Distilling New Data Types

Venkatesh Kannan and G. W. Hamilton

School of Computing, Dublin City University, Ireland
{vkannan, hamilton}@computing.dcu.ie

Abstract. Program transformation techniques are commonly used to improve the efficiency of programs. While many transformation techniques aim to remove inefficiencies in the algorithms used in a program, another source of inefficiency is the use of inappropriate datatypes whose structures do not match the algorithmic structure of the program. This mismatch will potentially result in inefficient consumption of the input by the program. Previously, Mogensen has shown how techniques similar to those used in supercompilation can be used to transform datatypes, but this was not fully automatic. In this paper, we present a fully automatic datatype transformation technique which can be applied in conjunction with distillation. The objective of the datatype transformation is to transform the original datatypes in a program so that the resulting structure matches the algorithmic structure of the distilled program. Consequently, the resulting transformed program potentially uses less pattern matching and as a result is more efficient than the original program.

1 Introduction

Fold/unfold 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. Transformation techniques such as supercompilation [9, 10] and distillation [2] are based on the unfold/fold transformation framework and achieve such improvements. In particular, the distillation transformation can potentially result in super-linear speedup of the distilled program.

While unfold/fold program transformation redefines functions for optimisation, the data types of the programs produced remain unaltered. For instance, we observe that 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 [5].

For instance, consider the simple program defined in Example 1 which reduces a given list by computing the sum of neighbouring pairs of elements in the list.

Example 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 2 on a new data type $T_{reducePairs}$.

Example 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 [6], Mogensen proposed one of the methods to address these issues by creating data types that suit the structure of programs based on the supercompilation transformation [10, 11]. The resulting transformed programs use fewer constructor applications and pattern-matchings. However, the transformation remains to be automated because functions that allow conversion between the original and new data types were not provided.

In this paper, we present a data type transformation technique to automatically define a new data type by transforming the original data types of a program. The new transformed data type is defined in such a way that its structure matches the algorithmic structure of the program. As a result, the transformed input argument is consumed in a more efficient fashion by the transformed program.

The proposed transformation is performed using the following two steps:

1. Apply the distillation transformation on a given program to obtain the distilled program. (Section 3)
2. Apply the proposed data type transformation on a distilled program to obtain the transformed program. (Section 4)

In Section 5, we demonstrate the proposed transformation with examples and present the results of evaluating the transformed programs. In Section 6, we discuss the merits and applications of the proposed transformation along with related work.

2 Language

The higher-order functional language used in this work is shown in Definition 1.

Definition 1 (Language Grammar).

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

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

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

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

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

3 Distillation

Given a program in the language from Definition 1, *distillation* [2] is a technique that transforms the program to remove intermediate data structures and yields a *distilled program*. It is an unfold/fold-based transformation that makes use of well-known transformation steps – unfold, generalise and fold [7] – and can potentially provide super-linear speedups to programs.

The syntax of a distilled program $de^{\{\}}$ is shown in Definition 3. Here, ρ is the set of variables introduced by **let**–expressions that are created during generalisation. These bound variables of **let**-expressions are not decomposed by pattern-matching in a distilled program. Consequently, $de^{\{\}}$ is an expression that has fewer intermediate data structures.

Definition 3 (Distilled Form Grammar).

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

4 Data Type Transformation

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

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

Definition 4 (General Form of Function in Distilled Program).

$$\begin{array}{lcl} f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N & & \\ \textbf{\textit{where}} & & \\ f\ p^1_1 \ldots p^1_M\ x_{(M+1)} \ldots x_N & = & e_1 \\ \vdots & & \vdots \\ f\ p^K_1 \ldots p^K_M\ x_{(M+1)} \ldots x_N & = & e_K \end{array}$$

The three steps to transform the pattern-matched arguments of function f are as follows:

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

Definition 5 (New Data Type T_f).

$$\begin{array}{lcl} \textbf{data}\ T_f\ \alpha_1 \ldots \alpha_G & ::= & c_1\ T^1_1 \ldots T^1_L\ (T_f\ \alpha_1 \ldots \alpha_G)^1_1 \ldots (T_f\ \alpha_1 \ldots \alpha_G)^1_J \\ & \vdots & \\ & | & c_K\ T^K_1 \ldots T^K_L\ (T_f\ \alpha_1 \ldots \alpha_G)^K_1 \ldots (T_f\ \alpha_1 \ldots \alpha_G)^K_J \end{array}$$

