquad\qquad Succ\ y'\ \rightarrow\ f_2\ y'\ z \\
\qquad\qquad Succ\ x'\ \rightarrow\ f_1\ x'\ y\ z
\end{array}
$$

The step-wise illustration of applying the Poitín proof rules to the distilled program in Example 3.2 is presented in Example 3.3 where the associativity predicate for *add* is proved to be true.

Example 3.3 (Proof of Associativity of Addition Operator):

$$
\begin{array}{l}
\quad \mathcal{P}[\![f_1\ x\ y\ z]\!]\ \{\}\ \{\}\ \{\}\ \phi \\
= \mathcal{P}[\![\textbf{case}\ x\ \textbf{of}\ Zero \rightarrow f_2\ y\ z \mid Succ\ x' \rightarrow f_1\ x'\ y\ z]\!]\ \{\}\ \{\}\ \{f_1\ x\ y\ z\}\ \phi \\
= (\mathcal{P}[\![f_2\ y\ z]\!]\ \{\}\ \{x\}\ \{f_1\ x\ y\ z\}\ \phi) \wedge (\mathcal{P}[\![f_1\ x'\ y\ z]\!]\ \{\}\ \{x\}\ \{f_1\ x\ y\ z\}\ \phi) \\
= (\mathcal{P}[\![f_2\ y\ z]\!]\ \{\}\ \{x\}\ \{f_1\ x\ y\ z\}\ \phi) \wedge True \\
= \mathcal{P}[\![\textbf{case}\ y\ \textbf{of}\ Zero \rightarrow f_3\ z \mid Succ\ y' \rightarrow f_2\ y'\ z]\!]\ \{\}\ \{x\}\ \{f_1\ x\ y\ z, f_2\ y\ z\}\ \phi \\
= (\mathcal{P}[\![f_3\ z]\!]\ \{\}\ \{x, y\}\ \{f_1\ x\ y\ z, f_2\ y\ z\}\ \phi) \wedge (\mathcal{P}[\![f_2\ y'\ z]\!]\ \{\}\ \{x, y\}\ \{f_1\ x\ y\ z, f_2\ y\ z\}\ \phi) \\
= (\mathcal{P}[\![f_3\ z]\!]\ \{\}\ \{x, y\}\ \{f_1\ x\ y\ z, f_2\ y\ z\}\ \phi) \wedge True \\
= \mathcal{P}[\![\textbf{case}\ z\ \textbf{of}\ Zero \rightarrow True \mid Succ\ z' \rightarrow f_3\ z']\!]\ \{\}\ \{x, y\}\ \{f_1\ x\ y\ z, f_2\ y\ z, f_3\ z\}\ \phi \\
= (\mathcal{P}[\![True]\!]\ \{\}\ \{x, y, z\}\ \{f_1\ x\ y\ z, f_2\ y\ z, f_3\ z\}\ \phi) \wedge \\
\quad (\mathcal{P}[\![f_3\ z']\!]\ \{\}\ \{x, y, z\}\ \{f_1\ x\ y\ z, f_2\ y\ z, f_3\ z\}\ \phi) \\
= True \wedge (\mathcal{P}[\![f_3\ z']\!]\ \{\}\ \{x, y, z\}\ \{f_1\ x\ y\ z, f_2\ y\ z, f_3\ z\}\ \phi) \wedge True \\
= True \wedge True \\
= True
\end{array}
$$

3.6 Summary

In this chapter, we have presented in detail an existing transformation called *distillation* [38] which is based on the unfold/fold transformation steps and uses a labelled transformation systems (LTS) framework to define the transformation steps. Distillation can reduce the number of intermediate data structures in a given program and potentially lead to super-linear speedups [38].

As illustrated in Figure 1.1, we use distillation in the first stage of the parallelisation transformation method proposed in this thesis. This stage of the transformation is applied on a given sequential program that may contain inefficient intermediate data

structures. By applying the distillation transformation, we obtain a distilled program that potentially contains fewer intermediate data structures. Following this, our objective is to identify parallel computations in the distilled program. As discussed in Section 3.5, distillation can also be used for automated theorem proving using the Poitín prover. This allows automatic verification of operator properties that was illustrated by testing the associativity of an addition operator. We discuss the application of the theorem prover in the context of the parallelisation transformation presented in this thesis in Chapter 4.

In the following Chapter 4, we discuss a technique for automatic identification of parallel computations in a given program. In Chapter 5, we present a data type transformation technique that facilitates identification of parallel computations in distilled programs. These two techniques form the core of the parallelisation transformation method presented in this thesis in addition to the distillation transformation.

Chapter 4

Parallelisation Using Skeletons

4.1 Introduction

In order to support our research hypothesis "*Program transformation can be used to automatically identify parallel computations in a given program, potentially leading to its efficient parallel execution*" stated in Section 1.2, our first objective is to automatically transform a given sequential program into a version that contains fewer intermediate data structures. This is because, as explained in Section 1.3.1, the repeated construction and decomposition of intermediate data structures in a program is a potential source of inefficiency. We reduce the intermediate data structures in a program using the distillation transformation described in Chapter 3.

Following this, we present a method in this chapter to analyse a given program and automatically identify potential parallel computations [56, 53, 52] as a solution to our first research question "***RQ-1:*** *How can potential parallel computations in a program be automatically identified?*". To achieve this, we encapsulate parallel computations as algorithmic skeletons. Using parallel implementations for the skeletons identified by our method, it is then possible to efficiently execute the program on parallel hardware. This provides a solution to our second research question "***RQ-2:*** *How can the transformed program be executed in parallel?*". As a result, we can produce a parallel version of a given program that contains fewer intermediate data structures and is defined using parallel skeletons.

Removing intermediate data structures from a program results in fusing computations that could potentially be evaluated in parallel. This presents an interesting challenge on judiciously removing intermediate data structures while preserving compu-

tations that can are free of data dependency and therefore evaluated in parallel. The transformation method presented in this thesis aims at identifying parallel skeletons in a distilled program aided by a data type transformation (presented later in Chapter 5). It would also be interesting to study an approach that eliminates intermediate data structures using distillation in a program that is defined using parallel skeletons. This is addressed in the section on further work in Chapter 7 with an example.

The remainder of this chapter is structured as follows: In Section 4.2, we discuss the skeletons that are of interest to us in the context of this thesis, and in Section 4.3, we discuss parallel implementations of the skeletons. In Section 4.4, we present our method to automatically identify skeletons in a given program. In Section 4.5, we discuss existing works that address program parallelisation by identifying potential parallel computations. In Section 4.6, we summarise and discuss the context that poses our third research question – "***RQ-3:*** *How can a given program be transformed to aid identification of parallel computations?*" – and present its solution in Chapter 5.

4.2 Parallel Skeletons

In this thesis, we are interested in identifying parallel computations that can be defined using algorithmic skeletons. In particular, we are interested in computations that can be defined using skeletons that model the map- and reduce-based operations – *map*, *reduce*, *map-reduce* and *accumulate*. This is because these skeletons are versatile and widely applicable in parallel programming. The *map*, *reduce*, *map-reduce* and *accumulate* skeletons are defined and can be parallelised as discussed in Section 2.2.3.

4.2.1 Parallel Reduction

It is important to note that the parallel execution of a reduction computation is contingent on the associativity and strictness of the reduction operator. For example, consider the following reduction expression defined using a binary operator $\oplus$.

$$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6$$

This expression can be evaluated in parallel by paranthesising it in different ways such as the following, where each sub-expression can be executed independently:

$$(x_1 \oplus x_2) \oplus (x_3 \oplus x_4) \oplus (x_5 \oplus x_6)$$
$$\big(x_1 \oplus (x_2 \oplus x_3)\big) \oplus \big((x_4 \oplus x_5) \oplus x_6\big)$$

Such parallelisation of the reduction is possible only if the operator $\oplus$ is associative. It is possible to automate the verification of such algebraic properties of operators using the theorem-proving rules $\mathcal{P}$ presented in Section 3.5.

To achieve this, we first define the equality operator $==_T$ for the data type T, shown in Definition 4.1, of the output of a given reduction operator $\oplus$ using the rules in Definition 4.2.

Definition 4.1 (Declaration of Data type T):
data $T\ \alpha_1 \ldots \alpha_M\ =\ c_1\ t_1^1 \ldots t_N^1\ |\ \ldots\ |\ c_K\ t_1^K \ldots t_N^K$

Definition 4.2 (Rules to Define Equality Operator for Data Type T):
The equality operator $==_T$ for two values x and y of type T is defined using the following rules:

$$(c\ x_1 \ldots x_N)\ ==_T\ (c\ y_1 \ldots y_N)\ \text{, if}\ \bigwedge_{n=1}^{N} \big((x_n :: T_n) ==_{T_n} (y_n :: T_n)\big)$$

Following this, we verify the associativity of $\oplus$ by testing the equality of all associative forms of $\oplus$ using $==_T$ with the theorem-proving rules $\mathcal{P}$ in Section 3.5.

Given a binary operator f, the two associative forms of f are $(f\ x_1\ (f\ x_2\ x_3))$ and $(f\ (f\ x_1\ x_2)\ x_3)$. Based on this, we can define the associative forms of an n-ary operator as shown in Definition 4.3.

Definition 4.3 (Associative Forms of an Operator):
The n associative forms of an n-ary operator f are defined as follows:

$$\begin{array}{l} f\ x_1 \ldots x_{n-1}\ (f\ x_n \ldots x_{2n-1}) \\ f\ x_1 \ldots x_{n-2}\ (f\ x_{n-1} \ldots x_{2n-2})\ x_{2n-1} \\ \vdots \\ f\ (f\ x_1 \ldots x_n)\ x_{n+1} \ldots x_{2n-1} \end{array}$$

Thus, we can prove that an n-ary operator f is associative if all its n associative forms are equal. For example, consider a binary reduction operator $\oplus :: T \to T \to T$ and its equality operator $==_T ::\ T \to T \to \mathit{Bool}$. We can verify that $\oplus$ is associative if the conjecture

$$\forall x_1, x_2, x_3 \cdot \mathcal{P}[\![\big((x_1 \oplus x_2) \oplus x_3\big) ==_T \big(x_1 \oplus (x_2 \oplus x_3)\big)]\!]$$

evaluates to *True*, where $\mathcal{P}$ are the theorem-proving rules. In Chapter 3, we present an example that shows the proof of associativity for the addition operator in Example 3.1.

4.3 Implementation of Parallel Skeletons

Given that we want to identify *map*, *reduce*, *map-reduce* and *accumulate* skeletons in a given program, our next objective is to obtain parallel implementations of these skeletons to execute the transformed programs on parallel hardware and evaluate their performances. In this section, we discuss the implementations of these skeletons defined over a generic data type a.k.a. polytypic skeletons (Section 4.3.1) and over a list data type (Section 4.3.2).

Polytypic skeletons facilitate parallelisation of a program defined over any data type. Their parallelisation has been widely studied and discussed [42, 44, 70]. However, many of the existing skeleton libraries provide parallel implementations of skeletons that operate over lists, arrays or binary trees only.

Therefore, in Section 4.3.1, we discuss a simplistic approach to create parallel implementations for polytypic skeletons that we use to execute the parallel programs that are produced by our transformation. In Section 4.3.2, we discuss existing libraries that provide efficient parallel implementations for skeletons that operate over list data types.

4.3.1 Implementation of Polytypic Parallel Skeletons

In Section 2.2.3, we discussed the *map*, *reduce*, *map-reduce* and *accumulate* skeletons. In [44], the authors of the *accumulate* skeleton conclude that parallel implementations of polytypic *accumulate* skeletons can be obtained using the tree contraction approach (Section 2.2.4). However, this approach remains to be exploited and no existing libraries provide an implementation of the polytypic *accumulate* skeleton. It is beyond the scope of this thesis to work on using tree contraction for parallel polytypic accumulate skeletons. Therefore, in this section, we present a simplistic approach to parallelise polytypic *map*, *reduce* and *map-reduce* skeletons using Glasgow Parallel Haskell (GpH) discussed earlier in Section 2.2.3.

The sequential definitions of the *map*, *reduce* and *map-reduce* skeletons that operate over a generic data type T with constructors $c_1, \ldots, c_K$ are shown in Definitions 4.4, 4.5 and 4.6, respectively.

The polytypic *map* skeleton is defined over a single recursive input x, which is pattern-matched using its type constructors $c_1, \ldots, c_K$, and the *map* operators $\overline{f}$. The output is constructed using each constructor to bind the results of applying the *map* operators $f_1^k, \ldots, f_L^k$ on the corresponding non-recursive components $z_1^k, \ldots, z_L^k$ of the input pattern and the results of recursively applying the *map* skeleton on the recursive components $y_1^k, \ldots, y_J^k$ of type T.

Definition 4.4 (Polytypic Map Skeleton):
$map\ x\ \overline{f}\ =\ \overline{f}\ x$

where
$\forall k \in \{1, \ldots, K\} \cdot \overline{f}\ (c_k\ x_1^k \ldots x_N^k) = c_k\ (f_1^k\ z_1^k) \ldots (f_L^k\ z_L^k)\ (map\ y_1^k\ \overline{f}) \ldots (map\ y_J^k\ \overline{f})$

$$\{y_1^k, \ldots, y_J^k\} = \{x \mid (x :: T) \in \{x_1^k, \ldots, x_N^k\}\}$$
$$\{z_1^k, \ldots, z_L^k\} = \{x_1^k, \ldots, x_N^k\} \setminus \{y_1^k, \ldots, y_J^k\}$$

$\overline{f} = \bigcup_{k=1}^{K} \{f_1^k, \ldots, f_L^k\}$

An example that maps a operators f_1 and f_2 over a binary tree input is shown in Example 4.1.

Example 4.1 (Map over Binary Tree):
data $BTree\ a\ =\ L\ a \mid B\ a\ (BTree\ a)\ (BTree\ a)$
$map_{BTree}\ (L\ x)\ f_1\ f_2 \quad = L\ (f_1\ x)$
$map_{BTree}\ (B\ x\ xt_1\ xt_2)\ f_1\ f_2\ =\ B\ (f_2\ x)\ (map_{BTree}\ xt_1\ f_1\ f_2)\ (map_{BTree}\ xt_2\ f_1\ f_2)$

The polytypic *reduce* skeleton is defined over a single recursive input x, which is pattern-matched using its type constructors $c_1, \ldots, c_K$, and the reduction operators $\overline{g}$. The output is computed using the corresponding reduction operator g_k to reduce the non-recursive components $z_1^k, \ldots, z_L^k$ of the input pattern and the results of reducing the recursive components $y_1^k, \ldots, y_J^k$ of type T.

Definition 4.5 (Polytypic Reduce Skeleton):
$reduce\ x\ \overline{g}\ =\ \overline{g}\ x$

where
$\forall k \in \{1, \ldots, K\} \cdot \overline{g}\ (c_k\ x_1^k \ldots x_N^k) = g_k\ z_1^k \ldots z_L^k\ (reduce\ y_1^k\ \overline{g}) \ldots (reduce\ y_J^k\ \overline{g})$

$$\{y_1^k, \ldots, y_J^k\} = \{x \mid (x :: T) \in \{x_1^k, \ldots, x_N^k\}\}$$
$$\{z_1^k, \ldots, z_L^k\} = \{x_1^k, \ldots, x_N^k\} \setminus \{y_1^k, \ldots, y_J^k\}$$

$\overline{g} = \{g_1, \ldots, g_K\}$

An example that reduces a binary tree input using an operator g is shown in Example 4.2.

Example 4.2 (Reduce over Binary Tree):
data $BTree\ a\ =\ L\ a \mid B\ a\ (BTree\ a)\ (BTree\ a)$
$reduce_{BTree}\ (L\ x)\ g_1\ g_2 \qquad = g_1\ x$
$reduce_{BTree}\ (B\ x\ xt_1\ xt_2)\ g_1\ g_2\ =\ g_2\ x\ (reduce_{BTree}\ xt_1\ g_1\ g_2)\ (reduce_{BTree}\ xt_2\ g_1\ g_2)$

The polytypic *mapReduce* skeleton is defined over a single recursive input x, which is pattern-matched using its type constructors $c_1, \ldots, c_K$, the map operators $\overline{f}$ and the reduction operators $\overline{g}$. The output is computed using the corresponding reduction operator g_k to reduce the results of applying the map operators $f_1^k, \ldots, f_L^k$ on the corresponding non-recursive components $z_1^k, \ldots, z_L^k$ of the input pattern and the results of recursively applying the *mapReduce* skeleton on the recursive components $y_1^k, \ldots, y_J^k$ of type T.

Definition 4.6 (Polytypic Map-Reduce Skeleton):
$mapReduce\ x\ \overline{g}\ \overline{f}\ =\ \overline{g}\ (\overline{f}\ x)$

where

$$\forall k \in \{1, \ldots, K\} \cdot \overline{g}\ (\overline{f}\ (c_k\ x_1^k \ldots x_N^k)) = g_k\ (f_1^k\ z_1^k) \ldots (f_L^k\ z_L^k)\ (mapReduce\ y_1^k\ \overline{g}\ \overline{f}) \vdots (mapReduce\ y_J^k\ \overline{g}\ \overline{f})$$

$$\{y_1^k, \ldots, y_J^k\} = \{x \mid (x :: T) \in \{x_1^k, \ldots, x_N^k\}\}$$

$$\{z_1^k, \ldots, z_L^k\} = \{x_1^k, \ldots, x_N^k\} \setminus \{y_1^k, \ldots, y_J^k\}$$

$$\overline{f} = \bigcup_{k=1}^{K} \{f_1^k, \ldots, f_L^k\}$$

$$\overline{g} = \{g_1, \ldots, g_K\}$$

An example that performs a map-reduce computation using map operators $f_1,\ f_2$ and reduce operator g over a binary tree input is shown in Example 4.3.

Example 4.3 (Map-Reduce over Binary Tree):
data $BTree\ a\ =\ L\ a \mid B\ a\ (BTree\ a)\ (BTree\ a)$
$mapReduce_{BTree}\ (L\ x)\ f_1\ f_2\ g_1\ g_2 \qquad = g_1\ (f_1\ x)$
$mapReduce_{BTree}\ (B\ x\ xt_1\ xt_2)\ f_1\ f_2\ g_1\ g_2\ =\ g_2\ (f_2\ x)\ (mapReduce_{BTree}\ xt_1\ f_1\ f_2\ g_1\ g_2)\ (mapReduce_{BTree}\ xt_2\ f_1\ f_2\ g_1\ g_2)$

Our approach to parallelising these skeletons is to evaluate each recursive call in the skeleton definitions simultaneously using the *rpar* and *rseq* constructs of Glasgow Parallel Haskell (GpH). The parallel versions of the polytypic *map*, *reduce* and *map-reduce* skeletons obtained using this approach are shown in Definitions 4.7, 4.8 and 4.9, respectively. Here, t is a threshold value that is used to control the number of parallel

threads created by the skeletons using the *rpar* construct for each recursive call. For instance, since a given the data type T is essentially an n-ary tree (since each recursive component in the data type can be a branch node in the n-ary tree), the initial value of t can be determined using the following simple rule-of-thumb where P is the number of processors:

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

The parallel *map* skeleton computes the result by evaluating the recursive calls to *map* in parallel using the *rpar* and *rseq* constructs if the threshold value t is greater than zero, and in sequence otherwise.

Definition 4.7 (Parallel Polytypic Map Skeleton):

$$\begin{aligned}
&map\ x\ t\ \overline{f}\ =\ \overline{f}\ x \\
&\text{where} \\
&\forall k \in \{1,\ldots,K\} \cdot \overline{f}\ (c_k\ x_1^k \ldots x_N^k) = h\ (t\ \leq\ 0) \\
&\qquad\qquad\qquad\qquad \textbf{where} \\
&\qquad\qquad\qquad\qquad h\ True\ = c_k\ (f_1^k\ z_1^k) \ldots (f_L^k\ z_L^k)\ (map\ y_1^k\ t\ \overline{f}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (map\ y_J^k\ t\ \overline{f}) \\
&\qquad\qquad\qquad\qquad h\ False = runEval\ \$\ do \\
&\qquad\qquad\qquad\qquad\qquad\qquad w_1^k \leftarrow rpar\ (map\ y_1^k\ (t-1)\ \overline{f}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad\qquad\qquad\qquad w_J^k \leftarrow rseq\ (map\ y_J^k\ (t-1)\ \overline{f}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad return\ (c_k\ (f_1^k\ z_1^k) \ldots (f_L^k\ z_L^k)\ w_1^k \ldots w_J^k) \\
&\qquad\qquad \{y_1^k,\ldots,y_J^k\} = \left\{x \mid (x :: T) \in \{x_1^k,\ldots,x_N^k\}\right\} \\
&\qquad\qquad \{z_1^k,\ldots,z_L^k\} = \{x_1^k,\ldots,x_N^k\} \setminus \{y_1^k,\ldots,y_J^k\} \\
&\overline{f} = \bigcup_{k=1}^{K} \{f_1^k,\ldots,f_L^k\}
\end{aligned}$$

The parallel *reduce* skeleton computes the result by evaluating the recursive calls to *reduce* in parallel using the *rpar* and *rseq* constructs if the threshold value t is greater than zero, and in sequence otherwise.

Definition 4.8 (Parallel Polytypic Reduce Skeleton):
$reduce\ x\ t\ \overline{g}\ =\ \overline{g}\ x$

where

$$\begin{aligned}
&\forall k \in \{1,\ldots,K\} \cdot \overline{g}\ (c_k\ x_1^k \ldots x_N^k) = h\ (t\ \leq\ 0)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\\
&\qquad\qquad\qquad\qquad\qquad\qquad h\ True\ = g_k\ z_1^k \ldots z_L^k\ (reduce\ y_1^k\ t\ \overline{g})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (reduce\ y_J^k\ t\ \overline{g})\\
&\qquad\qquad\qquad\qquad\qquad\qquad h\ False = runEval\ \$\ do\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad w_1^k \leftarrow rpar\ (reduce\ y_1^k\ (t-1)\ \overline{g})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad w_J^k \leftarrow rseq\ (reduce\ y_J^k\ (t-1)\ \overline{g})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad return\ (g_k\ z_1^k \ldots z_L^k\ w_1^k \ldots w_J^k)\\
&\qquad\qquad \{y_1^k,\ldots,y_J^k\} = \left\{x \mid (x :: T) \in \{x_1^k,\ldots,x_N^k\}\right\}\\
&\qquad\qquad \{z_1^k,\ldots,z_L^k\} = \{x_1^k,\ldots,x_N^k\} \setminus \{y_1^k,\ldots,y_J^k\}\\
&\overline{g} = \{g_1,\ldots,g_K\}
\end{aligned}$$

The parallel *mapReduce* skeleton computes the result by evaluating the recursive calls to *mapReduce* in parallel using the *rpar* and *rseq* constructs if the threshold value t is greater than zero, and in sequence otherwise.

Definition 4.9 (Parallel Polytypic Map-Reduce Skeleton):
$mapReduce\ x\ t\ \overline{g}\ \overline{f}\ =\ \overline{g}\ (\overline{f}\ x)$

where

$$\begin{aligned}
&\forall k \in \{1,\ldots,K\} \cdot \overline{g}\ (\overline{f}\ (c_k\ x_1^k \ldots x_N^k)) = h\ (t\ \leq\ 0)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\\
&\qquad\qquad\qquad\qquad\qquad\qquad h\ True = g_k\ (f_k\ z_1^k) \ldots (f_L^k\ z_L^k)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (mapReduce\ y_1^k\ t\ \overline{g}\ \overline{f})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (mapReduce\ y_J^k\ t\ \overline{g}\ \overline{f})\\
&\qquad\qquad\qquad\qquad\qquad\qquad h\ False = runEval\ \$\ do\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad w_1^k \leftarrow rpar\ (mapReduce\ y_1^k\ (t-1)\ \overline{g}\ \overline{f})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad w_J^k \leftarrow rseq\ (mapReduce\ y_J^k\ (t-1)\ \overline{g}\ \overline{f})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad return\ (g_k\ (f_k\ z_1^k) \ldots (f_L^k\ z_L^k)\ w_1^k \ldots w_J^k)\\
&\qquad\qquad \{y_1^k,\ldots,y_J^k\} = \left\{x \mid (x :: T) \in \{x_1^k,\ldots,x_N^k\}\right\}\\
&\qquad\qquad \{z_1^k,\ldots,z_L^k\} = \{x_1^k,\ldots,x_N^k\} \setminus \{y_1^k,\ldots,y_J^k\}\\
&\overline{f} = \bigcup_{k=1}^{K} \{f_1^k,\ldots,f_L^k\}\\
&\overline{g} = \{g_1,\ldots,g_K\}
\end{aligned}$$

4.3.2 Implementation of List-Based Parallel Skeletons

Parallel implementations of skeletons that operate over list or array data types are available in existing libraries such as Sketo [64], SkePU [30], FastFlow [5], Data Parallel Haskell (DPH) [15], Accelerate [16], Glasgow Distributed Haskell (GDH) [77] and Eden [61]. As discussed in Section 2.2.3, the SkeTo library offers C++ implementations of skeletons, including map and reduce, that operate over arrays, matrices and binary trees. However, a significant amount of effort is required to transform our functional programs to use SkeTo. The DPH and Accelerate libraries provide skeleton implementations for map, reduce and zip computations for execution on multi-core CPUs and GPUs, respectively. Both these libraries are designed to operate over custom array data types. While DPH provides parallel implementations only for their custom array-based skeletons, the Accelerate library restricts their skeleton operators (called "scalar operators" in Accelerate) to basic arithmetic, comparison, bitwise and logical operators. Finally, the GDH and Eden libraries provide parallel implementations for map- and reduce-based skeletons that operate over list data types.

In particular, we favour the use of the Eden library primarily because it is better designed for creating new skeletons using the basic constructs that are provided by the Eden language extension of Haskell and includes a trace viewing tool called EdenTV that aids performance visualisation and analysis. The Eden library skeletons that are of interest to us are presented below.

1. **Parallel Map**: We use the *farmB* skeleton in Eden to distribute a given list input to a number of processes and apply the mapped function in parallel on each. The *farmB* skeleton divides a given list into N sub-lists and creates N parallel processes, each of which can apply the map computation on a sub-list. The type signature of the *farmB* skeleton is as follows:

$$farmB :: (Trans\ a,\ Trans\ b) \Rightarrow Int \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

   Here, *Trans* is a type class defined in Eden for types that can be communicated between processing units. The *Trans* type class instances can be derived only for instances of *NFData* in Haskell, i.e. types that can be evaluated to normal form. For user-defined data types, instance declarations for the classes *NFData* and *Trans* must be provided to Eden. For example, the user-defined data type for binary tree (*BTree*) and the corresponding instances for *NFData* and *Trans* classes are shown below.

$$
\begin{aligned}
&\textbf{data}\ BTree\ a\ =\ Leaf\ a \mid Node\ a\ (BTree\ a)\ (BTree\ a)\\
&instance\ NFData\ a\ \Rightarrow\ NFData\ (BTree\ a)\ where\\
&\quad rnf\ (Leaf\ x)\ \ \ \ \ \ =\ rnf\ x\\
&\quad rnf\ (Node\ x\ l\ r)\ \ =\ rnf\ x\ \text{`}seq\text{`}\ rnf\ l\ \text{`}seq\text{`}\ rnf\ r\\
&instance\ Trans\ a\ \ \ \ \ \Rightarrow\ Trans\ (BTree\ a)
\end{aligned}
$$

The degree of parallelism of the *farmB* skeleton is controlled by dividing the input list $[a]$ into a number of sub-lists $[[a]]$, where the number of sub-lists is the desired number of parallel processes to be created as specified by the first argument of type *Int*. The Eden library uses the following *splitIntoN* function to distribute the input list block-wise into as many sub-lists as the first parameter determines. The lengths of the resulting sub-lists may differ at most by one.

$$splitIntoN :: Int \rightarrow [a] \rightarrow [[a]]$$

Similarly, the *farmS* skeleton in Eden distributes a given list using the *unshuffle* function that divides a list into sub-lists in a round-robin fashion. It is common in parallel programming to divide a given list into N sub-lists, where N is the number of processing units available in the hardware, which is provided in Eden as an integer constant *noPe*.

2. **Parallel Map-Reduce:** The parallel *map-reduce* skeleton is provided by the Eden library in two flavours as given below:

   parMapRedr This version of the parallel *map-reduce* skeleton is implemented using the *parMap* skeleton, which creates as many parallel processes for the map computation as the number of elements in a given list. The result of *parMap* is reduced sequentially using the conventional *foldr* function. The definition of the *parMapRedr* skeleton is as follows:

$$
\begin{aligned}
&parMapRedr :: (Trans\ a,\ Trans\ b) \Rightarrow (b \rightarrow b \rightarrow b) \rightarrow b \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow b\\
&parMapRedr\ g\ v\ f\ xs\ =\\
&\quad \begin{cases} h\ (noPe == 1)\\ \textbf{where}\\ h\ True\ \ =\ mapRedr\ g\ v\ f\ xs\\ h\ False\ =\ \big((foldr\ g\ v) \circ (parMap\ (mapRedr\ g\ v\ f)) \circ (splitIntoN\ noPe)\big)\ xs \end{cases}
\end{aligned}
$$

   A corresponding *parMapRedl* implementation is also provided using the *foldl* function in place of the *foldr* function.

offlineParMapRedr In the *parMapRedr* skeleton, the input lists of the processes are evaluated by the parent process, which creates the *parMap* processes, and are subsequently communicated via automatically created communication channels between the parent process and the *parMap* processes. This can create severe overhead for large lists because lists are transmitted as streams in Eden, i.e. each element is sent in a separate message.

To avoid this, the *offlineParMapRedr* implementation avoids stream communication of input lists and communicates only a process identification number that is used to identify the appropriate element of the list. This can be achieved in Eden by either incorporating the complete unevaluated list in a *worker* function that is mapped on the identification numbers, or locally recomputing the list in each processing element, or replicating arguments across processing elements. As a result, each process evaluates the (*splitIntoN noPe*) application and thereby reduces the communication overhead substantially. The definition of the *offlineParMapRedr* skeleton is as follows:

$$
\begin{array}{l}
\mathit{offlineParMapRedr} :: (\mathit{Trans}\ a,\ \mathit{Trans}\ b) \Rightarrow (b \to b \to b) \to b \to (a \to b) \to [a] \to b \\
\mathit{offlineParMapRedr}\ g\ v\ f\ xs\ = \\
\qquad \begin{cases}
h\ (\mathit{noPe} == 1) \\
\textbf{where} \\
h\ \mathit{True}\ \ = \ \mathit{mapRedr}\ g\ v\ f\ xs \\
h\ \mathit{False}\ = \ (\mathit{foldr}\ g\ v) \circ (\mathit{parMap}\ \mathit{worker}\ [0\ ..\ \mathit{noPe} - 1]) \\
\qquad\quad \textbf{where} \\
\qquad\quad \mathit{worker}\ i\ = \mathit{mapRedr}\ g\ v\ f\ ((\mathit{splitIntoN}\ \mathit{noPe}\ xs)\ !!\ i)
\end{cases}
\end{array}
$$

The *foldr* and *mapRedr* functions used in the parallel map-reduce skeletons are defined as follows:

$$
\begin{array}{l}
\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
\mathit{foldr}\ g\ v\ []\ \ \ \ \ \ \ \ = \ v \\
\mathit{foldr}\ g\ v\ (x : xs)\ \ = \ g\ x\ (\mathit{foldr}\ g\ v\ xs)
\end{array}
$$

$$
\begin{array}{l}
\mathit{mapRedr} :: (b \to c \to c) \to c \to (a \to b) \to [a] \to c \\
\mathit{mapRedr}\ g\ v\ f\ xs\ = \ \big((\mathit{foldr}\ g\ v) \circ (\mathit{map}\ f)\big)\ xs
\end{array}
$$

A corresponding *offlineParMapRedl* implementation is also provided in Eden using the *foldl* and *mapRedl* functions in place of the *foldr* and *mapRedr* functions.

4.3.3 Parallel Reduce in Eden

The Eden library does not provide a parallel implementation for the reduce skeleton. This can also be noted from its absence in the definition of *parMapRedr* and

offlineParMapRedr skeletons. Therefore, we define a parallel implementation of the *reduce* skeleton, *parRedr*, as shown below.

$$
\begin{aligned}
&parRedr :: (Trans\ a) \Rightarrow ([a] \to Bool) \to (a \to a \to a) \to a \to [a] \to a \\
&parRedr\ t\ g\ v\ xs\ = \\
&\begin{cases}
h\ \big((t\ xs)\ ||\ (noPe == 1)\big) \\
\textbf{where} \\
h\ True\ \ = \ foldr\ g\ v\ xs \\
h\ False\ = \ \big((parRedr\ t\ g\ v) \circ (parMap\ (foldr\ g\ v)) \circ (splitIntoN\ noPe)\big)\ xs
\end{cases}
\end{aligned}
$$

Here, a threshold function t is used to determine if the input list xs is split into sub-lists, using the *splitIntoN* function, which are then reduced in parallel using the *foldr* function. Thereafter, the results from reducing the sub-lists are reduced in parallel using the *parRedr* function. Additionally, an offline version for the *parRedr* function, similar to the *offlineParMapRedr* function, can be implemented as shown below.

$$
\begin{aligned}
&parRedr :: (Trans\ a) \Rightarrow ([a] \to Bool) \to (a \to a \to a) \to a \to [a] \to a \\
&offlineParRedr\ t\ g\ v\ xs\ = \\
&\qquad\begin{cases}
h\ \big((t\ xs)\ ||\ (noPe == 1)\big) \\
\textbf{where} \\
h\ True\ \ = \ foldr\ g\ v\ xs \\
h\ False\ = \ (parRedr\ t\ g\ v) \circ (parMap\ worker\ [0\ ..\ noPe - 1]) \\
\qquad\qquad \textbf{where} \\
\qquad\qquad worker\ i\ = foldr\ g\ v\ ((splitIntoN\ noPe\ xs)\ !!\ i)
\end{cases}
\end{aligned}
$$

4.3.4 Parallel Accumulate in Eden

The *accumulate* skeleton discussed in Section 2.2.3 of Chapter 2 is implemented in the Eden library though it is available in the SkeTo library that uses MPI and C++. In order to execute instances of the list-based *accumulate* skeletons identified by our technique, we implement a parallel version of the *accumulate* skeleton in Definition 2.11 using the parallel constructs available in the Eden library as follows.

$$
\begin{aligned}
&parAccumulate :: (Trans\ a,\ Trans\ b,\ Trans\ c) \Rightarrow \\
&\qquad (a \to b \to c) \to (c \to c \to c) \to (a \to b) \to (b \to b \to b) \to [a] \to b \to c \\
&parAccumulate\ p\ \oplus\ q\ \otimes\ xs\ c\ = \ h\ (noPe == 1) \\
&\qquad\qquad \textbf{where} \\
&\qquad\qquad h\ True\ \ = \ accumulate\ p\ \oplus\ q\ \otimes\ xs\ c \\
&\qquad\qquad h\ False\ = \ parZipWithRedr \oplus p\ xs\ (bs\ \oplus\ (q\ b)) \\
&\qquad\qquad\qquad\qquad \textbf{where} \\
&\qquad\qquad\qquad\qquad (bs,\ b)\ = \ parScan\ \otimes\ q\ c\ xs
\end{aligned}
$$

$$
\begin{aligned}
&accumulate :: (a \to b \to c) \to (c \to c \to c) \to (a \to b) \to (b \to b \to b) \to [a] \to b \to c \\
&accumulate\ p\ \oplus\ q\ \otimes\ xs\ c\ = \ zipWithRedr \oplus p\ xs\ (bs\ \oplus\ (q\ b)) \\
&\qquad\qquad\qquad\qquad \textbf{where} \\
&\qquad\qquad\qquad\qquad (bs, b)\ = \ scan\ \oslash\ c\ xs \\
&\qquad\qquad\qquad\qquad s\ \oslash\ t\ = \ s\ \otimes\ (q\ t)
\end{aligned}
$$

The definitions of *parZipWithRedr*, *parScan* and *zipWithRedr* are presented in Appendix C.

4.4 Identification of Parallel Skeletons

To automatically identify potential parallel computations in a given program, we introduce a method to identify instances of parallel skeletons. This is because, as explained in Section 2.2.3, skeletons can be used to encapsulate algorithmic forms of parallel computations. To achieve this, we use the LTS framework presented in Section 3.3 to represent and analyse the programs and skeletons.

An overview of the process to identify instances of skeletons and obtain a program defined with suitable calls to the identified skeletons is illustrated in Figure 4.1. Further, as explained earlier, our objective is to obtain a parallel program that contains fewer intermediate data structures. In order to achieve this, we perform the distillation transformation on the original program to obtain an efficient distilled program, and then apply the process illustrated in Figure 4.1 to identify potential instances of parallel skeletons in the distilled program.

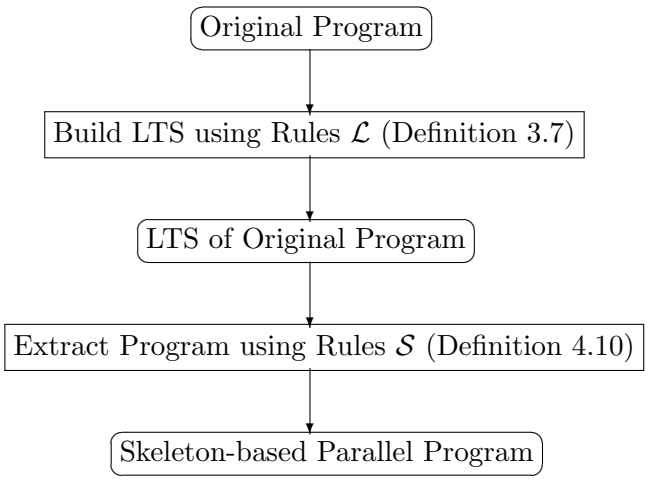


Figure 4.1: Identification of Skeletons in a Program

Here, using the rules $\mathcal{L}$ presented in Definition 3.7, we first build the LTS representations of a given program and the parallel skeletons that we want to identify. We build LTS representations for the skeletons that are of interest to us, whose definitions were presented in Sections 2.2.3 and 4.2. Following this, we adapt the rules $\mathcal{R}$ in Definition 3.8, which are used to obtain the program from its LTS representation, to include

rules that identify instances of skeletons. The resulting modified rules $\mathcal{S}$ presented in Definition 4.10 utilise LTS representations of the parallel skeletons whose instances are identified during program extraction and replaced with suitable calls to the corresponding skeletons.

Definition 4.10 (Extraction of Program from LTS with Skeletons):

$$\mathcal{S}[\![f : t]\!]\ \rho\ \omega = \begin{cases} f'\ e_1 \ldots e_N & \text{, if } \exists (f'\ x'_1 \ldots x'_N, t') \in \omega, \theta \cdot t'\theta = t \\ \text{where } \theta = \{x'_1 \mapsto t_1, \ldots, x'_N \mapsto t_N\} & \\ \qquad \forall n \in \{1, \ldots, N\} \cdot e_n = \mathcal{S}[\![t_n]\!]\ \rho\ \omega & \\ f\ x_1 \ldots x_N & \text{, if } f \in \rho \\ f\ x_1 \ldots x_N & \\ \textbf{where} & \\ f = \lambda x_1 \ldots \lambda x_N.(\mathcal{S}[\![t]\!]\ (\rho \cup \{f\})\ \omega) & \text{, otherwise} \end{cases}$$

$$\text{where } \{x_1, \ldots, x_N\} = fv(t)$$

$$\mathcal{S}[\![x : t]\!]\ \rho\ \omega = x\ x_1 \ldots x_N \quad \text{where } \{x_1, \ldots, x_N\} = fv(t)$$

$$\mathcal{S}[\![e \to (x, \mathbf{0})]\!]\ \rho\ \omega = x$$

$$\mathcal{S}[\![e \to (i, \mathbf{0})]\!]\ \rho\ \omega = i$$

$$\mathcal{S}[\![e \to (c, \mathbf{0}), (\#1, t_1), \ldots, (\#N, t_N)]\!]\ \rho\ \omega = c\ (\mathcal{S}[\![t_1]\!]\ \rho\ \omega) \ldots (\mathcal{S}[\![t_N]\!]\ \rho\ \omega)$$

$$\mathcal{S}[\![e \to (\lambda, t)]\!]\ \rho\ \omega = \lambda x.(\mathcal{S}[\![t]\!]\ \rho\ \omega) \quad \text{where } x \text{ is fresh}$$

$$\mathcal{S}[\![e \to (@, t_0), (\#1, t_1)]\!]\ \rho\ \omega = (\mathcal{S}[\![t_0]\!]\ \rho\ \omega)\ (\mathcal{S}[\![t_1]\!]\ \rho\ \omega)$$

$$\mathcal{S}[\![e \to (\mathbf{let}, x : t_0), (\mathbf{in}, t_1)]\!]\ \rho\ \omega = \mathbf{let}\ x = \lambda x_1 \ldots x_N.(\mathcal{S}[\![t_0]\!]\ \rho\ \omega)\ \mathbf{in}\ (\mathcal{S}[\![t_1]\!]\ \rho\ \omega)$$
$$\text{where } \{x_1, \ldots, x_N\} = fv(t_0)$$

$$\mathcal{S}[\![e \to (\mathbf{case}, t_0), (c_1, t_1), \ldots, (c_K, t_K)]\!]\ \rho\ \omega = \mathbf{case}\ (\mathcal{S}[\![t_0]\!]\ \rho\ \omega)\ \mathbf{of}\ p_1 \to (\mathcal{S}[\![t_1]\!]\ \rho\ \omega)\ |\ \ldots\ |\ p_K \to (\mathcal{S}[\![t_K]\!]\ \rho\ \omega)$$
$$\text{where } p_k = c_k\ x_1 \ldots x_N$$
$$c_k \text{ is of arity } k,\ x_1, \ldots, x_N \text{ are fresh.}$$

Here, the parameter ρ contains the set of new functions that are created and associates them with their corresponding states in the LTS. The 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 instance, $(farmB\ noPe\ f\ x, t)$ is a pair in ω where $farmB\ noPe\ f\ x$ is the application expression for the *farmB* skeleton and t is the LTS representation of the *map* skeleton defined in Section 2.2.3 obtained using rules $\mathcal{L}$ in Definition 3.7.

