Data Movement Optimisation in Point-Free Form

Brad Alexander and Andrew Wendelborn

School of Computer Science
University of Adelaide
5005, Adelaide, SA, Australia
{brad,andrew}@cs.adelaide.edu.au

Abstract. Programs written in point-free form express computation purely in terms of functions. Such programs are especially amenable to local transformation. In this paper, we describe a process for optimising the transport of data through point-free programs. This process systematically applies local transformations to achieve effective global optimisation. We describe the strategies we employ to ensure this process is tractable. This process has been implemented as an intermediate stage of a compiler. The optimiser is shown to be highly effective, producing code of comparable efficiency to hand-written code.

1 Introduction

Transformation is the key to any program improvement process. By using highly transformable programming notations we pave the way for the application of deep and pervasive transformation techniques. Programs written in point-free form are particularly amenable to transformation[5]. In point-free code all computation is expressed purely in terms of functions. Point-free code contains no variables to store values generated during program execution. As a consequence, functions cannot rely on variables as agents to transmit data through the program. Instead, the functions themselves are responsible for routing data through the code. This exposed data-transport maps well to distributed-memory platforms and there have been a number of experiments mapping functions found in point-free form to such platforms[18, 8, 6].

As well as providing a path to distributed implementation, exposing the transport of data also provides scope for direct optimisation of this transport. This avenue of research is less well explored. In this paper we outline an automated process to reduce the volume of data movement through point-free code through the systematic use of local transformations. We show that this process is highly effective and describe the techniques we found useful.

1.1 Outline of This Paper

The next section outlines the context in which our optimisation process takes place. Section 3 defines the broad strategies we applied in all parts of our optimisation process. Section 4 focuses on one part of the optimisation process -

the vector optimisation of map functions. Section 5 presents some of our results. Section 6 outlines related work and we present our conclusions in Sect. 7.

2 Context

The work described in this paper is part of a of a prototype implementation of a compiler mapping a simple functional language, Adl, to point-free code[2]. The focus of this paper is the optimisation phase, which reduces the flow of data through point-free programs. The optimiser was developed in CENTAUR[4], using rules expressed in Natural Semantics[11]. A simple translator from Adl to point-free form provides the input to the optimisation process. To provide a context for this paper, we outline salient features of Adl, point-free code, and the translation process next.

2.1 Adl

Adl is a small, strict, vector-oriented, functional language. Adl can be described as *point-wise* because, like most languages, it supports the use of variables as a means of storing data. Adl also provides standard operations for iteration - using `while`, conditional execution - using `if`, and scoping - using `let`. Adl supports implicit parallelism through second-order data-parallel operations over nestable single-dimensional arrays, called vectors, including `map`, `reduce` and `scan`. Other vector operations include a length operator (#), an index operator (!) and an iota operation to dynamically allocate a contiguous vector of indices. Adl also supports tuples of arbitrary arity and these are manipulated through pattern-matching. Adl places no restrictions on the nesting of datatypes and operations. Recursion is not supported in its initial implementation. Figure 1 shows an Adl program that adds corresponding elements of two input vectors.

```
main (a: vof int, b: vof int)
   := let
         f x := a!x + b!x
      in
         map(f,iota #a)
      endlet
```

Fig. 1. An Adl program to add corresponding elements of two vectors

We have built a number of applications in Adl and found it to be a simple and expressive language to use.

2.2 Point-Free Form

Our dialect of point-free form is derived from a point-free expression of the theory of concatenate-lists in the Bird-Meertens-Formalism (BMF)[7]. In this paper, we restrict ourselves to the functions required to express the translation and optimisation of Adl, omitting point-free equivalents of `reduce` and `scan`, which are beyond the immediate scope of this paper but discussed in[2].

Description	**Symbol(s)**	**Semantics**
Function Composition	$\cdot$	$(f \cdot g)\, x = f(g\, x)$
Vector map	$*$	$f * [x_0, \ldots, x_{n-1}] = [f\, x_0, \ldots, f\, x_{n-1}]$
All applied to for tuples	$(f_1, \ldots, f_n)^\circ$	$(f_1, \ldots, f_n)^\circ\, x = (f_1\, x, \ldots, f_n\, x)$
Identity function	id	$\mathsf{id}\, x = x$
Tuple access	${}^n\pi_i$	${}^n\pi_i\, (a_1, \ldots, a_n) = a_i$
Constant functions	K	$\mathsf{K}\, x = K$
Arithmetic operators	$+, -, \div, \times, \ldots$	$+\,(x, y) = x + y$ *etc.*
Left distribute	distl	$\mathsf{distl}\, (a, [x_0, \ldots, x_{n-1}]) = [(a, x_0), \ldots, (a, x_{n-1})]$
Zip	zip	$\mathsf{zip}\, ([x_0, \ldots, x_{n-1}], [y_0, \ldots, y_{n-1}]) = [(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})]$
Value repetition	repeat	$\mathsf{repeat}\, (a, n) = [\overbrace{a, \ldots, a}^{n\ times}]$
Vector transpose	$\mathsf{transpose}$	$\mathsf{transpose}\, a = b : \begin{array}{l} \forall (i,j) \in indices(a), a(i,j) = b(j,i) \wedge \\ \forall (i,j) \notin indices(a), a(i,j) = b(j,i) = \bot \end{array}$
Vector enumeration	iota	$\mathsf{iota}\, n = [0, 1, \ldots, n-1]$
Vector length	$\#$	$\#\, [x_0, \ldots, x_{n-1}] = n$
Vector indexing	$!$	$!\, ([x_0, \ldots, x_{n-1}], i) = x_i$
Vector selection	select	$\mathsf{select}\, (v, [x_0, \ldots, x_{n-1}]) = [v!x_0, \ldots, v!x_{n-1}]$