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

$\forall k \in \{1, \ldots, K\} \cdot$
c_k *is a fresh constructor for* T_f *corresponding to* $p^k_1 \ldots p^k_M$ *of the pattern-matched arguments.*

$$f\ p^k_1 \ldots p^k_M\ x_{(M+1)} \ldots x_N = E_k \begin{bmatrix} f\ x^1_1 \ldots x^1_M\ x^1_{(M+1)} \ldots x^1_N, \\ \ldots\ , \\ f\ x^J_1 \ldots x^J_M\ x^J_{(M+1)} \ldots x^J_N \end{bmatrix}$$

$$\{(z_1 :: T^k_1), \ldots, (z_L :: T^k_L)\} = fv(E_k) \setminus \{x_{(M+1)}, \ldots, x_N\}$$

Here, a new constructor c_k of the type T_f is created for each set $p^k_1 \ldots p^k_M$ of the pattern-matched inputs $x_1 \ldots x_M$ of function f that are encoded. As stated above, our objective is to transform the arguments of function f into a new type whose structure reflects the recursive structure of f. To achieve this, the components bound by constructor c_k correspond to the variables in $p^k_1 \ldots p^k_M$ that occur in the context E_k and the transformed arguments of the recursive calls to function f.

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

Definition 6 (Definition of Function $\textit{encode}_f$).

$$
\begin{array}{l}
\textit{encode}_f\ x_1 \ldots x_M \\
\textbf{\textit{where}} \\
\begin{array}{lcl}
\textit{encode}_f\ p_1^1 \ldots p_M^1 & = & e_1' \\
\vdots & & \vdots \\
\textit{encode}_f\ p_1^K \ldots p_M^K & = & e_K'
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textit{where} \\
\forall k \in \{1, \ldots, K\} \cdot \\
e_k' = c_k\ z_1^k \ldots z_L^k\ (\textit{encode}_f\ x_1^1 \ldots x_M^1) \ldots (\textit{encode}_f\ x_1^J \ldots x_M^J) \\
\{z_1^k, \ldots, z_L^k\} = fv(E_k) \setminus \{x_{(M+1)}, \ldots, x_N\} \\
f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots\ , \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}
\end{array}
$$

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

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

Definition 7 (Definition of Transformed Function Over Transformed Argument).

$$
\begin{array}{l}
f'\ x\ x_{(M+1)} \ldots x_N \\
\textbf{\textit{where}} \\
\begin{array}{ll}
f'\ \left(c_1\ z_1^1 \ldots z_L^1\ x_1^1 \ldots x_1^J\right)\ x_{(M+1)} \ldots x_N & = e_1' \\
\vdots & \vdots \\
f'\ \left(c_K\ z_1^K \ldots z_L^K\ x_K^1 \ldots x_K^J\right)\ x_{(M+1)} \ldots x_N & = e_K'
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textit{where} \\
\forall k \in \{1, \ldots, K\} \cdot \\
e_k' \ = \ E_k \left[f'\ x_k^1\ x_{(M+1)}^1 \ldots x_N^1,\ \ldots,\ f'\ x_k^J\ x_{(M+1)}^J \ldots x_N^J\right] \\
f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots\ , \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}
\end{array}
$$

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

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

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

4.1 Correctness

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

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

Proof:
The proof is by structural induction over the transformed data type T_f.

Base Case:
For the transformed argument $x_k = c_k\ z_1^k \ldots z_L^k$ computed by $encode_f\ p_1^k \ldots p_M^k$,

1. By Definition 4, L.H.S. evaluates to e_k.
2. By Definition 7, R.H.S. evaluates to e_k.

Inductive Case:
For the transformed argument $x_k = c_k\ z_1^k \ldots z_L^k\ x^{1_k} \ldots x^{J_k}$ which is computed by $encode_f\ p_1^k \ldots p_M^k$,

1. By Definition 4, 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 7, 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, $\big(f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N\big) = \big(f'\ x\ x_{(M+1)} \ldots x_N\big)$.

□

5 Examples

We demonstrate and evaluate the data type transformation presented in this paper using two simple examples, including the program introduced in Example 1, which are discussed in this section.

5.1 Reduce Neighbouring Pairs

The *reducePairs* program presented in Example 1 does not use any intermediate data structures. Consequently, the result of applying the distillation transformation yields the same program. Following this, Example 3 presents the transformed data type ($T_{reducePairs}$), the transformation function ($encode_{reducePairs}$) and the transformed program (*reducePairs′*) obtained for the *reducePairs* program in Example 1.