In rules $\mathcal{S}$, if the current LTS t being transformed is an instance of the LTS t' of a skeleton in ω, then a suitable call to the corresponding skeleton is created with the expressions for the substitution LTSs $t'_1, \ldots, t'_N$. Otherwise, rules $\mathcal{S}$ are defined to transform the current LTS using the same method as in rules $\mathcal{R}$ to extract the program.

In cases where a current LTS being transformed can be identified as an instance of multiple skeletons in ω, the ordering of ω determines the skeleton that is used to define the LTS. Here, ω can be initialised with skeletons in the order that is chosen by the user of the transformation method.

Consequently, by identifying potential instances of parallel skeletons in a given program, we obtain a parallel program that is defined using calls to the corresponding skeletons that are identified.

In this context, it is essential to discuss the preservation of evaluation strategy (laziness and strictness) of the original program and the skeleton-based program produced. Recursive functions in the original program that are identified as instances of the skeletons using the skeleton identification rules $\mathcal{S}$ are replaced with calls to the parallel skeletons. These recursive functions are strict on their recursive inputs which is preserved when replacing the recursive function applications with skeleton applications on the same inputs. Thus, the sequential evaluation of applying these recursive functions on the strict recursive inputs will be the semantically equivalent to the parallel evaluation of the skeleton's application on the same recursive input.

4.4.1 Example

Using the skeleton identification method presented here and the LTSs for the *map*, *reduce*, *map-reduce* and *accumulate* skeletons defined over lists that were presented in Sections 2.2.3 and 4.2, we can transform the original matrix multiplication program from Example 1.1 and identify instances of these skeletons. For example, the LTS for the *map* skeleton defined over a list (presented in Section 2.2.3) is shown in Figure 4.2 and the LTS for the *mMul* function in the matrix multiplication program is shown in Figure 4.3.

Here, we observe that the LTS for the function *mMul* is an instance of the LTS for the *map* skeleton based on the recursive structure. Therefore, according to the rules $\mathcal{S}$ from Definition 4.10, the LTS for *mMul* is extracted and replaced with a suitable call to the *farmB* Eden skeleton that provides a parallel implementation for the *map* skeleton.

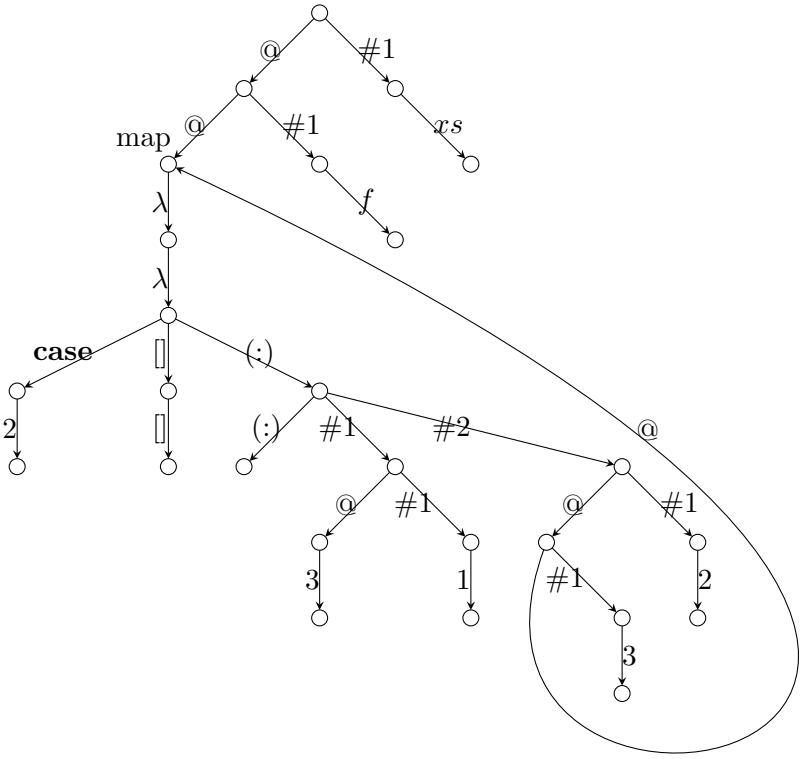


Figure 4.2: LTS for List-based *map* Skeleton

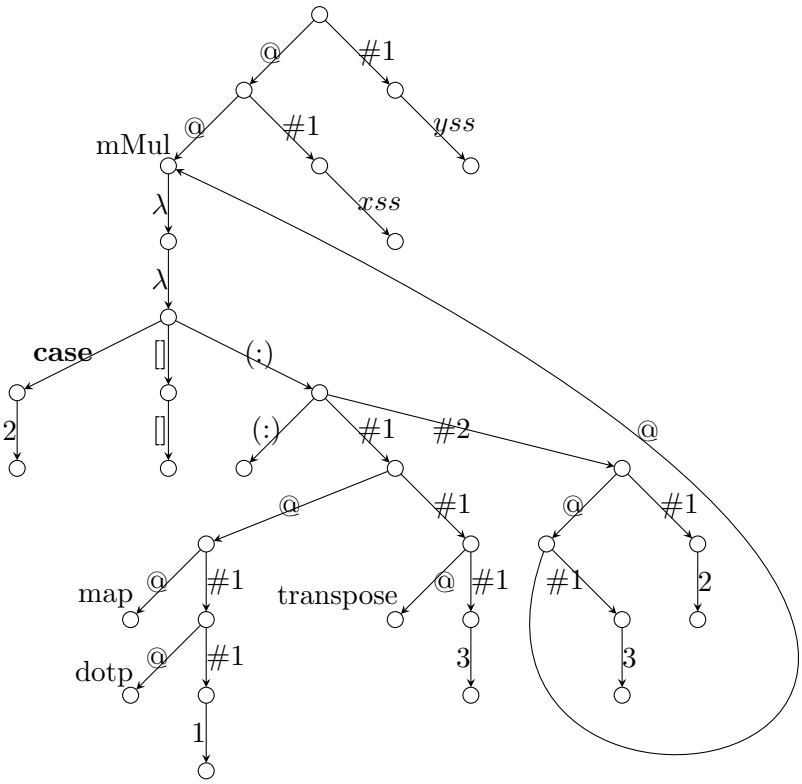


Figure 4.3: LTS for Function *mMul*

Similarly, the *map* and *reduce* functions in the matrix multiplication program are found to be instances of the *map* and *reduce* skeletons. Hence, these are replaced with suitable calls to the *farmB* and *parRedr* skeletons in the Eden library. The resulting matrix multiplication program, in which instances of the *map* and *reduce* skeletons have been identified and replaced with suitable calls to the *farmB* and *parRedr* functions using the rules $\mathcal{S}$ in Definition 4.10, is shown in Example 4.4. Here, *noPe* is the number of processing elements available in the Eden library as explained in Section 4.3.2.

Example 4.4 (Parallel Matrix Multiplication Using Skeleton Identification):

$$
\begin{array}{l}
mMul\ xss\ yss \\
\textbf{where} \\
mMul\ xss\ yss = farmB\ noPe\ f\ xss \\
\quad \textbf{where} \\
\quad f\ xs = farmB\ noPe\ (dotp\ xs)\ (transpose\ yss) \\
dotp\ xs\ ys = parRedr\ (\lambda xs.(length\ xs) < noPe)\ (+)\ 0\ (zipWith\ xs\ ys) \\
transpose\ yss = transpose'\ yss\ [] \\
transpose'\ []\ yss = yss \\
transpose'\ (xs : xss)\ yss = transpose'\ xss\ (rotate\ xs\ yss) \\
rotate\ []\ yss = yss \\
rotate\ (x : xs)\ [] = [x] : (rotate\ xs\ yss) \\
rotate\ (x : xs)\ (ys : yss) = (ys +\!\!+ [x]) : (rotate\ xs\ yss)
\end{array}
$$

An example of identifying a polytypic reduce skeleton is presented using a power tree program in Example 4.5.

Example 4.5 (Power Tree – Original Program (OP)):

$$
\begin{array}{l}
\textbf{data}\ BTree\ a = (L\ a) \mid B\ (BTree\ a)\ (BTree\ a) \\
power :: (BTree\ a) \to (BTree\ a) \to a \\
power\ xt \\
\textbf{where} \\
power\ (L\ x) = x * x \\
power\ (B\ xt_1\ xt_2) = (power\ xt_1) + (power\ xt_2)
\end{array}
$$

By applying the skeletons identification rules $\mathcal{S}$ on this program and using the polytypic skeletons defined in this chapter, we can identify that *power* as an instance of following parallel *reduce* skeleton.

$$
\begin{array}{l}
reduce\ (L\ x)\ t\ g_1\ g_2 = g_1\ x \\
reduce\ (B\ at\ bt)\ t\ g_1\ g_2 = \\
\quad \left\{ \begin{array}{l}
h\ (t \leq 0) \\
\textbf{where} \\
h\ True = g_2\ (reduce\ at\ t\ g_1\ g_2)\ (reduce\ bt\ t\ g_1\ g_2) \\
h\ False = runEval\ \$\ do \\
\qquad\qquad at' \leftarrow rpar\ (reduce\ at\ (t-1)\ g_1\ g_2) \\
\qquad\qquad bt' \leftarrow rseq\ (reduce\ bt\ (t-1)\ g_1\ g_2) \\
\qquad\qquad return\ (g_2\ at'\ bt')
\end{array} \right.
\end{array}
$$

Consequently, *power* is transformed and defined using the $reduce_{BTree}$ as shown below in Example 4.6.

Example 4.6 (Power Tree – Parallel Program):
power xt t
where
$power\ xt\ t\ =\ reduce\ xt\ t\ g_1\ g_2$
where
$g_1\ x\ =\ (x * x)$
$g_2\ at\ bt\ =\ at + bt$

Following this, our objective is to identify instances of the *map*, *reduce* and *map-reduce* skeletons in the distilled programs, for example the matrix multiplication program shown in Example 5.7. However, unlike the power tree example (where the distilled program is the same as the original program and hence is an instance of the *reduce* skeleton), the distilled matrix multiplication program does not contain any instance of the *map*, *reduce*, *map-reduce* or *accumulate* skeletons that are defined over lists. This is the result of a mismatch between the data type and the algorithm of the distilled version and those of the skeletons. Such potential mismatches may mitigate identification of skeletons in a distilled program, and are addressed by the data type transformation proposed in Chapter 5.

4.5 Related Work

Previously, much of the work to parallelise programs was based on list-homomorphisms [7, 79], particularly systematic derivation of parallel programs that are defined over lists [35, 24, 23, 32]. Based on this paradigm, the diffusion transformation [44] was proposed to decompose recursive functions in a given program to facilitate their definition using skeletons. The authors of diffusion also proposed the *accumulate* skeleton [48] that encapsulates the computational forms of *map* and *reduce* skeletons that use an accumulating parameter to build the result. However, such methods primarily address programs that are defined over lists. Even though diffusion can be extended to generic data types, it is applicable to programs that operate over one recursive input only.

To extend the homomorphism-based approach to a wider range of programs, tree contraction has been used to define parallel computations over trees [67, 4, 70]. In particular, Morihata et al. [71] proposed a method to decompose a binary tree into a list of sub-trees called a *zipper*. By defining functions using upward and downward computa-

tions on the zipper structure, it is then possible to use list-homomorphisms for parallel evaluation of the functions. However, such methods are often limited by the range of programs and data types they can transform. Also, a common aspect of such 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. [18, 19] 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 $+\!\!+$.

SkelML [78] is a compiler that parallelises higher-order functional programs. The compiler automatically extracts and exploits *map* and *fold* computations for execution on processor farms and processor trees, respectively. This is achieved by nesting the parallel skeletons in a processor topology that matches the structure of the Standard ML source. This work by Scaife et al. describes the analysis leading from a Standard ML input program through higher-order functions to an executable parallel program.

Ahn et al. [2] proposed an analytical method to transform general recursive functions into a composition of polytypic data parallel skeletons. This is achieved by classifying the sub-expressions in a given program based on their usage of the input arguments using program slicing. Following this classification, the sub-expressions are defined using the appropriate data parallel skeleton from among *map*, *reduce* and *scan*. Even though this method is applicable to a wider range of problems and does not need manual derivation of associative operators, the transformed programs are defined by composing skeletons and employ multiple intermediate data structures. Further their transformation does not address programs that may be defined over multiple inputs of any data type.

More recently, Dever et al. [28] proposed a transformation technique called *AutoPar*, which is capable of parallelising a given sequential program and reducing the number of intermediate data structures. AutoPar is designed to transform higher-order language programs that are defined over a single recursive input of any data type. The objective of this technique is to transform the recursive input of a given program into a well-partitioned join-list. Using a parallelisation algorithm to analyse the transformed program defined over the join-list, AutoPar explicitly specifies independent sub-expressions for parallel evaluation using the *par* and *pseq* parallel constructs in the Glasgow Haskell

Compiler (GHC). Even though AutoPar can parallelise higher-order language programs and handle functions defined over a recursive input of any data type, it cannot be used to parallelise programs that contain functions defined over multiple recursive inputs.

4.6 Summary

In this chapter, we have presented a technique to automatically identify potential skeleton instances in a given program. By using the distillation transformation described in Chapter 3 in conjunction with this method, we can transform a given program into a version that has fewer intermediate data structures and is defined using parallel skeletons. This provides a solution to our first research question "***RQ-1:*** *How can potential parallel computations in a program be automatically identified?*".

We discussed the *map*, *reduce*, *map-reduce* and *accumulate* skeletons that are of interest to us in the context of this thesis. We use Glasgow Parallel Haskell to provide a simplistic implementation of the polytypic skeletons. As a result of using GpH for parallelisation, the recursive calls in these skeletons are speculatively parallelised. That is, a recursive call's evaluation is *sparked* and is speculatively parallelised by the runtime system. The polytypic skeleton implementations used in this thesis are based on this speculative parallelisation model. Thus, their lazy evaluation is controlled using the *rpar* and *rseq* constructs. In future work, it is possible to use techniques such as evaluation strategies [84] to implement these polytypic skeletons to explicitly specify the strictness of their evaluation and also the degree of parallelisation. For list-based skeletons, we use the Eden library to obtain parallel implementations. The skeletons in Eden are implemented using strategies that force the execution of the parallel processes that are created. This means that a strict evaluation of the parallel skeletons in the transformed program is introduced by the using the skeletons in the Eden library. These implementations are required for execution and evaluation of the transformed programs on parallel hardware.

Further, in our skeleton identification rules, we replace recursive functions that are instances of the sequential form of the skeletons with calls to the parallel skeletons. The recursive functions that are identified as instances of skeletons will require strictness on their recursive inputs. Since the skeletons are also applied on the same recursive inputs in the transformed program, the sequential evaluation of applying these recursive functions on the strict recursive inputs will be the semantically equivalent to the parallel

evaluation of the skeleton's application on the same recursive input.

This provides a solution to our second research question "***RQ-2:*** *How can the transformed program be executed in parallel?*". However, there are two issues that need to be addressed:

1. **Mismatch of Programs and Skeletons**: While using fewer intermediate data structures, the distilled program still operates over the original inputs. However, the data types of the inputs and the algorithm of the distilled program may not match those of the parallel skeletons, which operate over a single recursive input. This would result in an inability to identify parallel computations that could potentially be encapsulated using the *map*, *reduce*, *map-reduce* or *accumulate* skeletons. This issue was posed in our third research question "***RQ-3:*** *How can a given program be transformed to aid identification of parallel computations?*". Therefore, we need to transform a distilled program into a form that is more likely to contain instances of these parallel skeletons.

2. **Existing Implementations of Parallel Skeletons**: Even though parallel skeletons that operate over generic data types (polytypic skeletons) have been extensively researched as discussed in Section 2.2.3, there are no libraries that implement the parallel *map*, *reduce*, *map-reduce* and *accumulate* skeletons over a generic data type. Currently existing libraries provide skeletons defined over flat data structures, such as lists and arrays, for parallel execution. Some libraries that implement parallel skeletons over generic data types (n-ary trees [70]) transform them implicitly into flat data types or binary trees and use skeletons defined over them.

To address the first issue of mismatch between programs and skeletons, we present a technique in Chapter 5 to transform the data type of a distilled program. The objective of this technique is to facilitate the identification of the skeletons discussed in this chapter in a distilled program, thus providing solutions to our third research question "***RQ-3:*** *How can a given program be transformed to aid identification of parallel computations?*".

To address the second issue of parallel implementations for polytypic skeletons, we use the simplistic approach of using Glasgow Parallel Haskell presented in this chapter. However, the use of tree contraction techniques to design and implement more efficient parallel polytypic skeletons has been widely discussed [44, 70]. Since this work is beyond the scope of this thesis, we address this in Chapter 7 as a part of future work.

Chapter 5

Data Type Transformation

5.1 Introduction

In Section 1.2, we hypothesised that a given program can be automatically transformed into an efficient parallel version by identifying potential parallel computations. To achieve this, we described an existing transformation called distillation in Chapter 3 which can reduce inefficient intermediate data structures in a given program. In Chapter 4, we presented a method to automatically identify computations in a program that can be implemented using skeletons. Using this technique, we identify potential instances of skeletons in a distilled program which can then be executed on parallel hardware using existing efficient implementations of the identified skeletons. This provides solutions to two of our research questions "***RQ-1:*** *How can potential parallel computations in a program be automatically identified?*" and "***RQ-2:*** *How can the transformed program be executed in parallel?*".

However, a distilled program may not be in a form that is suitable for automatic identification of the skeletons that are of interest to us – *map*, *reduce*, *map-reduce* and *accumulate*. This is because

1. The distilled program may be defined over multiple inputs of different data types which may not match with those of skeletons.

2. The algorithmic structure of the distilled program may not match with the recursive structure of the skeletons.

Consider the matrix multiplication and dot product of binary tree in distilled form presented in Examples 5.1 and 5.2.

Example 5.1 (Distilled Matrix Multiplication):

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

Example 5.2 (Distilled Dot Product of Binary Trees):

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

Here, the recursive functions $mMul_1$, $mMul_2$, $mMul_3$ or $dotP$ cannot be defined using the list-based or polytypic *map*, *reduce*, *mapReduce* or *accumulate* skeletons discussed in Chapter 4 because of the mismatch between the input data and the algorithm structures of the functions and the skeletons. To resolve this, we propose to transform the recursive functions in a distilled program by combining the inputs into a new data type. This is significantly different from methods such as tupling or currying as the new input data types are created based on the algorithmic structures of the recursive functions in the distilled program.

This issue is posed by our third research question in Section 1.2 "***RQ-3:*** *How can a given program be transformed to aid identification of parallel computations?*". To resolve this, we need to transform the distilled program into a form that is more likely to resemble the structure of the skeletons that are of interest to us before applying the skeleton identification technique presented in Chapter 4.

In this chapter, we present a technique called *encoding transformation* [56, 53, 52] that provides a solution to our third research question "***RQ-3:*** *How can a given program be transformed to aid identification of parallel computations?*". The encoding transformation combines all inputs of a distilled program into a single input belonging to a new data type whose structure reflects the algorithmic structure of the skeletons that are of interest to us. This facilitates identification of skeletons that are defined over the new data type in the resulting *encoded program*. Consequently, the encoded program can then be executed on parallel hardware using efficient parallel implementations of the skeletons that are available.

The remainder of this chapter is structured as follows: In Section 5.2, we introduce our encoding transformation and discuss how to parallelise a given program using the encoding transformation. In Section 5.3, we revisit the parallel skeletons presented in Section 4.3 to discuss them in the context of the encoding transformation. In Section 5.4, we discuss existing works on transforming the inputs of a program to facilitate their parallelisation. Finally, we present some remarks on our transformation in Section 5.5.

5.2 The Transformation

The data types of the inputs and the algorithm of a distilled program, which we want to parallelise, may not match with those of the skeletons that are of interest to us. As explained in Section 4.2, we are interested in identifying *map*, *reduce*, *map-reduce* and *accumulate* skeletons, and they are defined over a single recursive input. However, the distilled program may be defined over an arbitrary number of inputs each of which may be of any data type. Also, the algorithmic structure of the distilled program may not match the recursive structure of the skeletons. These factors will inhibit the identification of computations that can potentially be defined using the *map*, *reduce*, *map-reduce* or *accumulate* skeletons using the technique presented in Chapter 4. To resolve this, we present a transformation that *encodes* the inputs of a distilled program into a single input and transforms the distilled program to operate over the *encoded input*. The resulting *encoded program* is defined in a form that facilitates the identification of skeleton instances.

To achieve this, we first lift the definitions of all functions in a distilled program to the top-level using lambda lifting [6, 49]. Following this, for each recursive function

f defined in the top-level **where**-expression of the distilled program, we encode the pattern-matched inputs of f. Other inputs that are not pattern-matched in the definition of f are not included in the encoded input of f. 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.

5.2.1 Overview

Consider a recursive function f with inputs $x_1, \ldots, x_M, x_{(M+1)}, \ldots, x_N$ defined in a distilled program, where $x_1, \ldots, x_M$ are pattern-matched and $x_{(M+1)}, \ldots, x_N$ are not. The three steps to encode $x_1, \ldots, x_M$ into the encoded input x are illustrated in Figure 5.1 and summarised below:

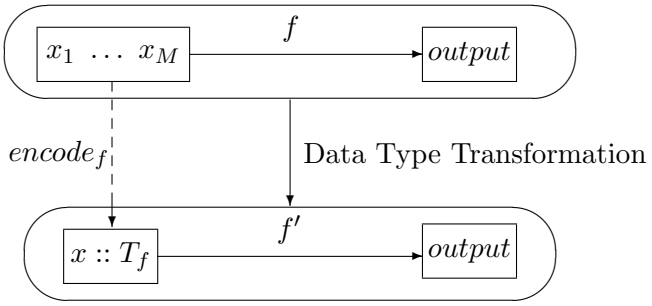


Figure 5.1: Steps to Encode Inputs of Function f

1. **Declare a new encoded data type T_f .**

   Firstly, we create a new data type T_f for the encoded input of recursive function f, which corresponds to the data types of the pattern-matched inputs $x_1, \ldots, x_M$.

2. **Define a function $encode_f$.**

   Secondly, we define a function $encode_f$ to encode the pattern-matched inputs of the recursive function f into the data type T_f.

3. **Transform function f .**

   Finally, we transform the recursive function f in the distilled program into function f' that operates over the encoded input of type T_f obtained using the function $encode_f$.

As explained earlier, the objective of the encoding transformation is to transform the pattern-matched inputs into a data structure that matches the recursive structure of the

function. Based on this approach, the encoding transformation for a function f can be defined using two different approaches:

- **Encoding Transformation Version 1 :** This version of the encoding transformation addresses recursive functions where at least one function body has more than one recursive call. The evaluation tree of such a function will be an n-ary tree where $n > 1$. Therefore, this version of the encoding transformation encodes the pattern-matched inputs into an n-ary data structure of type T_f, which matches the recursive structure of function f, using the $encode_f$ function.

  Consequently, the encoded program can potentially contain instances of polytypic skeletons that are defined over the new encoded data type T_f.

- **Encoding Transformation Version 2 :** This version of the encoding transformation addresses recursive functions where all function bodies contain at most one recursive call. The evaluation tree of such a function will be linear. Therefore, it is desirable in this case to encode the pattern-matched inputs of the linear-recursive function into a *cons*-list of elements of type T'_f, which matches the linear recursive structure of function f, using the $encode_f$ function.

  Consequently, the encoded program can potentially contain instances of skeletons that are defined over a *cons*-list.

Accordingly, in Section 5.2.3, we present version 1 of our encoding transformation to encode inputs into a new data type [53], and in Section 5.2.4, we present version 2 of our encoding transformation to encode inputs into a *cons*-list [52].

5.2.2 Parallelisation Using Encoding Transformation

Based on the two versions of the encoding transformation, the steps to parallelise a given program are as follows:

1. Apply the distillation transformation (Chapter 3) on a given program. This produces a *distilled program*.

2. For each recursive function f in the distilled program,

   (a) Compute the maximum number of recursive calls, n, in the bodies of the definition of function f.

(b) `IF` $n > 1$

`THEN` Apply encoding transformation version 1 (Section 5.2.3) on function f.

`ELSE` Apply encoding transformation version 2 (Section 5.2.4) on function f.

This produces an *encoded program*.

3. Apply the skeleton identification transformation presented in Chapter 4 on the encoded program using skeletons discussed in Chapter 4 that operate over lists and the new encoded data types created by the encoding transformation. This produces an *encoded parallel program*.

4. Execute the encoded parallel program using implementations for skeletons that operate over lists and the new data types created by the encoding transformation as discussed in Section 4.3.

Here, the maximum number of recursive calls, n, in the bodies of a recursive function is always greater than zero since we transform only recursive functions in the distilled program. Further, it is possible to transform mutually recursive functions into self-recursive functions [90], which is required for the proposed encoding transformation.

Consequently, the resulting encoded parallel program is potentially defined using skeletons, polytypic and/or list-based, that are most suited to the algorithmic structure of the program. Here, the encoded recursive functions that operate over an encoded n-ary data structure are potentially defined using polytypic skeletons and the encoded functions that operate over an encoded linear data structure are potentially defined using list-based skeletons.

5.2.3 Encoding Inputs into New Data Type

In this version of our encoding transformation, we encode the pattern-matched inputs of a recursive function f into an *encoded input* which is of a new data type T_f and whose structure reflects the recursive structure of f. We illustrate this with a program that computes the dot-product of two binary trees as shown in Example 5.3.

Example 5.3 (Dot-Product of Binary Trees):
data $BTree\ a\ ::=\ E \mid B\ (BTree\ a)\ (BTree\ a)$
$dotP\ xt\ yt$
where
$dotP\ E\ yt = 0$
$dotP\ (B\ x\ xt_1\ xt_2)\ E = 0$
$dotP\ (B\ x\ xt_1\ xt_2)\ (B\ y\ yt_1\ yt_2) = (x * y) + (dotP\ xt_1\ yt_1) + (dotP\ xt_2\ yt_2)$

Based on the language of a distilled program presented earlier in Definition 3.18, the definition of a recursive function f, with inputs $x_1, \ldots, x_M, x_{(M+1)}, \ldots, x_N$, of the form is shown in Definition 5.1. Here, at least one function body e_k corresponding to function header $f\ p_1^k \ldots p_M^k\ x_{(M+1)}^k \ldots x_N^k$ contains more than one recursive call to function f, and all arguments in a function call are variables.

Definition 5.1 (General Form of Recursive Function in Distilled Program):
$f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N$
where
$$\begin{aligned} f\ p_1^1 \ldots p_M^1\ x_{(M+1)} \ldots x_N &= e_1 \\ \vdots & \quad \vdots \\ f\ p_1^K \ldots p_M^K\ x_{(M+1)} \ldots x_N &= e_K \end{aligned}$$
where
$$\exists k \in \{1, \ldots, K\} \cdot e_k = E_k \left[f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1,\ \ldots\ , f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \right]$$

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

1. **Declare a new encoded data type T_f:**

   First, we declare a new data type T_f for the encoded input. This new data type corresponds to the data types of the pattern-matched inputs of function f that are encoded. The definition of the new data type T_f, which corresponds to the recursive function of the form in Definition 5.1, is shown in Definition 5.2.

   Definition 5.2 (New Encoded Data Type T_f):
   $$\begin{aligned} \textbf{data}\ T_f\ \alpha_1 \ldots \alpha_G = &\ c_1\ T_1^1 \ldots T_L^1\ (T_f\ \alpha_1 \ldots \alpha_G)_1^1 \ldots (T_f\ \alpha_1 \ldots \alpha_G)_J^1 \\ &\ \vdots \\ \mid &\ c_K\ T_1^K \ldots T_L^K\ (T_f\ \alpha_1 \ldots \alpha_G)_1^K \ldots (T_f\ \alpha_1 \ldots \alpha_G)_J^K \end{aligned}$$
   where
   $\alpha_1, \ldots, \alpha_G$ are the type parameters of the data types of pattern-matched inputs $x_1 \ldots x_M$
   $\forall k \in \{1, \ldots, K\} \cdot$
   c_k is a fresh constructor for the type T_f corresponding to $p_1^k \ldots p_M^k$ of the pattern-matched inputs.

$$f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots, \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}$$

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

$$\begin{cases} fv(E_k) \setminus \{x_{(M+1)}, \ldots, x_N\} & \text{, if } e_k = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots, \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix} \\ fv(e_k) \setminus \{x_{(M+1)}, \ldots, x_N\} & \text{, otherwise} \end{cases}$$

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 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 new type whose structure reflects the recursive structure of f. To achieve this, the arguments 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 inputs of the recursive calls to function f that may be present in the function body e_k.

The encoded data type obtained for the recursive function *dotP* in the dot-product of binary trees program from Example 5.3 are shown in Example 5.4.

Example 5.4 (Encoded Data Types For Dot Product of Binary Trees):

$$\begin{aligned} &\textbf{data}\ T_{dotP}\ a\ =\ c_1 \\ &\qquad\qquad \mid\ c_2 \\ &\qquad\qquad \mid\ c_3\ a\ a\ (T_{dotP}\ a)\ (T_{dotP}\ a) \end{aligned}$$

Here, the three constructors c_1, c_2 and c_3 for type T_{dotP} correspond to the three patterns $E\ yt$, $(B\ x\ xt_1\ xt_2)\ E$ and $(B\ x\ xt_1\ xt_2)\ (B\ y\ yt_1\ yt_2)$, respectively, of the inputs xt and yt in function *dotP* in Definition 5.3. The arguments bound by c_3 correspond to the variables x and y, and the encoding of the arguments xt_1, yt_1 and xt_2, yt_2 of the recursive calls to *dotP*.

2. **Define a function *encode*$_f$:**

   Given a recursive function f of the form shown in Definition 5.1, we define a function $encode_f$ as shown in Definition 5.3 to build the encoded input for function f.

Definition 5.3 (Definition of Function $encode_f$):
$encode_f\ x_1 \ldots x_M$
where
$encode_f\ p_1^1 \ldots p_M^1 \ = \ e'_1$
$\vdots \qquad\qquad\qquad\quad \vdots$
$encode_f\ p_1^K \ldots p_M^K \ = \ e'_K$

where
$\forall k \in \{1, \ldots, K\}\cdot$
$e'_k = c_k\ z_1^k \ldots z_L^k\ (encode_f\ x_1^1 \ldots x_M^1) \ldots (encode_f\ x_1^J \ldots x_M^J)$

$$f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots, \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}$$

$$\{z_1, \ldots, z_L\} = \begin{cases} fv(E_k) \setminus \{x_{(M+1)}, \ldots, x_N\} & \text{, if } e_k = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots, \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix} \\ fv(e_k) \setminus \{x_{(M+1)}, \ldots, x_N\} & \text{, otherwise} \end{cases}$$

Here, the inputs $x_1 \ldots x_M$ of function f are pattern-matched and recursively 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 inputs $x_1 \ldots x_M$, function $encode_f$ uses the corresponding constructor c_k whose arguments 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 inputs of the recursive calls to function f.

The encode function obtained for the recursive function *dotP* in the dot-product of binary trees program from Example 5.3 are shown in Example 5.5.

Example 5.5 (Encode Functions For Dot Product of Binary Trees):
$$\begin{aligned} encode_{dotP}\ E\ yt &= c_1 \\ encode_{dotP}\ (B\ x\ xt_1\ xt_2)\ E &= c_2 \\ encode_{dotP}\ (B\ x\ xt_1\ xt_2)\ (B\ y\ yt_1\ yt_2) &= c_3\ x\ y\ (encode_{dotP}\ xt_1\ yt_1) \\ &\qquad\qquad (encode_{dotP}\ xt_2\ yt_2) \end{aligned}$$

Here, $encode_{dotP}$ encodes the inputs xt and yt of function *dotP* in Definition 5.3 using the three patterns $E\ yt$, $(B\ x\ xt_1\ xt_2)\ E$ and $(B\ x\ xt_1\ xt_2)\ (B\ y\ yt_1\ yt_2)$ as in the definition of *dotP*. Also, the arguments bound by c_3 correspond to the variables x and y, and the encoding of the arguments xt_1, yt_1 and xt_2, yt_2 of the recursive calls to *dotP*.

3. **Transform the distilled program:**

After creating the encoded data type T_f and the $encode_f$ function for each recursive

function f, we transform the distilled program as shown in Definition 5.4 by defining a recursive function f', which operates over the encoded input, corresponding to function f.

Definition 5.4 (Definition of Transformed Function Over Encoded Input):
$f'\ x\ x_{(M+1)} \ldots x_N$
where
$$\begin{aligned} f'\ \left(c_1\ z_1^1 \ldots z_L^1\ x_1^1 \ldots x_1^J\right)\ x_{(M+1)} \ldots x_N &= e'_1 \\ \vdots & \quad \vdots \\ f'\ \left(c_K\ z_1^K \ldots z_L^K\ x_K^1 \ldots x_K^J\right)\ x_{(M+1)} \ldots x_N &= e'_K \end{aligned}$$
where
$$\forall k \in \{1, \ldots, K\} \cdot\ e'_k\ =\ E_k \left[f'\ x_k^1\ x_{(M+1)}^1 \ldots x_N^1,\ \ldots,\ f'\ x_k^J\ x_{(M+1)}^J \ldots x_N^J\right]$$
$$f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots\ , \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}$$

Here,

- In each function definition header of f, replace the pattern-matched inputs with the corresponding pattern of their encoding.

  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 inputs $p_1, \ldots, p_M$.

- 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 $f'\ x\ x_{(M+1)} \ldots x_N$, where x is the encoding of the pattern-matched inputs $x_1, \ldots, x_M$.

The encoded program obtained for the distilled matrix multiplication program from Example 5.3 is shown in Example 5.6.

Example 5.6 (Encoded Program for Dot Product of Binary Trees Defined Over Encoded Input):
$dotP'\ (encode_{dotP}\ xt\ yt)$
where
$$\begin{aligned} dotP'\ c_1 &= 0 \\ dotP'\ c_2 &= 0 \\ dotP'\ (c_3\ x\ y\ at\ bt) &= (x * y) + (dotP'\ at) + (dotP'\ bt) \end{aligned}$$

Here, the encoded function $dotP'$ is defined over the encoded input using the patterns for the encoded type T_{dotP}.

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,

$$\begin{aligned}&\big(f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N\big) = \big(f'\ x\ x_{(M+1)} \ldots x_N\big)\\ &\text{where } x = \mathit{encode}_f\ x_1 \ldots x_M\end{aligned}$$

Proof:

The proof is by structural induction over the encoded data type T_f.

Base Case:

For the encoded input $x_k = c_k\ z_1^k \ldots z_L^k$ computed by $\mathit{encode}_f\ p_1^k \ldots p_M^k$:

1. By Definition 5.1, L.H.S. evaluates to e_k.
2. By Definition 5.4, R.H.S. evaluates to e_k.

Inductive Case:

For the encoded input $x_k = c_k\ z_1^k \ldots z_L^k\ x^{1_k} \ldots x^{J_k}$ computed by $\mathit{encode}_f\ p_1^k \ldots p_M^k$:

1. By Definition 5.1, L.H.S. evaluates to $E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots\ , \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}$.
2. By Definition 5.4, R.H.S. evaluates to $E_k \begin{bmatrix} f'\ x_k^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots, \\ f'\ x_k^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}$.
3. By inductive hypothesis, $\big(f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N\big) = \big(f'\ x\ x_{(M+1)} \ldots x_N\big)$. □

Observation

From the rules to define an encode_f function, we can observe that the structure of the resulting encoded input reflects the recursive structure of recursive function f. Therefore, if the encoded function f' is potentially an instance of the *map*, *reduce*, *map-reduce* or *accumulate* skeleton, then these skeletons have to be defined over the encoded data type T_f, which is essentially an n-ary tree, and parallelised as discussed in Section 4.2. The parallelisation of the encoded program produced using this version of the encoding transformation is discussed in Section 5.3.1.

5.2.4 Encoding Inputs into Cons-List

In this version of our encoding transformation, we encode the pattern-matched inputs $x_1, \ldots, x_M$ of a recursive function f into a *cons*-list, referred to as the *encoded list*, so that the resulting encoded program can potentially contain instances of the well-known *map*, *reduce*, *map-reduce* or *accumulate* skeletons defined over a *cons*-list. The encoded list is of type $[T'_f]$, where T'_f is a new type created to encode the pattern-matched inputs $x_1, \ldots, x_M$. We illustrate this using the distilled matrix multiplication program shown in Example 5.7.

Example 5.7 (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 $(mMul_3\ xs\ yss\ v) : (mMul_2\ zs\ xs\ yss\ v')$
$mMul_3\ []\ yss\ v = 0$
$mMul_3\ (x : xs)\ []\ v = 0$
$mMul_3\ (x : xs)\ (ys : yss)\ v = (x * (v\ ys)) + (mMul_3\ xs\ yss\ v)$

Here, 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.

Based on the language of a distilled program presented earlier in Definition 3.18, 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 5.5 in a distilled program. Here, each function body e_k corresponding to function header $f\ p_1^k \ldots p_M^k\ x_{(M+1)}^k \ldots x_N^k$ contains at most one recursive call to f, and all arguments in a function call are variables.

Definition 5.5 (General Form of Linear Recursive Function in Distilled Program):

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

The three steps to encode the pattern-matched inputs of function f 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 definition of the new data type T'_f is shown in Definition 5.6.

   Definition 5.6 (New Encoded Data Type T'_f):
   data $T'_f\ \alpha_1 \dots \alpha_G\ =\ c_1\ T_1^1 \dots T_L^1\ |\ \dots\ |\ c_K\ T_1^K \dots T_L^K$
   where
   $\alpha_1, \dots, \alpha_G$ are type parameters of the data types of pattern-matched inputs $x_1 \dots x_M$
   $\forall k \in \{1, \dots, K\}$:
   c_k is a fresh constructor for the data type T'_f corresponding to $p_1^k \dots p_M^k$ of the pattern-matched inputs
   $f\ p_1^k \dots p_M^k\ x_{(M+1)} \dots x_N = e_k$
   $\{(z_1 :: T_1^k), \dots, (z_L :: T_L^k)\} =$
   $$\begin{cases} fv(E_k) \setminus \{x_{(M+1)}, \dots, x_N\}, & \text{if } e_k = E_k \left[f\ x_1^k \dots x_M^k\ x_{(M+1)}^k \dots x_N^k \right] \\ fv(e_k) \setminus \{x_{(M+1)}, \dots, x_N\}, & \text{otherwise} \end{cases}$$

   Here, a new constructor c_k of the type T'_f is created for each set $p_1^k \dots p_M^k$ of the pattern-matched inputs $x_1 \dots x_M$ 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 $z_1, \dots, z_L$ in $p_1^k \dots p_M^k$ 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 $z_1, \dots, z_L$.

The encoded data types obtained for the recursive functions $mMul_1$, $mMul_2$ and $mMul_3$ in the distilled matrix multiplication program from Example 5.7 are shown in Example 5.8.

Example 5.8 (Encoded Data Types for Distilled Matrix Multiplication):
data $T'_{mMul_1}\ a\ =\ c_1 \mid c_2 \mid c_3\ [a]\ [a]$
data $T'_{mMul_2}\ a\ =\ c_4 \mid c_5$
data $T'_{mMul_3}\ a\ =\ c_6 \mid c_7 \mid c_8\ a\ [a]$

Here, the three constructors c_1, c_2 and c_3 for type T'_{mMul_1} correspond to the three patterns $[]\ zss$, $xss\ []$ and $(xs : xss)\ (zs : zss)$, respectively, of the inputs xss and zss in function $mMul_1$ in Definition 5.7. The encoded data types T'_{mMul_2} and T'_{mMul_3} are declared in a similar fashion for the functions $mMul_2$ and $mMul_3$ in Definition 5.7.

2. **Define a function *encode*$_f$:**

Given a recursive function f of the form shown in Definition 5.5, we define a function $encode_f$ as shown in Definition 5.7 to build the encoded list in which each element is of type T'_f.

Definition 5.7 (Definition of Function $encode_f$):
$encode_f\ x_1 \ldots x_M$
where
$encode_f\ p_1^1 \ldots p_M^1\ =\ e'_1$
$\vdots \qquad\qquad\qquad\quad \vdots$
$encode_f\ p_1^K \ldots p_M^K\ =\ e'_K$
where
$\forall k \in \{1, \ldots, K\}$:

$$e'_k = \begin{cases} \left[c_k\ z_1^k \ldots z_L^k\right] +\!\!+ (encode_f\ x_1^k \ldots x_M^k), \text{ if } e_k = E_k\left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k\right] \\ \text{where } \{z_1^k, \ldots, z_L^k\} = fv(E_k) \setminus \{x_{(M+1)}, \ldots, x_N\} \\ \left[c_k\ z_1^k \ldots z_L^k\right], \qquad\qquad\qquad\qquad \text{otherwise} \\ \text{where } \{z_1^k, \ldots, z_L^k\} = fv(e_k) \setminus \{x_{(M+1)}, \ldots, x_N\} \\ \text{where } f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = e_k \end{cases}$$

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.

The encode functions obtained for the recursive functions $mMul_1$, $mMul_2$ and $mMul_3$ in the distilled matrix multiplication program from Example 5.7 are shown in Example 5.9.

Example 5.9 (Encode Functions For Distilled Matrix Multiplication):

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

Here, $encode_{mMul_1}$, $encode_{mMul_2}$ and $encode_{mMul_3}$ encode the inputs of functions $mMul_1$, $mMul_2$ and $mMul_3$ in Definition 5.7 using the patterns for inputs as in the definition of $mMul$.

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 as shown in Definition 5.8 by defining a recursive function f', which operates over the encoded list, corresponding to function f.

Definition 5.8 (Definition of Transformed Function Over Encoded List):

$f'\ x\ x_{(M+1)} \ldots x_N$
where

$$
\begin{array}{lcl}
f'\ \left((c_1\ z_1^1 \ldots z_L^1) : x^1\right)\ x_{(M+1)} \ldots x_N & = & e'_1 \\
\vdots & & \vdots \\
f'\ \left((c_K\ z_1^K \ldots z_L^K) : x^K\right)\ x_{(M+1)} \ldots x_N & = & e'_K
\end{array}
$$

where

$\forall k \in \{1, \ldots, K\}$:

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

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

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 T'_f.
  For instance, a function header $f\ p_1 \ldots p_M\ x_{(M+1)} \ldots x_N$ is transformed to $f'\ p\ x_{(M+1)} \ldots x_N$, where 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 $f'\ x\ x_{(M+1)} \ldots x_N$, where x is the encoding of the pattern-matched inputs $x_1, \ldots, x_M$.

The encoded program obtained for the distilled matrix multiplication program from Example 5.7 is shown in Example 5.10.