Table 1. A selection of functions in point-free form

Most point-free programs produced by our compiler consist of sequences of composed functions:

$$f_n \cdot f_{n-1} \cdot \ldots \cdot f_1$$

where input data enters at f_1 and flows toward f_n at the end of the program. In the remainder of this paper, we refer to the beginning of the program (f_1) as the *upstream* end of the program and we refer to the end of the program (f_n) as the *downstream* end of the program.

Translation Translation from point-wise Adl to point-free form strips all variable references from Adl code and replaces these with code to transport values between operations. A detailed description of the translation process is given in [2]. Similar translation processes have been defined in[5, 16, 13].

The translation process is conservative. It transports every value in the scope of an operation to the doorstep of that operation. This approach, though simple and robust, results in large surplus transport of data through translator-code. This can be seen in the translation of the Adl code from Fig. 1:

$$(+ \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ, ! \cdot (\pi_2 \cdot \pi_1, \pi_2)^\circ)^\circ) * \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot \#\pi_1)^\circ$$

where the distl operation distributes a copy of both input vectors to each instance of the map function downstream. The aim of the optimiser is to transform programs in order to reduce this volume of copying. We outline the general strategy of our optimiser next.

3 Optimisation Strategy

The optimiser works through the application of simple, semantics-preserving, rules. Taken alone this set of rules is neither confluent or terminating[1]. Moreover, the large number of steps typically required for optimisation, coupled with multiple rule and site choices at each step, leads to a case-explosion. To control these factors we must apply our rules strategically.

Our main strategy is to propagate the optimisation process, on a localised front[2], from the downstream end of the program to the upstream end. The front moves a step upstream by specialising functions immediately upstream of the front with respect to the needs of the optimised code immediately downstream. The front leaves a trail of optimised code in its wake as it steps upstream. The specialisation that takes place in in each step consists of three phases:

Pre-Processing: applies sets of normalisation rules to code at the front. These rules standardise the form of the code to make the application of key optimisation rules easier.

Key-Rule-Application: applies rules that substantially increase efficiency by either eliminating functions responsible for making redundant copies of data, or facilitating the removal of such functions further upstream.

Post-Processing: re-factors code to eliminate some small inefficiencies introduced by pre-processing. and exposes functions for optimisation further upstream.

The phases in each step are applied iteratively until the localised front of optimisation reaches the start of the program.

The broad pattern of processing we have just described applies to all optimisation stages of our implementation. A full description of these stages is beyond the scope of this paper. Instead, we illustrate the process by describing a key part of optimisation, the vector optimisation of map functions.

[1] Most rules could be applied in both directions which, trivially, leads to loops. It has been observed [9] that confluence seems an implausible objective in the context of program optimisation.

[2] In our implementation, we delineate this front by associating function compositions to make the functions on the front appear as an outermost term.

4 Vector Optimisation of Map Functions

The map operator is a second-order function that applies a parameter function to an input vector. In point-free programs, copies of all data required by the parameter function must be explicitly routed to that function. Vector optimisation of map functions reduces the amount of data that must be copied by changing code so that it selectively routes *vector* data to its destination[3]. Specifically, we seek to replace multiple applications of indexing operations on copies of vectors with a single bulk selection operator to direct data to where it is needed. Two general rules are used to achive this aim. For the purposes of explanation, we first introduce two specialised versions of these rules:

$$(!) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ \Rightarrow \mathsf{select} \cdot (\mathsf{id}, R)^\circ \tag{1}$$

$$\underline{(! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ)} * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ \Rightarrow \underline{\mathsf{repeat} \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \# \cdot \pi_2)^\circ} \cdot (\mathsf{id}, R)^\circ \tag{2}$$

The code $(\mathsf{id}, R)^\circ$, though not modified by these rules, is a product of the translation of all calls to `map` in Adl, and provides important context for later discussions. The code R can take various forms but, given an input value, a, always generates some vector of values $[x_0, \ldots, x_{n-1}]$.