Example 3 (Reduce Neighbouring Pairs – Transformed Program).

$$\begin{array}{lll}\textbf{data}\ T_{reducePairs}\ a & ::= & c_1 \\ & \mid & c_2\ a \\ & \mid & c_3\ a\ a\ (T_{reducePairs}\ a)\end{array}$$

$$\begin{array}{lll}encode_{reducePairs}\ [] & = & c_1 \\ encode_{reducePairs}\ (x : []) & = & c_2\ x \\ encode_{reducePairs}\ (x_1 : x_2 : xs) & = & c_3\ x_1\ x_2\ (encode_{reducePairs}\ xs)\end{array}$$

$$\begin{array}{lll}reducePairs'\ xs & & \\ \textbf{where} & & \\ reducePairs'\ c_1 & = & [] \\ reducePairs'\ (c_2\ x) & = & x : [] \\ reducePairs'\ (c_3\ x_1\ x_2\ xs) & = & (x_1 + x_2) : (reducePairs'\ xs)\end{array}$$

5.2 Reduce Trees

To demonstrate our data type transformation, we present another program in Example 4 that performs a reduction over a list of binary trees. Since this definition does not contain intermediate data structures, the result of applying the distillation transformation is the same program.

Example 4 (Reduce Trees – Original/Distilled Program).

$$\begin{array}{lll}\textbf{data}\ BTree\ a & ::= & L \\ & \mid & B\ a\ [BTree\ a]\ [BTree\ a]\end{array}$$

$$reduceTrees\ ::\ [BTree\ Int] \rightarrow Int$$

$$\begin{array}{lll}reduceTrees\ ts & & \\ \textbf{where} & & \\ reduceTrees\ [] & = & 0 \\ reduceTrees\ (L : xs) & = & reduceTrees\ xs \\ reduceTrees\ \big((B\ x\ lts\ rts) : xs\big) & = & x + (reduceTrees\ lts) \\ & & \quad +(reduceTrees\ rts) + (reduceTrees\ xs)\end{array}$$

Example 5 presents the transformed data type ($T_{reduceTrees}$), the transformation function ($encode_{reduceTrees}$) and the transformed program (*reduceTrees′*) obtained for the distilled *reduceTrees* program using the proposed transformation.

Example 5 (Reduce Trees – Transformed Program).

$$\begin{array}{lll} \textbf{data}\ T_{reduceTrees}\ a & ::= & c_1 \\ & \mid & c_2\ (T_{reduceTrees}\ a) \\ & \mid & c_3\ a\ (T_{reduceTrees}\ a)\ (T_{reduceTrees}\ a)\ (T_{reduceTrees}\ a) \end{array}$$

$$\begin{array}{lll} encode_{reduceTrees}\ [] & = & c_1 \\ encode_{reduceTrees}\ (L : xs) & = & c_2\ (encode_{reduceTrees}\ xs) \\ encode_{reduceTrees}\ \big((B\ x\ lts\ rts) : xs\big) & = & c_3\ x\ (encode_{reduceTrees}\ lts) \\ & & \quad (encode_{reduceTrees}\ rts) \\ & & \quad (encode_{reduceTrees}\ xs) \end{array}$$

$$\begin{array}{lll} reduceTrees'\ ts & & \\ \textbf{where} & & \\ reduceTrees'\ c_1 & = & 0 \\ reduceTrees'\ (c_2\ xs) & = & reduceTrees'\ xs \\ reduceTrees'\ (c_3\ x\ lts\ rts\ xs) & = & x + (reduceTrees'\ lts) \\ & & \quad +(reduceTrees'\ rts) + (reduceTrees'\ xs) \end{array}$$

5.3 Evaluation

For the two example programs presented in this section, we compare the execution times of the transformed functions *reducePairs′* and *reduceTrees′* against those of their original versions *reducePairs* and *reduceTrees*, respectively, for different input sizes. The resulting speedups achieved by these transformed programs are illustrated in Figure 1. Here, the input sizes for the *reduceTrees* program are the number of values that are present in the input tree that is reduced.

We observe that, as a result of the reduced pattern-matchings performed in the transformed programs, the transformed functions consume the transformed arguments more efficiently resulting in a speedup of `1.26x` – `1.67x` for the two examples evaluated in this section.

Additionally, Figure 2 illustrates the cost of transforming the arguments (using the $encode_f$ functions) in comparison with the total execution time of the transformed program.

We observe that the cost of transforming the arguments is non-trivial. However, given the relation between the original data type and the new transformed data type, which is defined by the $encode_f$ function, the user can benefit by producing the inputs in the proposed transformed data type and by using the efficient transformed program.