Example 5.10 (Encoded Program for Matrix Multiplication Defined Over Encoded List):

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

Here, the encoded functions $mMul'_1$, $mMul'_2$ and $mMul'_3$ are defined over the encoded inputs using the patterns for the encoded types T'_{mMul_1}, T'_{mMul_2} and T'_{mMul_3}.

Property 5.1 (Non-Empty Encoded List):

Given rules in Definition 5.7 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 5.5, $\exists k \in \{1, \ldots, K\} \cdot p_1^k \ldots p_M^k$ that matches inputs $x_1 \ldots x_M$ in function f. Consequently, from Definition 5.7

$$encode_f\ x_1 \ldots x_M \ = \begin{cases} [c_k\ z_1^k \ldots z_L^k] +\!\!+ \left(encode_f\ x_1^k \ldots x_M^k\right), \\ \text{if } f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \left[f\ x_1^k \ldots x_M^k\ x_{(M+1)}^k \ldots x_N^k\right] \\ [c_k\ z_1^k \ldots z_L^k], \qquad \text{otherwise} \end{cases}$$

Therefore, the list computed by $encode_f\ x_1 \ldots x_M$ contains at least one element. □

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,

$$\left(f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N\right) = \left(f'\ x\ x_{(M+1)} \ldots x_N\right)$$
$$\text{where } x = encode_f\ x_1 \ldots x_M$$

Proof:

The proof is by structural induction over the encoded list type $[T'_f]$, given a non-empty encoded list (Property 5.1).

Base Case:

For the encoded list $\left((c_k\ z_1^k \ldots z_L^k) : []\right)$ computed by $encode_f\ p_1^k \ldots p_M^k$:

1. By Definition 5.5, L.H.S. evaluates to e_k.
2. By Definition 5.8, R.H.S. evaluates to e_k.

Inductive Case:

For the encoded list $\left((c_k\ z_1^k \ldots z_L^k) : x^k\right)$ computed by $encode_f\ p_1^k \ldots p_M^k$ where $x^k \neq []$:

1. By Definition 5.5, 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 5.8, R.H.S. evaluates to $E_k \left[f'\ x^k\ x_{(M+1)}^k \ldots x_N^k\right]$.
3. By inductive hypothesis, $\left(f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N\right) = \left(f'\ x\ x_{(M+1)} \ldots x_N\right)$. □

Observation

It is evident from the rules to define an $encode_f$ function that the original inputs are encoded into a *cons*-list. Therefore, if the encoded function f' is potentially an instance of the *map*, *reduce*, *map-reduce* or *accumulate* skeleton, then we can use their list-based definitions discussed in Section 4.2 to define f' using a suitable call to the corresponding skeleton. The parallelisation of the encoded program produced using this version of the encoding transformation is discussed in Section 5.3.2.

5.3 Parallel Skeletons Revisited

In Section 4.3, we presented the parallelisation approach that we use in our transformation for executing polytypic and list-based skeletons on parallel hardware. In this section, we revisit and discuss the parallel skeletons and their implementations based on some observations we make about the encoded input produced by the two versions of our encoding transformation.

5.3.1 Polytypic Parallel Skeletons

In the context of encoding inputs into a new data type using the encoding transformation version 1 presented in Section 5.2.3, it is important to note that an encoded function f' that is defined over the new data type T_f cannot be an instance of a polytypic *map* skeleton. This is because, as shown in Section 4.3.1, a *map* skeleton that is defined over an input of type $T\ \alpha$ always produces an output of the same type $T\ \beta$, where α and β may be different type components. However, the encoding transformation version 1 creates a new data type T_f for the encoded input. Thus, the encoded function f' that is defined over the encoded input of type T_f operates over an input whose type is always different from that of the output. Consequently, we identify instances of only polytypic *reduce* and *map-reduce* skeletons that operate over the encoded data type which are parallelised as discussed in Section 4.3.1.

5.3.2 List-Based Parallel Skeletons

In the context of encoding inputs into a list as discussed in Section 5.2.4, it is important to note the following property of our encoding transformation version 2 presented in Section 5.2.4.

Property 5.2 (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 5.8, given an encoded function f',

   $$f' \ :: \ [T'_f] \ \rightarrow \ T_{(M+1)} \ \rightarrow \ \ldots \ \rightarrow \ T_N \ \rightarrow \ b$$

   where $[T'_f]$ is the encoded list data type.

   $T_{(M+1)}, \ldots, T_N$ 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 $\oplus :: T'_f \rightarrow b \rightarrow b$.

Given that T'_f is a newly created data type, it follows from (2) that the binary operator $\oplus$ is not associative because the input data type T'_f and the output data type b cannot not be equal. □

As a result of Property 5.2, any function that is an instance of a reduce skeleton in an encoded program that operates over an encoded list cannot be evaluated in parallel because the reduction operator will not be associative. Consequently, we identify instances of only *map*, *map-reduce* and *accumulate* skeletons in an encoded program obtained using the rules in Definition 5.2.4, and use their parallel implementations for execution. The *map-reduce* and *accumulate* skeletons are free from the operator non-associativity issue presented in Property 5.2. This is because the *map* operator in these skeletons allows transformation of the input to the output data type, which is then used for computations using the *reduce* or *scan* operators. Consequently, the input and output data types of the *reduce* and *scan* operators will match and are required to be associative for parallelisation.

Further, the *map-reduce* and *accumulate* skeletons defined in Section 4.3.2 were defined using the *foldr* and *foldl* functions that require a unit value for the reduction/fold operator to be provided as an input as discussed in Section 4.3.2. However, it is evident

from Property 5.1 that the skeletons that are potentially identified will always be applied on non-empty lists. Therefore, we use the *foldr1* and *foldl1* functions, which are defined for non-empty lists, for the *map-reduce* and *accumulate* skeletons, thereby avoiding the need to obtain a unit value for the reduction operator. To this end, we augment the skeletons in Eden by adding the parallel *map-reduce* (*parMapRedr1*, *offlineParMapRedr1*, *parMapRedl1*, *offlineParMapRedl1*) and *accumulate* (*parAccumulate1*) skeletons, which use the parallel *reduce* skeletons defined using the *foldr1* and *foldl1* functions as shown below.

Parallel Reduce

$$
\begin{array}{l}
parRedr1 :: (Trans\ a) \Rightarrow ([a] \to Bool) \to (a \to a \to a) \to [a] \to a \\
parRedr1\ t\ g\ xs\ = \\
\quad \left\{ \begin{array}{l}
h\ \big((t\ xs)\ ||\ (noPe == 1)\big) \\
\textbf{where} \\
h\ True\ \ = \ foldr1\ g\ xs \\
h\ False\ = \ (parRedr1\ t\ g) \circ (parMap\ (foldr1\ g)) \circ (splitIntoN\ noPe)\ xs
\end{array} \right.
\end{array}
$$

Using the *foldl1* function in place of the *foldr1* function, we can similarly define the *parRedl1* function. These functions are used to redefine the parallel *map-reduce* skeleton in Eden as shown below, so that both the map and reduce computations are parallelised.

Parallel Map-Reduce

$$
\begin{array}{l}
parMapRedr1 :: (Trans\ a,\ Trans\ b) \Rightarrow ([a] \to Bool) \to (b \to b \to b) \to (a \to b) \to [a] \to b \\
parMapRedr1\ t\ g\ f\ xs\ = \\
\quad \left\{ \begin{array}{l}
h\ \big((t\ xs)\ ||\ (noPe == 1)\big) \\
\textbf{where} \\
h\ True\ \ = \ mapRedr1\ g\ f\ xs \\
h\ False\ = \ (parRedr1\ t\ g) \circ (parMap\ (mapRedr1\ g\ f)) \circ (splitIntoN\ noPe)\ xs
\end{array} \right.
\end{array}
$$

$$
\begin{array}{l}
offlineParMapRedr1 :: (Trans\ a,\ Trans\ b) \Rightarrow \\
\qquad\qquad ([a] \to Bool) \to (b \to b \to b) \to (a \to b) \to [a] \to b \\
offlineParMapRedr1\ t\ g\ f\ xs\ = \\
\quad \left\{ \begin{array}{l}
h\ \big((t\ xs)\ ||\ (noPe == 1)\big) \\
\textbf{where} \\
h\ True\ \ = \ mapRedr1\ g\ f\ xs \\
h\ False\ = \ (parRedr1\ t\ g) \circ (parMap\ worker\ [0\ ..\ noPe - 1]) \\
\qquad\quad \textbf{where} \\
\qquad\quad worker\ i = mapRedr1\ g\ f\ ((splitIntoN\ noPe\ xs)\ !!\ i)
\end{array} \right.
\end{array}
$$

The *foldr1* and *mapRedr1* functions used in these skeletons are defined as follows:

$$
\begin{array}{l}
foldr1 :: (a \to a \to a) \to [a] \to a \\
foldr1\ [x] \quad\quad\quad = x \\
foldr1\ x : xs \quad\quad = g\ x\ (foldr1\ g\ xs)
\end{array}
$$

$$
\begin{array}{l}
mapRedr1 :: (b \to b \to b) \to (a \to b) \to [a] \to b \\
mapRedr1\ g\ f\ xs\ =\ (foldr1\ g) \circ (map\ f)\ xs
\end{array}
$$

Using the *foldl1*, *parRedl1* and *mapRedl1* functions in place of the *foldr1*, *parRedr1* and *mapRedr1* functions, we can similarly define the *parMapRedl1* and *offlineParMapRedl1* skeletons.

Parallel Accumulate

$$
\begin{array}{l}
parAccumulate1 :: (Trans\ a,\ Trans\ b,\ Trans\ c) \Rightarrow \\
\qquad\qquad (a \to b \to c) \to (c \to c \to c) \to (a \to b) \to (b \to b \to b) \to [a] \to b \to c \\
parAccumulate1\ p\ \oplus\ q\ \otimes\ xs\ c\ =\ h\ (noPe == 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad h\ True\ \ =\ accumulate1\ p\ \oplus\ q\ \otimes\ xs\ c \\
\qquad\qquad\qquad\qquad\qquad\qquad h\ False\ =\ parZipWithRedr1\ \oplus\ p\ xs\ bs \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad bs\ =\ parScan\ q\ \otimes\ c\ xs
\end{array}
$$

$$
\begin{array}{l}
accumulate1 :: (a \to b \to c) \to (c \to c \to c) \to (a \to b) \to (b \to b \to b) \to [a] \to b \to c \\
accumulate1\ p\ \oplus\ q\ \otimes\ xs\ c\ =\ zipWithRedr1\ \oplus\ p\ xs\ bs \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad bs\ \ \ \ \ =\ scanl\ \oslash\ c\ xs \\
\qquad\qquad\qquad\qquad\qquad\qquad s\ \oslash\ t\ =\ s\ \otimes\ (q\ t)
\end{array}
$$

The definitions of the auxiliary functions *parZipWithRedr1*, *parScan* and *zipWithRedr1* used in this definition are presented in Appendix C.

5.4 Related Work

Our encoding transformation version 1 presented in Section 5.2.3 builds an encoded input of a new data type that reflects the recursive structure of the function. Analogous to this, Mogensen's work on supercompilation for data types [69] uses a transformation based on supercompilation to combine groups of constructor applications of the inputs into a single constructor application in a given program. Consequently, the transformed program defined over the supercompiled input and output uses fewer pattern-matchings of the input and constructor applications to build the output. However, this transformation

currently lacks formal specification of methods to translate inputs and output of the transformed program from and to their original types. Further, the result of this data type transformation remains to be studied for its ability to aid program parallelisation. On the other hand, the parallel tree contraction techniques proposed by Morihata et al. [4, 70] present methods to transform n-ary trees into binary trees. This allows the use of parallel map and reduce skeletons defined over binary trees for parallelisation of the original programs that are defined over n-ary trees. However, this method requires that the programs be defined manually using the skeletons and does not address programs that are defined over multiple inputs.

With respect to our encoding transformation version 2 presented in Section 5.2.4 to encode the inputs of a program into a *cons*-list, there exist few methods that allow transformation of a generic data type into a list. One notable exception is the work of Morihata et al. [71] on adapting the third homomorphism theorem [32] to trees to allow the transformation of a program defined over a binary tree into one defined over a zipper, which is a list. While this work presents an interesting approach, it presents no concrete methodology for generating zippers from binary trees and assumes that the developer has provided such a function. Further, this approach is applicable only to programs that operate over one binary tree. More recently, Dever et al. [27] proposed a data transformation method that converts a recursive input of any data type into a list as a part of the AutoPar parallelisation technique [28]. Even though this method is capable of transforming an input of any data type into a well-partitioned list, the transformed programs may not lend themselves to being defined using list-based parallel skeletons because the recursive structure of the original program is not considered in building the list. Further, this data transformation method cannot be applied to programs with multiple inputs.

5.5 Summary

In Chapter 4, we presented a method to automatically identify instances of parallel skeletons in a given program. By applying this method on distilled programs, we can produce programs that have fewer intermediate data structures and are defined using parallel skeletons potentially leading to their efficient execution on parallel hardware. This provided solutions to our first and second research questions "***RQ-1:*** *How can*

potential parallel computations in a program be automatically identified?" and "***RQ-2:*** *How can the transformed program be executed in parallel?*".

However, we observed that mismatches may occur in the data types and algorithms of a distilled program and the parallel skeletons that are of interest to us (*map*, *reduce*, *map-reduce* and *accumulate*). Such mismatches would inhibit the identification of parallel computations that could potentially be encapsulated using the *map*, *reduce*, *map-reduce* and *accumulate* skeletons. This issue was posed in our third research question "***RQ-3:*** *How can a given program be transformed to aid identification of parallel computations?*". To resolve this, we have presented a technique called *encoding transformation* in this chapter, in which the pattern-matched inputs of a distilled program are encoded (combined) into a single input. We presented the encoding transformation in two versions:

- **Encoding Transformation Version 1**: We encode the original inputs into a new data type whose structure reflects the algorithmic structure of the distilled program. The encoded programs obtained through this version potentially contain instances of skeletons that are defined over the new data type, which is in essence an n-ary tree. Using parallel implementations for polytypic skeletons, it is possible to efficiently execute the encoded parallel programs that are defined using these parallel skeletons.

- **Encoding Transformation Version 2**: We encode the original inputs into a *cons*-list. Following this, we identify potential instances of parallel skeletons that are defined over lists, and use existing implementations of list-based skeletons that are available in libraries such as Eden.

By using these two versions of the encoding transformation, we can transform a distilled program into an encoded parallel program that is potentially defined using skeletons that are most suited to the algorithmic structure of the distilled program. As discussed in Section 5.2.2, the encoded parallel program is potentially defined using polytypic and/or list-based skeletons. Therefore, we can use parallel implementations of these skeletons to efficiently execute the encoded parallel program on parallel hardware.

We also highlight the following two attributes of the encoding transformation:

1. The encoding transformation may duplicate components from the original inputs in the encoded input. This is because the encoded input data structure is created

to match the recursive structure of functions in the program. Consequently, the encoded input may be larger than the original inputs.

2. The structure of the input data may already match the algorithmic structure of the distilled program. In this case, applying the encoding transformation may be redundant and introduce an unnecessary overhead in the parallelisation transformation.

These two attributes of the encoding transformation and possible approaches to overcome them are discussed as a part of further work in Chapter 7.

The encoding transformation technique presented in this chapter provides a solution to our third research question "***RQ-3:*** *How can a given program be transformed to aid identification of parallel computations?*". In Chapter 6, we apply our transformation to a set of benchmark programs and discuss the transformed programs that are produced along with the performance results of executing the benchmark programs. By analysing the results, we evaluate the effectiveness of our methods and the performance of the parallel programs produced.

Chapter 6

Evaluation of Benchmark Programs

6.1 Introduction

In Section 1.2, we presented our hypothesis "*Program transformation can be used to automatically identify parallel computations in a given program, potentially leading to its efficient parallel execution*". To support this hypothesis, we presented our transformation method that is composed of the following three stages:

1. Distillation of programs to reduce the number of inefficient intermediate data structures (Chapter 3).

2. Definition and implementation approaches for list-based and polytypic skeletons that encapsulate parallel computations that are of interest to us, in particular *map*, *reduce*, *map-reduce* and *accumulate* skeletons. This includes an automatic technique to identify skeletons in a given program (Chapter 4).

3. An encoding transformation to aid the identification of skeletons in a wider range of programs (Chapter 5). We presented two versions of the encoding transformation:

   (a) Encode all pattern-matched inputs into a single input (called the *encoded input*), whose structure reflects the recursive structure of the program.

   (b) Encode all pattern-matched inputs into a *cons*-list (called the *encoded list*).

Based upon these three stages, the parallel procedure defined in Section 5.2.2 allows a program to be transformed and potentially defined using polytypic and/or list-based

skeletons whose instances may be found in the transformed program. The skeletons identified and their parallel implementations are discussed in Chapter 4.

In this chapter, we present the transformation of a set of benchmark programs that we use to evaluate the proposed transformation method and the performance evaluations of the transformed benchmark programs. In Section 6.2, we present an overview of the transformation process that is used to obtain different versions of each benchmark program used in the evaluation. While we use the Eden library implementations for list-based skeletons that are used in the transformed parallel programs, we use a simplistic approach to implement the polytypic skeletons using GpH in the parallel programs. In Section 6.3, we present an overview of the process that is used to evaluate the benchmark programs. In Section 6.4, we present the details of the hardware and software setup that are used in the evaluation process.

In Section 6.5, we present the benchmark programs and their different transformed versions along with an analysis of the speedups achieved by the parallel programs that are produced against the sequential and hand-parallelised versions of the same programs. We also perform a cost-centre analysis and study how the execution of the parallel program behaves on multiple cores. In Section 6.6, we discuss a set of examples which our transformation method does not parallelise. We use these benchmark programs to study both the strengths and weaknesses of the proposed transformation. In Section 6.7, we summarise the observations that we make from the transformation of the benchmark programs and present a holistic assessment of the strengths and weaknesses of our parallelisation transformation method that are observed from these evaluations.

6.2 Transformation Process

As stated earlier, we use a set of benchmark programs to evaluate the transformation method presented in this thesis. The benchmark programs and their details are presented in Section 6.5, each of which is transformed using the parallelisation procedure introduced in Section 5.2.2 to obtain the encoded parallel program (EPP). For each benchmark program, the performance of the encoded parallel program (EPP) produced by our transformation method will be compared with the following versions of the same benchmark program:

- Original Program (OP).

- Distilled Program (DP).
- Hand-Parallelised Program (HPP).

The OP version is the given definition of a benchmark program. The DP version is obtained by transforming the OP version using the distillation transformation described in Chapter 3. Definitions of GHC library functions (such as *map*, *foldr*, *zipWith* and $+\!\!+$) that are used in the OP version are included in the distillation process. Operators such as arithmetic, *max* and *gcd* are considered built-in and their definitions are not used in the distillation transformation. For some benchmark programs, the OP and DP versions may be identical.

The HPP version is obtained by identifying instances of *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons in the OP version of a benchmark program. We consider this to be a fair practice because this is an approach of an average program developer who is not an expert in program optimisation or program parallelisation. This can be performed either by hand or using the skeleton identification rules presented in Chapter 4. In each of the benchmark programs presented, the HPP version is defined by identifying all instances of the *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons in the OP version.

6.3 Evaluation Process

Each benchmark program is evaluated over a range of input sizes. To ensure fairness, all versions of a benchmark program are evaluated on the same input that is read into the program in an identical fashion (either read from a file or constructed in the program). The parallel versions of a benchmark program are evaluated by running parallel threads on varying numbers of CPU cores. To reduce bias, the results presented are computed using the average numbers from 10 runs for each experiment. The hardware and software setup used for the evaluations are detailed in Section 6.4.

For all parallel versions of a benchmark program (HPP and EPP), only those skeletons that are present in the top-level expression are executed using their parallel implementations. That is, nesting of parallel skeletons is avoided. The nested skeletons that are present inside top-level skeletons are executed using their sequential versions. The objective of this approach is to avoid uncontrolled creation of too many threads that results in inefficient parallel execution where the cost of thread creation and management is greater than the cost of parallel execution. The scenario of poor performance

due to nested parallel skeletons is demonstrated using one of the benchmark programs in Section 6.5.9.

The performance factors evaluated for each benchmark program are as follows:

- **Speedup and Scalability**: This analysis is performed by comparing the total execution time of the EPP version, which is composed of the time to build the inputs, encode the inputs and the time to execute the function, with the total execution time of the other versions.

- **Parallel Execution Profile**: This analysis illustrates the execution states of the CPU cores when executing the EPP version of a benchmark program.

- **Cost-Centre**: This analysis is performed for the encoded version of each benchmark program. It shows the percentage of execution time spent on building the inputs, encoding the inputs and evaluating the functions, individually.

  For this, we use the sequential execution of the encoded version of each benchmark program. This is because profiling individual functions in a parallel execution adds significant profiling overhead to the execution time thereby making the total execution time and cost-centre statistics inconsistent.

6.4 Evaluation Environment

6.4.1 Hardware

The benchmark programs are evaluated on a Mac Pro (late 2013) computer with a 12-core Intel Xeon E5 processor each clocked at 2.7 GHz. The main memory is 64 GB composed of four 16 GB modules each of which is a DDR3 ECC clocked at 1866 MHz.

6.4.2 Software

The Mac Pro has OS X Yosemite version 10.10.5 installed. The standard Glasgow Haskell Compiler used for the sequential programs (OP and DP) is GHC version 7.10.2. For the parallel programs (HPP and EPP), the latest Eden compiler based on GHC version 7.8.2 is used. EdenTV version 4.10.0 is used to visualise the trace produced by Eden for the parallel programs.

6.4.3 Evaluation Steps for Sequential Programs

The sequential versions OP and DP of a benchmark program are evaluated as follows. For a given program in a file named `program.hs`, the command used to compile and generate an executable file is as follows:

```
ghc program.hs -rtsopts -eventlog -fforce-recomp
```

Here, the compilation flags `-rtsopts` and `-eventlog` enable logging of run-time statistics about the execution run of `program.hs`. The flag `-fforce-recomp` ensures that the executable file generated by compilation is always on the last saved version of `program.hs`. The executable file that is generated by this compilation, `program`, is then run using the following command:

```
./program input_sizes +RTS -ls -RTS
```

Here, the command-line argument `input_sizes` provides the executable file with the size of the input(s) that are to be used for this execution run of `program`. The flags `+RTS` and `-RTS` are used to delimit arguments for the runtime system, and the flag `-ls` enables event logging during the execution into a file named `program.eventlog` which can be analysed using the Threadscope or ghc-events tools. Upon successful execution of each sequential version of a benchmark program, the total execution time composed of the time to build the inputs and execute the functions that implement the benchmark program's algorithm are obtained from `program.eventlog`.

The encoded version of a benchmark program is also evaluated sequentially to obtain the individual execution time percentages for building inputs, encoding inputs and executing functions that implement the algorithm. For a given program in a file named `program.hs`, the command used to compile and generate an executable file for the encoded version is as follows:

```
ghc program.hs -prof -fprof-auto -rtsopts -fforce-recomp
```

Here, the compilation flags `-prof`, `-fprof-auto` and `-rtsopts` instruct the Haskell runtime to automatically generate statistics about the run of `program.hs`. The executable file that is generated by this compilation, `program`, is then run using the following command:

```
./program input_sizes +RTS -p -RTS
```

Here, the command-line argument `input_sizes` provides the executable file with the size of the input(s) that are to be used for this execution run of `program`. The flag `-p` instructs the Haskell run-time to save the run-time statistics in a file named `program.prof`. Upon successful execution, the execution time percentages to build inputs, encode inputs and to execute the functions that implement the benchmark program's algorithm are obtained from `program.prof`.

6.4.4 Evaluation Steps for Parallel Programs

The parallel versions HPP and EPP of a benchmark program that are defined using Eden skeletons are evaluated as follows. For a given program in a file named `program.hs`, the command used to compile and generate an executable file using the modified GHC in Eden is

```
eden/bin/ghc program.hs -parcp -eventlog -fforce-recomp
```

Here, the compilation flag `-parcp` enables parallel execution on shared-memory systems. The executable file that is generated by this compilation, `program`, is then run using the following command:

```
./program input_sizes +RTS -Nn -ls -RTS
```

The flag `-Nn` provides the number of cores to be used for the parallel execution, where `n` ranges from `2` to `12` cores in our execution runs. The flag `-ls` enables event logging during the execution into a file named `program_input_sizes_+RTS_-Nn_-ls_-RTS.parevents`. The `parevents` file with the event log information is then used to obtain the total execution time and for visualising the parallel execution statistics in EdenTV using the following command is

```
eden/bin/edentv program_input_sizes_+RTS_-Nn_-ls_-RTS.parevents
```

The parallel versions HPP and EPP of a benchmark program that are defined using polytypic skeletons implemented using GpH are evaluated as follows. For a given program in a file named `program.hs`, the command used to compile and generate an executable file:

```
eden/bin/ghc program.hs -threaded -feager-blackholing -rtsopts -eventlog
                                -fforce-recomp
```

Here, the compilation flag `-threaded` enables parallel execution and `-feager-blackholing` reduces redundant computations by GpH. The executable file that is generated by this compilation, `program`, is then run using the following command:

```
./program input_sizes +RTS -Nn -ls -RTS
```

The flag `-ls` enables event logging during the execution into a file named `program.eventlog` which is then used to obtain the total execution time and for visualising the parallel execution statistics in Threadscope using the following command:

```
threadscope program.eventlog
```

6.5 Benchmark Programs

We apply the proposed parallelisation transformation to the following benchmark programs that are discussed in Sections 6.5.1 to 6.5.7 and 6.6.1 to 6.6.4.

1. Matrix Multiplication.
2. Power Tree.
3. Dot Product of Binary Trees.
4. Totient.
5. Maximum Prefix Sum.
6. Sum Squares of List.
7. Fibonacci Series Sum.
8. Sum Append of Lists.
9. Maximum Segment Sum.
10. Reverse List.
11. Flatten Binary Tree.
12. Insertion Sort.

These programs are considered for our evaluation since they are based on the *nofib* benchmark suite [73] and known to be parallelisable by hand. Therefore, we believe they will serve to evaluate if the proposed transformation method can parallelise these programs and also to evaluate the performance of the parallel programs that are produced.

6.5.1 Evaluation of Matrix Multiplication

The OP version of the matrix multiplication program is presented in Example 6.5.1.1. Here, function *mMul* computes the dot-product of each row *xs* in the matrix *xss* with each row in the transpose of the matrix *yss* to obtain the result matrix. The built-in *map* function is used in the *mMul* function, and the *dotp* function that computes the dot-product of lists *xs* and *ys* is defined using the built-in functions *zipWith* and *foldr*. Here, the inefficient intermediate data structures are the result of *transpose yss*, which is decomposed by *map* for computing the dot-product using *dotp xs*, and the result of *zipWith* that is decomposed by *foldr*.

Example 6.5.1.1 (Matrix Multiplication – Original Program (OP)):

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

Based on the procedure in Section 6.2, we first apply the distillation transformation to reduce the number of intermediate data structures. Example 5.7 presents the distilled program (DP) obtained for the OP version in Example 6.5.1.1, where distillation uses the library definitions of *map*, *foldr*, *zipWith* and $(+\!\!+)$. In Example 5.7, the function *mMul* that multiplies matrices *xss* and *yss* without using any intermediate data structures is defined using three recursive functions $mMul_1$, $mMul_2$ and $mMul_3$.

Here, we observe that all three recursive functions $mMul_1$, $mMul_2$ and $mMul_3$ have a maximum of one recursive call in their function definition bodies. Consequently, based on the parallelisation steps presented in Section 5.2.2, we use the encoding transformation version 2 to encode the pattern-matched inputs into *cons*-list. Example 6.5.1.2 presents the encoded version of the DP version of matrix multiplication.

Example 6.5.1.2 (Matrix Multiplication – Encoded Program):

$$
\begin{array}{l}
\textbf{data}\ T'_{mMul_1}\ a = c_1 \mid c_2 \mid c_3\ [a]\ [a] \\
\textbf{data}\ T'_{mMul_2}\ a = c_4 \mid c_5 \\
\textbf{data}\ T'_{mMul_3}\ a = c_6 \mid c_7 \mid c_8\ a\ [a]
\end{array}
$$

$$
\begin{array}{lcl}
encode_{mMul_1}\ []\ zss & = & [c_1] \\
encode_{mMul_1}\ xss\ [] & = & [c_2] \\
encode_{mMul_1}\ (xs : xss)\ (zs : zss) & = & [c_3\ xs\ zs] +\!\!+ (encode_{mMul_1}\ xss\ zss)
\end{array}
$$

$$
\begin{array}{lcl}
encode_{mMul_2}\ [] & = & [c_4] \\
encode_{mMul_2}\ (z : zs) & = & [c_5] +\!\!+ (encode_{mMul_2}\ xs\ yss\ zs)
\end{array}
$$

$$
\begin{array}{lcl}
encode_{mMul_3}\ []\ yss & = & [c_6] \\
encode_{mMul_3}\ (x : xs)\ [] & = & [c_7] \\
encode_{mMul_3}\ (x : xs)\ (ys : yss) & = & [c_8\ x\ ys] +\!\!+ (encode_{mMul_3}\ xs\ yss)
\end{array}
$$

$$
\begin{array}{l}
mMul\ xss\ yss \\
\textbf{where} \\
mMul\ xss\ yss\ =\ mMul'_1\ (encode_{mMul_1}\ xss\ yss)\ yss
\end{array}
$$

$$
\begin{array}{lcl}
mMul'_1\ (c_1 : w)\ yss & = & [] \\
mMul'_1\ (c_2 : w)\ yss & = & [] \\
mMul'_1\ \big((c_3\ xs\ zs) : w\big)\ yss & = & \textbf{let}\ v = \lambda xs.g\ xs \\
 & & \qquad\qquad\qquad\qquad \textbf{where} \\
 & & \qquad\qquad\qquad\qquad g\ []\ = 0 \\
 & & \qquad\qquad\qquad\qquad g\ (x : xs)\ = x \\
 & & \quad \textbf{in}\ (mMul'_2\ (encode_{mMul_2}\ zs)\ xs\ yss\ v) : (mMul'_1\ w\ yss)
\end{array}
$$

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

$$
\begin{array}{lcl}
mMul'_3\ (c_6 : w)\ v & = & 0 \\
mMul'_3\ (c_7 : w)\ v & = & 0 \\
mMul'_3\ \big((c_8\ x\ ys) : w\big)\ v & = & (x * (v\ ys)) + (mMul'_3\ w\ v)
\end{array}
$$

Using the encoded program in Example 6.5.1.2, we apply the skeleton identification technique to identify potential instances of the *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons to obtain the EPP version. Consequently, the functions $mMul'_1$ and $mMul'_3$ are identified as instances of the *map* and *mapRedr* skeletons, respectively. Even though $mMul'_2$ is identified as an instance of the *accumulate* skeleton defined over a list, it cannot be parallelised as the reduction operator (:) is not associative, and hence its distilled form is used in the EPP version. Example 6.5.1.3 presents the EPP version defined using suitable calls to the corresponding Eden skeletons.

Example 6.5.1.3 (Matrix Multiplication – Encoded Parallel Program (EPP)):

$$
\begin{array}{l}
\textbf{data}\ T'_{mMul_1}\ a\ =\ c_1 \mid c_2 \mid c_3\ [a]\ [a] \\
\textbf{data}\ T'_{mMul_2}\ a\ =\ c_4 \mid c_5 \\
\textbf{data}\ T'_{mMul_3}\ a\ =\ c_6 \mid c_7 \mid c_8\ a\ [a]
\end{array}
$$

$$
\begin{array}{lll}
encode_{mMul_1}\ []\ zss & = & [c_1] \\
encode_{mMul_1}\ xss\ [] & = & [c_2] \\
encode_{mMul_1}\ (xs : xss)\ (zs : zss) & = & [c_3\ xs\ zs] +\!\!+ (encode_{mMul_1}\ xss\ zss)
\end{array}
$$

$$
\begin{array}{lll}
encode_{mMul_2}\ [] & = & [c_4] \\
encode_{mMul_2}\ (z : zs) & = & [c_5] +\!\!+ (encode_{mMul_2}\ xs\ yss\ zs)
\end{array}
$$

$$
\begin{array}{lll}
encode_{mMul_3}\ []\ yss & = & [c_6] \\
encode_{mMul_3}\ (x : xs)\ [] & = & [c_7] \\
encode_{mMul_3}\ (x : xs)\ (ys : yss) & = & [c_8\ x\ ys] +\!\!+ (encode_{mMul_3}\ xs\ yss)
\end{array}
$$

$mMul\ xss\ yss$
where

$$
\begin{array}{lll}
mMul\ xss\ yss & = & mMul''_1\ (encode_{mMul_1}\ xss\ yss)\ yss \\
mMul''_1\ w\ yss & = & farmB\ noPe\ f\ w \\
& & \textbf{where} \\
& & f\ c_1 = [] \\
& & f\ c_2 = [] \\
& & f\ (c_3\ xs\ zs) = \textbf{let}\ v = \lambda xs.g\ xs \\
& & \qquad\qquad\qquad\qquad \textbf{where} \\
& & \qquad\qquad\qquad\qquad g\ [] = 0 \\
& & \qquad\qquad\qquad\qquad g\ (x : xs) = x \\
& & \qquad\qquad \textbf{in}\ mMul''_2\ zs\ xs\ yss\ v \\
mMul''_2\ []\ xs\ yss\ v & = & [] \\
mMul''_2\ (z : zs)\ xs\ yss\ v & = & \textbf{let}\ v' = \lambda xs.g\ xs \\
& & \qquad\qquad\qquad\qquad \textbf{where} \\
& & \qquad\qquad\qquad\qquad g\ [] = 0 \\
& & \qquad\qquad\qquad\qquad g\ (x : xs) = v\ xs \\
& & \qquad\qquad \textbf{in}\ (mMul''_3\ (encode_{mMul_3}\ xs\ yss)\ v) : (mMul''_2\ zs\ xs\ yss\ v') \\
mMul''_3\ w\ v & = & offlineParMapRedr1\ g\ f\ w \\
& & \textbf{where} \\
& & g = (+) \\
& & f\ c_6 = 0 \\
& & f\ c_7 = 0 \\
& & f\ (c_8\ x\ ys) = x * (v\ ys)
\end{array}
$$

A hand-parallelised version (HPP) of the OP version is presented in Example 6.5.1.4, which is obtained by manually identifying instances of list-based *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons in the OP version. As a result, we identify that functions *mMul* and *map* can be defined using the *farmB* skeleton and *foldr* can be defined using the *offlineParMapRedr* skeleton presented in Section 4.2 to obtain the HPP version.

Example 6.5.1.4 (Matrix Multiplication – Hand-Parallelised Program (HPP)):

$mMul\ xss\ yss$
where

$$
\begin{array}{lll}
mMul\ xss\ yss & = & farmB\ noPe\ f\ xss \\
& & \textbf{where} \\
& & f\ xs = farmB\ noPe\ (dotp\ xs)\ (transpose\ yss)
\end{array}
$$

$$
\begin{array}{lcl}
dotp\ xs\ ys & = & offlineParMapRedr\ (+)\ 0\ id\ (zipWith\ (*)\ xs\ ys) \\
transpose\ xss & = & transpose'\ xss\ [] \\
transpose'\ []\ yss & = & yss \\
transpose'\ (xs : xss)\ yss & = & transpose'\ xss\ (rotate\ xs\ yss) \\
rotate\ []\ yss & = & yss \\
rotate\ (x : xs)\ [] & = & [x] : (rotate\ xs\ yss) \\
rotate\ (x : xs)\ (ys : yss) & = & (ys +\!\!+ [x]) : (rotate\ xs\ yss)
\end{array}
$$

Speedup Analysis

Figure 6.1 presents the speedup achieved by the distilled matrix multiplication program against the original program (OP) for various input sizes. An input size indicated by `NxM` denotes the multiplication of matrices of sizes `NxM` and `MxN`.

We observe that the speedup achieved from distillation depends on the size of the data structure that is eliminated. For instance, in cases where the intermediate data structures are smaller (for input sizes `100x100` and `1000x100` where the intermediate data structures are of lists of size 100), the speedup achieved from distillation is limited. However for larger intermediate data structures of sizes 250 and 1000 for inputs `250x250` and `100x1000`, respectively, the speedups achieved are significantly higher i.e. `4.34` and `25.16`, respectively.

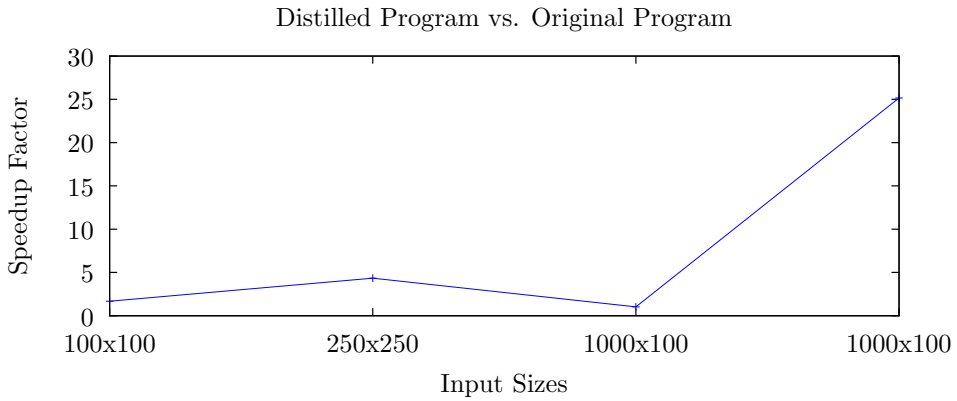


Figure 6.1: Speedup of Distilled Matrix Multiplication

Figure 6.2 presents the speedups of the EPP version compared to the OP, DP and HPP versions for varying input sizes. We observe that the speedup achieved by the EPP version in comparison with the OP version varies significantly depending on the input size. The speedup is roughly linear with the number of cores used for parallel execution for all input sizes except the input size `100x1000`. In the case with input size `100x1000`, the speedup achieved is around `6x-25x` more than the speedups achieved

for the other input sizes. This is due to the cost of the intermediate data structure *transpose yss* used for the *dotp* computation in the OP version. For the input size `100x1000`, *dotp* is applied over input lists of size `1000`, while for other input sizes, *dotp* is applied over lists of sizes `100-250`. Since the EPP version is free of intermediate data structures, the speedup achieved from both elimination of intermediate data structures and parallelisation results in a much greater speedup for the input size `100x1000`.

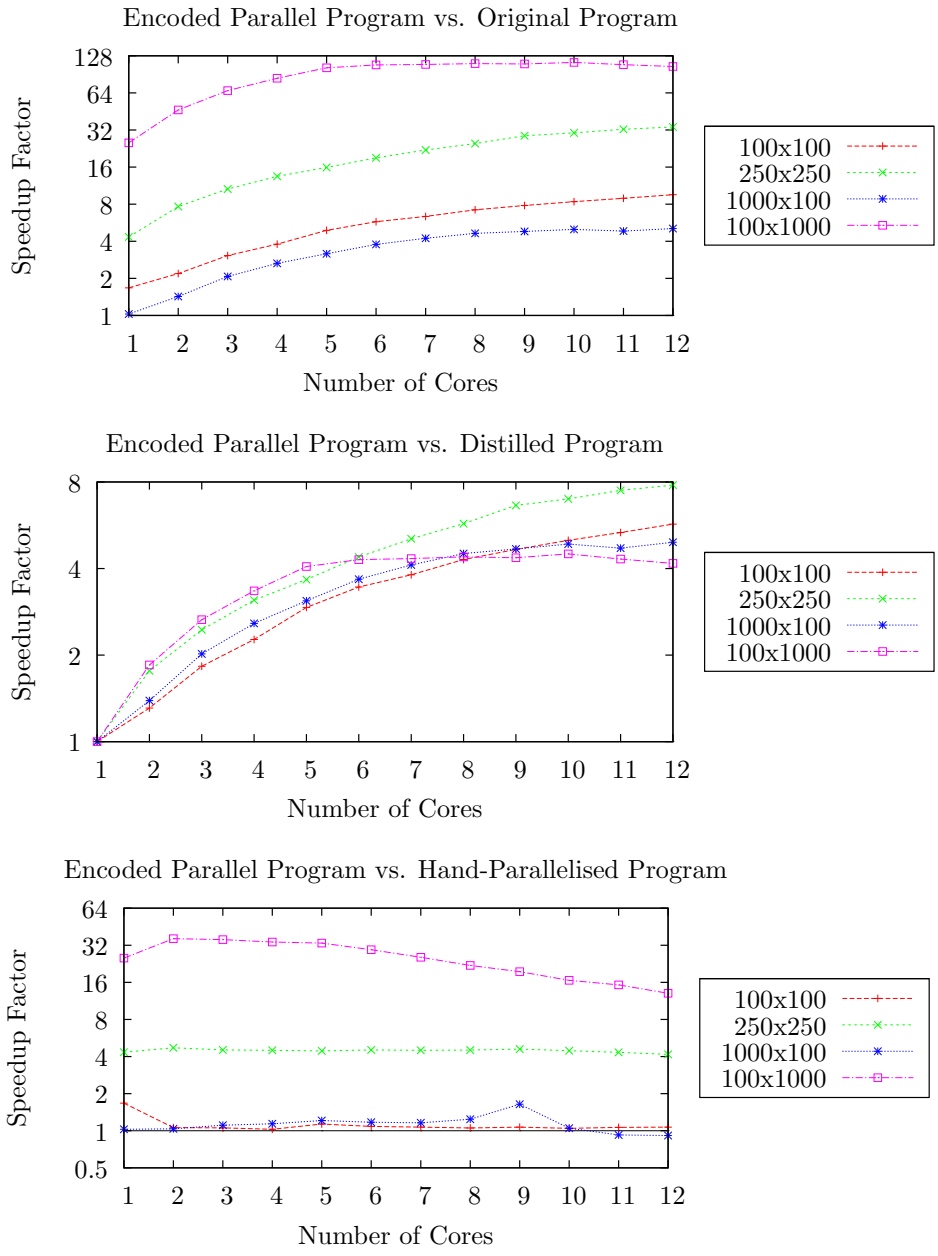


Figure 6.2: Speedup of Matrix Multiplication

This observation about the influence of intermediate data structures can be verified

when the speedup achieved by the EPP version is compared with the DP version. Here, since both the DP and EPP versions are free of intermediate data structures, parallelisation of the EPP version results in similar speedups for all input sizes and tails off when more than 8 cores are used. In particular, the speedup of the EPP version for input size `100x1000` against the DP version, which was markedly higher against the OP version, now scales on the same level as the other input sizes.