Rule 1, above, fuses the vector indexing function and a corresponding distl function into a select. If we factor out $(\mathsf{id}, R)^\circ$ from both sides, the equivalence underlying rule 1 can be informally stated:

$$\begin{aligned}(!) * \cdot \mathsf{distl}\ (a, [x_0, \ldots, x_{n-1}]) &= \\ \mathsf{select}\ (a, [x_0, \ldots, x_{n-1}]) &= \\ [a!x_0, \ldots, a!x_{n-1}]&\end{aligned}$$

The important difference between the two sides is that distl creates the, often large, intermediate structure: $[(a, x_0), \ldots, (a, x_{n-1})]$ whereas select avoids this.

Rule 2 applies where the code that carries out the the indexing (underlined) accesses only the first element in each of the tuples in the vector produced by distl[4]. Again, factoring out $(\mathsf{id}, R)^\circ$, the equivalence underlying rule 2 can be informally stated:

$$\begin{aligned}(! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ) * \cdot \mathsf{distl}\ ((a, x), [y_0, \ldots, y_{n-1}]) &= \\ \mathsf{repeat} \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \# \cdot \pi_2)^\circ\ ((a, x), [y_0, \ldots, y_{n-1}]) &= \\ [a!x, \ldots, a!x]&\end{aligned}$$

where $\#\ [a!x, \ldots, a!x] = \#\ [y_0, \ldots, y_{n-1}] = n$. It should be noted that while rule 2 reduces the size of intermediate structures, it also moves the indexing function further upstream where it can be accessed by subsequent optimisation steps.

[3] The impact of this optimisation is most strongly felt in a distributed parallel context where explicit routing of data is required and this routing incurs a cost.

[4] We know this because both $\pi_1 \cdot \pi_1$ and $\pi_2 \cdot \pi_1$ first execute π_1 which accesses the first element of a tuple.

We emphasise that rules 1 and 2 are specialisations of corresponding, more general, rules in our vector optimiser implementation. We present these rules shortly, but first we describe the pre-processing steps that allow such rules to be applied effectively.

4.1 Pre-processing

The pre-processing of code for vector optimisation of map functions consists of three main stages:

1. **Compaction**: composed functions are coalesced to help reveal true data dependencies and to bring functions of interest as far upstream as possible.
2. **Isolation**: functions are isolated from each other to allow them to be processed individually.
3. **Reorientation**: transpose functions are added, where necessary, to reorient the vectors to aid further processing.

We describe each of these stages in turn.

Compaction Compaction transforms code to minimise the number of composed functions between functions of interest, in this case, indexing functions and the code further upstream. When compaction is complete, index functions are "on the surface" and exposed for further processing. As an example - prior to compaction, the code:

$$(\underline{!.(\pi_2, \pi_1)^\circ \cdot (\pi_2, \pi_1)^\circ}) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ$$

is not amenable to immediate optimisation because the contents of the map function, underlined, is recognisable to neither rule 1 or 2. However, after compacting: $(\pi_2, \pi_1)^\circ \cdot (\pi_2, \pi_1)$ to $(\pi_1, \pi_2)^\circ$ the code takes the form:

$$(! \cdot (\pi_1, \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ$$

and rule 1 can be applied after eliminating the redundant identity $(\pi_1, \pi_2)^\circ$.

Isolation Often, the combination of functions in code confounds the matching of optimisation rules. For example the code:

$$(+ \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ, ! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ$$

matches neither rule 1 or 2. However, if the indexing components are isolated from each other to produce the equivalent code:

$$\begin{aligned}(+) * \cdot \mathsf{zip} \cdot (\, & (! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ, \\ & (! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ)^\circ\end{aligned}$$

to which rule 2 can be applied immediately, and to which a more general form of rule 1 can be applied.

Reorientation In code where indexing functions are nested it is sometimes the case that the dimensions of the input vector are accessed in an order that defeats immediate optimisation. For example, it is not instantly clear how to optimise:

$$(! \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ, \pi_1 \cdot \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ$$

However, if we transpose the vector we can switch the functions used to create the indices giving:

$$(! \cdot (! \cdot (\mathsf{transpose} \cdot \pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ$$

which *almost* matches the specialised rule 1 and actually *does* match the corresponding rule (rule 3) described next.