6 Conclusion

6.1 Summary

The data type transformation presented in this paper allows us to modify the program data types into a structure that reflects the structure of the program in distilled form. This is achieved by combining the original pattern-matched arguments of each function in the distilled program. The transformation combines

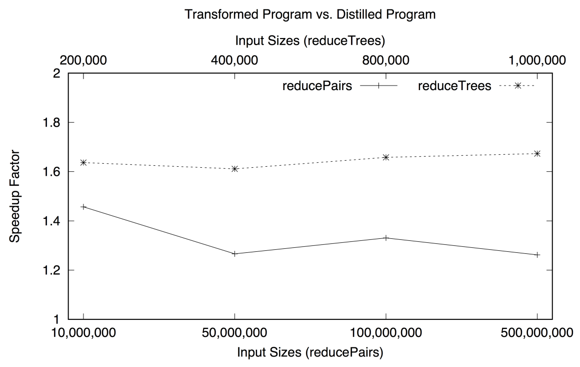

Fig. 1. Speedups of Transformed Programs vs. Distilled Programs

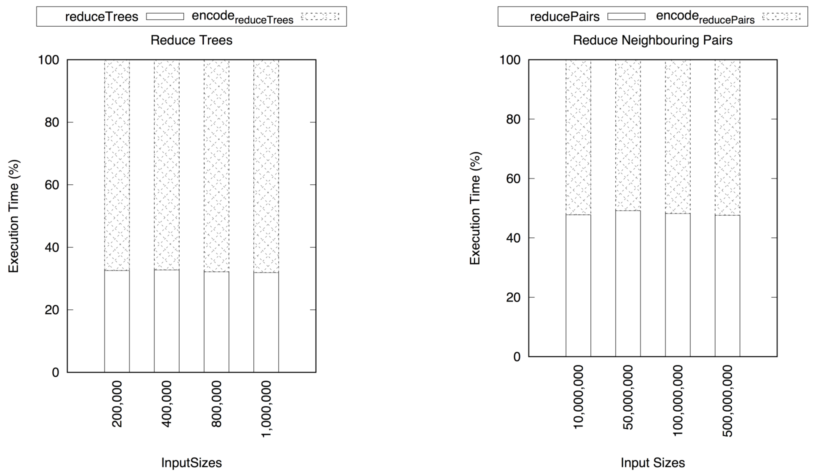

Fig. 2. Cost Centre of Transformed Programs

groups of patterns that are matched with the arguments into a single pattern for the transformed argument. By using the transformation function ($encode_f$) that specifies the correspondence between the original data types and the transformed data type, the user can produce a transformed argument which requires less pattern-matching. Consequently, the transformed program can potentially consume the input data in an optimised fashion.

Furthermore, this data type transformation can also be used to facilitate automatic parallelisation of a given program. By defining algorithmic skeletons that operate over the newly defined transformed data type, we can identify instances of the skeletons – polytypic [4] and list-based [3] – in the transformed program. Following this, we can use efficient parallel implementations of the skeletons to execute the transformed program on parallel hardware.

6.2 Related Work

The importance of such data type transformation methods has been discussed in other works such as [1,5]. Creating specialised data types that suit the structure of a program can provide flexibility to statically typed languages that is similar to dynamically typed languages.

Mogensen presented one of the initial ideas in [5] 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 [5], Dussart et al. proposed a polyvariant constructor specialisation in [1]. 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 [6], 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 [8], 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.

$$\begin{array}{lcl} last\ [] & = & error\ \text{``}last\text{''} \\ last\ (x : xs) & = & last'\ x\ xs \\ & & \textbf{where} \\ & & \begin{array}{lcl} last'\ x\ [] & = & x \\ last'\ x\ (y : ys) & = & last'\ y\ ys \end{array} \end{array}$$

This transformation was implemented as a part of the Glasgow Haskell Compiler for evaluation and results in an average run-time improvement of 10%.

Acknowledgement

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

References

[1] 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.

[2] 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*, 2012.

[3] Venkatesh Kannan and G. W. Hamilton. Program Transformation to Identify List-Based Parallel Skeletons. *4th International Workshop on Verification and Program Transformation (VPT)*, 2016.

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

[5] Torben Æ. Mogensen. Constructor specialization. *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 22–32, 1993.

[6] Torben Æ. Mogensen. Supercompilation for datatypes. *Perspectives of System Informatics (PSI)*, 8974:232–247, 2014.

[7] A. Pettorossi, M. Proietti, and R. Dicembre. Rules And Strategies For Transforming Functional And Logic Programs. *ACM Computing Surveys*, 1996.

[8] Simon Peyton Jones. Call-pattern specialisation for haskell programs. *SIGPLAN Not.*, 42(9):327–337, 2007.

[9] M. H. Sørensen, R. Glück, and N. D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 1996.

[10] Valentin F. Turchin. A Supercompiler System Based on the Language Refal. *SIGPLAN Notices*, 1979.

[11] Valentin F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 1986.