Figure 6.2 also presents a comparison of the speedup achieved by the EPP version with that achieved by the HPP version. It can be observed from Section 6.5.1 that both the HPP and EPP versions parallelise the equivalent computations that multiply the rows in the first matrix with the columns in the second matrix using the *farmB* skeleton. However, the EPP version is marginally faster than the HPP version for input sizes `100x100` and `1000x100`, and `4x-4.5x` faster for input size `250x250`. This is due to the use of intermediate data structures present in the HPP version, which is of size `100` for input sizes `100x100` and `1000x100`, and of size `250` for the size `250x250`. As discussed earlier, due to the large intermediate data structures for the input size `100x1000`, the speedup achieved by the EPP version in this case is `32x-15x` more than the HPP version. We observe that the HPP version scales better with a higher number of cores than the EPP version for the input size `100x1000`. This is because the EPP 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 HPP version.

Parallel Execution Analysis

Figure 6.3 presents the status of the cores from EdenTV when executing the EPP version of matrix multiplication on an input of size `250x250` on 6 cores.

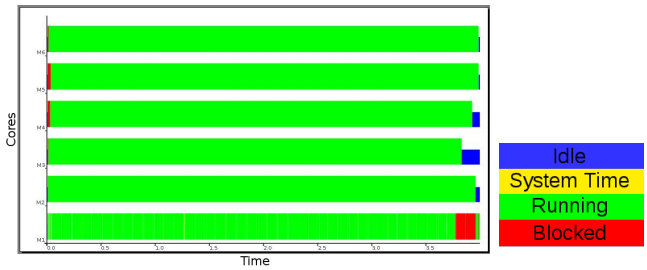


Figure 6.3: Matrix Multiplication – EPP Execution Profile from EdenTV – 250x250 on 6 cores

We observe from this parallel execution profile that the parallel computations are

well balanced across all cores without any idle (in blue) or waiting/blocked (in red) time. The first core has a waiting time (in red) at the end of computation because it runs the master thread that waits for the other threads to finish. This indicates a balanced workload across the cores when executing of the encoded parallel program.

Cost-Centre Analysis

The speedup of programs are computed using the execution time of the *mMul* function for all versions and include the execution time to build the input matrices and to encode them for function $mMul_1$ using the $encode_{mMul_1}$ function for the encoded version. Therefore, it is important to individually measure the overhead to build the inputs, encode the inputs and execute the $mMul$ function. Figure 6.4 illustrates that for all input sizes the overhead to build and encode the inputs is negligible compared to the total execution time for the encoded matrix multiplication program.

Input Size	*buildInputs* (%)	$encode_{mMul_1}$ (%)	$mMul$ (%)
`100x100`	1.2	0.0	98.8
`250x250`	0.2	0.0	99.8
`1000x100`	0.2	0.0	99.8
`100x1000`	0.2	0.0	99.8

Figure 6.4: Matrix Multiplication – Cost Centre of Encoded Program

Related Work

It is well known that matrix multiplication is well-suited for parallelisation. Thus, the speedup achieved is expected to scale almost linearly with the number of cores used for parallel evaluation [61]. We observed that the speedup achieved by the EPP version against the DP version scales linearly upto 5-6 cores, but saturated for additional number of cores. This is due to our approach to parallelise only the top-level skeletons to avoid nested parallel skeletons. Thus, while the dot-product of different rows and columns are performed in parallel, each dot-product computation is performed sequentially.

6.5.2 Evaluation of Power Tree

The original program (OP) to compute the sum of squares of a set of numbers placed at the leaves of a binary tree of type *BTree* is presented in Example 6.5.2.1. Since

this definition of *power* does not contain any intermediate data structures, the distilled program (DP) is the same as the OP version.

Example 6.5.2.1 (Power Tree – Original/Distilled Program (OP/DP)):

$$\textbf{data}\ BTree\ a\ =\ (L\ a)\ |\ B\ (BTree\ a)\ (BTree\ a)$$

$$power\ ::\ (BTree\ a) \rightarrow (BTree\ a) \rightarrow a$$
$$power\ xt$$
where
$$power\ (L\ x)\ =\ x * x$$
$$power\ (B\ xt_1\ xt_2)\ =\ (power\ xt_1) + (power\ xt_2)$$

Here, we observe that the recursive function *power* has a maximum of two recursive calls in its definition body. Consequently, based on the parallelisation steps presented in Section 5.2.2, we use the encoding transformation version 1 to encode the pattern-matched inputs into a new data type. Example 6.5.2.2 presents the resultant encoded program that encodes the input xt and yt into a new data type T_{power}. Here, we observe that the encoded data type created is of the same form as the original data type *BTree* because the data structure already matches the algorithmic structure of the program.

Example 6.5.2.2 (Power Tree – Encoded Program):

$$\textbf{data}\ T_{power}\ a\ =\ c_1\ a\ |\ c_2\ a\ (T_{power}\ a)\ (T_{power}\ a)$$
$$encode_{power}\ (L\ x)\ =\ c_1\ x$$
$$encode_{power}\ (B\ xt_1\ xt_2)\ =\ c_2\ (encode_{power}\ xt_1)\ (encode_{flip}\ xt_2)$$
$$power'\ (encode_{power}\ xt\ yt)$$
where
$$power'\ (c_1\ x)\ =\ x * x$$
$$power'\ (c_2\ x\ at\ bt)\ =\ (power'\ at) + (power'\ bt)$$

As discussed in Section 4.3.1, we define the reduce and map-reduce skeletons that operate over the encoded data type T_{power}. We identify that the $power'$ function is an instance of the $reduce_{power}$ skeleton shown in Example 6.5.2.3.

Example 6.5.2.3 (Reduce Skeleton Defined over T_{power}):

$$reduce_{power}\ (c_1\ x)\ g_1\ g_2\ =\ g_1\ x$$
$$reduce_{power}\ (c_2\ at\ bt)\ g_1\ g_2\ =\ g_2\ (reduce_{power}\ at)\ (reduce_{power}\ bt)$$

Consequently, the encoded function $power'$ is defined using a suitable call to the parallel $reduce_{power}$ skeleton as shown in Example 6.5.2.4.

Example 6.5.2.4 (Power Tree – Encoded Parallel Program (EPP)):

$$\textbf{data}\ T_{power}\ a\ =\ c_1\ a\ |\ c_2\ (T_{power}\ a)\ (T_{power}\ a)$$
$$encode_{power}\ (L\ x)\ =\ c_1\ x$$
$$encode_{power}\ (B\ xt_1\ xt_2)\ =\ c_2\ (encode_{power}\ xt_1)\ (encode_{power}\ xt_2)$$

$$
\begin{array}{l}
reduce_{power}\ (c_1\ x)\ t\ g_1\ g_2 \quad = g_1\ x \\
reduce_{power}\ (c_2\ at\ bt)\ t\ g_1\ g_2\ = \\
\qquad \left\{ \begin{array}{l} h\ (t \leq 0) \\ \textbf{where} \\ h\ True\ = g_2\ (reduce_{power}\ at\ t\ g_1\ g_2)\ (reduce_{power}\ bt\ t\ g_1\ g_2) \\ h\ False = runEval\ \$\ do \\ \qquad\qquad at' \leftarrow rpar\ (reduce_{power}\ at\ (t-1)\ g_1\ g_2) \\ \qquad\qquad bt' \leftarrow rseq\ (reduce_{power}\ bt\ (t-1)\ g_1\ g_2) \\ \qquad\qquad return\ (g_2\ at'\ bt') \end{array} \right. \\
power''\ (encode_{power}\ xt)\ t \\
\textbf{where} \\
power''\ w\ t\ =\ reduce_{power}\ w\ t\ g_1\ g_2 \\
\qquad\qquad \textbf{where} \\
\qquad\qquad g_1\ x \quad = (x * x) \\
\qquad\qquad g_2\ at\ bt\ =\ at + bt
\end{array}
$$

Here, the threshold value t for the $reduce_{power}$ skeleton can be initialised to $\lceil log_2\ P \rceil + 1$, where P is the number of processors used for parallel execution, based on the rule presented in Section 4.3.1.

Since the OP version of power tree does not contain any instance of skeletons whose implementations are available in Eden, the HPP version for the power tree program is defined using Glasgow Parallel Haskell (GpH) as shown in Example 6.5.2.5. Here, the evaluations of the two recursive calls $(power\ xt_1)$ and $(power\ xt_2)$ are parallelised using the *rpar* and *rseq* constructs.

Example 6.5.2.5 (Power Tree – Hand-Parallelised Program (HPP)):

$$
\begin{array}{l}
\textbf{data}\ BTree\ a\ =\ (L\ a)\ |\ B\ (BTree\ a)\ (BTree\ a) \\
power\ ::\ (BTree\ a) \rightarrow (BTree\ a) \rightarrow a \\
power\ xt\ t \\
\textbf{where} \\
power\ (L\ x)\ t \quad = x * x \\
power\ (B\ xt_1\ xt_2)\ t\ =\ h\ (t \leq 0) \\
\qquad\qquad \textbf{where} \\
\qquad\qquad h\ True\ = (power\ xt_1\ t) + (power\ xt_2\ t) \\
\qquad\qquad h\ False = runEval\ \$\ do \\
\qquad\qquad\qquad\qquad xt' \leftarrow rpar\ (power\ xt_1\ (t-1)) \\
\qquad\qquad\qquad\qquad yt' \leftarrow rseq\ (power\ xt_2\ (t-1)) \\
\qquad\qquad\qquad\qquad return\ xt' + yt'
\end{array}
$$

Speedup Analysis

Figure 6.5 presents the speedups achieved by the EPP version of the power tree program presented in Section 6.5.2 in comparison with the OP/DP and HPP versions. Here, an input of size `N` denotes a balanced binary tree with `N` leaf nodes.

When compared to the OP version, we observe that the EPP version achieves a positive speedup of `1.5x-2.7x` for all input sizes. The speedup achieved for the different input sizes scales equally for varying numbers of cores. When compared to the HPP version, we observe that both programs achieve similar speedups for all input sizes. This is primarily because both the EPP and HPP versions essentially parallelise the dot-product computation in a similar fashion as can be observed from their definitions in Section 6.5.3. The EPP version is marginally slower than the HPP version due to the added cost of encoding the inputs in the EPP version.

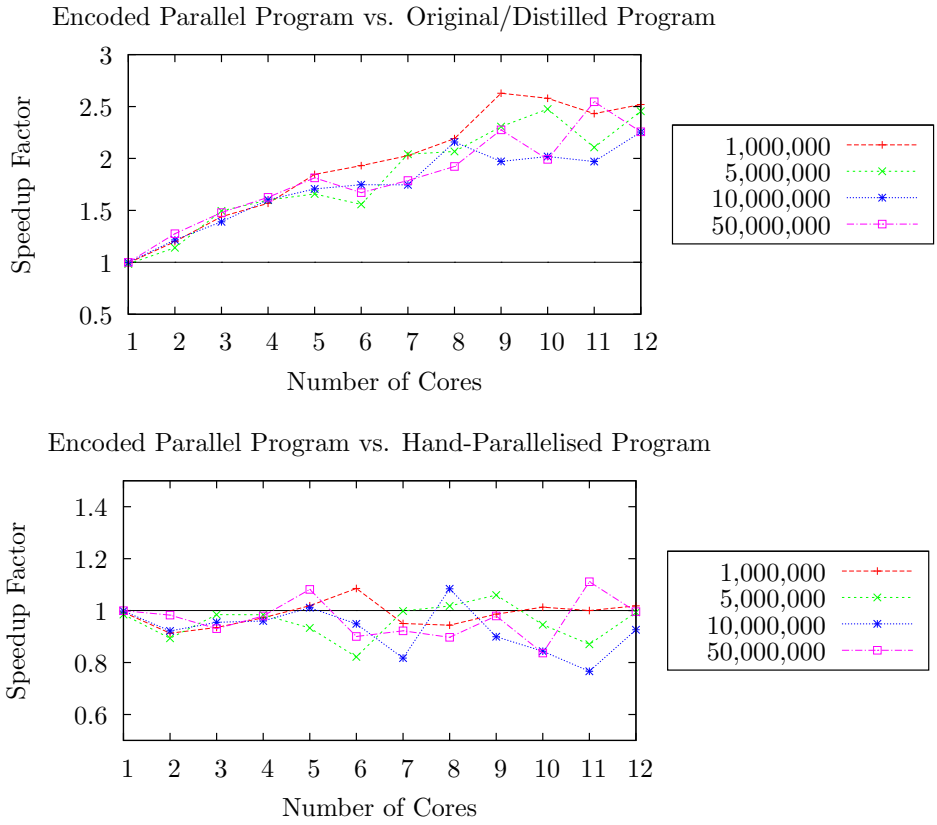


Figure 6.5: Speedup of Power Tree

We note from our experiments that if the input trees are not well-balanced, then speedups achieved by both the encoded parallel and hand-parallelised versions are reduced. This is because, from the definitions of these parallel versions in Section 6.5.2, it is evident that their parallelisation and workloads of the parallel threads are dictated by the structure of the input tree.

Parallel Execution Analysis

Figure 6.6 presents the status of the cores from Threadscope when executing the EPP version of the power tree program on an input of size `10,000,000` on 6 cores. Here, we observe from this parallel execution profile that the workloads of the parallel threads are nearly well-balanced across all cores. The execution times spent by the threads (in green) overlap thus indicating good parallelisation, and are interrupted only by the garbage collection phases (in orange).

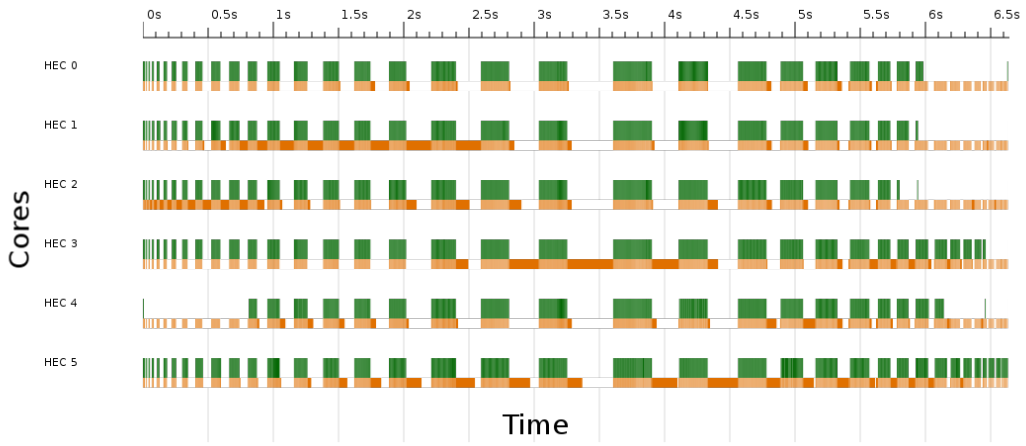


Figure 6.6: Power Tree – EPP Execution Profile from Threadscope – 10,000,000 on 6 cores

To understand the garbage collection (GC) behaviour, we consider the four phases involved in GC of Glasgow Parallel Haskell: (1) synchronisation, (2) initialise, (3) trace and (4) cleanup. In the evaluation experiments, the input trees and their encodings are precomputed before the parallel evaluation of the power tree program is performed. As a result, the synchronisation overhead during garbage collection due to shared data structures is not a significant part of the GC. From the Threadscope execution profile, we observe that the initialise phase (at the start of execution to setup the memory for GC) and the cleanup phase (at the end of execution to release the memory for GC) are negligible. However, the trace phase requires all threads to trace the live data and copy it. For the experiment illustrated in Figure 6.6, the size of the tree evaluated by each active thread is a binary tree with approximately 1,250,000 nodes, which results in copying 5.6 GB of data in total during the trace phase of GC. Repeating the experiments by varying the size of memory used for GC and the heap size did not significantly improve this behaviour. Consequently, the cost of the trace phase imparts a significant overhead

during the garbage collection operation.

Figure 6.7 presents the spark statistics when executing the EPP version of the power tree program on an input of size `10,000,000` on varying number of cores. Here, we observe that based on the thresholding technique introduced for the polytypic skeletons in Section 4.3.1, almost all sparks that are created are converted into useful parallel computation.

Cores	Total Sparks	Converted Sparks	Fizzled Sparks
2	3	2	1
3	7	6	1
4	7	6	1
5	15	14	1
6	15	14	1
7	15	15	0
8	15	15	0
9	31	30	1
10	31	31	0
11	31	30	1
12	31	31	0

Figure 6.7: Power Tree – Spark Statistics – 10,000,000 on 6 cores

Cost-Centre Analysis

For the encoded version of the power tree program, we measure the individual execution times for the *buildInputs*, $encode_{powerTree}$ and parallel *powerTree* functions. Since the *buildInputs* function constructs the input binary tree, its execution has a significant contribution to the total execution time. Figure 6.8 illustrates that for all input sizes the overhead to encode the input is `4%-5%` compared to the total execution time for the encoded version and the fraction of the program that can be parallelised, i.e. *powerTree* amounts to `8%-11.5%` of the total execution time.

Input Size	*buildInputs* (%)	$encode_{powerTree}$ (%)	*powerTree* (%)
`1,000,000`	83.3	5.2	11.6
`5,000,000`	86.1	4.1	9.8
`10,000,000`	86.6	4.3	9.1
`50,000,000`	88.2	3.9	7.9

Figure 6.8: Power Tree – Cost Centre of Encoded Program

Related Work

In [26], the authors evaluate a parallel power tree example where the input tree is flatted into a join-list which is split into sub-lists for parallel evaluation of each half. The speedup achieved by this method was estimated to be around `1.2x` on average when using between 2 and 12 cores.

6.5.3 Evaluation of Dot-Product of Binary Trees

The original program (OP) for computing the dot-product of two binary trees is presented in Example 6.5.3.1. Here, function *dotP* computes the product of values at the corresponding branch nodes of trees xt and yt, and adds the dot-products of the left and right sub-trees. Since this definition of *dotP* does not contain any intermediate data structures, the distilled program (DP) is the same as the OP version.

Example 6.5.3.1 (Dot Product of Binary Trees – OP/DP):
data $BTree\ a\ =\ E \mid B\ a\ (BTree\ a)\ (BTree\ a)$

$dotP\ ::\ (BTree\ a) \rightarrow (BTree\ a) \rightarrow a$
$dotP\ xt\ yt$
where
$dotP\ E\ yt \quad = 0$
$dotP\ (B\ x\ xt_1\ xt_2)\ E \quad = 0$
$dotP\ (B\ x\ xt_1\ xt_2)\ (B\ y\ yt_1\ yt_2) \ = (x * y) + (dotP\ xt_1\ yt_1) + (dotP\ xt_2\ yt_2)$

Here, we observe that recursive function *dotP* has a maximum of two recursive calls in its definition body. Consequently, based on the parallelisation steps presented in Section 5.2.2, we use the encoding transformation version 1 to encode the pattern-matched inputs into a new data type. Example 6.5.3.2 presents the resultant encoded program that encodes the inputs xt and yt into a new data type T_{dotP} that suits the recursive structure of *dotP*.

Example 6.5.3.2 (Dot Product of Binary Trees – Encoded Program):

$$
\begin{array}{l}
\textbf{data}\ T_{dotP}\ a\ =\ c_1 \mid c_2 \mid c_3\ a\ a\ (T_{dotP}\ a)\ (T_{dotP}\ a) \\
encode_{dotP}\ E\ yt \qquad\qquad\qquad\qquad\qquad\qquad = c_1 \\
encode_{dotP}\ (B\ x\ xt_1\ xt_2)\ E \qquad\qquad\qquad\qquad = c_2 \\
encode_{dotP}\ (B\ x\ xt_1\ xt_2)\ (B\ y\ yt_1\ yt_2) = c_3\ x\ y\ (encode_{dotP}\ xt_1\ yt_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (encode_{dotP}\ xt_2\ yt_2) \\
\\
dotP'\ (encode_{dotP}\ xt\ yt) \\
\textbf{where} \\
dotP'\ c_1 \qquad\qquad = 0 \\
dotP'\ c_2 \qquad\qquad = 0 \\
dotP'\ (c_3\ x\ y\ at\ bt)\ =\ (x * y) + (dotP'\ at) + (dotP'\ bt)
\end{array}
$$

As discussed in Section 4.3.1, we define the reduce and map-reduce skeletons that operate over the encoded data type T_{dotP}. We identify that the $dotP'$ function is an instance of the $reduce_{dotP}$ skeleton shown in Example 6.5.3.3.

Example 6.5.3.3 (Reduce Skeleton Defined over T_{dotP}):

$$
\begin{array}{l}
reduce_{dotP}\ c_1\ g_1\ g_2\ g_3 \qquad\qquad\qquad = g_1 \\
reduce_{dotP}\ c_2\ g_1\ g_2\ g_3 \qquad\qquad\qquad = g_2 \\
reduce_{dotP}\ (c_3\ x\ y\ at\ bt)\ g_1\ g_2\ g_3\ =\ g_3\ x\ y\ (reduce_{dotP}\ at)\ (reduce_{dotP}\ bt)
\end{array}
$$

Consequently, the encoded function $dotP'$ is defined using a suitable call to the parallel $reduce_{dotP}$ skeleton as shown in Example 6.5.3.4.

Example 6.5.3.4 (Dot Product of Binary Trees – EPP):

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

$dotP''\ (encode_{dotP}\ xt\ yt)\ t$
where
$dotP''\ w\ t\ =\ reduce_{dotP}\ w\ t\ g_1\ g_2\ g_3$
where
$g_1 = 0$
$g_2 = 0$
$g_3\ x\ y\ x'\ y' = (x * y) + x' + y'$

Here, the threshold value t for the $mapReduce_{dotP}$ skeleton can be initialised to $\lceil log_2\ P \rceil + 1$, where P is the number of processors used for parallel execution, based on the rule presented in Section 4.3.1.

Since the OP version of dot-product of binary trees does not contain any instance of skeletons whose implementations are available in Eden, the HPP version for the dot-product of binary trees program is defined using Glasgow Parallel Haskell (GpH) as shown in Example 6.5.3.5. Here, the evaluations of the two recursive calls ($dotP\ xt_1\ yt_1$) and ($dotP\ xt_2\ yt_2$) are parallelised using the *rpar* and *rseq* constructs.

Example 6.5.3.5 (Dot Product of Binary Trees – HPP):
data $BTree\ a\ =\ E \mid B\ a\ (BTree\ a)\ (BTree\ a)$
$dotP\ ::\ (BTree\ a) \rightarrow (BTree\ a) \rightarrow a$
$dotP\ xt\ yt\ t$
where
$dotP\ E\ yt\ t = 0$
$dotP\ (B\ x\ xt_1\ xt_2)\ E\ t = 0$
$dotP\ (B\ x\ xt_1\ xt_2)\ (B\ y\ yt_1\ yt_2)\ t = h\ (t \leq 0)$
where
$h\ True\ = (dotP\ xt_1\ yt_1\ t) + (dotP\ xt_2\ yt_2\ t)$
$h\ False = runEval\ \$\ do$
$x' \leftarrow rpar\ (dotP\ xt_1\ yt_1\ (t-1))$
$y' \leftarrow rseq\ (dotP\ xt_2\ yt_2\ (t-1))$
$return\ (x * y) + x' + y'$

Speedup Analysis

Figure 6.9 presents the speedups of the encoded parallel version compared to the OP version and the HPP version. An input size indicated by `N` denotes the dot-product of two identical balanced binary trees with `N` nodes each.

Overall, we observe that the performance of the EPP version is similar to that of the EPP version of the power tree program. When compared to the OP version, we observe that the EPP version of dot-product 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 HPP version, we observe that both programs

achieve similar speedups for all input sizes. Similar to the performance of *powerTree* in Section 6.5.2, this is primarily because both the EPP and HPP versions essentially parallelise the dot-product computation in a similar fashion as can be observed from their definitions in Section 6.5.3.

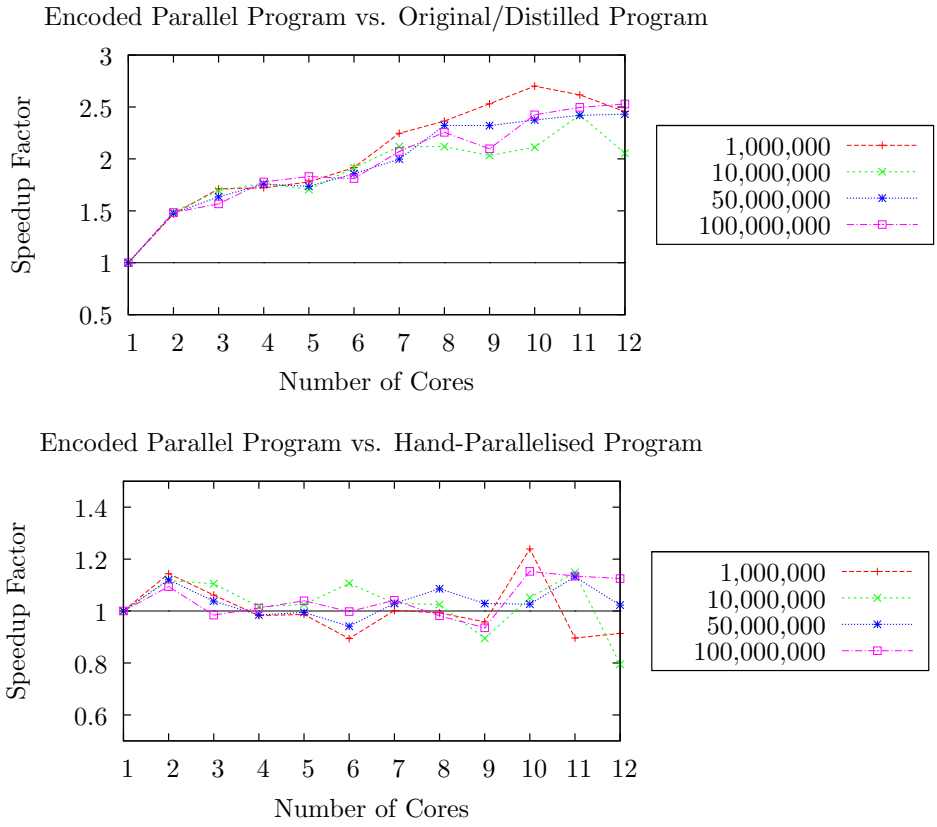


Figure 6.9: Speedup of Dot Product of Binary Trees

Similar to the power tree program, 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-parallelised versions are reduced.

Parallel Execution Analysis

Figure 6.10 presents the status of the cores from Threadscope when executing the EPP version of the dot-product program on an input of size `10,000,000` on 6 cores. Here, we observe from this parallel execution profile that the workloads of the parallel threads are nearly well-balanced across all cores. The execution times spent by the threads (in green) overlap thus indicating good parallelisation, and are interrupted only by the garbage collection phases (in orange). The reason behind the overhead from garbage

collection is the same as observed in the power tree example.

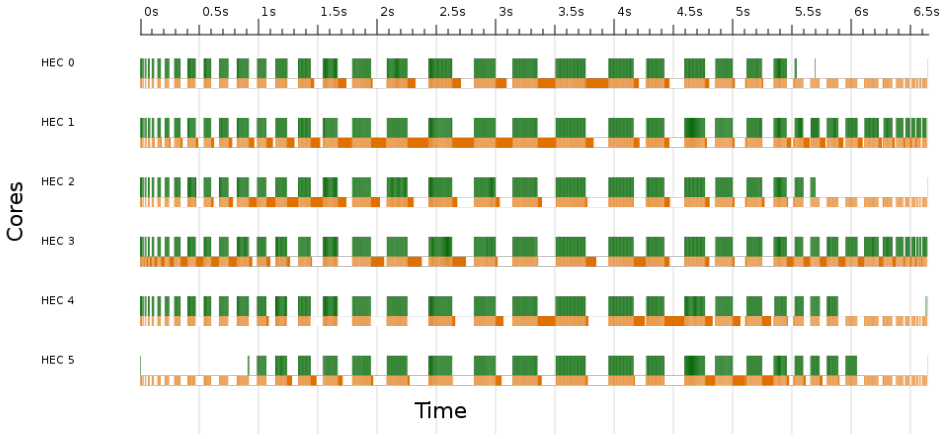


Figure 6.10: Dot Product of Binary Trees – EPP Execution Profile from Threadscope – 10,000,000 on 6 cores

Figure 6.11 presents the spark statistics when executing the EPP version of the dot product of binary trees program on an input of size `10,000,000` on varying number of cores. Here, we observe that based on the thresholding technique introduced for the polytypic skeletons in Section 4.3.1, almost all sparks that are created are converted into useful parallel computation.

Cores	Total Sparks	Converted Sparks	Fizzled Sparks
2	3	2	1
3	7	6	1
4	7	6	1
5	15	15	0
6	15	15	0
7	15	15	0
8	15	15	0
9	31	31	0
10	31	31	0
11	31	30	1
12	31	31	0

Figure 6.11: Power Tree – Spark Statistics – 10,000,000 on 6 cores

Cost-Centre Analysis

For the encoded version of the dot-product program, we measure the individual execution times for the *buildInputs*, $encode_{dotP}$ and parallel *dotP* functions. Since the *buildInputs* function constructs the two input binary trees, its execution has a significant contribution to the total execution time. Figure 6.12 illustrates that for all input sizes the overhead to encode the input is `5%-6%` compared to the total execution time for the encoded version and the fraction of the program that can be parallelised, i.e. *dotP* amounts to `13%-16%` of the total execution time.

Input Size	*buildInputs* (%)	$encode_{dotP}$ (%)	*dotP* (%)
`1,000,000`	77.8	6.3	15.9
`10,000,000`	80.6	5.0	14.4
`50,000,000`	80.7	6.3	13.0
`100,000,000`	82.4	4.9	12.7

Figure 6.12: Dot Product of Binary Trees – Cost Centre of Encoded Program

6.5.4 Evaluation of Totient

The OP version of a program to compute the totient of a given value is presented in Example 6.5.4.1. Here, the function $totient_1$ is defined over one input argument x and counts the number of values from *Zero* to x that are relatively prime to x using the recursive function $totient_2$. Since this definition of $totient_1$ does not contain any intermediate data structures, the distilled program (DP) is the same as the OP version.

Example 6.5.4.1 (Totient – Original/Distilled Program (OP/DP)):

```
data IntValue = Zero | Succ IntValue

totient1 :: IntValue → Int
totient1 x
where
totient1 x          = totient2 x x

totient2 Zero y     = 0
totient2 (Succ x) y = (coPrime x y) + (totient2 x y)

coPrime x y         = h (1 == (gcd x y))
                      where
                      h True  = 1
                      h False = 0
```

Here, we observe that recursive function $totient_2$ has a maximum of one recursive

call in its definition body. Consequently, based on the parallelisation steps presented in Section 5.2.2, we use the encoding transformation version 2 to encode the pattern-matched inputs into a *cons*-list. Example 6.5.4.2 presents the encoded version of the DP version.

Example 6.5.4.2 (Totient – Encoded Program):

$$
\begin{array}{l}
\textbf{data}\ T'_{totient_2}\ a\ =\ c_1 \mid c_2\ a \\
\\
encode_{totient_2}\ Zero\ \ \ \ \ \ \ \ =\ [c_1] \\
encode_{totient_2}\ (Succ\ x)\ =\ [c_2\ x] +\!\!+ (encode_{totient_2}\ x) \\
\\
totient_1\ x \\
\textbf{where} \\
totient_1\ x\ \ \ \ \ \ \ \ \ \ \ \ \ \ =\ totient'_2\ (encode_{totient_2}\ x)\ x \\
\\
totient'_2\ (c_1 : w)\ y\ \ \ \ \ \ =\ 0 \\
totient'_2\ \big((c_2\ x) : w\big)\ y\ =\ (coPrime\ x\ y) + (totient'_2\ w\ y) \\
\\
coPrime\ x\ y\ \ \ \ \ \ \ \ \ \ \ =\ h\ (1 == (gcd\ x\ y)) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\quad h\ True\ \ =\ 1 \\
\qquad\qquad\qquad\qquad\qquad\quad h\ False\ =\ 0
\end{array}
$$

Using the encoded program in Example 6.5.4.2, we apply the skeleton identification technique to identify potential instances of the *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons. Consequently, the function $totient'_2$ is identified as an instance of the *mapRedr* skeleton. Example 6.5.4.3 presents the encoded parallel program (EPP) defined using a suitable call to the *offlineParMapRedr1* skeleton in Eden.

Example 6.5.4.3 (Totient – Encoded Parallel Program (EPP)):

$$
\begin{array}{l}
\textbf{data}\ T'_{totient_2}\ a\ =\ c_1 \mid c_2\ a \\
\\
encode_{totient_2}\ Zero\ \ \ \ \ \ \ \ =\ [c_1] \\
encode_{totient_2}\ (Succ\ x)\ =\ [c_2\ x] +\!\!+ (encode_{totient_2}\ x) \\
\\
totient_1\ x \\
\textbf{where} \\
totient_1\ x\ \ \ \ \ \ \ \ \ \ \ \ \ \ =\ totient''_2\ (encode_{totient_2}\ x)\ x \\
totient''_2\ w\ y\ \ =\ offlineParMapRedr1\ g\ f\ w \\
\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\quad g\ \ \ \ \ \ \ \ =\ (+) \\
\qquad\qquad\qquad\quad f\ c_1\ \ \ \ \ \ =\ 0 \\
\qquad\qquad\qquad\quad f\ (c_2\ x)\ =\ coPrime\ x\ y \\
coPrime\ x\ y\ =\ h\ (1 == (gcd\ x\ y)) \\
\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\quad h\ True\ \ =\ 1 \\
\qquad\qquad\qquad\quad h\ False\ =\ 0
\end{array}
$$

Since the OP version of the totient program does not contain any instance of skeletons

whose implementations are available in Eden, the HPP version for the totient program is defined using GpH as shown in Example 6.5.4.4. Here, to compute the totient for a value x, we first build a list of values $[x, \ldots, (Succ\ Zero)]$ using $(buildList\ x)$ following which each element in the list is to be checked for relative primality with x. To achieve this, the totient computation is parallelised in the function $totient_2$ by splitting the input list into halves and then checking the relative primality on elements in each half simultaneously using the *rpar* and *rseq* constructs. Here, the threshold value t for the parallel definition can be initialised to $\lceil log_2\ P \rceil + 1$, where P is the number of processors used for parallel execution, as the evaluation tree follows the structure of a binary tree.

Example 6.5.4.4 (Totient – Hand-Parallelised Program (HPP)):

$$
\begin{array}{lcl}
totient_1\ x\ t & & \\
\textbf{where} & & \\
totient_1\ x\ t & = & totient_2\ (buildList\ x)\ x\ t \\
totient_2\ []\ y\ t & = & 0 \\
totient_2\ (x:xs)\ y\ t & = & (coPrime\ x\ y) + \\
 & & \left(\begin{array}{l}
h\ (t \leq 1) \\
\textbf{where} \\
h\ True\ = totient_2\ xs\ y\ t \\
h\ False\ = \textbf{let}\ (xs_1, xs_2) = split\ xs \\
\qquad\qquad \textbf{in}\ runEval\ \$\ do \\
\qquad\qquad\quad x_1\ \rightarrow\ rpar\ (totient_2\ xs_1\ y\ (t-1)) \\
\qquad\qquad\quad x_2\ \rightarrow\ rseq\ (totient_2\ xs_2\ y\ (t-1)) \\
\qquad\qquad\quad return\ (x_1 + x_2)
\end{array}\right) \\
coPrime\ x\ y & = & h\ (1 == (gcd\ x\ y)) \\
 & & \textbf{where} \\
 & & h\ True\ = 1 \\
 & & h\ False\ = 0 \\
buildList\ Zero & = & [] \\
buildList\ (Succ\ x) & = & (Succ\ x) : (buildList\ x)
\end{array}
$$

Speedup Analysis

Figure 6.13 presents the speedups of the EPP version compared to the OP/DP and the HPP versions for varying input sizes. An input size indicated by `N` denotes the computation of the totient function for the value `N`.

We observe that the EPP version achieves a positive speedup of `1.1x-1.8x` on average for all input sizes when more than two cores are used. We also observe good scalability of the parallel execution for varying input sizes. However, the speedup gained tails off beyond the use of 8 cores and the overall speedup gained does not scale linearly with the number of cores used. A closer inspection of the parallel execution profile

explains the reason for this behaviour.

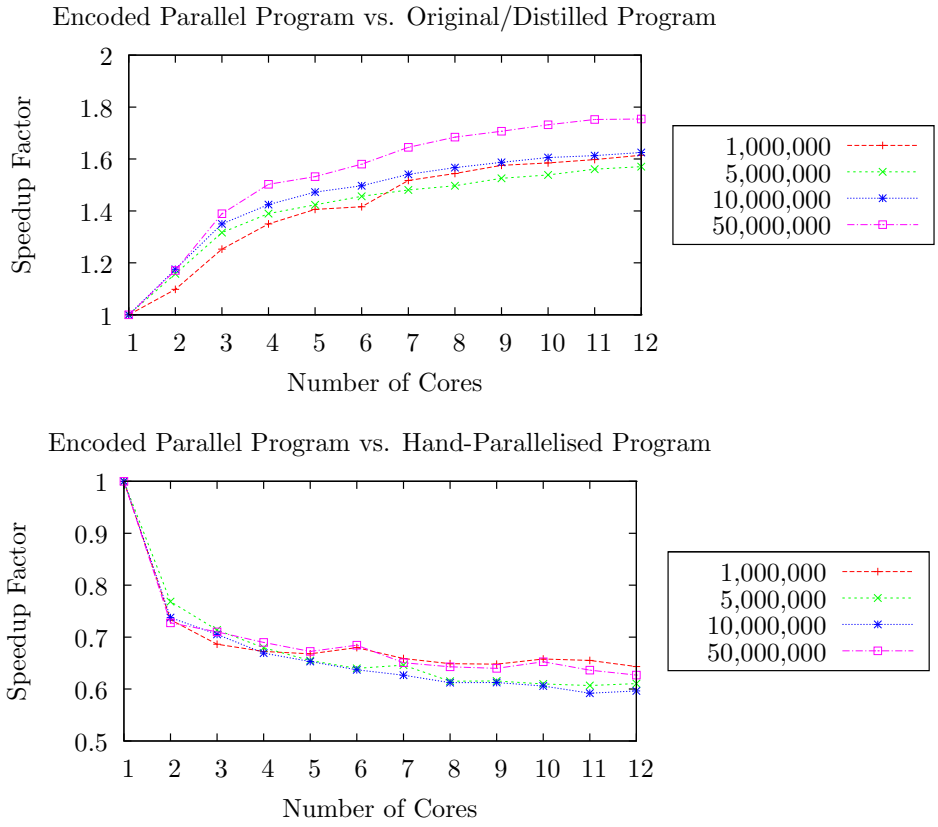


Figure 6.13: Speedup of Totient

When compared to the HPP version, we observe that the EPP version of the totient program does not deliver as much speedup as the hand-parallelised program. In particular, we observe that the HPP version is faster by a factor of `1.3x-1.6x` than the EPP version. We believe that this is due to the cost of encoding the inputs that is performed by the EPP version. The cost of encoding is discussed in the cost-centre analysis in Section 6.5.4.

Parallel Execution Analysis

Figure 6.14 presents the status of the cores when executing the EPP version of the totient program on an input of size `10,000,000` on 6 cores. We observe from this parallel execution profile that the parallel computations are well balanced across all cores without any idle (in blue) or waiting/blocked (in red) time, except the fractional idle times at the end when the results from the threads are aggregated and reduced by the master thread. This indicates a balanced workload across the cores when executing

of the encoded parallel program.

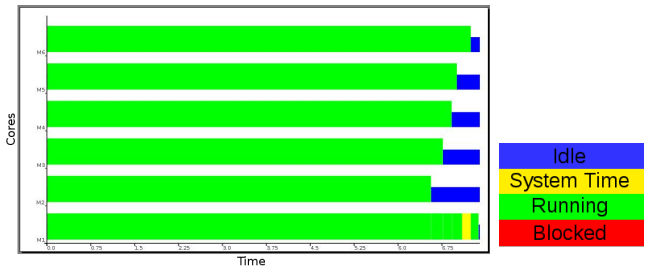


Figure 6.14: Totient – EPP Execution Profile from EdenTV – 10,000,000 on 6 cores

Cost-Centre Analysis

We observe that despite good parallel execution as illustrated in Figure 6.14, the speedup achieved by the EPP version does not scale linearly with the number of cores used even though it scales well with varying input sizes. This is because, as shown in the cost-centre analysis in Figure 6.15, even though the function $totient_2$ contributes to `85%` of the execution time and is defined using the *offlineParMapRedr1* skeleton, the cost of the computation which is executed by the parallel threads i.e. *coPrime* is only `2.5%`. This shows that the cost of the computation done in parallel is significantly low compared to the cost of parallelisation. Thus, a majority of the computation time is spent in the $totient_2''$ function to perform the sequential computations including splitting the input into sub-lists.

As a result, the speedup achieved from the parallelising the low-cost *coPrime* computation is also limited. This may be addressed in future by using a technique to split the input list in such a way that the number of sub-lists is decided based on the cost of parallel computation performed on each sub-list, instead of creating `P` sub-lists for parallel computation on `P` processors. This could improve the cost of computation performed by each parallel process leading to a better gain from parallelisation.

Input Size	*buildInputs* (%)	$encode_{totient_2}$ (%)	$totient_2$ (%)	*coPrime* (%)
`1,000,000`	0.0	14.0	83.5	2.5
`5,000,000`	0.0	14.2	83.3	2.5
`10,000,000`	0.0	12.6	84.9	2.5
`50,000,000`	0.0	12.2	85.5	2.3

Figure 6.15: Totient – Cost Centre of Encoded Program

Related Work

A refactoring-based parallelisation of the totient program was evalauated in [10]. Here, the authors report a near-linear speedup for an input list of size 10,000 when using upto 8 cores for parallel execution. Upon additing more cores, the speedup achieved saturates due to increased communication and synchronisation costs. The speedup achieved by our HPP version of the totient program is `2x-4x` when using 2 to 12 cores. We believe that this reduced speedup of the HPP versions is due to the additional cost of building the input list included in our evaluations, but is not in the results presented in [10].