4.2 Key-Rule Application

The key vector optimisation rules rules are shown in Fig. 2. Rules 3 and 4

Select introduction

$$\frac{\begin{array}{c} \mathit{Not\_In\_oexp}(f_1, \pi_2) \\ \mathit{In\_oexp}(f_2, \pi_2) \\ \mathit{Opt}(f_2 \Rightarrow f_2') \end{array}}{(! \cdot (f_1, f_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ \Rightarrow \mathsf{select} \cdot (f_1, f_2')^\circ \cdot (\mathsf{id}, R)^\circ} \quad (3)$$

Repeat introduction

$$\frac{\begin{array}{c} \mathit{Not\_In\_oexp}(f_1, \pi_2) \\ \mathit{Not\_In\_oexp}(f_2, \pi_2) \end{array}}{(! \cdot (f_1, f_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ \Rightarrow \mathsf{repeat} \cdot (! \cdot (f_1, f_2)^\circ, \# \cdot \pi_2)^\circ \cdot (\mathsf{id}, R)^\circ} \quad (4)$$

Fig. 2. Two key rules of the vector optimiser

correspond to the archetype rules 1 and 2 respectively.

Rules 3 and 4 are expressed in Natural Semantics. The parts above the line in each rule are premises that must be true in order to apply the transformation specified below the line. These rules capture a wider variety of code than their respective archetypes and are thus more useful in an actual implementation.

Both rules hinge on calls to the predicates *In_oexp* and *Not_In_oexp* which test for the presence, and absence, respectively, of π_2 as a most-upstream function in f_1 and f_2. The presence of a π_2 function as the most upstream functions indicates a reference to the output value of R.

During the application of rules 3 and 4, the fate of the f_2 function in each rule differs. In rule 3, the truth of $\mathit{In\_oexp}(f_2, \pi_2)$ implies that f_2 references R. This referencing means that at least some code in f_2 cannot be carried upstream

of R for further processing. In light of this constraint, the recursive call to the vector optimiser, $Opt(f_2 \Rightarrow f_2')$, is made to exploit a last opportunity to, locally, optimise f_2 before the process moves upstream. In rule 4, f_2 does not reference the output of R and thus *can* be carried upstream of R for further processing.

On a related note, some thought about the premises of both rules reveals that code such as:

$$(! \cdot (! \cdot (\pi_1, \pi_2)^\circ, \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ$$

will match neither rule 3 or 4. In these cases we apply a default rule, not shown here, leaves outer index function intact. The post-processing phase is then left to salvage what it can from the code that generates its parameters for optimisation upstream.

4.3 Post-processing

After the application of the key rules in the last section, code is often in no fit state for immediate processing further upstream. It is the task of post-processing to *compact* and *combine* optimisable code fragments to prepare them for the next optimisation step. As an example of compaction, after applying rule 4 to the code:

$$(! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ$$

we have:

$$\mathsf{repeat} \cdot (! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \# \cdot \pi_2)^\circ \cdot (\mathsf{id}, R)^\circ$$

where the functions of interest: $! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ$, and $\# \cdot \pi_2$ are not in the most-upstream section of code, ready for further processing. Post-processing compacts these functions into $(\mathsf{id}, R)^\circ$ producing the, more accessible, code:

$$\mathsf{repeat} \cdot (! \cdot (\pi_1, \pi_2)^\circ, \# \cdot R)^\circ$$

As an example of combination, the application of rules 3 and 4, plus compaction, to the code:

$$\begin{array}{c}(+) * \cdot \mathsf{zip} \cdot (\, (((! \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ, \\ ! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ) * \cdot \mathsf{distl} \cdot (\mathsf{id}, R)^\circ)^\circ\end{array}$$

produces:

$$\begin{array}{c}(+) * \cdot \mathsf{zip} \cdot (\, \mathsf{repeat} \cdot (! \cdot (\pi_1, \pi_2)^\circ, \# \cdot R)^\circ, \\ \mathsf{select} \cdot (\pi_1, R)^\circ)^\circ)^\circ\end{array}$$

Subsequently, further post-processing *combines* the two zipped sections of code to produce the more easily processed:

$$(+) * \cdot \mathsf{distl} \cdot (! \cdot (\pi_1, \pi_2)^\circ, \mathsf{select} \cdot (\pi_1, R)^\circ)^\circ$$

Note that the re-introduced distl function now transports just the required values to downstream code rather than broadcasting copies of whole vectors.

This concludes our description of the vector optimisation of map. The code resulting from vector optimisation has a significantly reduced flow of surplus vector elements. However, surplus flows of other values remain. Our implementation reduces these flows during, much-simpler, subsequent passes of optimisation. A discussion of these other passes is beyond the scope of this paper but their effects are evident in the performance of code produced by the optimiser in its entirety. We examine this performance next.

5 Results

We now examine the impact of the optimisation process on the performance of point-free program code. After this, we briefly discuss the influence that point-free form has on the design of the optimiser.

5.1 Performance Model

To measure the effect of data movement optimisation we created an instrumented model for measuring execution time and space consumption on point-free code. To keep the design of the model simple and consistent we implemented the following basic strategy for memory allocation and deallocation:

- memory for each data element is *a*llocated just prior to when it is needed.
- memory for each data element is *d*e-allocated just after its last use.