6.5.5 Evaluation of Maximum Prefix Sum

The OP version of a program to compute the maximum prefix sum of a given list is presented in Example 6.5.5.1, 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 6.5.5.1 (Maximum Prefix Sum – Original Program (OP)):

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

The result of distilling the OP version of the maximum prefix sum program is the presented in Example 6.5.5.2.

Example 6.5.5.2 (Maximum Prefix Sum – Distilled Program (OP)):

$$
\begin{array}{l}
mps_1 \ :: \ [Int] \rightarrow Int \\
mps_1 \ xs \\
\textbf{where} \\
mps_1 \ xs \ = \ mps_2 \ xs \ 0 \\
mps_2 \ [] \ v \quad\quad\quad = \ v \\
mps_2 \ (x : xs) \ v \ = \ \textbf{let} \ v' = x + v \\
\qquad\qquad\qquad\quad\ \ \textbf{in} \ max \ v \ (mps_2 \ xs \ v')
\end{array}
$$

Here, we observe that recursive function mps_2 has a maximum of one recursive call in its definition body. Consequently, based on the parallelisation steps presented in Section 5.2.2, we use the encoding transformation version 2 to encode the pattern-matched inputs into a *cons*-list. Example 6.5.5.3 presents the encoded version of the DP version.

Example 6.5.5.3 (Maximum Prefix Sum – Encoded Program):

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

Using the encoded program in Example 6.5.5.3, we apply the skeleton identification technique to identify potential instances of the *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons. Consequently, the function mps'_2 is identified as an instance of the *accumulate* skeleton. Example 6.5.5.4 presents the encoded parallel program (EPP) defined using a suitable call to the *parAccumulate1* skeleton defined using Eden. The parallel execution of this definition is possible due to the associativity of the operators *max* and +, which can be verified as described in Section 4.2.1.

Example 6.5.5.4 (Maximum Prefix Sum – Encoded Parallel Program (EPP)):

$$
\begin{array}{l}
\textbf{data}\ T'_{mps_2}\ a\ =\ c_1 \mid c_2\ a \\
\\
encode_{mps_2}\ []\ \ \ \ \ \ \ \ = [c_1] \\
encode_{mps_2}\ (x : xs)\ = [c_2\ x] +\!\!+ (encode_{mps_2}\ xs) \\
\\
mps_1\ xs \\
\textbf{where} \\
mps_1\ xs\ \ \ \ \ \ \ \ \ \ \ \ = mps''_2\ (encode_{mps_2}\ xs)\ 0 \\
mps''_2\ w\ v\ = parAccumulate1\ p\ \oplus\ q\ \otimes\ w\ v \\
\qquad\qquad \textbf{where} \\
\qquad\qquad p\ c_1\ v\ \ \ \ \ \ = v \\
\qquad\qquad p\ (c_2\ x)\ v\ = v \\
\qquad\qquad \oplus\ \ \ \ \ \ \ \ = max \\
\qquad\qquad q\ c_1\ \ \ \ \ \ \ = v \\
\qquad\qquad q\ (c_2\ x)\ \ \ = x \\
\qquad\qquad \otimes\ \ \ \ \ \ \ \ = (+)
\end{array}
$$

The OP version does not contain any instances of list-based skeletons. However, a hand-parallelised version (HPP) of the DP version is presented in Example 6.5.5.5. As explained in Section 6.3, the HPP version is obtained by manually identifying instances of list-based *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons, and nesting of parallel skeletons is avoided. As a result, we identify that function mps_2 is an instance of the *accumulate* skeleton presented in Section 4.2 and therefore define it using a suitable call to the *parAccumulate* skeleton that was added to the Eden library.

Example 6.5.5.5 (Maximum Prefix Sum – Hand-Parallelised Program (HPP)):

$mps_1\ xs$
where
$mps_1\ xs \quad = mps_2\ xs\ 0$
$mps_2\ xs\ v\ = parAccumulate\ p\ \oplus\ q\ \otimes\ xs\ v$
where
$p\ x\ v\ = max\ v\ (x + v)$
$\oplus \quad = max$
$q \quad = \lambda x.x$
$\otimes \quad = (+)$

Speedup Analysis

Figure 6.16 presents the speedup achieved by the distilled maximum prefix sum program against the original program (OP) for various input sizes. An input size indicated by `N` denotes the computation of the maximum prefix sum for a list of size `N`. We observe that distillation produces a speedup of around `2x` for all input sizes presented here.

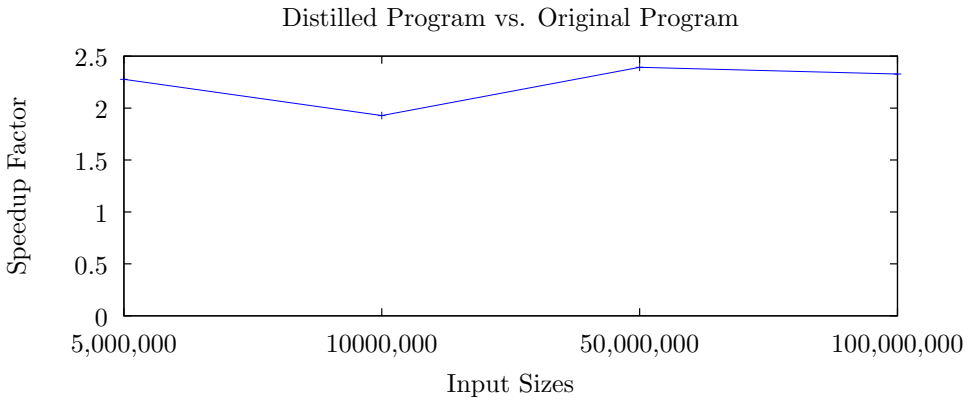


Figure 6.16: Speedup of Distilled Maximum Prefix Sum

Figure 6.17 presents the speedups of the EPP version compared to the OP, DP and HPP versions for varying input sizes.

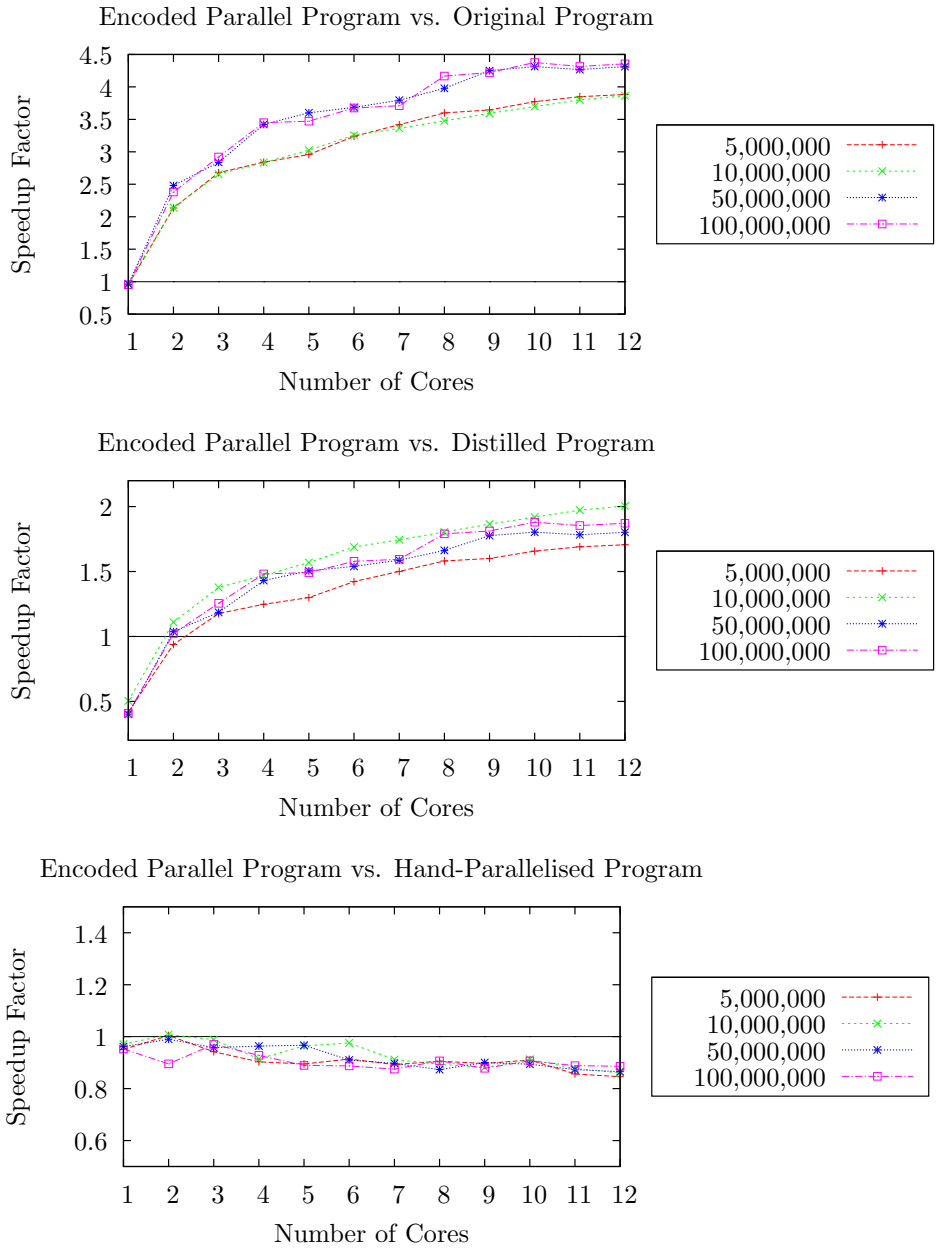


Figure 6.17: Speedup of Maximum Prefix Sum

In comparison to the OP version, we observe that the EPP 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 EPP version scales well for different input sizes. However, the overall speedup gained does not scale linearly with the number of cores used. From the comparison of the EPP version with the DP version, we observe that around `50%` of the speedup achieved by the EPP version is the result of the distillation transformation. This is because the DP version has fewer *max*

comparisons as can be observed from the transformed programs in Section 6.5.5.

We also observed that both the HPP and EPP versions parallelise the maximum prefix sum program in precisely the same way using the *accumulate* skeleton, with the HPP version defined over the original input list and the EPP version defined over the encoded list. Consequently, we observe in our evaluations that the HPP version is marginally faster than the EPP version for all input sizes. This difference in speedups is due to the cost of encoding inputs performed by the EPP version. The function to encode the inputs adds a constant factor to the total execution time of the EPP version resulting in a marginal slowdown by a constant factor.

Parallel Execution Analysis

Figure 6.18 presents the status of the cores when executing the EPP version of the maximum prefix sum program on an input of size `5,000,000` on 6 cores. We observe from this parallel execution profile that the parallel computations are well balanced across all cores without any idle (in blue) or waiting/blocked (in red) time. The fractional idle times at the end on cores 2-5 are due to the time to aggregate the results, respectively. This indicates a balanced workload across the cores when executing of the encoded parallel program.

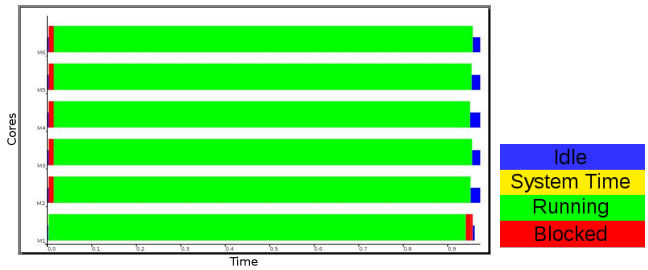


Figure 6.18: Maximum Prefix Sum – EPP Execution Profile from EdenTV – 5,000,000 on 6 cores

However, despite the good parallel execution, the speedup achieved by the EPP version does not scale linearly with the number of cores used even though it scales well with varying input sizes. We attribute this behaviour to the cost of the parallel threads; since the gain from parallel execution is limited when compared to the cost of parallelisation, the overall speedup achieved from parallel execution is also limited. This reasoning about the scalability is also justified by the performance of the HPP version which is identical to that of the EPP version (when the cost to encode inputs is omitted).

Cost-Centre Analysis

For the EPP version of the maximum prefix sum program, we measure the individual execution times for the *buildInputs*, $encode_{mps_2}$, mps_2 and mps_1 functions. Figure 6.19 illustrates that for all input sizes the cost of encoding the input is around `62%` of the total execution time for the encoded version while the execution of mps_2 defined using the *accumulate* skeleton amount to only `38%`. This further justifies our reasoning that the parallel execution of the EPP version is marginally slower than the HPP version due to the cost of encoding the inputs.

Input Size	*buildInputs* (%)	$encode_{mps_2}$ (%)	mps_2 (%)	mps_1 (%)
`5,000,000`	0.0	62.4	37.6	0.0
`10,000,000`	0.0	62.4	37.6	0.0
`50,000,000`	0.0	61.2	38.8	0.0
`100,000,000`	0.0	61.5	38.5	0.0

Figure 6.19: Maximum Prefix Sum – Cost Centre of Encoded Program

Related Work

In [26], the authors evaluate a parallel maximum prefix sum program. For input sizes between 10,000 to 1,000,000 elements, the reported speedup achieved was around `1.4x` when using 2 to 12 cores. Even though parallelisation of maximum prefix sum has been discussed in [24], the speedup achieved has not been evaluated in such work.

6.5.6 Evaluation of Sum Squares of List

The OP version of the sum of squares program is presented in Example 6.5.6.1. Here, function *sumSquares* computes the sum of squares of each element in a given list using the Haskell library function *sum*. Here, the intermediate data structure of produced by the *map* function is decomposed by *sum* for computing the result.

Example 6.5.6.1 (Sum Squares of List – Original Program (OP)):

$sumSquares\ xs\ =\ sum\ (map\ (\lambda x.x * x)\ xs)$

Based on the procedure in Section 6.2, we first apply the distillation transformation to reduce the number of intermediate data structures. Example 6.5.6.2 presents the distilled program (DP) obtained for the OP version, where distillation uses the library

definitions of *sum* and *map*. In Example 6.5.6.2, the function *sumSquares* computes the sum of the squared elements without the use of any intermediate data structure.

Example 6.5.6.2 (Sum Squares of List – Distilled Program (DP)):

$$
\begin{array}{l}
sumSquares\ xs \\
\textbf{where} \\
sumSquares\ [] \quad = 0 \\
sumSquares\ (x : xs) = (x * x) + (sumSquares\ xs)
\end{array}
$$

Here, we observe that recursive function *sumSquares* has a maximum of one recursive call in its definition body. Consequently, based on the parallelisation steps presented in Section 5.2.2, we use the encoding transformation version 2 to encode the pattern-matched inputs into a *cons*-list. Example 6.5.6.3 presents the encoded version of the DP version.

Example 6.5.6.3 (Sum Squares of List – Encoded Program):

$$
\begin{array}{l}
\textbf{data}\ T'_{sumSquares}\ a = c_1 \mid c_2\ a\ (T'_{sumSquares}\ a) \\
encode_{sumSquares}\ [] \quad = [c_1] \\
encode_{sumSquares}\ (x : xs) = [c_2\ x] +\!\!+ (encode_{sumSquares}\ xs)
\end{array}
$$

$$
\begin{array}{l}
sumSquares'\ xs \\
\textbf{where} \\
sumSquares'\ (c_1 : w) \quad = 0 \\
sumSquares'\ ((c_3\ x) : w) = (x * x) + (sumSquares'\ w)
\end{array}
$$

Using the encoded program in Example 6.5.6.3, we apply the skeleton identification technique to identify potential instances of the *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons to obtain the EPP version. Consequently, the function $sumSquares'$ is identified as an instance of the *mapRedr* skeleton. Example 6.5.6.4 presents the EPP version defined using suitable calls to the corresponding Eden skeleton.

Example 6.5.6.4 (Sum Squares of List – Encoded Parallel Program (EPP)):

$$
\begin{array}{l}
\textbf{data}\ T'_{sumSquares}\ a = c_1 \mid c_2\ a\ (T'_{sumSquares}\ a) \\
encode_{sumSquares}\ [] \quad = [c_1] \\
encode_{sumSquares}\ (x : xs) = [c_2\ x] +\!\!+ (encode_{sumSquares}\ xs)
\end{array}
$$

$$
\begin{array}{l}
sumSquares''\ (encode_{sumApp}\ xs) \\
\textbf{where} \\
sumSquares''\ w = offlineParMapRedr1\ g\ f\ w \\
\qquad\qquad \textbf{where} \\
\qquad\qquad g \quad = (+) \\
\qquad\qquad f\ c_1 \quad = 0 \\
\qquad\qquad f\ (c_2\ x) = x * x
\end{array}
$$

A hand-parallelised version (HPP) of the DP version is presented in Example 6.5.6.5, which is obtained by manually identifying instances of list-based *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons in the DP version as it is free of intermediate data structures. Following this, we identify that the distilled *sumSquares* function is an instance of the *mapRedr* skeleton presented in Section 4.2 and hence can be defined using the *offlineParMapRedr* skeleton in the Eden library.

Example 6.5.6.5 (Sum Squares of List – Hand-Parallelised Program (HPP)):

$$
\begin{array}{l}
sumSquares\ xs \\
\textbf{where} \\
sumSquares\ xs\ =\ offlineParMapRedr\ g\ v\ f\ xs \\
\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\quad g\ \ \ \ =\ (+) \\
\qquad\qquad\quad v\ \ \ \ =\ 0 \\
\qquad\qquad\quad f\ x\ =\ x * x
\end{array}
$$

Speedup Analysis

Figure 6.20 presents the speedup achieved by the distilled sum squares of list program against the original program for various input sizes. Here, an input of size `N` denotes the *sumSquares* computation applied on a list of length `N`. We observe that distillation produces a speedup of around `2.5x` for all input sizes presented here.

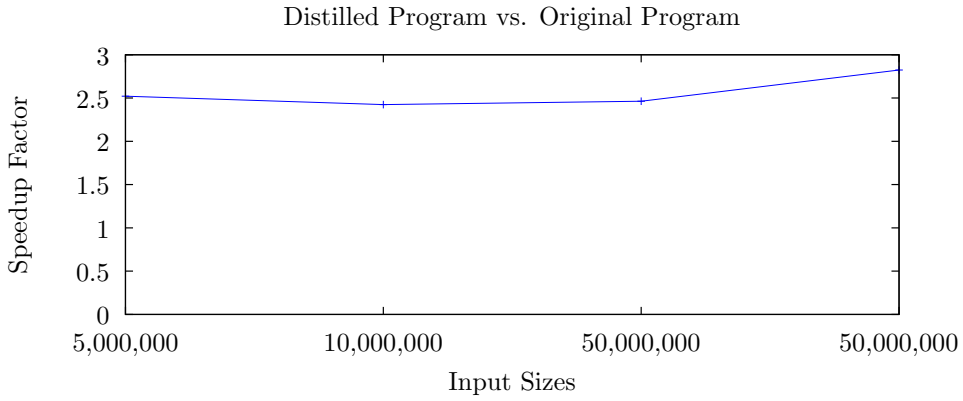


Figure 6.20: Speedup of Distilled Sum Squares of List

Figure 6.21 presents the speedups achieved by the EPP version of the sum squares benchmark program presented in Section 6.5.6 in comparison with the OP, DP and HPP versions. We observe that the EPP version achieves a positive speedup of around `2x` in comparison with the OP version for all input sizes, which is primarily due to eliminating the intermediate data structure present in the OP version. This can be inferred because

the EPP version is slower than the DP version, both of which are free of intermediate data structures. The EPP version is slower than the DP version due to the additional cost of encoding the inputs in the EPP version. Further, we observe that the EPP version is faster than the HPP version by a factor of around `1.2x` when executed on two or more cores. This positive speedup is also associated to the intermediate data structure present in the HPP version that is absent in the EPP version.

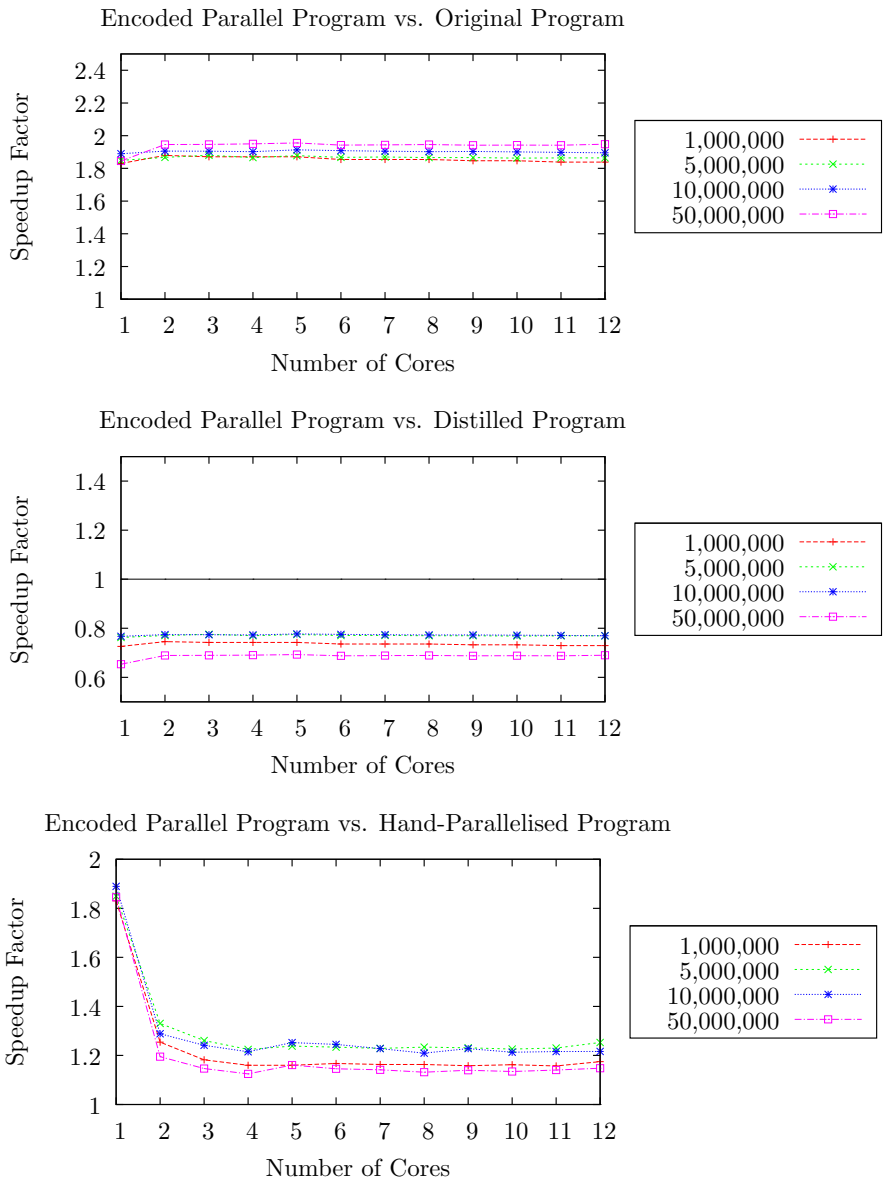


Figure 6.21: Speedup of Sum Squares of List

The EPP and HPP versions do not scale well with increased number of cores due to the significantly high cost of encoding the inputs (for the EPP version) and paralleli-

saiton (for the EPP and HPP versions) when compared to the cost associated with the computation done by each parallel thread as seen in the cost-centre analysis.

Parallel Execution Analysis

Figure 6.22 presents the status of the cores when executing the EPP version of the sum squares program on an input of size `10,000,000` on 6 cores. We observe from this parallel execution profile that the parallel computations are well balanced across all cores without any idle (in blue) or waiting/blocked (in red) time. This indicates a balanced workload across the cores when executing of the encoded parallel program.

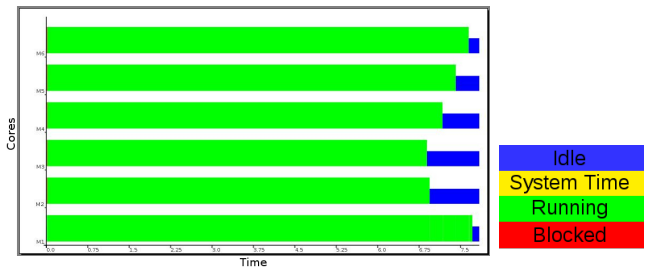


Figure 6.22: Sum Squares of List – EPP Execution Profile from EdenTV – 10,000,000 on 6 cores

However, despite the good parallel execution as illustrated in Figure 6.22, the EPP version does not achieve a positive speedup compared to the DP version. As mentioned earlier, we believe this is due to the cost of encoding which is significantly higher than the cost of parallel execution. This is discussed further in the cost-centre analysis.

Cost-Centre Analysis

For the EPP version of the sum squares program, the cost centre for the $encode_{sumSquares}$, *sumSquares* and *main* functions in the encoded program are as shown in Figure 6.23. Here, we observe that the cost of encoding the input is approximately equal to the cost of the sum squares computation which is parallelised. Consequently, the cost of encoding the input is a significant overhead in this benchmark program, which results in the EPP version performing slower than the DP version.

Input Size	*main* (%)	$encode_{sumSquares}$ (%)	*sumSquares* (%)
`1,000,000`	10.5	48.1	41.4
`5,000,000`	10.6	44.4	45.0
`10,000,000`	10.8	42.3	46.9
`50,000,000`	11.8	44.3	44.0

Figure 6.23: Sum Squares of List – Cost Centre of Encoded Program

Related Work

In [26], the authors evaluate a parallel sum squares of list example where the input list transformed to a join-list which is split into sub-lists for parallel evaluation of each half. The speedup reported by this method for varying input sizes is around `1.4x` on average when using between 2 and 12 cores.

6.5.7 Evaluation of Fibonacci Series Sum

The OP version of the program to compute the Fibonacci series sum is presented in Example 6.5.7.1. Here, function *fibSum* is defined to compute the sum of the first `N` numbers in the Fibonacci series, where `N` is the input argument to *fibSum*. This definition uses the intermediate data structure $(Succ\ x)$ in the second recursive call to the function *fibSum*.

Example 6.5.7.1 (Fibonacci Series Sum – Original Program (OP)):
data $IntValue\ =\ Zero \mid Succ\ IntValue$

$fibSum\ ::\ IntValue \rightarrow Int$
$fibSum\ x$
where
$fibSum\ Zero\ =\ 1$
$fibSum\ (Succ\ Zero)\ =\ 1$
$fibSum\ (Succ\ (Succ\ x))\ =\ (fibSum\ x) + (fibSum\ (Succ\ x))$

Based on the procedure in Section 6.2, we first apply the distillation transformation to reduce the number of intermediate data structures. Example 6.5.7.2 presents the distilled program (DP) obtained for the OP version in Example 6.5.7.1. Here, the Fibonacci series sum program is defined using two recursive functions $fibSum_1$ and $fibSum_2$ and does not use intermediate data structures.

Example 6.5.7.2 (Fibonacci Series Sum – Distilled Program (DP)):

$$
\begin{array}{l}
fibSum_1\ x \\
\textbf{where} \\
fibSum_1\ Zero \quad = 1 \\
fibSum_1\ (Succ\ Zero) \quad = 1 \\
fibSum_1\ (Succ\ (Succ\ x)) \ = \ (fibSum_2\ x) + (fibSum_1\ x) \\
\\
fibSum_2\ Zero \quad = 1 \\
fibSum_2\ (Succ\ x) \ = \ (fibSum_1\ x) + (fibSum_2\ x)
\end{array}
$$

Here, we observe that recursive functions $fibSum_1$ and $fibSum_2$ have a maximum of one recursive call in their function definition bodies. Consequently, based on the parallelisation steps presented in Section 5.2.2, we use the encoding transformation version 2 to encode their pattern-matched inputs into *cons*-lists. Example 6.5.7.3 presents the encoded version of the DP version.

Example 6.5.7.3 (Fibonacci Series Sum – Encoded Program):

$$
\begin{array}{l}
\textbf{data}\ T'_{fibSum_1}\ a \ = \ c_1 \mid c_2 \mid c_3\ a \\
\textbf{data}\ T'_{fibSum_2}\ a \ = \ c_4 \mid c_5\ a \\
\\
encode_{fibSum_1}\ Zero \quad = [c_1] \\
encode_{fibSum_1}\ (Succ\ Zero) \quad = [c_2] \\
encode_{fibSum_1}\ (Succ\ (Succ\ x)) \ = \ [c_3\ x] +\!\!+ (encode_{fibSum_1}\ x) \\
\\
encode_{fibSum_2}\ Zero \quad = [c_4] \\
encode_{fibSum_2}\ (Succ\ x) \quad = [c_5\ x] +\!\!+ (encode_{fibSum_2}\ x) \\
\\
fibSum'_1\ (encode_{fibSum_1}\ x) \\
\textbf{where} \\
fibSum'_1\ (c_1 : w) \quad = 1 \\
fibSum'_1\ (c_2 : w) \quad = 1 \\
fibSum'_1\ \big((c_3\ x) : w\big) \ = \ (fibSum'_2\ (encode_{fibSum_2}\ x)) + (fibSum'_1\ w) \\
fibSum'_2\ (c_4 : w) \quad = 1 \\
fibSum'_2\ \big((c_5\ x) : w\big) \ = \ (fibSum'_1\ (encode_{fibSum_1}\ x)) + (fibSum'_2\ w)
\end{array}
$$

Using the encoded program in Example 6.5.7.3, we apply the skeleton identification technique to identify potential instances of the *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons. Consequently, the functions $fibSum'_1$ and $fibSum'_2$ are identified as instances of the *mapRedr* skeleton. Example 6.5.7.4 presents the encoded parallel program (EPP) defined using a suitable call to the *offlineParMapRedr1* skeleton in Eden.

Example 6.5.7.4 (Fibonacci Series Sum – Encoded Parallel Program (EPP)):

$$
\begin{array}{l}
\textbf{data}\ T'_{fibSum_1}\ a \ = \ c_1 \mid c_2 \mid c_3\ a \\
\textbf{data}\ T'_{fibSum_2}\ a \ = \ c_4 \mid c_5\ a
\end{array}
$$

$$
\begin{array}{lll}
encode_{fibSum_1}\ Zero & = & [c_1] \\
encode_{fibSum_1}\ (Succ\ Zero) & = & [c_2] \\
encode_{fibSum_1}\ (Succ\ (Succ\ x)) & = & [c_3\ x] +\!\!+ (encode_{fibSum_1}\ x)
\end{array}
$$

$$
\begin{array}{lll}
encode_{fibSum_2}\ Zero & = & [c_4] \\
encode_{fibSum_2}\ (Succ\ x) & = & [c_5\ x] +\!\!+ (encode_{fibSum_2}\ x)
\end{array}
$$

$fibSum_1\ (encode_{fibSum_1}\ x)$
where

$$
\begin{array}{lll}
fibSum''_1\ w & = & offlineParMapRedr1\ g\ f\ w \\
 & & \textbf{where} \\
 & & g = (+) \\
 & & f\ c_1 = 1 \\
 & & f\ c_2 = 1 \\
 & & f\ (c_3\ x) = fibSum''_2\ (encode_{fibSum_2}\ x) \\
fibSum''_2\ w & = & offlineParMapRedr1\ g\ f\ w \\
 & & \textbf{where} \\
 & & g = (+) \\
 & & f\ c_4 = 1 \\
 & & f\ (c_5\ x) = fibSum''_1\ (encode_{fibSum_1}\ x)
\end{array}
$$

Since the OP version of the Fibonacci series sum does not contain any instance of skeletons whose implementations are available in Eden, the HPP version for the Fibonacci series sum program is defined using GpH as shown in Example 6.5.7.5. Here, the evaluation of the two recursive calls $(fibSum\ x)$ and $(fibSum\ (Succ\ x))$ are parallelised using the *rpar* and *rseq* constructs. Here, the threshold value t for the parallel definition can be initialised to $\lceil log_2 P \rceil + 1$, where P is the number of processors used for parallel execution, as the evaluation tree follows the structure of a binary tree.

Example 6.5.7.5 (Fibonacci Series Sum – HPP):

$$
\begin{array}{lll}
fibSum\ x\ t & & \\
\textbf{where} & & \\
fibSum\ Zero\ t & = 1 & \\
fibSum\ (Succ\ Zero)\ t & = 1 & \\
fibSum\ (Succ\ (Succ\ x))\ t & = h\ (t \leq 1) & \\
 & \textbf{where} & \\
 & h\ True\ = (fibSum\ x\ t) + (fibSum\ (Succ\ x)\ t) & \\
 & h\ False = runEval\ \$\ do & \\
 & \qquad x' \leftarrow rpar\ (fibSum\ x\ (t-1)) & \\
 & \qquad y' \leftarrow rseq\ (fibSum\ (Succ\ x)\ (t-1)) & \\
 & \qquad return\ (x' + y') &
\end{array}
$$

Speedup Analysis

Figure 6.24 presents the speedup achieved by the distilled Fibonacci series sum program against the original program for various input sizes. An input size indicated by `N` denotes

the summation of the first `N` elements in the Fibonacci series.

We observe that the distilled program performs on par with the original program. This is primarily attributed to the cost of computing the intermediate data structure (which is of the order of the inputs sizes of 40 to 46 elements) which is negligible compared to the significantly higher cost of computing the Fibonacci series sum (which is of the order of (40*40) to (46*46) elements).

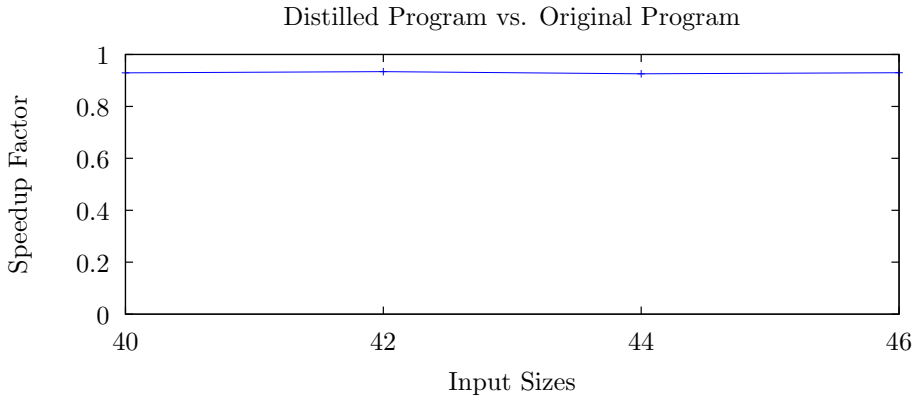


Figure 6.24: Speedup of Distilled Fibonacci Series Sum

Figure 6.25 presents the speedups of the EPP version compared to the OP, DP and HPP versions. When compared to the OP version, we observe that the EPP version achieves a positive speedup of `1.3x-2.0x` for all input sizes when more than two cores are used. The speedup achieved for different input sizes scales well for upto 5 cores (except for 2 cores) and does not scale when more than 5 cores are used. When compared to the DP version the EPP version performs better by the same factor of `1.2x-1.8x` as against the OP version when more than two cores are used. This marginal difference is because the DP version performs slightly better than the OP version as a result of the intermediate data structure that was eliminated. This was discussed in Section 6.5.7.

When compared to the HPP version, we observe that the EPP version does not deliver as much speedup as the hand-parallelised program. In particular, the HPP version is faster than the EPP version by a factor of `1.3x-4.5x` for all input sizes on different number cores. This is because the workload for the threads in the HPP version are well-balanced unlike in the EPP version. This can be observed from the parallel execution analysis presented in Section 6.5.7.

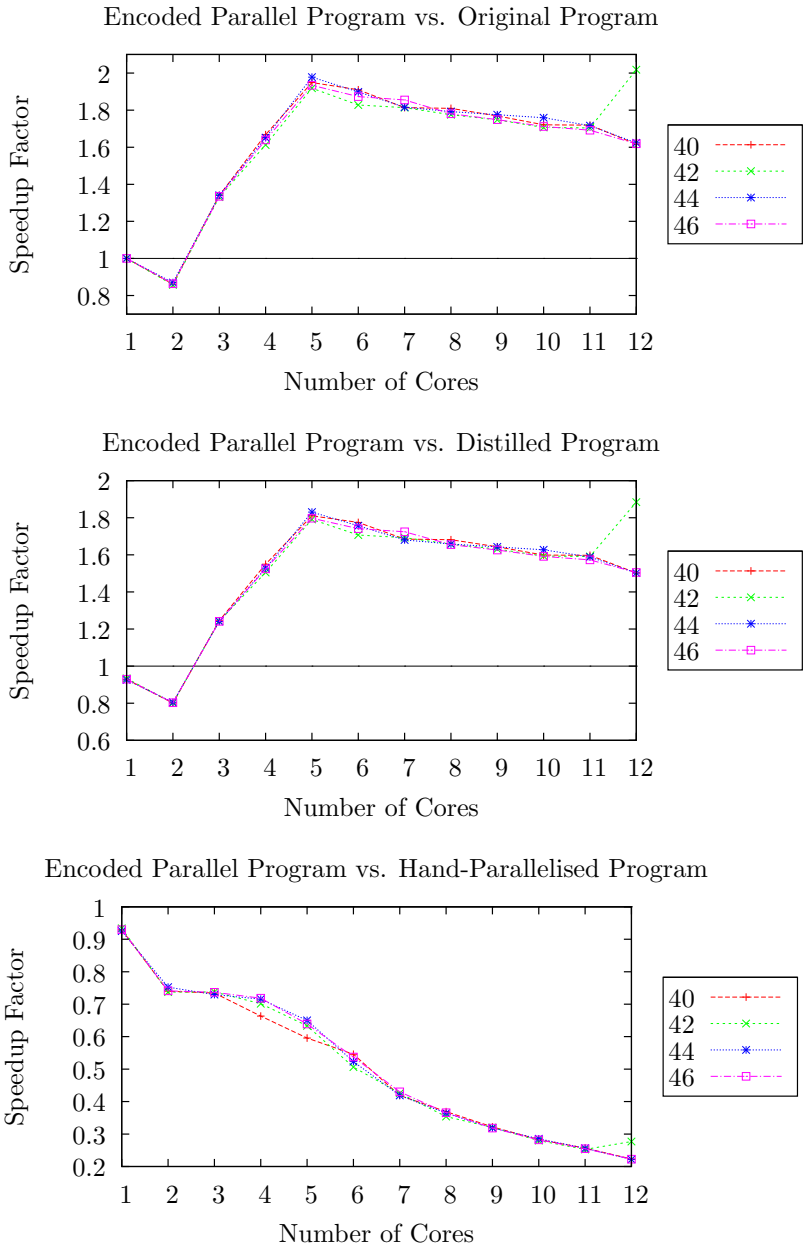


Figure 6.25: Speedup of Fibonacci Series Sum

Parallel Execution Analysis

A sample of the parallel execution profiles of the EPP and HPP versions on 8 cores are shown in Figures 6.26 and 6.27, respectively. Here, we observe that the cores have significant idle times when executing the EPP version due to sub-optimal workload balance unlike in the HPP versions where the workloads of the parallel threads are well-balanced.

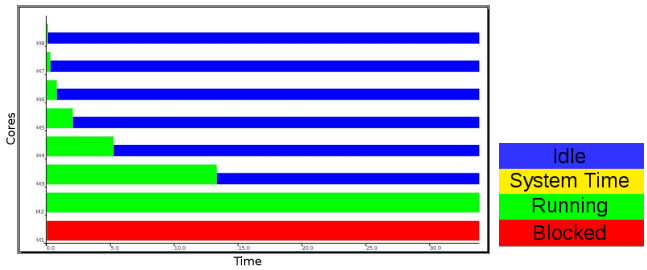


Figure 6.26: Fibonacci Series Sum – EPP Execution Profile from EdenTV – 44 on 8 cores

This can be explained from the definitions of the EPP and HPP versions. The workload of the two threads created at each point of parallelism in the HPP version is decided by the costs of recursive calls $fibSum\ x$ and $fibSum\ (Succ\ x)$ which are nearly identical computations. However, the workload of the threads in the EPP version is decided by the cost of function call $fibSum_2\ x$ which is dependent on the magnitude of the element x in the encoded list and executed sequentially as explained earlier to avoid nested parallel skeletons. Since the magnitude of x in the encoded list progressively decreases from the first to the last element, which can be observed from the $encode_{fibSum_1}$ function, the workloads of the threads created for the EPP version also vary significantly. This results in the sub-par parallel performance of the EPP version for the Fibonacci series sum despite a positive speedup.

Figure 6.27: Fibonacci Series Sum – HPP Execution Profile from Threadscope – 44 on 8 cores

Cost-Centre Analysis

For the encoded version of the Fibonacci series sum program, we measure the individual execution times to build the inputs, encode the inputs and for executing the $fibSum_1$

and $fibSum_2$ functions. Figure 6.28 illustrates that for all input sizes the overhead to encode the input is negligible compared to the total execution time for the encoded version. Also, since we parallelise only the top-level skeleton as explained earlier, the function $fibSum_1$ is executed using the parallel skeleton while $fibSum_2$ is executed using its sequential definition. Therefore, we observe from the cost-centre that we parallelise only around `47%-48%` of the Fibonacci series sum program.

Input Size	*buildInputs* (%)	$encode_{fibSum_1}$ (%)	$fibSum_1$ (%)	$fibSum_2$ (%)
`40`	0.0	0.0	48.1	51.9
`42`	0.0	0.0	47.8	52.2
`44`	0.0	0.0	47.4	52.3
`46`	0.0	0.0	47.8	52.2

Figure 6.28: Fibonacci Series Sum – Cost Centre of Encoded Program

Related Work

When the original Fibonacci series sum program is parallelised using Glasgow Parallel Haskell, the authors in [88] report a speedup of `2x-4x` when executed on 2 to 8 cores. This is consistent with the speedup achieved by our HPP version. However, the speedup achieved by the EPP version is significantly lower due to the poor load balance across the parallel processes as discussed earlier.

6.5.8 Evaluation of Sum Append of Lists

The OP version of the sum append program is presented in Example 6.5.8.1. Here, function *sumApp* computes the sum of a list (using the Haskell built-in operator *sum*) that is built by appending two lists using the Haskell library function $+\!\!+$. Here, the inefficient intermediate data structure produced by the append operation ($xs +\!\!+ ys$) is decomposed by *sum* for computing the result.

Example 6.5.8.1 (Sum Append of Lists – Original Program (OP)):

$sumApp\ xs\ ys\ =\ sum\ (xs +\!\!+ ys)$

Based on the procedure in Section 6.2, we first apply the distillation transformation to reduce the number of intermediate data structures. Example 6.5.8.2 presents the distilled program (DP) obtained for the OP version in Example 6.5.8.1, where distillation uses the

library definitions of *sum* and $(+\!\!+)$. In Example 6.5.8.2, the function *sumApp* computes the sum of the appended list without the use of any intermediate data structure.

Example 6.5.8.2 (Sum Append of Lists – Distilled Program (DP)):

$$
\begin{array}{l}
sumApp_1\ xs\ ys \\
\textbf{where} \\
sumApp_1\ []\ ys \ = \ sumApp_2\ ys \\
sumApp_1\ (x : xs)\ ys \ = \ x + (sumApp_1\ xs\ ys) \\
\\
sumApp_2\ [] \ = \ 0 \\
sumApp_2\ (y : ys) \ = \ y + (sumApp_2\ ys)
\end{array}
$$

Here, we observe that the recursive functions $sumApp_1$ and $sumApp_2$ have a maximum of one recursive call in their definition body. Consequently, based on the parallelisation steps presented in Section 5.2.2, we use the encoding transformation version 2 to encode the pattern-matched inputs into a *cons*-list. Example 6.5.8.3 presents the encoded version of the DP version.

Example 6.5.8.3 (Sum Append of Lists – Encoded Program):

$$
\begin{array}{l}
\textbf{data}\ T'_{sumApp_1}\ a \ = \ c_1 \mid c_2\ a\ (T'_{sumApp_1}\ a) \\
\\
\textbf{data}\ T'_{sumApp_2}\ a \ = \ c_4 \mid c_3\ a\ (T'_{sumApp_2}\ a) \\
encode_{sumApp_1}\ [] \ = \ [c_1] \\
encode_{sumApp_1}\ (x : xs) \ = \ [c_2\ x] +\!\!+ (encode_{sumApp_1}\ xs) \\
\\
encode_{sumApp_2}\ [] \ = \ [c_3] \\
encode_{sumApp_2}\ (y : ys) \ = \ [c_4\ y] +\!\!+ (encode_{sumApp_2}\ ys) \\
\\
sumApp'_1\ (encode_{sumApp_1}\ xs\ ys) \\
\textbf{where} \\
sumApp'_1\ []\ ys \ = \ sumApp'_2\ (encode_{sumApp_2}\ ys) \\
sumApp'_1\ (x : xs)\ ys \ = \ x + (sumApp'_1\ xs\ ys) \\
\\
sumApp'_2\ [] \ = \ 0 \\
sumApp'_2\ (y : ys) \ = \ y + (sumApp_2\ ys)
\end{array}
$$

Using the encoded program in Example 6.5.8.3, we apply the skeleton identification technique to identify potential instances of the *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons to obtain the EPP version. Consequently, the function $sumApp'$ is identified as an instance of the *mapRedr* skeleton. Example 6.5.8.4 presents the EPP version defined using suitable calls to the corresponding Eden skeleton.

Example 6.5.8.4 (Sum Append of Lists – Encoded Parallel Program (EPP)):

$$
\begin{array}{l}
\textbf{data}\ T'_{sumApp}\ a \ = \ c_1 \mid c_2\ a\ (T'_{sumApp}\ a) \mid c_3\ a\ (T'_{sumApp}\ a) \\
encode_{sumApp}\ []\ [] \ = \ [c_1] \\
encode_{sumApp}\ []\ (y : ys) \ = \ [c_2\ y] +\!\!+ (encode_{sumApp}\ []\ ys) \\
encode_{sumApp}\ (x : xs)\ ys \ = \ [c_3\ z] +\!\!+ (encode_{sumApp}\ xs\ ys)
\end{array}
$$

$sumApp''\ (encode_{sumApp}\ xs\ ys)$
where
$sumApp''\ w\ =\ offlineParMapRedr1\ g\ f\ w$
where
$g\ =\ (+)$
$f\ c_1\ =\ 0$
$f\ (c_2\ y)\ =\ y$
$f\ (c_3\ x)\ =\ x$

A hand-parallelised version (HPP) of the OP version is presented in Example 6.5.8.5, which is obtained by manually identifying instances of list-based *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons in the OP version. As a result, we identify that the *sum* operation can be applied in parallel, using the *offlineParMapRedr* skeleton presented in Section 4.2, on the list produced by the append operation. The append operation is not trivially parallelisable using skeletons as the Eden library does not offer parallel implementation for skeletons that operate over multiple inputs.

Example 6.5.8.5 (Sum Append of Lists – Hand-Parallelised Program (HPP)):
$sumApp\ xs\ ys\ =\ offlineParMapRedr\ (+)\ 0\ id\ (xs\ ++\ ys)$

Speedup Analysis

Figure 6.29 presents the speedup achieved by the distilled sum append of lists program against the original program for various input sizes. Here, an input of size `N` denotes the *sumApp* computation applied on two lists each of length `N`.

We observe that distillation is faster than the original program by a factor of around `2x`.

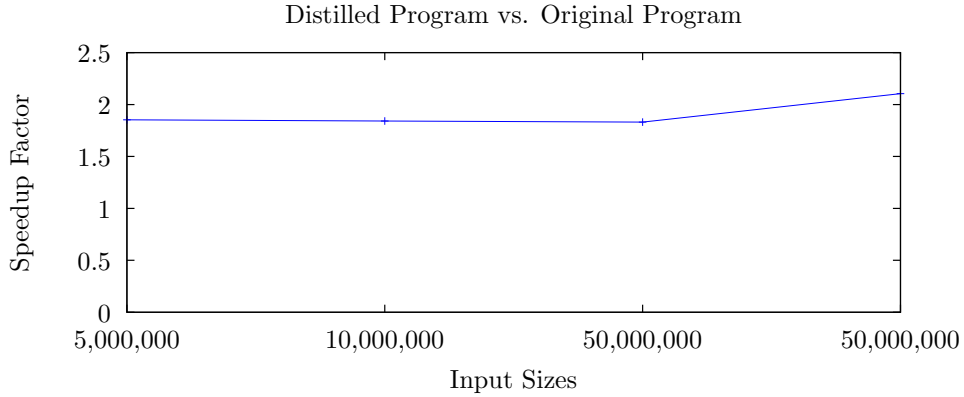


Figure 6.29: Speedup of Distilled Sum Append of Lists

Figure 6.30 presents the speedups achieved by the EPP version of the sum append

benchmark program presented in Section 6.5.8 in comparison with the OP, DP and HPP versions.

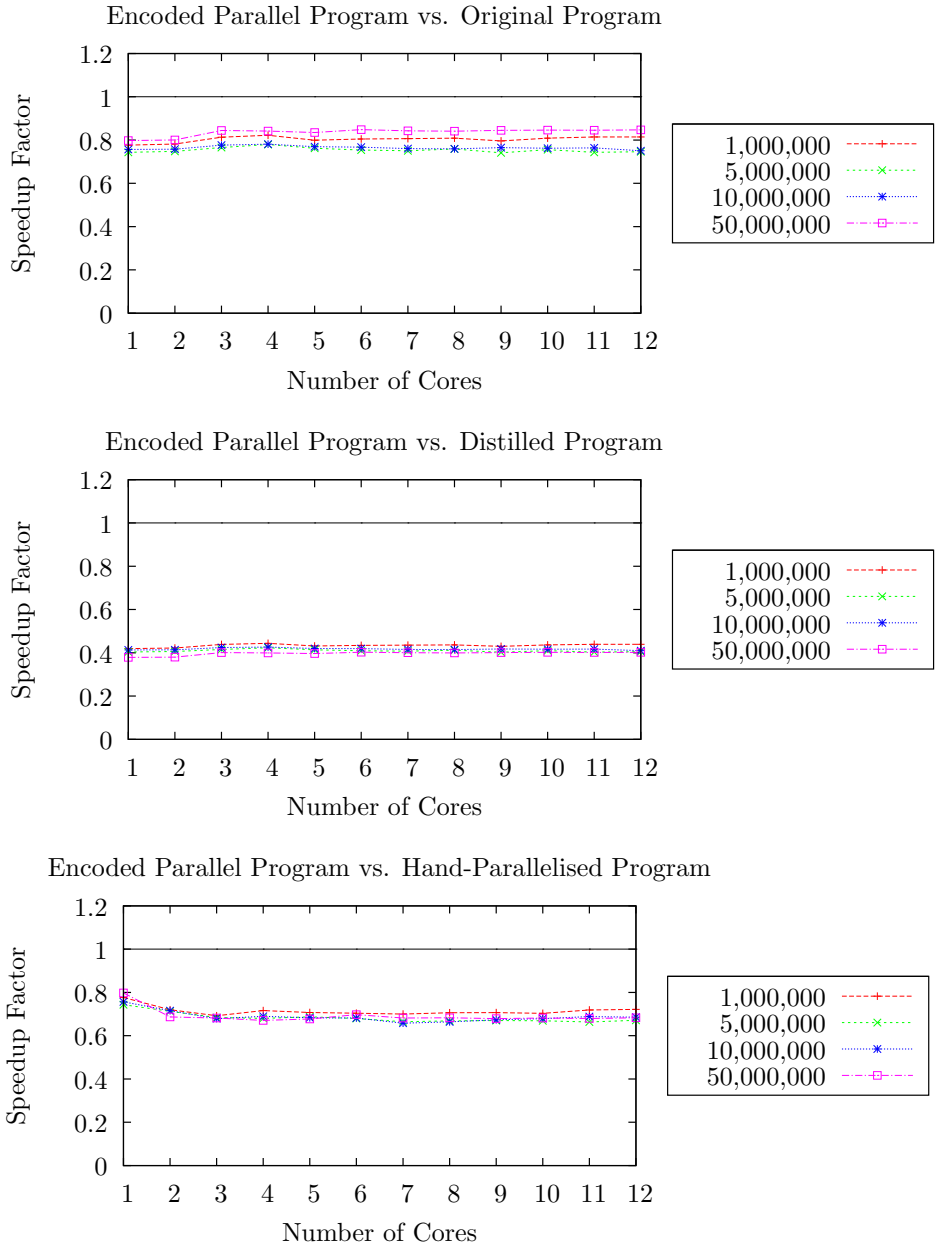


Figure 6.30: Speedup of Sum Append of Lists

We observe that the EPP version, even though parallelisable using the *mapReduce* skeleton without the use of intermediate data structures, does not achieve a positive speedup in comparison with the OP, DP or HPP versions. Upon closer inspection, we observe that this can be attributed to the additional cost of input encoding performed only by the EPP version and not the OP, DP and HPP versions. This reasoning is

investigated further and validated in the following sections by analysing the parallel execution profile and cost-centre.

Parallel Execution Analysis

Figure 6.31 presents the status of the cores when executing the EPP version of the sum append program on an input of size `10,000,000` on 6 cores. We observe from this parallel execution profile that the parallel computations are well balanced across all cores without any idle (in blue) or waiting/blocked (in red) time. This indicates a balanced workload across the cores when executing of the encoded parallel program.

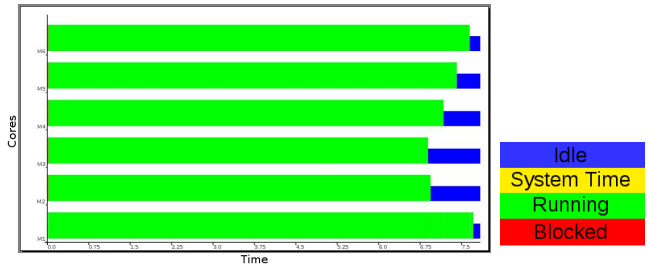


Figure 6.31: Sum Append of Lists – EPP Execution Profile from EdenTV – 10,000,000 on 6 cores

However, despite the good parallel execution as illustrated in Figure 6.31, the EPP version does not achieve a positive speedup compared to the OP, DP and HPP versions. We believe this is due to the cost of encoding which is significantly higher than the cost of parallel execution. This is discussed further in the cost-centre analysis.

Cost-Centre Analysis

For the EPP version of the sum prefix program, the cost centre for the $encode_{sumApp}$, *sumApp* and *main* functions in the encoded program are as shown in Figure 6.32. Here, we observe that the computation to encode the input amounts on an average to around 50% of the execution time, while the *sumApp* and *main* functions amount to around 32% and 17%, respectively. This makes the cost of encoding significantly higher than that of executing the *sumApp* function which is the potentially parallelisation computation. Thus, the EPP version performs poorer in this benchmark program due to high encoding overhead.

Input Size	*main* (%)	$encode_{sumApp}$ (%)	*sumApp* (%)
1,000,000	14.7	54.6	30.7
5,000,000	17.2	48.6	34.2
10,000,000	19.2	48.8	32.0
50,000,000	16.0	51.4	32.6

Figure 6.32: Sum Append of Lists – Cost Centre of Encoded Program

6.5.9 Performance of Nested Parallel Skeletons

As stated in Section 6.3, for all parallel versions of a benchmark program (HPP and EPP), nesting of parallel skeletons is avoided. This is achieved by using parallel implementations only for those skeletons that are present in the top-level while any nested skeleton instances are executed using their sequential versions. The objective of this approach is to avoid uncontrolled creation of too many threads that results in inefficient parallel execution where the cost of thread creation and management is greater than the cost of parallel execution. This is demonstrated in this section using the matrix multiplication benchmark program evaluated in Section 6.5.1. The HPP and EPP versions of the matrix multiplication program defined using nested parallel skeletons are presented in Examples 6.5.9.1 and 6.5.9.2, respectively.

In the HPP version presented here, in addition to the *mMul* function that is defined using the *farmB* skeleton, the *map* and *foldr* computations are defined using the *farmB* and *offlineParMapRedr* skeletons, respectively, from Eden.

Example 6.5.9.1 (Matrix Multiplication – Hand-Parallelised Program (HPP) with Nested Skeletons):

$mMul\ xss\ yss$
where
$mMul\ xss\ yss\ =\ farmB\ noPe\ f\ xss$
where
$f\ xs\ =\ farmB\ noPe\ (dotp\ xs)\ (transpose\ yss)$
$dotp\ xs\ ys\ =\ offlineParMapRedr\ (+)\ 0\ id\ (zipWith\ (*)\ xs\ ys)$
$transpose\ yss\ =\ transpose'\ yss\ []$
$transpose'\ []\ yss\ =\ yss$
$transpose'\ (xs : xss)\ yss\ =\ transpose'\ xss\ (rotate\ xs\ yss)$
$rotate\ []\ yss\ =\ yss$
$rotate\ (x : xs)\ []\ =\ [x] : (rotate\ xs\ yss)$
$rotate\ (x : xs)\ (ys : yss)\ =\ (ys +\!\!+ [x]) : (rotate\ xs\ yss)$

In the EPP version presented below, the $mMul_1$ function that is an instance of the *map* skeleton was defined using the *farmB* skeleton. Additionally, we also find function $mMul_3$ to be an instance of the *mapRedr* skeleton and hence define it using the *offlineParMapRedr1* skeleton.

Example 6.5.9.2 (Matrix Multiplication – Encoded Parallel Program (EPP) with Nested Skeletons):

$$
\begin{array}{l}
mMul\ xss\ yss \\
\textbf{where} \\
mMul\ xss\ yss\ =\ mMul_1\ (encode_{mMul_1}\ xss\ yss)\ yss \\
\\
mMul_1\ w\ yss\ =\ farmB\ noPe\ f\ w \\
\qquad\qquad \textbf{where} \\
\qquad\qquad f\ c_1 \qquad\quad = [] \\
\qquad\qquad f\ c_2 \qquad\quad = [] \\
\qquad\qquad f\ (c_3\ xs\ zs)\ =\ \textbf{let}\ v = \lambda xs.g\ xs \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad g\ [] \qquad = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad g\ (x:xs)\ =\ x \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ mMul_2\ zs\ xs\ yss\ v \\
\\
mMul_2\ []\ xs\ yss\ v \qquad = [] \\
mMul_2\ (z:zs)\ xs\ yss\ v\ =\ \textbf{let}\ v' = \lambda xs.g\ xs \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad g\ [] \qquad = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad g\ (x:xs)\ =\ v\ xs \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (mMul_3\ (encode_{mMul_3}\ xs\ yss)\ v) : (mMul_2\ zs\ xs\ yss\ v') \\
\\
mMul_3\ w\ v\ =\ offlineParMapRedr1\ g\ f\ w \\
\qquad\qquad \textbf{where} \\
\qquad\qquad g \qquad\quad = (+) \\
\qquad\qquad f\ c_6 \qquad\quad = 0 \\
\qquad\qquad f\ c_7 \qquad\quad = 0 \\
\qquad\qquad f\ (c_8\ x\ ys)\ =\ x * (v\ ys)
\end{array}
$$

Following the evaluation of the HPP and EPP versions in Examples 6.5.9.1 and 6.5.9.2, the performance of these parallel versions that use nested skeletons are presented in Figure 6.33 for an input of size `100x100`. We observe that the HPP and EPP versions are significantly slower than the OP version in Example 6.5.1.1 by factors of `6x-20x` and `60x-100x`, respectively. This is due to the massive overhead incurred from the creation and management of the additional threads that result from the use of the nested skeletons. This is evident from the statistics presented in Figure 6.34.

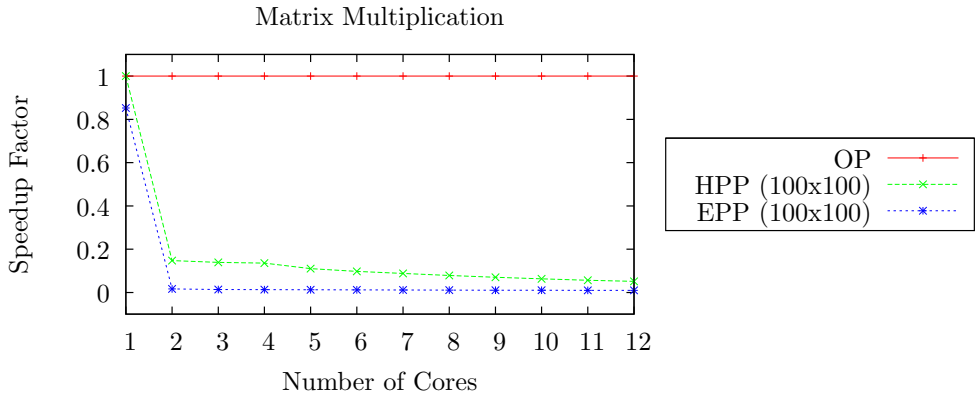


Figure 6.33: Speedup of HPP and EPP versions Using Nested Skeletons vs. Original Program (OP)

	EPP / HPP with Top-Level Skeletons	EPP with Nested Skeletons	HPP with Nested Skeletons
Cores Used	6	6	6
Processes Created	7	60,007	60,607
Messages Exchanged	110	210,110	230,002

Figure 6.34: Matrix Multiplication – Thread Creation and Management Statistics for Top-Level vs. Nested Skeletons

Here, we notice that for both the HPP and EPP versions that use nested skeletons, a far greater number of parallel processes and inter-thread messages are created in comparison with the HPP and EPP versions that use only top-level skeletons. In this example of input size `100x100` executed on 6 cores, the top-level skeleton creates 6 additional parallel processes to multiply 100 rows and columns. Each of these processes in turn computes the dot-product of inputs of size `100`. Given this scenario, the nested skeletons create an additional `60,000` (`60,600`) processes in the EPP (HPP) version and, consequently, additional messages.

The poor parallelisation that results from this uncontrolled creation of threads is further illustrated in Figures 6.35 and 6.36. Here, we observe that the execution state of the cores constantly changes among busy (in green), blocked and waiting for results (in red), and system time (in yellow).

Upon closer inspection of the program definitions, we understand the reason behind the poorer performance of the EPP version using nested skeletons when compared to

its HPP counterpart. In the EPP version, the parallel threads created for $mMul_1$ have a significantly large sequential computation in $mMul_2$ even though the dot-product computation performed thereafter by $mMul_3$ is parallelised. That is, the parallelised dot-product computations are spawned sequentially by $mMul_2$ in the EPP version. In contrast, the HPP version spawns the dot-product computations in parallel using the *farmB* skeleton. As a result, the HPP version is significantly faster than the EPP version in this scenario.

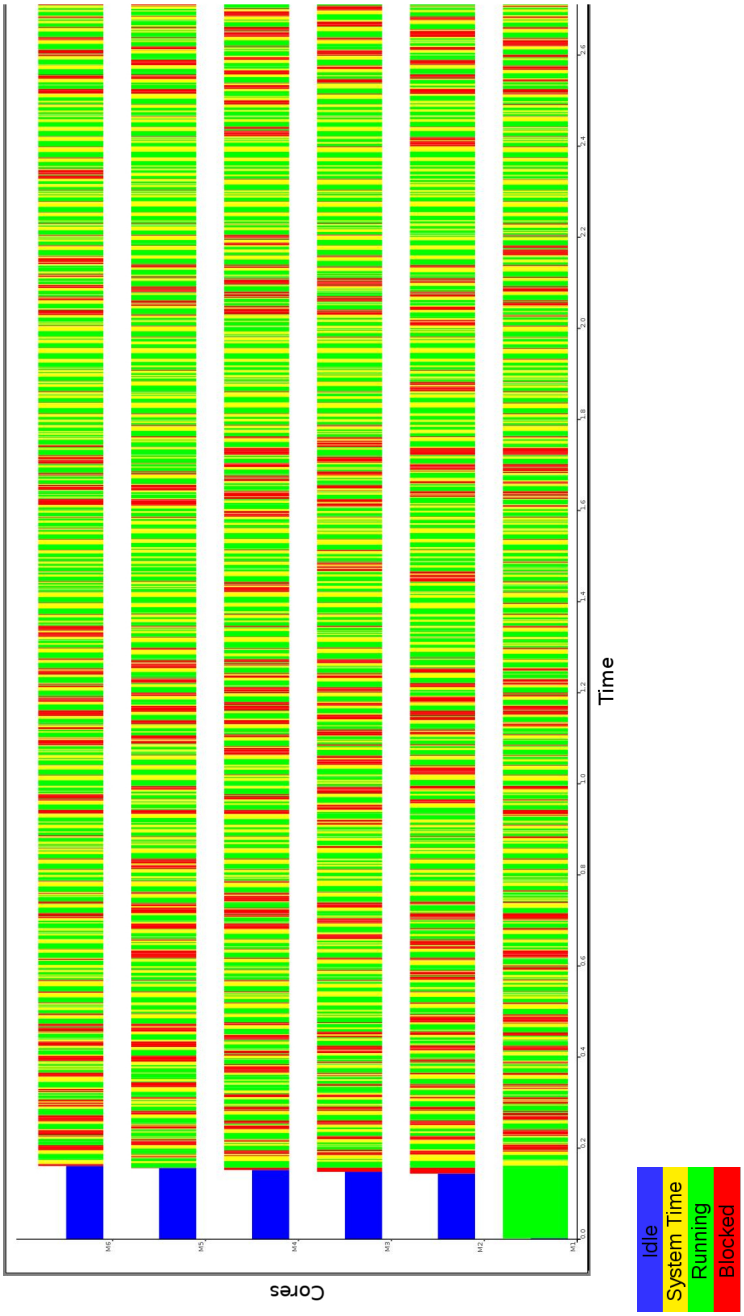


Figure 6.35: Matrix Multiplication – EPP using Nested Skeletons – Execution Profile from EdenTV – 100x100 on 6 cores

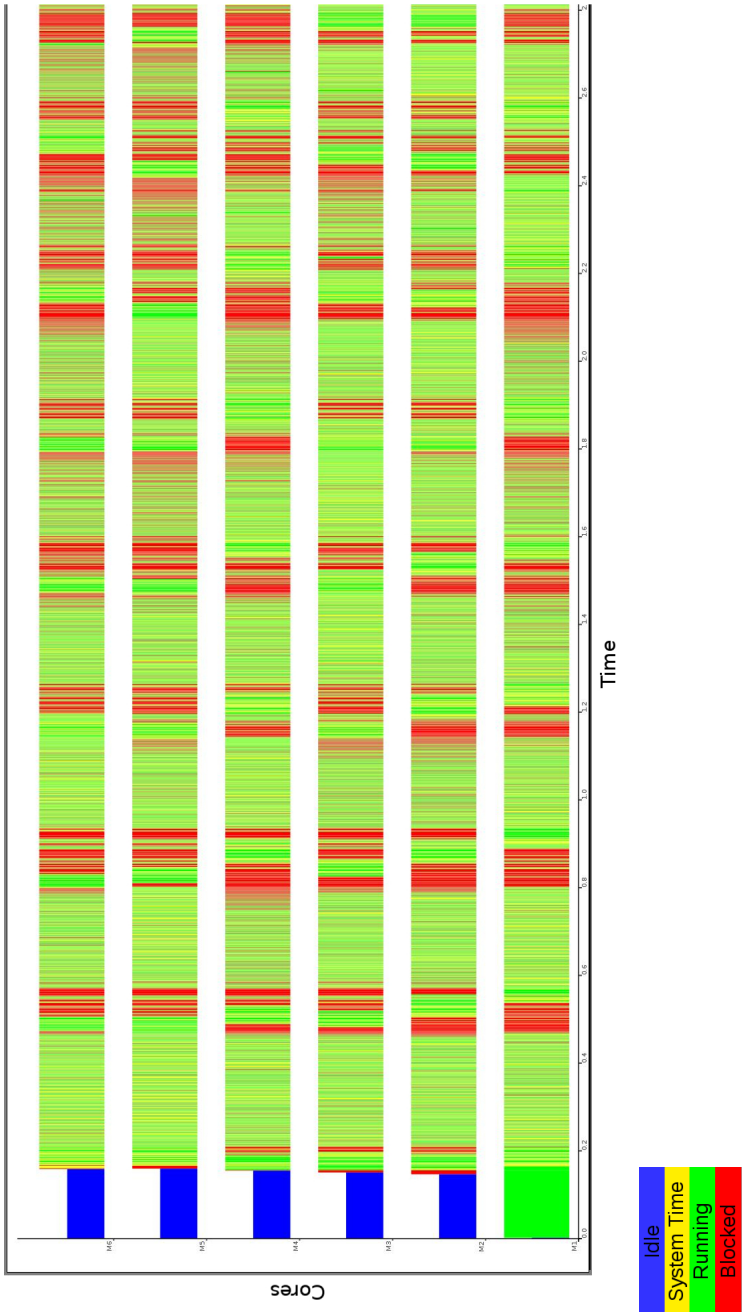


Figure 6.36: Matrix Multiplication – HPP using Nested Skeletons – Execution Profile from EdenTV – 100x100 on 6 cores

6.6 Problem Cases

In this section, we present a set of examples for which the transformation using the method proposed in this thesis does not lead to their parallelisation. We study these examples particularly because their sequential versions (OP) are parallelisable by hand, while the application of our transformation method does not produce a parallel version.

6.6.1 Maximum Segment Sum

The OP version of a program to compute the maximum segment sum of a given list is presented in Example 6.6.1.1, where *mss* computes the maximum segment sum of list *xs* by building a list of segments using the function *segments*.

Example 6.6.1.1 (Maximum Segment Sum – Original Program (OP)):

$$
\begin{array}{l}
mss\ ::\ [Int] \rightarrow Int \\
mss\ xs \\
\textbf{where} \\
mss\ xs = maxList\ (map\ sum\ (segments\ xs)) \\
maxList\ [] = 0 \\
maxList\ (x : xs) = maxList'\ xs\ x \\
maxList'\ []\ m = m \\
maxList'\ (x : xs)\ m = maxList'\ xs\ (max\ x\ m) \\
segments\ xs = concat\ (map\ inits\ (tails\ xs))
\end{array}
$$

The distilled version (DP) of maximum segment sum is presented in Example 6.6.1.2. Here, we observe that the distillation transformation has optimised the OP version, which has a cubic run-time complexity, into a version that has linear run-time complexity.

Example 6.6.1.2 (Maximum Segment Sum – Distilled Program (DP)):

$$
\begin{array}{l}
mss_1\ xs \\
\textbf{where} \\
mss_1\ xs = mss_2\ xs\ 0\ 0 \\
mss_2\ []\ y\ z = y \\
mss_2\ (x : xs)\ y\ z = \textbf{let}\ y' = max\ y\ (x + z) \\
\qquad\qquad\qquad\qquad\quad z' = max\ x\ (x + z) \\
\qquad\qquad\qquad\qquad \textbf{in}\ mss_2\ xs\ y'\ z'
\end{array}
$$

Here, we observe that recursive function mss_2 has a maximum of one recursive call in its definition body. Consequently, based on the parallelisation steps presented in Section 5.2.2, we use the encoding transformation version 2 to encode the pattern-matched inputs into a *cons*-list. Example 6.6.1.3 presents the encoded version of the distilled maximum segment sum program.

Example 6.6.1.3 (Maximum Segment Sum – Encoded Program):

$$
\begin{array}{l}
\textbf{data}\ T_{mss_2}\ a = c_1 \mid c_2\ a \\
encode_{mss_2}\ [] = [c_1] \\
encode_{mss_2}\ (x : xs) = [c_2\ x] +\!\!+ (encode_{mss_2}\ xs)
\end{array}
$$

$mss_1\ xs$
where
$mss'_1\ xs = mss_2\ (encode_{mss_2}\ xs)\ 0\ 0$
$mss'_2\ c_1\ y\ z = y$
$mss'_2\ ((c_2\ x) : w)\ y\ z = \mathbf{let}\ y' = max\ y\ (x + z)$
$z' = max\ x\ (x + z)$
$\mathbf{in}\ mss'_2\ w\ y'\ z'$

Using the encoded program in Example 6.5.5.3, we apply the skeleton identification technique to identify potential instances of the *map*, *mapRedr*, *mapRedl* and *accumulate* skeletons. However, we see that the encoded program cannot be defined using a list-based skeleton. Upon closer inspection, we observe that this is because the distilled version and hence the encoded function mss'_2, though defined over a single list input, is defined similar to a reduction computation using two accumulating arguments y and z. In contrast, the reduce skeletons *mapRedl* and *accumulate* are defined using a single accumulating argument. Consequently, we are not able to define the function mss_2 using the skeletons that we use in our transformation method.

Alternately, if we adopt an approach to tuple the arguments that are not pattern-matched (y and z), then we obtain an encoded program that operates over a single accumulating argument as shown in Example 6.6.1.4. Here, the expression $(\lambda(y, z).y)$ is introduced in mss_1 to extract the output from the result of mss_2.

Example 6.6.1.4 (Maximum Segment Sum – Encoded Program with Tupling):

$\mathbf{data}\ T_{mss_2}\ a = c_1 \mid c_2\ a$
$encode_{mss_2}\ [] = [c_1]$
$encode_{mss_2}\ (x : xs) = [c_2\ x] +\!\!+ (encode_{mss_2}\ xs)$
$mss_1\ xs$
where
$mss'_1\ xs = (\lambda(y, z).y)\ (mss_2\ (encode_{mss_2}\ xs)\ (0, 0))$
$mss'_2\ c_1\ v = v$
$mss'_2\ ((c_2\ x) : w)\ v = \mathbf{let}\ v' = h\ v$
$\mathbf{where}$
$h\ (y, z) = (max\ y\ (x + z),\ max\ x\ (x + z))$
$\mathbf{in}\ mss'_2\ w\ v'$

Here, we may identify that mss'_2 is an instance of the *mapRedl* skeleton, where the map operator f and the reduction operator g are identified as follows:

$$f\ (y, z) = (y, z)$$

$$g\ (y, z)\ x = (max\ y\ (x + z),\ max\ x\ (x + z))$$

However, this cannot be parallelised because the reduction operator g is not associative.

On the other hand, the HPP version of the maximum segment sum program can be obtained by parallelising *map sum (segments xs)* using the *farmB* skeleton, and parallelising $maxList'$ using the *offlineParMapRedr* skeleton as shown in Example 6.6.1.5. We also note that the HPP version can be automatically obtained by applying our skeleton identification technique from Chapter 4 on the OP version without using the distillation and encoding transformations.

Example 6.6.1.5 (Maximum Segment Sum – Hand-Parallelised Program (HPP)):

$mss\ ::\ [Int] \rightarrow Int$
$mss\ xs$
where
$mss\ xs = maxList\ (farmB\ noPe\ sum\ (segments\ xs))$
$maxList\ [] = 0$
$maxList\ (x : xs) = maxList'\ xs\ x$
$maxList'\ xs\ m = offlineParMapRedr\ max\ m\ id\ xs$
$segments\ xs = concat\ (farmB\ noPe\ inits\ (tails\ xs))$

6.6.2 Reverse List

The OP version of a program to reverse a given list is presented in Example 6.6.2.1. Here, the result of $(nrev\ xs)$ is decomposed by the function *app* that appends two lists.

Example 6.6.2.1 (Reverse List – Original Program (OP)):

$nrev\ ::\ [a] \rightarrow [a]$
$nrev\ xs$
where
$nrev\ [] = []$
$nrev\ (x : xs) = app\ (nrev\ xs)\ [x]$
$app\ []\ ys = ys$
$app\ (x : xs)\ ys = x : (app\ xs\ ys)$

The DP version of this program that is free of the intermediate data structure is presented in Example 6.6.2.2.

Example 6.6.2.2 (Reverse List – Distilled Program (OP)):

$arev\ ::\ [a] \rightarrow [a]$
$arev_1\ xs = arev_2\ xs\ []$
where
$arev_2\ []\ ys = ys$
$arev_2\ (x : xs)\ ys = \textbf{let}\ ys' = x : ys\ \textbf{in}\ arev_2\ xs\ ys'$

Here, we observe that recursive function $arev_2$ has a maximum of one recursive call in its definition body. Consequently, based on the parallelisation steps presented in Section

5.2.2, we use the encoding transformation version 2 to encode the pattern-matched inputs into a *cons*-list. Example 6.6.2.3 presents the encoded version of the distilled reverse list program.

Example 6.6.2.3 (Reverse List – Encoded Program):
data $T_{arev_2}\ a\ =\ c_1 \mid c_2\ a$

$encode_{arev_2}\ []\ \ \ \ \ \ \ \ = [c_1]$
$encode_{arev_2}\ (x : xs)\ = [c_2\ x] +\!\!+ (encode_{arev_2}\ xs)$
$arev_1\ xs\ \ \ \ \ \ \ \ \ \ \ \ \ \ = arev_2'\ (encode_{arev_2}\ xs)\ []$
where
$arev_2'\ (c_1 : w)\ ys\ \ \ \ \ \ = ys$
$arev_2'\ ((c_2\ x) : w)\ ys\ = \textbf{let}\ ys' = x : ys\ \textbf{in}\ arev_2'\ w\ ys'$

In the encoded program, we observe that the new encoded data type T_{arev_2} has the same structure as the original type *cons*-list input. Consequently, the encoded program has the same algorithmic structure as the distilled program. However, the encoded program cannot be defined using a list-based map or map-reduce skeleton. This is because the *Cons* constructor (:) that is used to construct the accumulating argument $(x : ys)$ is not associative. Therefore, the recursive function $arev_2'$ cannot be an instance of a parallel map-reduce skeleton.

On the other hand, if the accumulating argument in the DP version is constructed using the built-in associative append operator as $([x] +\!\!+ ys)$, then the DP version of the reverse list program can be parallelised using the *mapRedl* skeleton as shown in Example 6.6.2.4.

Example 6.6.2.4 (Reverse List – Hand-Parallelised Program (HPP) with Built-In $+\!\!+$):
$arev_1\ xs\ \ \ \ = arev_2\ xs\ []$
where
$arev_2\ xs\ ys\ = offlineParMapRedl\ g\ ys\ f\ xs$
where
$g\ \ \ \ = (+\!\!+)$
$f\ x\ = [x]$

If the encoding transformation version 2 is applied on the DP version that uses the built-in $(+\!\!+)$ to construct the accumulating argument, then the DP version can be parallelised by the encoding transformation as shown in Example 6.6.2.5, which parallelises the $arev_2''$ function in the same way $arev_2$ is parallelised in the HPP version with the exception of being defined over the encoded list.

Example 6.6.2.5 (Reverse List – Encoded Parallel Program (EPP) with Built-In $+\!\!+$):

$$
\begin{array}{l}
\textbf{data}\ T_{arev_2}\ a\ =\ c_1 \mid c_2\ a \\
encode_{arev_2}\ []\ \ \ \ \ \ \ \ =\ [c_1] \\
encode_{arev_2}\ (x : xs)\ =\ [c_2\ x] +\!\!+ (encode_{arev_2}\ xs) \\
arev_1\ xs\ \ \ \ \ =\ arev_2''\ (encode_{arev_2}\ xs)\ [] \\
\textbf{where} \\
arev_2''\ xs\ ys\ =\ offlineParMapRedl1\ g\ f\ xs \\
\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad g\ \ \ \ \ \ \ \ =\ (+\!\!+) \\
\qquad\qquad\qquad f\ c_1\ \ \ \ \ =\ ys \\
\qquad\qquad\qquad f\ (c_2\ x)\ =\ [x]
\end{array}
$$

6.6.3 Flatten Binary Tree

The OP version of a program to flatten a binary tree into a list is presented in Example 6.6.3.1. Here, even though the append operator $+\!\!+$ is built-in in Haskell, the results of $(flatten\ lt)$ and $(flatten\ rt)$ are intermediate data structures that are decomposed by the append operator to compute the result.

Example 6.6.3.1 (Flatten Binary Tree – Original Program (OP)):

$$
\begin{array}{l}
\textbf{data}\ BTree\ a\ =\ E \mid B\ a\ (BTree\ a)\ (BTree\ a) \\
flatten\ ::\ (BTree\ a) \rightarrow [a] \\
flatten\ xt \\
\textbf{where} \\
flatten\ E\ \ \ \ \ \ \ \ \ \ =\ [] \\
flatten\ (B\ x\ lt\ rt)\ =\ [x] +\!\!+ (flatten\ lt) +\!\!+ (flatten\ rt)
\end{array}
$$

A hand-parallelised version of the OP version can be obtained by simultaneously evaluating of the recursive calls $(flatten\ lt)$ and $(flatten\ rt)$ using GpH as shown in Example 6.6.3.2. We observe that this hand-parallelised version retains the intermediate data structures that are the result of the recursive calls to $flatten$.

Example 6.6.3.2 (Flatten Binary Tree – Hand-Parallelised Program (HPP)):

$$
\begin{array}{l}
flatten\ xt\ t \\
\textbf{where} \\
flatten\ E\ t\ \ \ \ \ \ \ \ \ \ =\ [] \\
flatten\ (B\ x\ lt\ rt)\ t\ = \\
\qquad \left\{ \begin{array}{l}
h\ (t \leq 0) \\
\textbf{where} \\
h\ True\ = [x] +\!\!+ (flatten\ lt\ t) +\!\!+ (flatten\ rt\ t) \\
h\ False = runEval\ \$\ do \\
\qquad\qquad\qquad ls \leftarrow rpar\ (flatten\ lt\ (t-1)) \\
\qquad\qquad\qquad rs \leftarrow rseq\ (flatten\ rt\ (t-1)) \\
\qquad\qquad\qquad return\ ([x] +\!\!+ ls +\!\!+ rs)
\end{array} \right.
\end{array}
$$

The distilled version of the *flatten* program is presented in Example 6.6.3.3, which is defined using two functions $flatten_1$ and $flatten_2$. Here, we observe that the intermediate data structure has been removed by the distillation transformation. This is achieved by using the definition of the append operator $+\!\!+$ during the distillation process.

Example 6.6.3.3 (Flatten Binary Tree – Distilled Program (DP)):

$$
\begin{array}{l}
flatten_1\ xt\\
\textbf{where}\\
\begin{array}{lcl}
flatten_1\ xt & = & flatten_2\ xt\ []\\
flatten_2\ E\ ys & = & ys\\
flatten_2\ (B\ x\ lt\ rt)\ ys & = & \textbf{let}\ ys' = (flatten_2\ rt\ ys)\\
 & & \textbf{in}\ x : (flatten_2\ lt\ ys')
\end{array}
\end{array}
$$

Here, we observe that recursive function $flatten_2$ has a maximum of two recursive calls in its definition body. Consequently, based on the parallelisation steps presented in Section 5.2.2, we use the encoding transformation version 1 to encode the pattern-matched inputs into a new data type. Example 6.6.3.4 presents the encoded version of the distilled flatten binary tree program.

Example 6.6.3.4 (Flatten Binary Tree – Encoded Program):

$$
\begin{array}{l}
\textbf{data}\ T_{flatten_2}\ a\ =\ c_1 \mid c_2\ a\ (T_{flatten_2}\ a)\ (T_{flatten_2}\ a)\\
\\
\begin{array}{lcl}
encode_{flatten_2}\ E & = & c_1\\
encode_{flatten_2}\ (B\ x\ lt\ rt) & = & c_2\ x\ (encode_{flatten_2}\ lt)\ (encode_{flatten_2}\ rt)
\end{array}\\
flatten_1\ xt\\
\textbf{where}\\
\begin{array}{lcl}
flatten_1\ xt & = & flatten'_2\ (encode_{flatten_2}\ xt)\ []\\
flatten'_2\ c_1\ ys & = & ys\\
flatten'_2\ (c_2\ x\ lt\ rt)\ ys & = & \textbf{let}\ ys' = (flatten'_2\ rt\ ys)\\
 & & \textbf{in}\ x : (flatten'_2\ lt\ ys')
\end{array}
\end{array}
$$

In the encoded program, we observe that the new encoded data type $T_{flatten_2}$ has the same structure as the original type *BTree* of the binary tree. Consequently, the encoded program has the same algorithmic structure as the distilled program. However, the encoded program cannot be defined using a polytypic reduce or map-reduce skeleton shown in Example 6.6.3.5 that are defined over the type $T_{flatten_2}$.