Our model assigned unit costs to all scalar operations and unit costs to allocating and to copying scalars. Vectors and tuples were treated as collections of scalars.

It must be noted that, when mapping point free code to imperative sequential code there are optimisations that could be applied that our model doesn't reflect. However, in a distributed context, we have found that high data-transport costs in this model map to high communications costs[1].

5.2 Experiments

We ran the translator and then the optimiser over a series of Adl programs and used an implementation of the model above to compare the performance of translator and optimiser code. As a benchmark, we hand-coded efficient solutions in point-free form and ran these against the model too. The results of two of these experiments are presented next.

Adding Corresponding Elements of Nested Vectors The source code for `map_map_addpairs.Adl` is shown in Fig. 3(a). This program uses a nested map operation to add corresponding elements of two nested input vectors. The translator code, the optimiser code and the handed-coded point-free version are shown in parts (b), (c) and (d) respectively. The translator code distributes large amounts of data to the inner map function to be accessed by index functions. The

```
main (a: vof vof int, b: vof vof int)
 := let
       f x :=
          let
              g y := a!x!y + b!x!y
          in
              map(g,iota #(a!x))
          endlet
    in
       map(f,iota (# a))
    endlet
```

(a)

$$
\begin{array}{l}
((+\cdot\ (!\cdot(!\cdot(\pi_1\cdot\pi_1\cdot\pi_1, \pi_2\cdot\pi_1)^\circ, \pi_2)^\circ, \\
\qquad !\cdot(!\cdot(\pi_2\cdot\pi_1\cdot\pi_1, \pi_2\cdot\pi_1)^\circ, \pi_2)^\circ)^\circ) * \cdot \\
\quad \mathsf{distl}\cdot(\mathsf{id}, \mathsf{iota}\cdot\#\cdot!\cdot(\pi_1\cdot\pi_1, \pi_2)^\circ)^\circ\cdot\mathsf{id}) * \cdot \\
\ \mathsf{distl}\cdot(\mathsf{id}, \mathsf{iota}\cdot\#\cdot\pi_1)^\circ\cdot\mathsf{id}
\end{array}
$$

(b)

$$
\begin{array}{l}
((+) * \cdot \mathsf{zip} \\
\qquad \cdot(\mathsf{select}\cdot({}^3\pi_1, {}^3\pi_2)^\circ, \mathsf{select}\cdot({}^3\pi_3, {}^3\pi_2)^\circ)^\circ\cdot \\
\qquad (\pi_1\cdot\pi_1, \mathsf{iota}\cdot\pi_2, \pi_2\cdot\pi_1)^\circ) * \cdot \\
\mathsf{zip}\cdot(\,((\pi_2, \pi_1)^\circ) * \cdot \mathsf{zip}\cdot \\
\qquad (\mathsf{select}\cdot({}^3\pi_1, {}^3\pi_2)^\circ, \mathsf{select}\cdot({}^3\pi_3, {}^3\pi_2)^\circ)^\circ, \\
\qquad (\#) * \cdot \mathsf{select}\cdot({}^3\pi_3, {}^3\pi_2)^\circ)^\circ\cdot \\
(\pi_1, \mathsf{iota}\cdot\#\cdot\pi_2, \pi_2)^\circ\cdot(\pi_2, \pi_1)^\circ
\end{array}
$$

(c)

$$
\begin{array}{l}
(\,(+) * \cdot \mathsf{zip}\cdot \\
\quad (\mathsf{select}\cdot(\pi_1\cdot\pi_1, \pi_2)^\circ \\
\quad \mathsf{select}\cdot(\pi_2\cdot\pi_1, \pi_2)^\circ)^\circ\cdot \\
\quad (\mathsf{id}, \mathsf{iota}\cdot\#\cdot\pi_1)^\circ) * \cdot \\
\mathsf{zip}\cdot \\
(\mathsf{select}\cdot(\pi_1\cdot\pi_1, \pi_2)^\circ \\
\mathsf{select}\cdot(\pi_2\cdot\pi_1, \pi_2)^\circ)^\circ\cdot \\
(\mathsf{id}, \mathsf{iota}\cdot\#\cdot\pi_1)^\circ
\end{array}
$$

(d)

Fig. 3. Source code - part (a), translator code - part (b), optimiser code - part (c), and hand-crafted code - part(d) for `map_map_addpairs.Adl`

optimiser code in part (c) has replaced all of the indexing operations by select

operations. The hand-coded version in part (d) has the same basic structure as the code in part (c) but forms fewer intermediate tuples.