Example 6.6.3.5 (Reduce and Map-Reduce Skeletons over $T_{flatten_2}$):

$$
\begin{array}{lcl}
reduce_{flatten_2}\ c_1\ g_1\ g_2 & = & g_1\\
reduce_{flatten_2}\ (c_2\ x\ lt\ rt)\ g_1\ g_2 & = & g_2\ x\ (reduce_{flatten_2}\ lt\ g_1\ g_2)\ (reduce_{flatten_2}\ rt\ g_1\ g_2)
\end{array}
$$

$$
\begin{array}{lcl}
mapReduce_{flatten_2}\ c_1\ g_1\ g_2\ f_2 & = & g_1\\
mapReduce_{flatten_2}\ (c_2\ x\ lt\ rt)\ g_1\ g_2\ f_2 & = & g_2\ (f_2\ x)\ (reduce_{flatten_2}\ lt\ g_1\ g_2\ f_2)\\
 & & \qquad (reduce_{flatten_2}\ rt\ g_1\ g_2\ f_2)
\end{array}
$$

This is due to the mismatch in the algorithmic structure of the skeletons and the $flatten_2$ function the reason for which is as follows: the distillation transformation uses the definition of the append operator $+\!\!+$ to remove the intermediate data structures created by $(flatten_2\ lt)$ and $(flatten_2\ rt)$. In this process, the generalisation step in distillation extracts the expression $(flatten_2\ rt)$ as a **let**-variable ys' bound to $(flatten'_2\ rt\ ys)$ which is then used to compute the result as an accumulating argument in $x : (flatten'_2\ lt\ ys')$. This introduces a data dependency as can be seen from the use of the **let**-variable ys' between the recursive calls to $flatten'_2$ in Example 6.6.3.4. However, the polytypic skeletons in Example 6.6.3.5, which follow the widely used forms of polytypic map- and reduce-based skeletons in literature, require that the recursive calls do not have any dependency. As a result of this mismatch, the encoded version of the flatten binary tree program cannot be defined using these polytypic skeletons. This can be addressed by adapting the polytypic skeletons that may allow data dependencies between the recursive calls and speculatively evaluate the recursive calls in parallel using Glasgow Parallel Haskell as explained later using Example 6.6.3.8.

On the other hand, if we consider the append operator $+\!\!+$ to be built-in, then the encoded version of the flatten binary tree program is as shown in Example 6.6.3.6.

Example 6.6.3.6 (Flatten Binary Tree – Encoded Program with Build-In $+\!\!+$):
data $T_{flatten}\ a\ =\ c_1 \mid c_2\ a\ (T_{flatten}\ a)\ (T_{flatten}\ a)$

$encode_{flatten}\ E\ =\ c_1$
$encode_{flatten}\ (B\ x\ lt\ rt)\ =\ c_2\ x\ (encode_{flatten}\ lt)\ (encode_{flatten}\ rt)$
$flatten'\ (encode_{flatten}\ xt)$
where
$flatten'\ c_1\ =\ []$
$flatten'\ (c_2\ x\ lt\ rt)\ =\ [x] +\!\!+ (flatten'\ lt) +\!\!+ (flatten'\ rt)$

By applying our transformation on this encoded version, we observe that it can be parallelised using the $reduce_{flatten}$ skeleton defined over the type $T_{flatten}$ as shown in Example 6.6.3.7. It is important to note that this encoded parallel program parallelises the flatten binary tree program in the same way as the hand-parallelised version in Example 6.6.3.2.

Example 6.6.3.7 (Flatten Binary Tree – Encoded Parallel Program (EPP)):
data $T_{flatten}\ a\ =\ c_1 \mid c_2\ a\ (T_{flatten}\ a)\ (T_{flatten}\ a)$

$encode_{flatten}\ E\ =\ c_1$
$encode_{flatten}\ (B\ x\ lt\ rt)\ =\ c_2\ x\ (encode_{flatten}\ lt)\ (encode_{flatten}\ rt)$

$$
\begin{array}{l}
reduce_{flatten}\ c_1\ t\ g_1\ g_2 \quad = \ g_1 \\
reduce_{flatten}\ (c_2\ x\ lt\ rt)\ t\ g_1\ g_2 \ = \\
\qquad \left\{ \begin{array}{l}
h\ (t \leq 0) \\
\textbf{where} \\
h\ True\ = g_2\ x\ (reduce_{flatten}\ lt\ t\ g_1\ g_2)\ (reduce_{flatten}\ rt\ t\ g_1\ g_2) \\
h\ False = runEval\ \$\ do \\
\qquad\qquad ls \leftarrow rpar\ (reduce_{flatten}\ lt\ (t-1)\ g_1\ g_2) \\
\qquad\qquad rs \leftarrow rseq\ (reduce_{flatten}\ rt\ (t-1)\ g_1\ g_2) \\
\qquad\qquad return\ (g_2\ x\ ls\ rs)
\end{array} \right. \\
flatten''\ (encode_{flatten}\ xt)\ t \\
\textbf{where} \\
flatten''\ xs\ t\ =\ reduce_{flatten}\ xs\ t\ g_1\ g_2 \\
\qquad\qquad \textbf{where} \\
\qquad\qquad g_1 \qquad\quad [] \\
\qquad\qquad g_2\ x\ ls\ rs\ =\ [x] +\!\!+ ls +\!\!+ rs
\end{array}
$$

Alternately, upon further analysis we observe that the encoded program in Example 6.6.3.4 can be speculatively parallelised the two recursive calls to $flatten'_2$ using GpH as shown in Example 6.6.3.8. This is based on the automatic parallelisation technique proposed by Dever in [26].

Example 6.6.3.8 (Flatten Binary Tree – Parallelised using GpH):

$$
\begin{array}{l}
\textbf{data}\ T_{flatten_2}\ a\ =\ c_1 \mid c_2\ a\ (T_{flatten_2}\ a)\ (T_{flatten_2}\ a) \\
encode_{flatten_2}\ E \quad = \ c_1 \\
encode_{flatten_2}\ (B\ x\ lt\ rt)\ =\ c_2\ x\ (encode_{flatten_2}\ lt)\ (encode_{flatten_2}\ rt) \\
flatten_1\ xt\ t \\
\textbf{where} \\
flatten_1\ xt\ t \quad = \ flatten'_2\ (encode_{flatten_2}\ xt)\ []\ t \\
flatten'_2\ c_1\ ys\ t \quad = \ ys \\
flatten'_2\ (c_2\ x\ lt\ rt)\ ys\ t\ =\ h\ (t \leq 0) \\
\qquad\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\qquad h\ True\ =\ \textbf{let}\ ys' = (flatten'_2\ rt\ ys\ t) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ x : (flatten'_2\ lt\ ys'\ t) \\
\qquad\qquad\qquad\qquad h\ False\ =\ runEval\ \$\ do \\
\qquad\qquad\qquad\qquad\qquad\qquad ys' \leftarrow rpar\ (flatten'_2\ rt\ ys\ (t-1)) \\
\qquad\qquad\qquad\qquad\qquad\qquad ys \leftarrow rseq\ (flatten'_2\ lt\ ys'\ (t-1)) \\
\qquad\qquad\qquad\qquad\qquad\qquad return\ (x : ys)
\end{array}
$$

6.6.4 Insertion Sort

The OP version of a program to sort a given list using insertion sort is presented in Example 6.6.4.1. The function *isort* inserts each element x in a given list into the correct position in the sorted tail xs using the function *insert*. Here, the intermediate data structures are the results of $(isort\ xs)$ that are decomposed by the *insert* function.

Example 6.6.4.1 (Insertion Sort – Original Program (OP)):

$$
\begin{array}{l}
isort\ ::\ [a] \rightarrow [a] \\
isort\ xs \\
\textbf{where} \\
isort\ [] \quad = [] \\
isort\ (x : xs) \quad = insert\ x\ (isort\ xs) \\
\\
insert\ y\ [] \quad = y : [] \\
insert\ y\ (z : zs) \ = h\ (y \leq z) \\
\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\quad h\ True \ = y : z : zs \\
\qquad\qquad\qquad\quad h\ False \ = z : (insert\ y\ zs)
\end{array}
$$

The DP version of this program that is free of intermediate data structures is presented in Example 6.6.4.2.

Example 6.6.4.2 (Insertion Sort – Distilled Program (DP)):

$$
\begin{array}{l}
arev\ ::\ [a] \rightarrow [a] \\
isort\ xs \\
\textbf{where} \\
isort\ [] \quad = [] \\
isort\ (x : xs) \ = insert\ x\ xs\ (\lambda x.(x : [])) \\
\\
insert\ x\ []\ f_1 \quad = f_1\ x \\
insert\ x\ (y : ys)\ f_1 \ = h_1\ (x \leq y) \\
\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\quad h_1\ True \ = \textbf{let}\ f_2 = \lambda z.h_2\ (y \leq z) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_2\ True \ = y : (f_1\ z) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_2\ False \ = z : (f_1\ y) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ insert\ x\ ys\ f_2 \\
\qquad\qquad\qquad\quad h_1\ False \ = \textbf{let}\ f_2 = \lambda z.h_2\ (x \leq z) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_2\ True \ = x : (f_1\ z) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad h_2\ False \ = z : (f_1\ x) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ insert\ y\ ys\ f_2
\end{array}
$$

Here, we observe that recursive function *insert* has two recursive calls in its definition body. Consequently, based on the parallelisation steps presented in Section 5.2.2, we use the encoding transformation version 1 to encode the pattern-matched inputs into a new data type T_{insert}. Example 6.6.4.3 presents the encoded version of the distilled insertion sort program.

Example 6.6.4.3 (Insertion Sort – Encoded Program):

$$
\begin{array}{l}
\textbf{data}\ T_{insert}\ a\ =\ c_1 \mid c_2\ a\ (T_{insert}\ a)\ (T_{insert}\ a) \\[4pt]
\begin{array}{lcl}
encode_{insert}\ [] & = & c_1 \\
encode_{insert}\ (y : ys) & = & c_2\ y\ (encode_{insert}\ ys)\ (encode_{insert}\ ys)
\end{array} \\[4pt]
isort\ xs \\
\textbf{where} \\
\begin{array}{lcl}
isort\ [] & = & [] \\
isort\ (x : xs) & = & insert'\ x\ (encode_{insert}\ xs)\ (\lambda x.(x : []))
\end{array} \\[4pt]
\begin{array}{lcl}
insert'\ x\ c_1\ f_1 & = & f_1\ x \\
insert'\ x\ (c_2\ y\ ys_1\ ys_2)\ f_1 & = & h_1\ (x \leq y) \\
& & \textbf{where} \\
& & \begin{array}{lcl}
h_1\ True & = & \textbf{let}\ f_2 = \lambda z.h_2\ (y \leq z) \\
& & \quad\quad \textbf{where} \\
& & \quad\quad \begin{array}{lcl} h_2\ True & = & y : (f_1\ z) \\ h_2\ False & = & z : (f_1\ y) \end{array} \\
& & \textbf{in}\ insert'\ x\ ys_1\ f_2 \\
h_1\ False & = & \textbf{let}\ f_2 = \lambda z.h_2\ (x \leq z) \\
& & \quad\quad \textbf{where} \\
& & \quad\quad \begin{array}{lcl} h_2\ True & = & x : (f_1\ z) \\ h_2\ False & = & z : (f_1\ x) \end{array} \\
& & \textbf{in}\ insert'\ y\ ys_2\ f_2
\end{array}
\end{array}
\end{array}
$$

Here, the recursive function $insert'$ is defined over the binary data type T_{insert} and uses an accumulating parameter f_1. However, the parallel polytypic reduce skeletons that are used in this thesis based on literature do not support accumulating arguments. Therefore, the encoded version of insertion sort presented in Example 6.6.4.3 is not automatically parallelised using our parallelisation technique presented in this thesis.

Additionally, we observe that parallel polytypic skeletons need to be extended and implemented to support accumulating parameters. Further work in this direction can be led by the polytypic accumulate skeleton approach introduced in [44].

6.7 Summary

In this chapter, we introduced a few benchmark programs that are used to evaluate the transformation method proposed in this thesis. Given the sequential version of a benchmark program, we obtain its distilled version using the distillation transformation. Following this, the distilled version is parallelised using the strategy proposed in Section 5.2.2 that makes use of the encoding transformation presented in Chapter 5. Further, we define the hand-parallelised version of each benchmark program using the skeletons in Eden library or Glasgow Parallel Haskell (GpH), whichever is suitable to the benchmark

program. Following this, we evaluated the performance of each EPP version of the benchmark programs by comparing it with the original sequential program (OP), the distilled program (DP) and a hand-parallelised program (HPP).

From the benchmark program transformations presented in Section 6.5, we observe that both the distillation and the encoding transformation can facilitate parallelisation of a given program using fewer intermediate data structures. For instance, the original versions of the matrix multiplication and Fibonacci series sum programs are defined using intermediate data structures. While the distillation transformation removes these intermediate data structures, the distilled forms could not be parallelised using skeletons. Here, the programs produced by the encoding transformation are defined in such a way that they contain instances of parallel map- and reduce-based skeletons. Similarly, it is the data type transformation performed by the encoding transformation that facilitates identification of skeletons in the dot-product of binary trees and totient benchmark programs. In a few exceptions such as the maximum prefix sum and sum of squares programs, we observe that the original or distilled versions contain instances of list-based skeletons before applying the encoding transformation. The consequence of applying the encoding transformation in this case is that we again identify the encoded program to be an instance of the same skeletons.

6.7.1 Observations from Parallelisation

From our evaluations, we analyse the speedup achieved, the parallel execution profile and cost-centre of the encoded parallel programs produced. Based on this analysis, we make the following observations about the parallel programs produced by our transformation method based on their performances:

- **Elimination of Intermediate Data Structures**: It is evident that the use of intermediate data structures can have a significant impact on the performances of the sequential and parallel versions of a given program. For instance, we observe from the matrix multiplication benchmark evaluation that eliminating intermediate data structures can potentially result in significant speedups in the EPP version when compared to the OP and HPP versions. We also observe the performance gain from eliminating intermediate data structures in the maximum prefix sum and sum squares of list benchmark programs.

- **Encoding Inputs**: The proposed encoding transformation allows parallelisation of programs that do not initially contain instances of parallel skeletons. This is evident from the transformation of benchmark programs such as dot-product of binary trees, Fibonacci series sum and totient computation. Therefore, we observe that encoding the inputs can facilitate the identification of list-based and/or polytypic parallel skeletons in the transformed programs.

  Additionally, from the evaluations, we observe that the cost of encoding is an important factor to achieve efficient parallel execution. If the cost of encoding inputs is relatively low when compared to the cost of the computation executed in parallel, then we observe efficient parallel execution that is comparable to or better than hand-parallelised versions. This can be observed in examples such as matrix multiplication, power tree, dot-product of binary trees and maximum prefix sum. However, if the cost of encoding the inputs is significantly higher, then the encoded parallel programs do not exhibit efficient parallel execution as the overhead from encoding outweighs the gain from parallelisation. This can lead to the EPP version performing better than the sequential version but slower than hand-parallelised versions (as observed in examples totient computation, maximum prefix sum and Fibonacci series sum), or the EPP version performing slower than the sequential and hand-parallelised versions (as observed in the sum squares of list example).

  Therefore, it is beneficial to apply the encoding transformation only in cases where the cost of encoding is lower than the cost of the parallel computation to achieve efficient parallelisation. This can be estimated by comparing the cost of a recursive function with that of its corresponding encode function in an encoded parallel program.

- **Efficient Parallelisation Using Skeletons**: We observe that it is possible to achieve positive speedups for our transformed programs by using existing implementations of list-based and polytypic parallel skeletons. It is important to note that even in cases where a good parallel execution profile is achieved, such as in the power tree, totient computation and maximum prefix sum programs, the speedup factors achieved depend upon the gain from parallel execution as compared to the cost of encoding or parallelisation.

We also observe that when using nested skeletons in parallel execution, the cost of parallelisation may result in in poor performance (for both EPP and HPP versions) if the creation of parallel threads is poorly controlled which increases the cost of thread creation and management. Our naïve strategy to avoid nested skeletons and use parallel implementations only for top-level skeletons results in positive speedups of the EPP versions of the benchmark programs. This is explored in detail in the NESL programming language [8] which particularly addresses nested data parallelism.

Furthermore, we observe that the thresholding technique used to limit the number of sparks created by the polytypic skeletons implemented with GpH may be improved with a more efficient throttling technique. For instance, instead of determining the number of parallel processes based on the number of cores used and creating one chunk of the input for each parallel process, the input data structure may also be partitioned into chunks depending on the cost of the computation that is applied on each chunk. The objective is to reduce the cost of parallelisation when compared to the cost of the computations performed in parallel. This could be addressed in future along with a tree-contraction based approach to implement polytypic skeletons discussed below.

The workload of parallel threads needs to be well-balanced for good parallel execution. For instance, the parallelisation of polytypic skeletons is dictated by the encoded n-ary data structure. Hence, an unbalanced encoded data structure will potentially lead to unbalanced parallel execution as can be seen from the power tree and dot-product of binary tree benchmark programs. For list-based skeletons, the workload of the threads can be dictated by the list elements. This can be seen in the Fibonacci series sum evaluation where a simplistic distribution (such as round-robin distribution) of the encoded list among the parallel threads results in poor workload distribution leading to sub-optimal performance. Therefore, we require efficient data partitioning and workload balancing techniques in skeleton implementations. We suggest that this can be addressed by using tree-contraction for implementing the polytypic skeletons and a more sophisticated workload distribution for list-based skeletons. However, this needs to be verified by evaluations. Detailed work on efficient implementations of parallel skeletons is beyond the scope of this thesis and future work in this direction is discussed in Chapter 7 under Sec-

tion 7.5 on future work.

6.7.2 Observations from Problem Cases

Additionally, we also discussed a set of problem cases in Section 6.6 to illustrate cases where our proposed transformation method does not produce a parallel program by using the distillation and encoding transformations. Analysing these examples is of interest to us because the given sequential versions of these problem cases can be parallelised using skeletons albeit by retaining the intermediate data structures. The primary reasons we observe for not parallelising these problem cases are:

1. Even if the encoded program is defined over a single pattern-matched input as in a skeleton, there may be multiple input arguments that are not pattern-matched and hence not encoded. This potentially causes mismatch if the result is computed using multiple accumulating arguments in the distilled or encoded program, while the *map-reduce* and *accumulate* skeletons are defined using only a single accumulating argument. This was observed in examples such as the maximum segment sum program. Even though we observe that tupling the accumulating arguments may help identifying instances of *map-reduce* or *accumulate* skeletons, it may not be parallelisable due to the non-associativity of the resulting reduction operator.

   From the insertion sort example, we also observe that the encoded programs which are defined over an n-ary data type and use accumulating parameters are not automatically parallelisable using the parallel polytypic skeletons used in this thesis since they do not use accumulating parameters.

2. In some programs that are defined using built-in operators which may be associative (such as $+\!\!+$ for lists), the distillation transformation uses the definition of the operator to remove intermediate data structures. This may, in some cases, introduce non-associative operators that are used in the definition of the associative built-in operators. Consequently, the distilled program, and hence the encoded program, may be defined using non-associative operators which hinder parallel evaluation of the transformed program. This was observed in examples such as the reverse list and insertion sort programs.

3. Some programs may originally be defined in such a way that recursive calls do not have data dependencies among each other, though they may exist in the expression

that aggregates the results of the recursive calls. During the process of removing intermediate data structures, the distillation transformation uses the generalisation strategy discussed in Section 2.1.1 where expressions are extracted as variables to enable folding. In some programs, this shifts the data dependencies between recursive calls in the distilled program. Consequently, the distilled or encoded program may not match the polytypic map- or reduce-based skeletons, which in their current forms from literature, do not support such dependencies. This was observed in problem cases such as the flatten binary tree program.

The transformation of programs such as quick sort, merge sort, n-queens problem and ray-tracer algorithm were also performed. However, they were not parallelised for the same reasons discussed above that have identified using the problem cases presented in this chapter.

Potential directions to address these problem cases by adapting our skeletons and transformation method are discussed in Chapter 7, which summarises the research presented in this thesis, as a part of future work in Section 7.5.

Chapter 7

Conclusions

7.1 Introduction

The work presented in this thesis investigates our research hypothesis:

"*Program transformation can be used to automatically identify parallel computations in a given program, potentially leading to its efficient parallel execution*".

In this chapter, we present concluding remarks on our transformation method composed of the following three stages to address this hypothesis:

1. Using an existing transformation technique called distillation, we reduce the number of inefficient intermediate data structures used in a given program. This technique was discussed in Chapter 3.

2. Using map- and reduce-based skeletons to encapsulate parallel computations and a skeleton identification technique, we automate the identification of parallel computations in a program. The skeletons of interest to us and the skeleton identification technique were discussed in Chapter 4.

3. Using an encoding transformation, we transform the program produced by distillation by combining pattern-matched inputs into a single input to facilitate automatic identification of the parallel skeletons. The encoding transformation was introduced and discussed in Chapter 5.

The remainder of this chapter is organised as follows: In Section 7.2, we present a summary of the research presented in this thesis with observations about our transformation methods and how they address our research hypothesis. In Section 7.3, we discuss the

contributions of this research work to the field of using program transformation to parallelise functional programs. In Section 7.5, we present a summary of future directions and improvements to the work presented in this thesis and the open problems and difficulties that remain in this field.

7.2 Research Summary

We summarise the research presented in this thesis and our observations by revisiting our research questions to discuss our methods proposed to answer them.

- "***RQ-1:*** *How can potential parallel computations in a program be automatically identified?*"

  Our proposed solution to this question was presented in Chapter 4. To answer this question, we use algorithmic skeletons that abstract parallelisable map- and reduce-based computations which are commonly used in parallel program development. We presented both polytypic and list-based definitions of these skeletons to address a wide range of programs that may be defined over any data type.

  Following this, we proposed a skeleton identification technique to automatically identify computations in a given program that are instances of the parallel skeletons and implement them using suitable calls to the corresponding skeletons. This technique can use the distillation-based proof rules presented in Chapter 3 to automatically verify properties that skeleton operators are required to satisfy to enable their parallel evaluation. Here, by representing a given program and the skeletons as labelled transition systems (LTSs), we are able to find potential instances of the skeletons in a program by abstracting away from function names and comparing their definitions based on their recursive structures.

  We observe that this technique is widely applicable to any given set of skeletons on any program, without placing restrictions on either of them, and automatically extract the skeleton operators from the program. Further, the automatic verification of skeleton operator properties ensures that the parallel program produced can indeed be executed in parallel.

- "***RQ-2:*** *How can the transformed program be executed in parallel?*"

  Our solution to this question was also discussed in Chapter 4. Given that we identify

parallel computations as instances of parallel algorithmic skeletons, we require efficient parallel implementations of these skeletons to efficiently execute the transformed program on parallel hardware. Since we identify both polytypic and list-based skeletons (map, map-reduce and accumulate), we require parallel implementations of both polytypic and list-based skeletons.

From existing works, we observe that a majority of the effort has been in creating parallel implementations for list-based skeletons. A few existing libraries provide implementations of skeletons that operate over binary trees. However, there is a lack of skeleton implementations that operate over n-ary trees which are required for our polytypic skeletons. Consequently, we use the existing Eden library to execute our list-based skeletons and a simplistic approach using Glasgow Parallel Haskell (GpH) to execute our polytypic skeletons.

We observe that while this approach allows parallel execution of the parallel programs produced by our transformation method, efficient execution of the programs depends on a well-balanced workload across the parallel threads created by the skeleton implementations. In particular, the parallel implementation of polytypic skeletons should not be dictated by the data structure on which the parallel computation is applied. This will ensure that an unbalanced data structure does not result in an unbalanced parallel execution. For this, we suggest approaches such as tree contraction for efficient parallel implementations of polytypic skeletons.

- "***RQ-3:*** *How can a given program be transformed to aid identification of parallel computations?*"

  The encoding transformation presented in Chapter 5 that is applied to distilled programs was proposed as a solution to this question. The motivation of this transformation is to address potential mismatches in the algorithms and data types of a given program and the skeletons that we aim to identify.

  While the distillation transformation is used to reduce inefficient intermediate data structures in a given program, the encoding transformation transforms the distilled program by encoding the pattern-matched inputs into a single input belonging to a data type whose structure matches the recursive structure of the distilled program. The objective of this approach is to facilitate identification of skeletons defined over the encoded input to be identified in the programs produced by the encoding trans-

formation.

While we find that the proposed transformation can lead to parallelisation of a given program, we make the following broad observations based on our evaluations of a set of benchmark programs:

– **Efficiently parallelised programs:**

This result is observed in a program whose original definition contains large intermediate data structures, and has significant mismatch between its algorithmic structure and input data structure that does not allow it to be defined using skeletons. Here, the distillation transformation results in significant speedup and the encoding transformation facilitates parallelisation using skeletons in an otherwise difficult to parallelise original or distilled program. Consequently, the parallel program produced by our transformation performs better than or on par with the hand-parallelised version.

– **Moderately parallelised programs:**

This result is observed in a parallel program produced by our transformation if the cost of encoding inputs is significantly high compared to the cost of the computation executed in parallel. This can make encoding a large overhead due to which the encoded parallel program is less efficient than the hand-parallelised version or even the distilled version in some cases.

– **Programs not parallelised:**

Through some problem cases, we observed that some potentially hand-parallelisable programs are not parallelised by our proposed transformation method. The three main reasons we identify for this result are as follows:

1. **Mismatch of accumulating parameters:**

   In some problem cases, we observe that the distilled program may be defined using more than one accumulating argument which will not be pattern-matched. However, the polytypic skeletons used in this thesis do not use accumulating arguments and the list-based skeletons use a single accumulating argument. This creates a potential mismatch between an encoded program and the skeletons.

   Even though tupling the accumulating arguments allows identification of list-based skeleton instances, the resulting program may not be parallelisable as the skeleton operator (such as the reduction operator) may not satisfy desired

properties (such as associativity) for parallelisation.

2. **Not parallelisable due to skeleton operator:**
   In some problem cases, we observe that the distillation transformation, in the process of removing intermediate data structures, may use the library definition of an operator with properties that are desirable for parallel evaluation and produces programs that may be defined using operators that do not satisfy properties required for parallel evaluation (such as associativity).
   In these cases, distillation using the library definitions of such operators works against the objectives of our transformation method. Therefore, it will be beneficial to not use the library definition of built-in operators with these desirable properties (such as associativity) in the distillation transformation.
3. **Dependencies between recursive calls:**
   Based on some problem cases, we observe that the distillation transformation used to eliminate intermediate data structures may fuse expressions in such a way that the distilled program may contain data dependencies between recursive function calls. This prevents the definition of such programs using skeletons presented in this thesis, which require that the recursive calls be free of data dependency.

7.3 Contributions

The overall objective of this research was to facilitate automatic identification of parallel computations in a given program and parallel execution of the transformed program using parallel skeletons. According to the detailed study of the existing body of work presented in Chapter 2, the different existing approaches to this research problem have varying strengths and limitations. Based on this, the major contributions of our proposed transformation method are as follows:

1. **Skeleton-based programs with fewer intermediate data structures:**
   The programs produced by our transformation method are potentially defined using parallel skeletons. While this provides a good level of abstraction to the developer by hiding the parallel implementation details, the parallel programs also use fewer inefficient intermediate data structures. This was achieved by using the distillation transformation as a part of our transformation method. As discussed earlier, a dis-

tilled program that potentially does not match the structure of parallel skeletons is transformed using our encoding technique to facilitate its match with the map, map-reduce or accumulate skeletons.

Existing approaches to skeleton-based parallel programming make heavy use of inefficient intermediate data structures. Even though some approaches, such as the Accelerate library, use "skeleton fusion" techniques to merge neighbouring skeletons, this is not always easy to achieve. This is because the algorithmic structures of transformed programs produced by techniques that eliminate intermediate data structures may not match those of the parallel skeletons.

2. **Fully automatable program parallelisation:**
   Our proposed parallelisation process is completely automated. This is aided by the distillation and encoding transformations, and the skeleton identification technique. The skeleton identification technique also allows automatic extraction of skeleton operators and their verification for specific properties to enable parallel execution.

   In most existing unfold/fold-based and calculation-based transformation methods, certain operators or auxiliary functions required by the parallel program need to be derived by hand as they are required to satisfy certain properties, such as associativity, for parallel execution.

3. **No restrictions on programs:**
   Our proposed transformation method is applicable to all programs that are defined over any number of inputs of any data type. This is facilitated by the encoding transformation technique that combines all inputs that are pattern-matched and consumed by a given program into a single input.

   Most existing techniques to systematically derive parallel programs or transform a given program to identify skeletons enforce restrictions on the program or its data type. While some methods are applicable only to programs that are defined over lists, arrays or binary trees, other methods are applicable to programs that have only a single recursive input.

4. **Data type specialisation:**
   As evidenced by the related work discussed in Appendix A, data type transformation and pattern specialisation can improve the efficiency of sequential programs. The

encoding transformation proposed in this thesis achieves this by matching the pattern-matching of inputs with the algorithmic structure of the program.

While there exist a few other techniques [69] to achieve this, they are not fully automated. However, the encoding transformation proposed allows complete automation by presenting the forms of the new data types to be created and the conversion functions for the data transformation.

We observe that these attributes of the transformation method proposed in this thesis offer improvements and key insights to fill gaps that presently exist in automatic program parallelisation using program transformation.

7.4 Limitations

Having analysed the results of the transformation proposed in this thesis and discussed the major contributions, we acknowledge the following limitations in the parallelisation method presented in this thesis:

1. **Parallel workload balance:**

   The workload balance across parallel processes created by skeletons is not explicitly addressed in this thesis. This is primarily because the objective of this work is to address program transformation methods to aid identification of skeleton instances in a given program. The workload imbalance can be addressed in future either as a part of the data type transformation or in the implementations of parallel skeletons.

2. **Threshold for degree of parallelism:**

   The simplistic approach of parallel evaluation of only top-level skeletons is one of the potential reasons for not achieving the expected near-linear speedups even in examples such as matrix multiplication that are known to scale well. Therefore, we need a better threshold mechanism to control the degree of parallelism for nested skeletons in place of the current simplistic approach. This can be based upon the approach in NESL [8] for nested data parallel programs that are defined over arrays.

3. **Map- and reduce-based skeletons:**

   We have addressed the most commonly used data parallel skeletons by selecting map- and reduce-based skeletons. This is primarily because of our approach to

address the mismatch between the input data and the algorithmic structure of a given program using the data type transformation which is mainly observed in map and reduce-based computations.

However, there are other skeletons such as high-level divide-and-conquer and task parallel skeletons that we have not explored and would be of interest. Addressing such skeletons may require further study of the issues in identifying their instances in a given program.

4. **Processor and memory models:**

   For the skeleton implementations and the evaluations, we have considered a shared-memory multi-core CPU architecture. However, there are other processor and memory architectures such as GPUs and distributed-memory systems where the costs associated with communication and data transfer may be higher. Even though we consider these to be beyond the scope of this thesis, they can be addressed both as a part of the data type transformation and the skeleton implementations.

7.5 Future Work

7.5.1 Efficient Implementation of Polytypic Skeletons

The parallel implementation of polytypic skeletons presented in this thesis in Section 4.3 is simplistic and its efficiency is dictated by the structure of the n-ary input tree. As a result, an unbalanced input tree will result in a poor parallel evaluation where parallel processes are created at every branch node to evaluate its sub-trees in parallel.

Consider the reduction of an unbalanced binary tree illustrated in Figure 7.1 where the result is computed by adding all values at the nodes in the tree. The simplistic parallelisation approach will result in serialisation of the addition operations, thus requiring 7 addition steps (one for each branch node) to compute the result. This is because the parallel processes are created from the root node downward and efficiency depends on similar sized sub-trees for a balanced workload across the parallel processes.

A more efficient approach to parallel computations on trees is by using tree contraction described in Section 2.2.4. This is because tree contraction applies computations on trees from leaf nodes rather than the root node, and thus performs better on unbalanced trees.

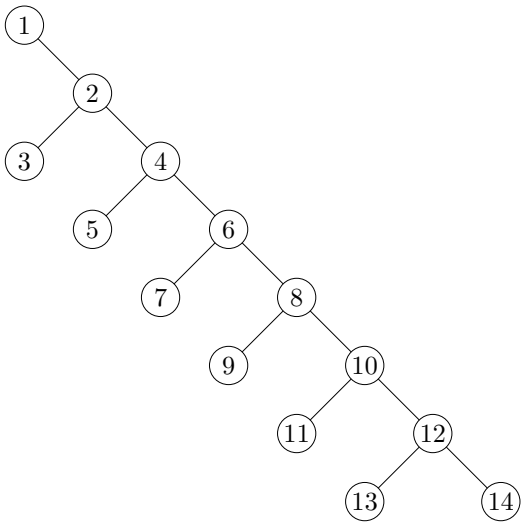


Figure 7.1: An Unbalanced Binary Tree

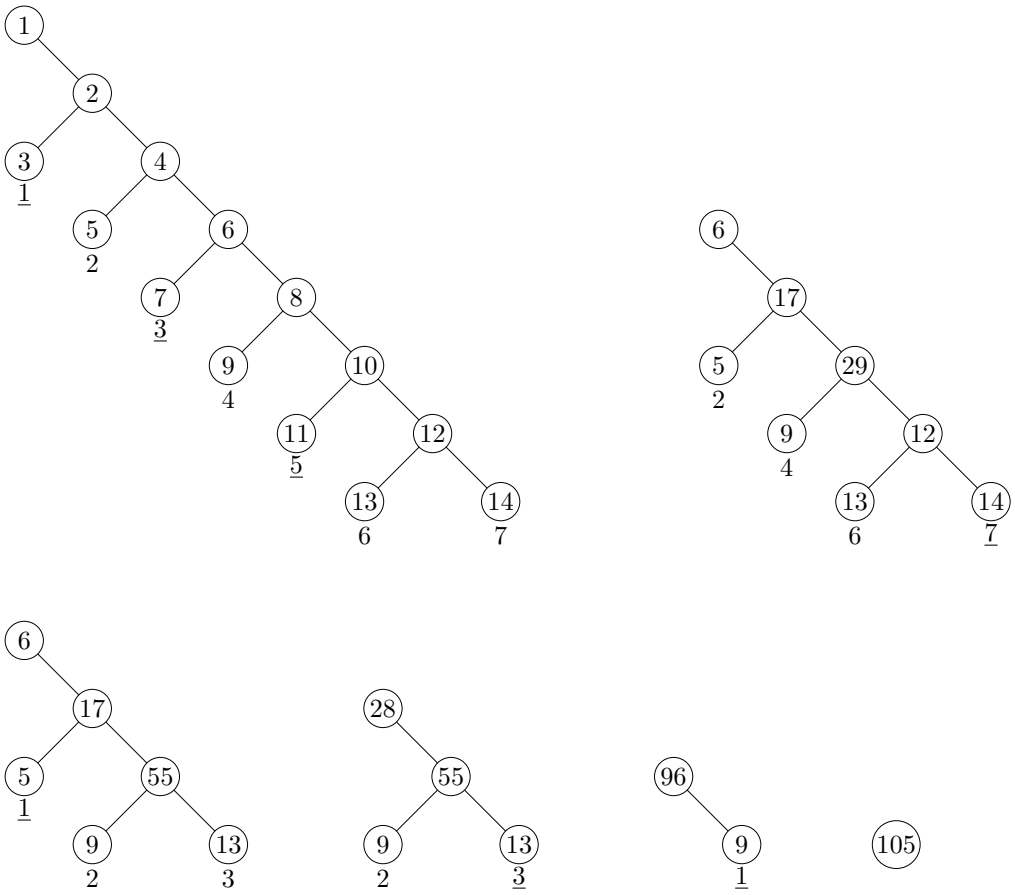


Figure 7.2: Reduction by Tree Contraction

A step-by-step illustration of reducing the unbalanced tree from Figure 7.1 is presented in Figure 7.2 based on the tree contraction procedure listed in [70]. Here, the contraction operations in each step are applied in parallel on the numbered leaf nodes that are underlined, thus requiring 5 addition steps to compute the result.

While the simplistic parallellisation of tree computations used in this thesis offers a time complexity of `O(log`$_n$ `N)` for balanced n-ary trees, and `O(N)` for skewed trees, where `N` is the number of nodes in a tree, the tree contraction-based approach guarantees a time complexity of `O(N/P + log`$_n$ `N)` for n-ary trees, where `P` is the number of processing elements in the parallel evaluation [70]. Despite being well studied and incorporated into some skeleton libraries such as SkeTo [64] for C++ programs, polytypic skeleton libraries implemented using tree contraction still do not exist for functional languages such as Haskell. Thus, creating such a polytypic skeleton library will be one of the next interesting steps.

7.5.2 Improvements to Proposed Transformation

The potential limitations of the data type transformation were discussed in Section 7.2 based on the problem cases and cost of performing the data type transformation. To address these limitations, we propose the following modifications to the data type transformation:

1. **Selective data type transformation:**

   From the evaluation of the benchmark programs such as totient computation, maximum prefix sum, Fibonacci series sum and sum squares of list, we observe that the higher cost of encoding the inputs results in the EPP version performing slower than the sequential and/or hand-parallelised counterparts.

   Among these benchmark programs, we also observe that the distilled versions already contain instances of parallel skeletons. This is because the structure of the input data type already matches the algorithmic structure of the program. In such cases, the encoding transformation is redundant and hence an overhead. This was observed in the maximum prefix sum programs and sum squares of list.

   A solution to this is to selectively encode inputs only when it is desirable according to the following modified steps to parallelise a given program:

   (a) Apply the distillation transformation (Chapter 3) on the given program. This

produces a *distilled program*.

(b) For each recursive function f in the distilled program,

i. Compute the maximum number of recursive calls, n, in the bodies of the definition of function f.

ii. `IF` f operates over a single pattern-matched input of type T and the structure of T matches the recursive structure of f

`THEN` Goto Step (c).

`ELSE IF` $n > 1$

`THEN` Apply encoding transformation version 1 (Section 5.2.3) on f.

`ELSE` Apply encoding transformation version 2 (Section 5.2.4) on f.

(c) Apply the skeleton identification transformation presented in Chapter 4 on the program from Step (b) using skeletons that operate over the input data types of the recursive functions. This produces a *transformed parallel program*.

(d) Execute the transformed parallel program using implementations for skeletons defined in Step (c) as discussed in Section 4.3.

2. **Operator property-aware distillation:**

In some programs, we observe that the original version can be defined using skeletons and their evaluation can be parallelised due to the use of skeleton operators that satisfy the desired properties. For example, the original naïve reverse benchmark program discussed in Chapter 6 can be defined using the *map-reduce* skeleton where the associative reduction operator $+\!\!+$ allows parallel evaluation.

However, we observe that even if the encoded program can be defined using skeletons, it is not parallelisable if the skeleton operators do not satisfy the desired properties. This case may arise if the distillation transformation uses the library definitions of the operators used in the original program. For example, the encoded version of the naïve reverse program is defined using the *Cons* operator that is not associative (obtained from the library definition of $+\!\!+$) and hence does not allow parallelisation using the *map-reduce* skeleton.

As a solution to this, we could adapt the transformation steps in distillation to be aware of desired properties of operators that are present in the original program using the Poitín theorem proving rules discussed in Chapter 3. For example, distillation can leave in place associative operators used in the original program without using

their library definitions. Consequently, the distilled program will still be defined using operators whose properties could lead to parallelisation using skeletons.

3. **Skeleton-aware distillation:**
   Instead of first distilling programs and then performing data type transformation with an objective of identifying skeleton instances, an alternate approach is to first perform data type transformation on a given program and then apply distillation by memoising the skeletons to be identified. In this case, during distillation, instances of skeletons that appear in the program defined over encoded data types will be replaced with calls to skeletons, followed by distillation of the skeleton operators.

   The objective here is to retain certain intermediate data structures in the distilled program that may be necessary to identify instances of the skeletons of interest. Removing such intermediate data structures was the reason for not parallelising programs such as maximum segment sum, flatten binary tree and insertion sort. However, this requires the data type transformation to be defined over the more general grammar in Definition 1.1, which would be a more complicated task, instead of the distilled form in Definition 3.18 where all arguments of a function call are variables. This can be addressed in future with modifications to the data type transformation.

7.5.3 Transformation for Execution on GPU

To address parallel programming for GPUs, there exist few libraries such as SkePU [30] and SkelCL [82] that provide skeletons with efficient implementations for execution on GPUs. These skeleton implementations operate over flat data types such as arrays, lists or vectors. Consider the dot-product program that operates over array/list data structures. Examples 7.1 and 7.2 present the definition of the dot-product program using the skeletons in the SkePU and SkelCL libraries, respectively.

Example 7.1 (Dot-Product using SkePU Skeletons):

```
BINARY_FUNC(plus, double, a, b, return a + b;)
BINARY_FUNC(mult, double, a, b, return a * b;)
int main ()
{
  /* create skeletons */
  skepu :: MapReduce < mult, plus > dotProduct(new mult, new plus);
  /* create input vectors */
  skepu :: Vector < double > v0(1000, 2);
  skepu :: Vector < double > v1(1000, 2);
  /* execute skeletons */
  double r = dotProduct(v0, v1);
  return 0;
}
```

The SkePU library offers implementations of the map, reduce, map-reduce and scan skeletons among others. While the map skeleton can operate over one, two or three vector or matrix inputs, the other skeletons operate over a single input.

Example 7.2 (Dot-Product using SkelCL Skeletons):

```
int main ()
{
  SkelCL :: init();
  /* create skeletons */
  SkelCL :: Reduce < double >
                    sum("double sum (double x, double y) {return x + y;}");
  SkelCL :: Zip < double >
                    mult("double mult (double x, double y) {return x * y;}");
  /* allocate and initialise input arrays */
  float *a_ptr = new double [ARRAY_SIZE];
  float *b_ptr = new double [ARRAY_SIZE];
  fillArray(a_ptr, ARRAY_SIZE);
  fillArray(b_ptr, ARRAY_SIZE); [2mm]
  /* create input vectors */
  SkelCL :: Vector < double > A(a_ptr, ARRAY_SIZE);
  SkelCL :: Vector < double > B(b_ptr, ARRAY_SIZE);
  /* execute skeletons */
  SkelCL :: Scalar < double > C = sum(mult(A, B));
  return 0;
}
```

The SkelCL library offers implementations of the map, reduce, zip and scan skeletons. While the map, reduce and scan skeletons operate over a single vector input, the zip skeleton operated over two inputs.

In this context, the following attributes of the transformation presented in this thesis allow translation of the skeletons identified into OpenCL/CUDA to allow execution on GPUs using libraries such as SkePU and SkelCL.

- Encoding inputs into a single input which may be a flat data type or an n-ary tree.
- Automatic extraction of map and reduce skeleton operators and verification of properties required for parallel evaluation.

Based on this, another interesting direction for future work will be the transformation of the identified skeleton instances to OpenCL kernels to facilitate their execution on GPUs. This requires the following steps to be completed:

1. Translation of skeleton operators defined in Haskell to C for use by libraries such as SkePU or SkelCL.
2. Transformation of skeleton inputs in the Haskell program to suit the SkePU or SkelCL library.
3. Add skeletons to such libraries to support non-linear data types that have shown to be parallelisable by our transformation method.

Bibliography

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A Simple Parallel Tree Contraction Algorithm. *J. Algorithms*, 10(2):287–302, 1989.