The performance of the three versions of point-free code, applied to a pair of nested input vectors, is shown in Fig. 4. The translator code fares the worst

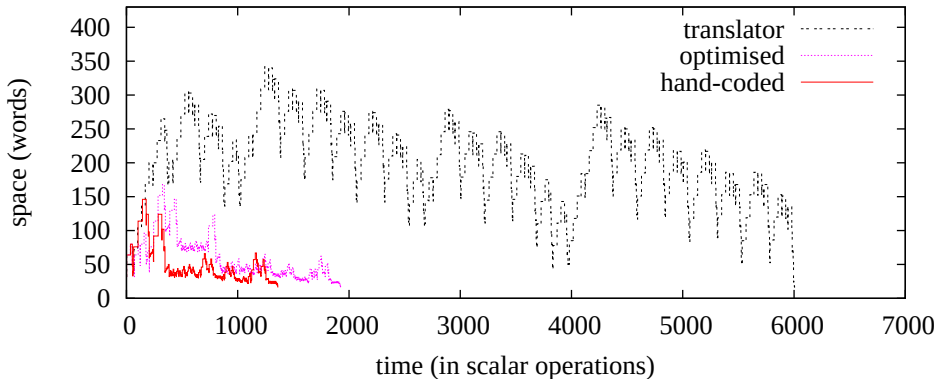

Fig. 4. Performance plots for map_map_addpair applied to the pair of ragged vectors: $([[1],[2,3,4],[5,6],[],[7,8,9,10]],[[1],[2,3,4],[5,6],[],[7,8,9,10]])$.

due to the cost of distributing aggregate values. The optimiser code exhibits substantially better performance. The hand optimised version performs even better, with a similar pattern of data allocation on a slightly smaller scale. Close inspection of the code reveals that the optimiser has been more aggressive than necessary in moving the # (length) function out of the inner-map function. This resulted in extra code to transmit the output of the # that has now been moved upstream. The movement wasn't warranted in this case because the input vector to this invocation of # had to be transmitted to the inner map function anyway. This result indicates that there is scope for tempering the aggression of the vector optimiser in certain cases.

A Simple Stencil Operation The source code for `finite_diff.Adl` is shown in Fig. 3(a). This program applies a very simple stencil operation to a one-dimensional vector. It is a good example of the use of multiple indexing operations into a single vector and the use of arithmetic operators on vector indices. The translator code, the optimiser code and the handed-coded point-free version are shown in parts (b), (c) and (d) respectively. Again the translator code distributes surplus data through the distl operations. The optimiser code in part (c) has removed this distribution. The hand-coded solution in part (d) is similar to the hand-coded version but avoids the use of repeat.

Figure 6 shows performance of the three versions of point-free code. Again, the translator code is the worst-performing. The efficiencies of the optimised

```
main a: vof int :=
  let
    stencil x
      := a!x + a!(x-1) + a!(x+1);
    addone x := x + 1;
    element_index
      := map(addone,iota ((# a)-2))
  in
    map (stencil, element_index)
  endlet
```

(a)

$$
\begin{array}{l}
(+ \cdot (+ \cdot (\,! \cdot (\pi_1 \cdot \pi_1, \pi_2)^\circ, \\
\qquad\quad ! \cdot (\pi_1 \cdot \pi_1, - \cdot (\pi_2, 1)^\circ)^\circ)^\circ, \\
\qquad ! \cdot (\pi_1 \cdot \pi_1, + \cdot (\pi_2, 1)^\circ)^\circ)^\circ) * \cdot \\
\mathsf{distl} \cdot (\mathsf{id}, \pi_2)^\circ \cdot \mathsf{id} \cdot \\
(\mathsf{id}, (+ \cdot (\pi_2, 1)^\circ) * \mathsf{distl} \cdot (\mathsf{id}, \mathsf{iota} \cdot - \cdot (\# \cdot \mathsf{id}, 2)^\circ)^\circ)^\circ
\end{array}
$$

(b)

$$
\begin{array}{l}
(+ \cdot (+ \cdot \pi_1, \pi_2)^\circ) * \cdot \mathsf{zip} \cdot \\
(\mathsf{zip} \cdot \\
(\,\mathsf{select}, \\
\;\mathsf{select} \cdot (\pi_1, (-) * \cdot \mathsf{zip} \cdot \\
\quad (\mathsf{id}, \mathsf{repeat} \cdot (1, \#)^\circ)^\circ \cdot \pi_2)^\circ)^\circ, \\
\mathsf{select} \cdot (\pi_1, (+) * \cdot \mathsf{zip} \cdot (\mathsf{id}, \mathsf{repeat} \cdot (1, \#)^\circ)^\circ \cdot \pi_2)^\circ)^\circ \cdot \\
(\mathsf{id}, (+ \cdot (\mathsf{id}, 1)^\circ) * \cdot \mathsf{iota} \cdot - \cdot (\#, 2)^\circ)^\circ
\end{array}
$$

(c)

$$
\begin{array}{l}
(+ \cdot (\pi_1 \cdot \pi_1, + \cdot (\pi_2 \cdot \pi_1, \pi_2)^\circ)^\circ) * \cdot \\
\mathsf{zip} \cdot (\mathsf{zip} \cdot (\pi_1 \cdot \pi_1, \pi_2 \cdot \pi_1)^\circ, \pi_2)^\circ \cdot \\
((\,\mathsf{select}, \\
\quad \mathsf{select} \cdot (\pi_1, (+ \cdot (\mathsf{id}, 1)^\circ) * \cdot \pi_2)^\circ)^\circ, \\
\;\mathsf{select} \cdot (\pi_1, (- \cdot (\mathsf{id}, 1)^\circ) * \cdot \pi_2)^\circ)^\circ \cdot \\
(\mathsf{id}, (+ \cdot (\mathsf{id}, 1)^\circ) * \cdot \mathsf{iota} \cdot - \cdot (\#, 2)^\circ)^\circ
\end{array}
$$

(d)

Fig. 5. Source code - part (a), translator code - part (b), optimiser code - part (c), and hand-crafted code - part(d) for `finite_diff.Adl`

and hand-coded versions are very similar with the optimiser code very slightly ahead on time and space performance. Close inspection of the code shows that the optimiser code in part (c) has been more thorough in eliminating transport of tuple elements in downstream parts of the code.

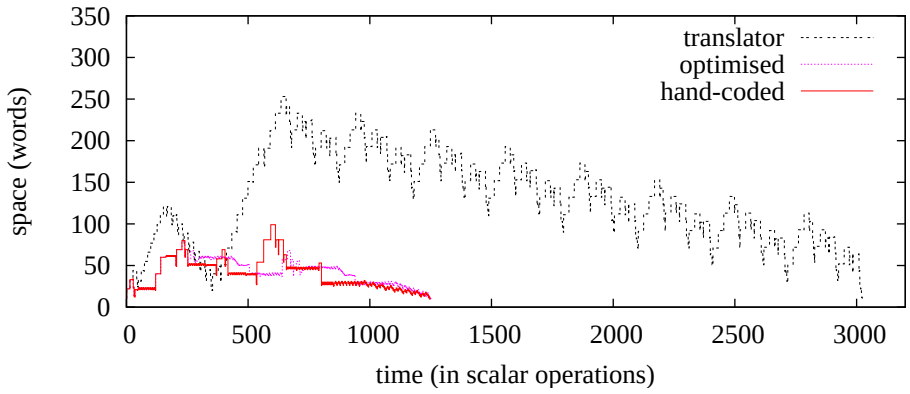

Fig. 6. Performance plots for `finite_diff` applied to a short one-dimensional vector.

5.3 Point-Free Form and Optimiser Design

The performance of code produced by the optimiser in the experiments presented here, and in other experiments we have carried out, is encouraging. In addition to these results we make the following general observations regarding the influence of point-free form on optimiser design.

First, pre-processing, using normalisation rules, is an essential part of automatically processing point-free form. It is infeasible to write enough rules to match code as-is, so code needs to be processed to match the rules. We applied, and reused, normalisation rules quite frequently and in a variety of circumstances to maximise the effect of other transformation rules.

Second, it pays to have interfaces. If there is a mismatch between the output of one transformation component and the expected input of another, the cause of the resulting error can be difficult to find. The use of interfaces, even on an informal, ad-hoc, basis makes matching, and constructing compilation components much less error-prone.

Last, it is difficult to write intelligent rules. It is not easy to trace dependencies through, and then alter, at a distance, point-free code. It is easier to propagate dependencies by local transformations to the code itself.

6 Related Work

There is a large body of work exploring program transformation in functional notations. [17] summarises many of the important issues that arise during transformation. The use and transformation of point-free programs was espoused in [3]. A broad range of rules and techniques applicable to both point-free and point-wise programs were developed with Bird-Meertens Formalism[7] we used a number of the algebraic identities developed in this work in our optimiser.

The idea of making programs more efficient by reducing the intermediate data produced is strongly related to concept of program fusion. Automated systems have been developed to perform fusion[14, 10] in point-wise recursive definitions. Our work, while using some of the rules derived by research into fusion, is more specialised in its purpose. However, there remains broad scope for further applications of techniques from the work above in refinements of our optimiser.

Point-free notation has been used in a few experiments and implementations including FP*[21] and EL*[16]. Some early experiments with transforming point-free code are described in[12]. More recently a complete translation process from recursive functions in Haskell to point-free form has been defined[5]. However, none of these implementations perform data movement optimisation to the extent of the implementation described here[5].

7 Conclusions and Future Work

We have described an optimisation process to reduce data movement through point-free code. We have shown that this process is effective in significantly reducing the flow of data through such programs. This process is incremental with the program itself serving as the sole repository of the state of the transformation process.