[2] Joonseon Ahn and Taisook Han. An Analytical Method for Parallelization of Recursive Functions. *Parallel Processing Letters*, 10(1):87–98, 2000.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers - Principles, Techniques and Tools. *Addison-Wesley*, 1986.

[4] Kiminori Matsuzaki Akimasa Morihata. A Parallel Tree Contraction Algorithm on Non-Binary Trees. *Technical Report, Department of Mathematical Engineering and Information Physics, University of Tokyo*, 2008.

[5] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating Code on Multi-cores with Fastflow. *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, pages 170–181, 2011.

[6] Lennart Augustsson. A Compiler for Lazy ML. *Proceedings of ACM Symposium on List and Functional Programming*, 1984.

[7] R. S. Bird. An Introduction to the Theory of Lists. *Proceedings of the NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*, pages 5–42, 1987.

[8] Guy E. Blelloch. NESL: A Nested Data-Parallel Language. *Technical Report, Carnegie Mellon University*, 1992.

[9] J. Brotherston. Cyclic Proofs for First-Order Logic with Inductive Definitions. *TABLEAUX-14, Volume 3702 of LNAI, Springer-Verlag*, 2005.

[10] Christopher Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick, and Archibald Elliott. Cost-Directed Refactoring for Parallel Erlang Programs. *International Journal of Parallel Programming*, 42(4):564–582, 2014.

[11] Christopher Brown, Kevin Hammond, Marco Danelutto, and Peter Kilpatrick. A Language-independent Parallel Refactoring Framework. *Proceedings of the Fifth Workshop on Refactoring Tools (WRT)*, pages 54–58, 2012.

[12] Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Schöner, and Tino Breddin. Paraphrasing: Generating Parallel Programs Using Refactoring. *10th International Symposium on Formal Methods for Components and Objects (FMCO), Revised Selected Papers*, 2013.

[13] Christopher Brown, Hans-Wolfgang Loidl, and Kevin Hammond. ParaForming: Forming Parallel Haskell Programs Using Novel Refactoring Techniques. *12th International Symposium on Trends in Functional Programming (TFP), Revised Selected Papers*, pages 82–97, 2012.

[14] R. M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the Association for Computer Machinery*, 24(1):44–67, 1977.

[15] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A Status Report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18. ACM, 2007.

[16] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14. ACM, 2011.

[17] Wei-Ngan Chin. Generalising Deforestation for All First-Order Functional Programs. *In Journées de Travail sur L'Analyse Statique en Programmation Equationnelle,Fonctionnelle et Logique*, pages 173 – 181, 1991.

[18] Wei-Ngan Chin. Synthesizing Efficient Parallel Programs from Sequential Specifications. *Technical Report (TRA9/95), Department of IS/CS, NUS*, 1995.

[19] Wei-Ngan Chin, A. Takano, and Zhenjiang Hu. Parallelization via Context Preservation. *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 153–162, May 1998.

[20] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58:345–363, April 1936.

[21] Alonzo Church. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

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

[23] Murray Cole. List Homomorphic Parallel Algorithms for Bracket Matching. *Technical Report, Department of Computer Science, University of Edinburgh*, 1993.

[24] Murray Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. *PARCO*, pages 489–492, 1993.

[25] John Darlington, A. J. Field, Peter G. Harrison, Paul Kelly, D W N Sharp, Qiang Wu, and R. Lyndon While. Parallel Programming Using Skeleton Functions. *Lecture Notes in Computer Science, PARLE'93, 5th International PARLE Conference on Parallel Architectures and Languages Europe*, 694:146–160, 1993.

[26] Michael Dever. AutoPar: Automating the Parallelization of Functional Programs. *Ph.D. Thesis, Dublin City University*, 2015.

[27] Michael Dever and G. W. Hamilton. Automatically Partitioning Data to Facilitate the Parallelization of Functional Programs. *8th International Andrei Ershov Memorial Conference (PSI)*, 2014.

[28] Michael Dever and G.W. Hamilton. Autopar: Automating the Parallelization of Functional Programs. *Fourth International Valentin Turchin Workshop on Metacomputation (META)*, 2014.

[29] Dirk Dussart, Eddy Bevers, and Karel De Vlaminck. Polyvariant Constructor Specialisation. *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1995.

[30] Johan Enmyren and Christoph W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, pages 5–14, 2010.

[31] Martinus Maria Fokkinga. Law and Order in Algorithmics. *Thesis, University of Twente*, 1992.

[32] Jeremy Gibbons. The Third Homomorphism Theorem. *J. Funct. Program.*, 6(4):657–665, 1996.

[33] Allen Goldberg, Peter Mills, Lars Nyland, Jan Prins, John Reif, and James Riely. Specification and development of parallel algorithms with the Proteus system. *Specification of Parallel Algorithms*, 1994.

[34] Horacio González-Vélez and Mario Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[35] Sergei Gorlatch. Extracting and Implementing List Homomorphisms in Parallel Program Development. *Sci. Comput. Program.*, 33(1):1–27, 1999.

[36] G. W. Hamilton. Higher Order Deforestation. In *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*, PLILP '96, pages 213–227. Springer-Verlag, 1996.

[37] G. W. Hamilton. Distillation: Extracting the Essence of Programs. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 61–70, 2007.

[38] G. W. Hamilton and Neil D. Jones. Distillation with Labelled Transition Systems. *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pages 15–24, 2012.

[39] G.W. Hamilton. Poitín: Distilling Theorems From Conjectures. *Proceedings of The Twelfth Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Electronic Notes in Theoretical Computer Science (ENTCS)*, 2005.

[40] Kevin Hammond, Marco Aldinucci, Christopher Brown, Francesco Cesarini, Marco Danelutto, Horacio González-Vélez, Peter Kilpatrick, Rainer Keller, Michael Rossbory, and Gilad Shainer. The ParaPhrase Project: Parallel Patterns for Adaptive

Heterogeneous Multicore Systems. *10th International Symposium on Formal Methods for Components and Objects (FMCO), Revised Selected Papers*, pages 218–236, 2013.

[41] Kevin Hammond and Greg Michaelson. Research Directions in Parallel Functional Programming. *Springer-Verlag*, 2000.

[42] Z. Hu, M. Takeichi, and H. Iwasaki. Towards polytypic parallel programming. *Technical Report METR 98-09, University of Tokyo*, 1998.

[43] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving Structural Hylomorphisms from Recursive Definitions. *ACM SIGPLAN International Conference on Functional Programming*, 31(6):73–82, June 1996.

[44] Zhenjiang Hu, Masato Takeichi, and Hideya Iwasaki. Diffusion: Calculating Efficient Parallel Programs. *ACM SIGPLAN Workshop On Partial Evaluation And Semantics-Based Program Manipulation (PEPM '99*, pages 85–94, 1999.

[45] Zhenjiang Hu, Tetsuo Yokoyama, and Masato Takeichi. Program Optimizations and Transformations in Calculation Form. *GTTSE*, pages 144–168, 2005.

[46] G. Huet. The Zipper. *The Implementation of Functional Programming*, 1997.

[47] J. Hughes. Why Functional Programming Matters. *Comput. J.*, 32(2):98–107, April 1989.

[48] Hideya Iwasaki and Zhenjiang Hu. A New Parallel Skeleton for General Accumulative Computations. *International Journal of Parallel Programming*, 32(5):389–414, 2004.

[49] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. *Proceedings of a Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, 1985.

[50] M. H. Kabir. Automatic Inductive Theorem Proving and Program Construction Methods Using Program Transformation. *Ph.D. Thesis, Dublin City University, Ireland*, 2007.

[51] M. H. Kabir and G. W. Hamilton. Extending Poitin to Handle Explicit Quantification. *International Workshop on First-Order Theorem Proving*, 2007.

[52] Venkatesh Kannan and G. W. Hamilton. Program Transformation To Identify List-Based Parallel Skeletons. *Fourth International Workshop on Verification and Program Transformation (VPT)*, 2016.

[53] Venkatesh Kannan and G. W. Hamilton. Program Transformation To Identify Parallel Skeletons. *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), revised version under review in International Journal of Parallel Programming (IJPP)*, 2016.

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

[55] Venkatesh Kannan and G.W. Hamilton. Distilling New Data Types. *Fifth International Valentin Turchin Workshop on Metacomputation (META)*, 2016.

[56] Venkatesh Kannan and G.W. Hamilton. Functional Program Transformation for Parallelisation using Skeletons. *9th International Symposium on High-Level Parallel Programming and Applications (HLPP), revised version in International Journal of Parallel Programming (IJPP)*, 2016.

[57] Christopher Brown Francesco Cesarini Marco Daneluto Horacio Gonzalez-Valez Peter Kilpatrick Rainer Keller Michael Rossbory Gilad Shainer Kevin Hammond, Marco Aldinucci. The Paraphrase Project: Parallel Patterns for Adaptive Heterogeneous Muticore Systems. *Formal Methods for Components and Objects*, pages 218–236, 2013.

[58] D.E. Knuth, Morris J.H., and Pratt V.R. Fast Pattern Matching in Strings. *The SIAM Journal on Computing*, 1977.

[59] Lindsey Kuper and Ryan R. Newton. LVars: Lattice-based Data Structures for Deterministic Parallelism. *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, pages 71–84, 2013.

[60] Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 2–14, 2014.

[61] Rita Loogen. Eden – Parallel Functional Programming with Haskell. *Lecture Notes in Computer Science, Central European Functional Programming School, Springer Berlin Heidelberg*, pages 142–206, 2012.

[62] G. Malcolm. Data Structures and Program Transformation. *Sci. Comput. Program.*, 14(2-3):255–279, September 1990.

[63] S. Marlow. Deforestation for Higher-Order Functional Programs. *PhD thesis*, 1996.

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

[65] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs. *SIGPLAN Not.*, 48(9):49–60, September 2013.

[66] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. *Functional Programming Languages and Computer Architecture*, 523:124–144, 1991.

[67] Gary L. Miller and John H. Reif. Parallel Tree Contraction and Its Application. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 478–489. IEEE Computer Society, 1985.

[68] Torben Æ. Mogensen. Constructor Specialization. *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 22–32, 1993.

[69] Torben Ægidius Mogensen. Supercompilation for Datatypes. *Perspectives of System Informatics (PSI)*, 8974:232–247, 2014.

[70] Akimasa Morihata and Kiminori Matsuzaki. A Practical Tree Contraction Algorithm for Parallel Skeletons on Trees of Unbounded Degree. *Procedia Computer Science*, 4:7 – 16, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[71] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. The Third Homomorphism Theorem on Trees: Downward & Upward Lead to Divide-and-Conquer. *POPL*, pages 177–185, 2009.

[72] Y. Onoue, Z. Hu, M. Takeichi, and H. Iwasaki. A Calculational Fusion System HYLO. In *Proceedings of the IFIP TC 2 WG 2.1 International Workshop on Algorithmic Languages and Calculi*, pages 76–106, London, UK, UK, 1997. Chapman & Hall, Ltd.

[73] Will Partain. The nofib Benchmark Suite of Haskell Programs. *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, 1993.

[74] A. Pettorossi, M. Proietti, and R. Dicembre. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28:360–414, 1996.

[75] Simon Peyton Jones. Call-pattern Specialisation for Haskell Programs. *SIGPLAN Not.*, 42(9):327–337, 2007.

[76] Simon Peyton Jones. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138. Springer-Verlag, 2008.

[77] Robert Pointon Hans Wolfgang Loidl Phil Trinder, Jan Henry Nystrom. Glasgow Distributed Haskell (GDH). *http://www.macs.hw.ac.uk/ dsg/gdh/*, 2011.

[78] Norman Scaife, Susumi Horiguchi, Greg Michaelson, and Paul Bristow. A Parallel SML Compiler Based on Algorithmic Skeletons. *Journal of Functional Program.*, 15(4):615–650, July 2005.

[79] D.B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. *Software for Parallel Computation*, 106:120–133, 1993.

[80] M. H. Sörensen, R. Glück, and N. D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6:811–838, 1996.

[81] Morten Heine Sörensen and Robert Glück. An Algorithm of Generalization in Positive Supercompilation. *ILPS*, pages 465–479, 1995.

[82] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182, May 2011.

[83] Akihiko Takano and Erik Meijer. Shortcut Deforestation in Calculational Form. *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, 1995.

[84] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Funct. Program.*, 8(1):23–60, January 1998.

[85] Valentin F. Turchin. The Concept of a Supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, June 1986.

[86] Philip Wadler. Efficient Compilation of Pattern Matching. *The Implementation of Functional Programming Languages, Prentice Hall, International Series in Computer Science*, pages 78–103, 1987.

[87] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73(2):231 – 248, 1990.

[88] Malcolm Watt. Benchmarking a New Parallel Language Implementation. *M.Sc. Thesis, Heriot-Watt University*, 2011.

[89] Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449. IEEE Press, 1981.

[90] Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards. Hardware Synthesis from a Recursive Functional Language. *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 83–93, 2015.

Appendix A

Encoding Transformation for Pattern Specialisation

In this appendix, we discuss the role of the data type transformation proposed in Chapter 5, outside its intended use for program parallelisation, to potentially improve the efficiency of a given program [55].

As discussed in Chapter 2, unfold/fold program transformation has been used to obtain more efficient versions of programs. One of the primary improvements achieved by such transformation techniques is through the elimination of intermediate data structures that are used in a given program, referred to as *fusion* – combining multiple functions in a program into a single function thereby eliminating the intermediate data structure used between them. In particular, the distillation transformation can potentially result in super-linear speedup of the distilled program as mentioned previously.

We also discussed that while such program transformation techniques redefine functions for optimisation, the data types of the programs produced remain unaltered. For instance, the programs produced by the distillation transformation are still defined over the original data types. Thus, another source of inefficiency in a program is the potential mismatch of the structures of the data types in comparison to the algorithmic structure of the program [68]. For instance, consider the simple program defined in Example A.1 which reduces a given list by computing the sum of neighbouring pairs of elements in the list.

Example A.1 (Reduce Neighbouring Pairs):

$reducePairs :: [Int] \rightarrow [Int]$
$reducePairs\ xs$
where
$reducePairs\ [] = []$
$reducePairs\ (x : []) = x : []$
$reducePairs\ (x_1 : x_2 : xs) = (x_1 + x_2) : (reducePairs\ xs)$

Here, we observe that in order to pattern-match a non-empty list, *reducePairs* checks if the tail is non-empty (in which case the second pattern $(x : [])$ is excluded), and then the tail is matched again in the third pattern $(x_1 : x_2 : xs)$. Also, the third pattern is nested to obtain the first two elements x_1 and x_2 in the list. While this pattern is used to obtain the elements that are used in the function body, we observe that the structure of the pattern-matching performed is inefficient and does not match the structure of the *reducePairs* function definition. It desirable to have the input argument structured in such a way that the elements x_1 and x_2 are obtained using a single pattern-match and redundant pattern-matchings are avoided. One such definition of the *reducePairs* function is presented in Example A.2 on a new data type $T_{reducePairs}$.

Example A.2 (Reduce Neighbouring Pairs – Desired Program):

data $T_{reducePairs} = c_1 \mid c_2\ Int \mid c_3\ Int\ Int\ T_{reducePairs}$
$reducePairs\ xs$
where
$reducePairs\ c_1 = []$
$reducePairs\ (c_2\ x) = x : []$
$reducePairs\ (c_3\ x_1\ x_2\ xs) = (x_1 + x_2) : (reducePairs\ xs)$

In this context, we observe that the encoding transformation allows the pattern-matched inputs to be combined in such a way that the structure of the encoded input matches that of the program algorithm. As a result, we evaluated the performance of the encoded programs obtained for the reduce pairs program in Example A.1 and the reduce trees program in Example A.3; their respective encoded programs are shown in Examples A.4 and A.5. Here, we observe that the patterns of the encoded inputs in both encoded programs matches the recursive structures of the programs.

Example A.3 (Reduce Trees):

data $BTree\ a = L$
$\mid\ B\ a\ [BTree\ a]\ [BTree\ a]$

$$
\begin{array}{l}
reduceTrees\ ::\ [BTree\ Int] \rightarrow Int \\
reduceTrees\ ts \\
\textbf{where} \\
\begin{array}{lcl}
reduceTrees\ [] & = & 0 \\
reduceTrees\ (L : xs) & = & reduceTrees\ xs \\
reduceTrees\ \big((B\ x\ lts\ rts) : xs\big) & = & x + (reduceTrees\ lts) + (reduceTrees\ rts) \\
 & & \quad +(reduceTrees\ xs)
\end{array}
\end{array}
$$

Example A.4 (Reduce Neighbouring Pairs – Encoded Program):

$$
\begin{array}{l}
\textbf{data}\ T_{reducePairs}\ a\ =\ c_1 \\
\qquad\qquad\quad \mid\ c_2\ a \\
\qquad\qquad\quad \mid\ c_3\ a\ a\ (T_{reducePairs}\ a) \\
\begin{array}{lcl}
encode_{reducePairs}\ [] & = & c_1 \\
encode_{reducePairs}\ (x : []) & = & c_2\ x \\
encode_{reducePairs}\ (x_1 : x_2 : xs) & = & c_3\ x_1\ x_2\ (encode_{reducePairs}\ xs)
\end{array} \\
reducePairs'\ xs \\
\textbf{where} \\
\begin{array}{lcl}
reducePairs'\ c_1 & = & [] \\
reducePairs'\ (c_2\ x) & = & x : [] \\
reducePairs'\ (c_3\ x_1\ x_2\ xs) & = & (x_1 + x_2) : (reducePairs'\ xs)
\end{array}
\end{array}
$$

Example A.5 (Reduce Trees – Encoded Program):

$$
\begin{array}{l}
\textbf{data}\ T_{reduceTrees}\ a\ =\ c_1 \\
\qquad\qquad\quad \mid\ c_2\ (T_{reduceTrees}\ a) \\
\qquad\qquad\quad \mid\ c_3\ a\ (T_{reduceTrees}\ a)\ (T_{reduceTrees}\ a)\ (T_{reduceTrees}\ a) \\
\begin{array}{lcl}
encode_{reduceTrees}\ [] & = & c_1 \\
encode_{reduceTrees}\ (L : xs) & = & c_2\ (encode_{reduceTrees}\ xs) \\
encode_{reduceTrees}\ \big((B\ x\ lts\ rts) : xs\big) & = & c_3\ x\ (encode_{reduceTrees}\ lts) \\
 & & \quad (encode_{reduceTrees}\ rts) \\
 & & \quad (encode_{reduceTrees}\ xs)
\end{array} \\
reduceTrees'\ ts \\
\textbf{where} \\
\begin{array}{lcl}
reduceTrees'\ c_1 & = & 0 \\
reduceTrees'\ (c_2\ xs) & = & reduceTrees'\ xs \\
reduceTrees'\ (c_3\ x\ lts\ rts\ xs) & = & x + (reduceTrees'\ lts) + (reduceTrees'\ rts) \\
 & & \quad +(reduceTrees'\ xs)
\end{array}
\end{array}
$$

Following this, we compare the execution times of the encoded programs *reducePairs′* and *reduceTrees′* (excluding the times to encode inputs) against those of their distilled versions *reducePairs* and *reduceTrees*, respectively, for different input sizes. An input of size `N` for the *reducePairs* program denotes the length of the input list, and the number of nodes in the input tree for the *reduceTrees* program. The resulting speedups achieved by these encoded programs are illustrated in Figure A.1.

We observe that, as a result of the reduced pattern-matchings performed in the encoded programs, they consume the encoded inputs more efficiently resulting in a speedup

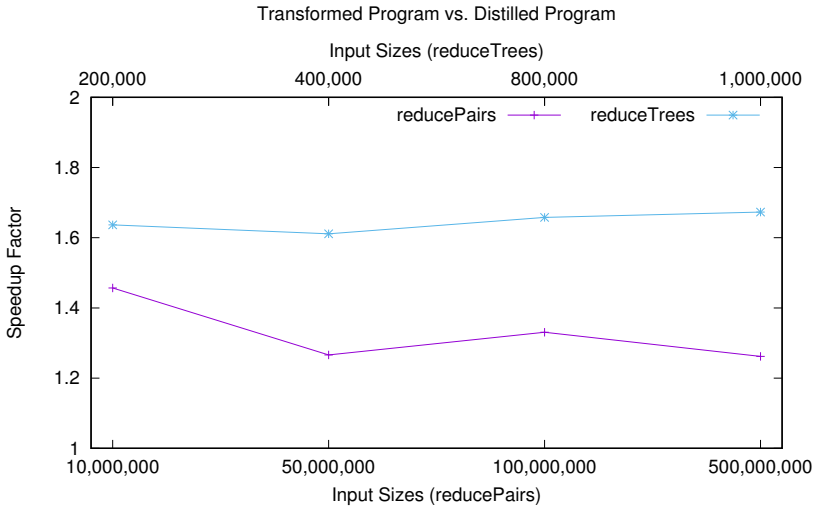


Figure A.1: Speedups of Transformed Programs vs. Distilled Programs

of `1.26x` – `1.67x` for the two examples. Additionally, Figure A.2 illustrates the cost of encoding the inputs (using the $encode_f$ functions) in comparison with the total execution time of the encoded programs.

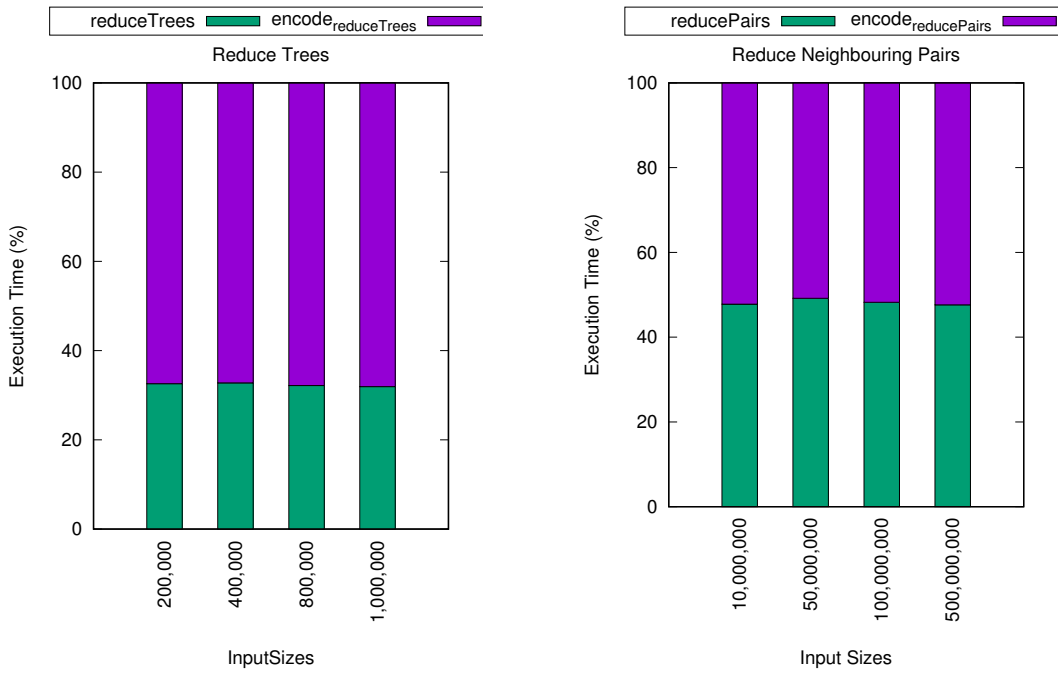


Figure A.2: Cost Centre of Transformed Programs

We observe that the cost of encoding inputs is non-trivial. However, given the relation between the original data type and the new encoded data type, which is defined by the $encode_f$ function, the user can benefit by producing the inputs in the proposed encoded data type and by using the efficient encoded program.

A.1 Related Work

The importance of such data type transformation methods has been discussed in other works such as [68, 29]. Mogensen presented one of the initial ideas in [68] to address data type transformation using constructor specialisation. This method improves the quality of the transformed programs (such as compiled programs) by inventing new data types based on the pattern-matchings performed on the original data types. It is explained that such data type transformation approaches can impact the performance of a program that uses limited data types to encode a larger family of data structures as required by the program.

To improve on Mogensen's work in [68], Dussart et al. proposed a polyvariant constructor specialisation in [29]. The authors highlight that the earlier work by Mogensen was monovariant since each data type, irrespective of how it is dynamically used for pattern-matching in different parts of a program, is statically analysed and transformed. This monovariant design potentially produces dead code in the transformed programs. Dussart et al. improved this by presenting a polyvariant version where a data type is transformed by specialising it based on the context in which it is used. This is achieved in three steps: (1) compute properties for each pattern-matching expression in the program based on its context, (2) specialise the pattern-matching expression using these properties, and (3) generate new data type definitions using the specialisations performed.

More recently, in [69], Mogensen presents supercompilation for data types. Similar to the unfold, fold and special-casing steps used in the supercompilation transformation, the author presents a technique for supercompiling data types using the three steps designed for data types. This technique combines groups of constructor applications in a given program into a single constructor application of a new data type that is created analogous to how supercompilation combines groups of function calls into a single function call. As a result, the number of constructor applications and pattern-matchings in the transformed program are fewer compared to the regular supercompiled programs. What remains to be done in this technique is the design of functions that allow automatic conversion between the original and supercompiled data types. We address this aspect in our proposed transformation technique by providing automatic steps to declare the transformed data type and to define the transformation function.

In [75], Simon Jones presents a method to achieve the same objective of matching

the data types used by a program and the definition of the program. The main difference to this approach is that their transformation specialises each recursive function according to the structure of its arguments. This is achieved by creating a specialised version of the function for each distinct pattern. Following this, the calls to the function are replaced with calls to the appropriate specialised versions. To illustrate this transformation, consider the following definition of function *last*, where the tail of the input list is redundantly checked by the patterns $(x : [])$ and $(x : xs)$.

$$\begin{array}{lcl} last\ [] & = & error\ \text{“}last\text{”} \\ last\ (x : []) & = & x \\ last\ (x : xs) & = & last\ xs \end{array}$$

Such a definition is transformed by creating a specialised version of the *last* function based on the patterns for the list tail, resulting in the following definition for the *last* function which avoids redundant pattern-matching.

$$\begin{array}{lcl} last\ [] & = & error\ \text{“}last\text{”} \\ last\ (x : xs) & = & last'\ x\ xs \\ & & \textbf{where} \\ & & last'\ x\ [] = x \\ & & last'\ x\ (y : ys) = last'\ y\ ys \end{array}$$

This transformation was implemented as a part of the Glasgow Haskell Compiler for evaluation and results in an average run-time improvement of 10%.

Appendix B

Execution Times of Benchmark Programs

B.1 Matrix Multiplication Execution Times

Input Size	OP total (s)	DP total (s)
100x100	0.554	0.331
250x250	75.042	17.305
1000x100	53.434	52.031
100x1000	115.052	4.573

	100x100		250x250	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	0.331	0.554	17.305	75.042
2	0.253	0.268	9.816	46.216
3	0.181	0.191	7.067	31.947
4	0.146	0.150	5.569	25.045
5	0.113	0.128	4.725	20.978
6	0.096	0.104	3.946	17.839
7	0.087	0.093	3.409	15.331
8	0.077	0.081	3.02	13.631
9	0.071	0.076	2.609	11.987
10	0.066	0.069	2.476	11.043
11	0.062	0.066	2.314	10.020
12	0.058	0.062	2.218	9.245

	1000x100		100x1000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	52.031	53.434	4.573	115.052
2	37.469	38.854	2.474	89.544
3	25.756	28.545	1.722	61.275
4	20.176	22.964	1.367	46.444
5	16.851	20.408	1.125	37.492
6	14.175	16.631	1.065	31.396
7	12.636	14.664	1.056	26.973
8	11.532	14.293	1.041	22.873
9	11.116	18.263	1.048	20.458
10	10.691	11.217	1.017	16.893
11	11.047	10.204	1.060	16.178
12	10.538	9.632	1.097	14.316

B.2 Power Tree Execution Times

Input Size	OP/DP total (s)
1,000,000	0.975
5,000,000	5.516
10,000,000	11.582
50,000,000	62.849

	1,000,000		5,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	0.975	0.975	5.516	5.516
2	0.813	0.742	4.836	4.321
3	0.677	0.633	3.690	3.633
4	0.622	0.605	3.450	3.395
5	0.528	0.538	3.326	3.104
6	0.505	0.548	3.541	2.910
7	0.481	0.457	2.703	2.699
8	0.445	0.420	2.669	2.717
9	0.371	0.366	2.391	2.534
10	0.378	0.383	2.228	2.107
11	0.401	0.401	2.618	2.279
12	0.387	0.394	2.247	2.232

	10,000,000		50,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	11.582	11.582	62.849	62.849
2	9.531	8.783	49.289	48.425
3	8.335	7.960	42.494	39.534
4	7.234	6.946	38.669	37.865
5	6.785	6.863	34.655	37.476
6	6.628	6.288	37.607	33.868
7	6.632	5.420	35.152	32.420
8	5.366	5.816	32.685	29.322
9	5.874	5.282	27.629	27.061
10	5.736	4.838	31.559	26.400
11	5.879	4.504	24.687	27.433
12	5.136	4.757	27.855	27.750

B.3 Dot Product of Binary Trees Execution Times

Input Size	OP/DP total (s)
1,000,000	1.083
10,000,000	12.706
50,000,000	69.683
100,000,000	147.014

	1,000,000		10,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	1.083	1.083	12.706	12.706
2	0.737	0.843	8.522	9.519
3	0.633	0.672	7.507	8.297
4	0.629	0.618	7.19	7.312
5	0.609	0.601	7.451	7.638
6	0.565	0.505	6.637	7.345
7	0.482	0.483	5.995	6.175
8	0.458	0.455	5.998	6.150
9	0.428	0.410	6.25	5.594
10	0.401	0.497	6.016	6.326
11	0.414	0.371	5.243	6.023
12	0.441	0.403	6.185	4.917

	50,000,000		100,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	69.683	69.683	147.014	147.014
2	47.207	52.884	99.109	108.454
3	42.614	44.247	93.824	92.365
4	39.672	39.079	82.693	83.676
5	40.147	39.929	80.323	83.486
6	37.530	35.339	81.174	81.019
7	34.877	35.900	70.944	73.868
8	29.995	32.565	65.169	63.984
9	30.033	30.894	70.082	65.554
10	29.347	30.126	60.649	69.911
11	28.804	32.611	58.889	66.826
12	28.672	29.331	58.153	65.422

B.4 Totient Execution Times

Input Size	OP/DP total (s)
1,000,000	0.783
5,000,000	4.108
10,000,000	8.59
500,000,000	44.904

	1,000,000		5,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	0.783	0.783	4.108	4.108
2	0.713	0.523	3.548	2.726
3	0.625	0.429	3.12	2.227
4	0.58	0.39	2.956	2.009
5	0.557	0.372	2.886	1.89
6	0.553	0.376	2.821	1.805
7	0.516	0.34	2.774	1.792
8	0.507	0.329	2.744	1.688
9	0.497	0.322	2.693	1.658
10	0.494	0.325	2.67	1.628
11	0.49	0.321	2.632	1.597
12	0.485	0.312	2.616	1.597

	10,000,000		50,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	8.59	8.59	44.904	44.904
2	7.308	5.39	38.365	27.91
3	6.363	4.487	32.323	22.951
4	6.03	4.036	29.889	20.614
5	5.833	3.809	29.309	19.708
6	5.738	3.656	28.415	19.448
7	5.575	3.494	27.302	17.763
8	5.482	3.357	26.662	17.138
9	5.413	3.316	26.309	16.832
10	5.35	3.242	25.926	16.917
11	5.325	3.153	25.628	16.309
12	5.286	3.152	25.597	16.048

B.5 Maximum Prefix Sum Execution Times

Input Size	OP total (s)	DP total (s)
5,000,000	2.898	1.273
10,000,000	5.676	2.945
50,000,000	32.513	13.589
100,000,000	65.206	28.011

	5,000,000		10,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	3.048	2.898	5.846	5.676
2	1.358	1.364	2.653	2.674
3	1.082	1.019	2.137	2.112
4	1.02	0.921	2.003	1.832
5	0.98	0.877	1.876	1.809
6	0.895	0.818	1.744	1.702
7	0.848	0.757	1.688	1.539
8	0.805	0.729	1.632	1.459
9	0.795	0.713	1.579	1.391
10	0.768	0.699	1.535	1.401
11	0.753	0.645	1.493	1.31
12	0.746	0.631	1.469	1.264

	50,000,000		100,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	33.813	32.513	68.466	65.206
2	13.108	12.979	27.379	24.516
3	11.462	10.975	22.327	21.657
4	9.501	9.158	18.904	17.541
5	9.03	8.733	18.777	16.697
6	8.824	8.029	17.739	15.742
7	8.565	7.685	17.573	15.379
8	8.172	7.134	15.647	14.192
9	7.651	6.889	15.45	13.567
10	7.537	6.738	14.899	13.514
11	7.62	6.654	15.108	13.424
12	7.539	6.524	14.967	13.264

B.6 Sum Squares of List Execution Times

Input Size	OP total (s)	DP total (s)
1,000,000	0.421	0.167
5,000,000	2.034	0.839
10,000,000	4.157	1.688
50,000,000	22.910	8.114

	1,000,000		5,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	0.167	0.421	0.339	2.034
2	0.224	0.281	1.089	1.449
3	0.225	0.266	1.082	1.364
4	0.225	0.261	1.090	1.335
5	0.225	0.261	1.083	1.341
6	0.227	0.265	1.089	1.344
7	0.227	0.264	1.088	1.337
8	0.227	0.264	1.090	1.345
9	0.228	0.264	1.090	1.342
10	0.228	0.265	1.092	1.339
11	0.229	0.265	1.091	1.342
12	0.229	0.269	1.091	1.367

	10,000,000		50,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	1.688	4.157	8.114	22.910
2	2.181	2.810	11.773	14.069
3	2.182	2.709	11.769	13.493
4	2.185	2.655	11.749	13.216
5	2.173	2.722	12.415	13.598
6	2.178	2.712	11.795	13.515
7	2.182	2.679	11.785	13.454
8	2.185	2.643	11.775	13.325
9	2.184	2.682	11.797	13.443
10	2.187	2.653	11.793	13.382
11	2.190	2.663	11.800	13.461
12	2.193	2.667	11.762	13.500

B.7 Fibonacci Series Sum Execution Times

Input Size	OP total (s)	DP total (s)
40	4.215	4.536
42	11.046	11.83
44	28.885	31.214
46	75.597	81.309

	40		42	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	4.536	4.215	11.83	11.046
2	5.264	3.896	13.785	10.164
3	3.387	2.487	8.887	6.555
4	2.72	1.805	7.339	5.149
5	2.328	1.387	6.167	3.906
6	2.375	1.296	6.473	3.277
7	2.502	1.048	6.525	2.776
8	2.507	0.925	6.665	2.358
9	2.565	0.826	6.771	2.169
10	2.636	0.746	6.932	1.94
11	2.639	0.677	6.944	1.753
12	2.806	0.627	5.862	1.623

	44		46	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	31.214	28.885	81.309	75.597
2	35.863	26.992	94.163	69.77
3	23.283	16.984	60.903	44.854
4	18.904	13.525	49.598	35.642
5	15.782	10.262	42.056	26.873
6	16.445	8.607	43.419	23.329
7	17.188	7.203	43.837	18.854
8	17.421	6.348	45.679	16.699
9	17.587	5.604	46.502	14.8
10	17.748	5.068	47.5	13.392
11	18.185	4.657	48.067	12.243
12	19.226	4.267	50.242	11.181

B.8 Sum Append of Lists Execution Times

Input Size	OP total (s)	DP total (s)
1,000,000	0.606	0.327
5,000,000	2.952	1.603
10,000,000	6.016	3.286
50,000,000	33.265	15.801

	1,000,000		5,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	0.327	0.606	1.603	2.952
2	0.775	0.558	3.946	2.814
3	0.745	0.516	3.856	2.638
4	0.737	0.528	3.786	2.575
5	0.758	0.536	3.872	2.662
6	0.753	0.530	3.916	2.662
7	0.751	0.526	3.933	2.614
8	0.749	0.529	3.890	2.601
9	0.761	0.538	3.982	2.669
10	0.749	0.527	3.908	2.615
11	0.744	0.535	3.972	2.636
12	0.744	0.537	3.955	2.656

	10,000,000		50,000,000	
cores	EPP total (s)	HPP total (s)	EPP total (s)	HPP total (s)
1	3.286	6.010	41.700	33.265
2	7.938	5.679	41.559	28.536
3	7.744	5.258	39.399	26.858
4	7.702	5.313	39.531	26.510
5	7.821	5.343	39.842	26.987
6	7.839	5.344	39.217	27.346
7	7.909	5.204	39.463	26.897
8	7.919	5.258	39.545	27.002
9	7.863	5.296	39.372	26.745
10	7.890	5.357	39.310	26.810
11	7.874	5.423	39.332	26.732
12	8.015	5.498	39.270	26.814

Appendix C

Parallel Skeleton Implementations

Example C.1 (ZipWith-Reduce):

$$\begin{aligned}&\mathit{zipWithRedr} :: (c \to d \to d) \to d \to (a \to b \to c) \to [a] \to [b] \to d\\&\mathit{zipWithRedr}\ g\ v\ f\ xs\ ys = \mathit{foldr}\ g\ v\ (\mathit{zipWith}\ f\ xs\ ys)\end{aligned}$$

Example C.2 (ZipWith-Reduce on Non-Empty List):

$$\begin{aligned}&\mathit{zipWithRedr1} :: (c \to d \to d) \to (a \to b \to c) \to [a] \to [b] \to d\\&\mathit{zipWithRedr1}\ g\ f\ xs\ ys = \mathit{foldr1}\ g\ (\mathit{zipWith}\ f\ xs\ ys)\end{aligned}$$

Example C.3 (Parallel ZipWith-Reduce):

$$\begin{aligned}&\mathit{parZipWithRedr} :: (\mathit{Trans}\ a,\ \mathit{Trans}\ b,\ \mathit{Trans}\ c) \Rightarrow\\&\qquad (c \to c \to c) \to c \to (a \to b \to c) \to [a] \to [b] \to c\\&\mathit{parZipWithRedr}\ g\ v\ f\ xs\ ys =\\&\quad \begin{cases} h\ (\mathit{noPe} == 1)\\ \textbf{where}\\ h\ \mathit{True} = \mathit{zipWithRedr}\ g\ v\ f\ xs\ ys\\ h\ \mathit{False} = \mathit{foldr}\ g\ v\ \Big(\mathit{parMap}\ \big(\lambda xs'.\lambda ys'.\mathit{zipWithRedr}\ g\ v\ f\ xs'\ ys'\big)\\ \qquad\qquad \big(\mathit{zip}\ (\mathit{splitIntoN}\ \mathit{noPe}\ xs)\ (\mathit{splitIntoN}\ \mathit{noPe}\ ys)\big)\Big) \end{cases}\end{aligned}$$

Example C.4 (Parallel ZipWith-Reduce on Non-Empty List):

$$\begin{aligned}&\mathit{parZipWithRedr1} :: (\mathit{Trans}\ a,\ \mathit{Trans}\ b,\ \mathit{Trans}\ c) \Rightarrow\\&\qquad (c \to c \to c) \to (a \to b \to c) \to [a] \to [b] \to c\\&\mathit{parZipWithRedr1}\ g\ f\ xs\ ys =\\&\quad \begin{cases} h\ (\mathit{noPe} == 1)\\ \textbf{where}\\ h\ \mathit{True} = \mathit{zipWithRedr1}\ g\ f\ xs\ ys\\ h\ \mathit{False} = \mathit{foldr1}\ g\ \Big(\mathit{parMap}\ \big(\lambda(xs', ys').\mathit{zipWithRedr1}\ g\ f\ xs'\ ys'\big)\\ \qquad\qquad \big(\mathit{zip}\ (\mathit{splitIntoN}\ \mathit{noPe}\ xs)\ (\mathit{splitIntoN}\ \mathit{noPe}\ ys)\big)\Big) \end{cases}\end{aligned}$$

Example C.5 (Parallel ZipWith):

$$
\begin{array}{lll}
\mathit{parZipWith} & :: & (\mathit{Trans}\ a,\ \mathit{Trans}\ b,\ \mathit{Trans}\ c) \\
 & & (a \to b \to c) \to [a] \to [b] \to [c] \\
\mathit{parZipWith}\ f\ xs\ ys\ = & & \\
\quad \left\{ \begin{array}{l}
h\ (\mathit{noPe} == 1) \\
\textbf{where} \\
h\ \mathit{True}\ = \mathit{zipWith}\ f\ xs\ ys \\
h\ \mathit{False} = \mathit{unSplit}\ \circ\ \mathit{parMap}\ \big(\lambda(xs', ys').\mathit{zipWith}\ f\ xs'\ ys'\big) \\
\qquad\qquad\qquad\qquad \big(\mathit{zip}\ (\mathit{splitIntoN}\ \mathit{noPe}\ xs)\ (\mathit{splitIntoN}\ \mathit{noPe}\ ys)\big)
\end{array} \right.
\end{array}
$$

Example C.6 (Parallel Scan):

$$
\begin{array}{lll}
\mathit{parScan} & :: & (\mathit{Trans}\ a,\ \mathit{Trans}\ b) \Rightarrow \\
 & & (b \to b \to b) \to (a \to b) \to b \to [a] \to [b] \\
\mathit{parScan}\ g\ f\ v\ xs\ = & & \textbf{let}\ (ds, zs) = \mathit{unzip}\ \big(\mathit{parMap}\ (\mathit{scanRes}\ g\ v\ f)\ (\mathit{splitInto}\ \mathit{noPe}\ xs)\big) \\
 & & \quad zs' = \mathit{init}\ (\mathit{scanl}\ g\ v\ zs) \\
 & & \textbf{in}\ \mathit{unSplit}\ (\mathit{parZipWith}\ (\mathit{map}\ \circ\ g)\ zs'\ ds) \\
\mathit{scanRes} & :: & (\mathit{Trans}\ a,\ \mathit{Trans}\ b) \Rightarrow \\
 & & (b \to b \to b) \to b \to (a \to b) \to [a] \to ([b], b) \\
\mathit{scanRes}\ g\ v\ f\ xs\ = & & \textbf{let}\ ys = \mathit{scanl}\ (\lambda v.\lambda x.g\ v\ (f\ x))\ v\ xs \\
 & & \textbf{in}\ (\mathit{init}\ ys, \mathit{last}\ ys)
\end{array}
$$