We envisage three major improvements to the implementation as defined so far. First, the system could be made more extensible by the formal definition of syntax interfaces for transformation components as in[19]. Second, the volume of transformation rules may be significantly reduced by separating rule application strategies from the rewrite rules themselves[20]. Last, an efficient mapping of point free code to imperative code on sequential architectures needs to be defined, work toward this goal is underway[15].

To conclude, optimisation in point free form is a highly effective process that requires different strategies to more traditional approaches. Like all optimisation, this process is open-ended and much interesting work remains to be done.

References

1. Alexander, B. , Wendelborn, A. L. 2004, *Automated Transformation of BMF Programs*, The First International Workshop on Object Systems and Software Architectures., pp. 133-141,
   URL: `http://www.cs.adelaide.edu.au/ wossa2004/HTML/19-brad-2.pdf`
2. Alexander, B. 2006, *Compilation of Parallel Applications via Automated Transformation of BMF Programs*, Phd. Thesis, University of Adelaide,
   URL: `http://www.cs.adelaide.edu.au/ brad/thesis/main.pdf`
3. Backus, J. 1978, *Can Programming Be Liberated from the von Neumann Style? A functional Style and Its Algebra of Programs*, 'Communications of the ACM', Vol. 21, No. 8, pp. 613 – 641.

[5] FP* performs some optimisation as code is being mapped to an imperative language but explicit data transfer information, useful for mapping to a distributed machine, is lost in the process.

4. Borras, P., Clément, D., Despeyroux, Th., Incerpi, J., 1989 , Kahn, G., *CENTAUR: the system*, SIGSOFT software engineering notes / SIGPLAN: SIGPLAN notices, Vol. 24, No. 2, The ACM.
5. Cunha, A., Pinto, S. P., , Proenca, J., 2005, *Down with Variables*, Technical report, No. DI-PURe-05.06.01.
6. Crooke, D. C., 1999, *Practical Structured Parallelism Using BMF*, Thesis, University of Edinburgh.
7. Gibbons, J., 1994, *An introduction to the Bird-Meertens Formalism*, 'New Zealand Formal Program Development Colloquium', Hamilton, NZ.
8. Hamdan, M., 2000 , *A Combinational Framework for Parallel Programming Using Algorithmic Skeletons*, Thesis, Department of Computing and Electrical Engineering. Heriot-Watt University.
9. Jones, S., Hoare, T. and T. Hoare., Tolmach, A., 2001,*Playing by the rules: rewriting as a practical optimisation technique*, ACM SIGPLAN Haskell Workshop.
10. Johann, P., Visser, E., 1997 , *Warm fusion in Stratego: A case study in the generation of program transformation systems*, Technical report, Department of Computer Science, Universiteit Utrecht, 1999.
11. Kahn, G., 1987 , *Natural Semantics*, Fourth Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, Vol. 247, Springer-Verlag.
12. Martin, U. , Nipkow, T., 1990 , *Automating Squiggol*, Programming Concepts and Methods, pp. 233–247, Eds. Broy, M., Jones, C. D., North-Holland.
13. Mauny, M. , Ascander Suarez, 1986 , *Implementing functional languages in the Categorical Abstract Machine*,
LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming, pp. 266–278, ACM Press.
14. Onoue, Y., Hu, Z., Takeichi, M., , Iwasaki H. 1997 , *A calculational fusion system HYLO*, 'Proceedings of the IFIP TC 2 WG 2.1 international workshop on Algorithmic languages and calculi', pp. 76–106, Chapman & Hall, Ltd.
15. Pettge, S. 2005, *A Fast Code Generator for Point-Free Form*, Hons. Thesis, University of Adelaide,
URL: `http://www.cs.adelaide.edu.au/ brad/students/seanp.pdf`
16. Rao, P. , Walinsky, C., 1993, *An equational language for data-parallelism*, Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 112–118, ACM Press.
17. Sands, D., 1996, *Total correctness by local improvement in the transformation of functional programs*, 'ACM Trans. Program. Lang. Syst.', Vol. 18, No. 2.
18. Skillicorn, D. B. , Cai, W., 1994, *Equational code generation: Implementing categorical data types for data parallelism*, TENCON '94, Singapore, IEEE.
19. de Jonge, M. , Visser, J., 2001, *Grammars as Contracts*, Lecture Notes in Computer Science, Vol. 2177, Springer-Verlag.
20. Visser, E., 2001, *Stratego: A Language for Program Transformation Based on Rewriting Strategies*,
RTA '01: Proceedings of the 12th International Conference on Rewriting Techniques and Applications, pp. 357–362, Springer-Verlag.
21. Walinksy, C., Banerjee, D., 1994, *A Data-Parallel FP Compiler*, Journal of Parallel and Distributed Computing, Vol. 22, pp. 138–153.