Proceedings of PASCO'94

The vectorisation monad

Jonathan M.D. Hill, Keith M. Clarke, and Richard Bornat*
Department of Computer Science
Queen Mary & Westfield College
University of London

Abstract: Traditionally a vectorising compiler matches the iterative constructs of a program against a set of predefined templates. If a loop contains no dependency cycles then a *map* template can be used; other simple dependencies can often be expressed in terms of *fold* or *scan* templates. This paper addresses the template matching problem within the context of functional programming. A small collection of program identities are used to specify vectorisable for-loops. By incorporating these program identities within a monad, *all* well-typed for-loops in which the body of the loop is expressed using the *vectorisation monad* can be vectorised. This technique enables the elimination of template matching from a vectorising compiler, and the proof of the safety of vectorisation can be performed by a type inference mechanism.

Keywords: Data parallelism; monads; vectorisation; Bird-Meertens formalism; program transformation; category theory; imperative functional programming; non-strict semantics; Haskell.

1 Introduction

> " *'Should declarative languages be mixed with imperative languages?', clearly has the answer that they should, because at the moment we don't know how to do everything in pure declarative languages.*" C. Strachey, 1966 [13].

It has long been known that some of the most common uses of `for`-loops in imperative programs can easily be expressed using the standard higher-order functions *fold* and *map*. With this correspondence as a starting point, we derive parallel implementations of various iterative constructs, each having a better complexity than their sequential counterparts, and explore the use of monads to guarantee the soundness of the parallel implementation.

As an aid to the presentation of the material, we use the proposed syntax for **p**arallel **H**askell [18] (figure 1) as a vehicle in which imperative functional programs will be expressed. Incorporating imperative features into a purely functional language has become an active area of research within the functional programming community [20, 14, 23]. One of the techniques gaining widespread acceptance as a model for imperative functional programming is monads [24, 17]. Typically monads are used to guarantee single threadedness, enabling side effects to be incorporated into a purely functional language without losing referential transparency. We take a different approach. First, for-loops are translated into a monadic framework. Next, by ensuring that the monad satisfies the programming identities usually associated with the successful vectorisation of imperative constructs, *all* well-typed for-loops in which the body is expressed using the *vectorisation monad* can be parallelised.

The technique is not just restricted to a data-parallel environment. It could be implemented by a divide and conquer technique on a multiprocessor platform, or by a parallel implementation of graph reduction.

*Email: {hilly,keithc,richard}@dcs.qmw.ac.uk

2 Background

We use a model of parallel computation based upon the Bird Meertens Formalism [2]. To make things simple, all parallelism is expressed in terms of the *map*, *fold*, and *scan* functions. Given a function f which has a $\mathcal{O}(1)$, then $(map\ f)$ is $\mathcal{O}(1)$, while $(scanPar\ f)$ and $(foldPar\ f)$ are each $\mathcal{O}(\log N)$ if f is associative. Unlike our earlier work, we use the BMF as is—all the operations presented here can be interpreted *as though they were being expressed on lists.* This stance contradicts an earlier paper [7] in which we said that lists were unsuitable for data-parallel evaluation in a non-strict language. We still believe this to be true, but we have tried to sugar the pill somewhat. By using a purely combinator approach to programming, it is possible to give the impression of using lists, whilst actually implementing these list-like objects on top of more suitable data-parallel data structures (see [7, 8]).

$$
\mathsf{TPH}\left[\!\!\left[\begin{array}{l}\texttt{for i <-}\mathit{range}\ \texttt{do}\\ \quad pat_1 = exp_1\\ \quad .\qquad ..\\ \quad pat_n = exp_n\\ \texttt{finally}\ exp_{fin}\end{array}\right]\!\!\right]
$$

$$
= \texttt{case}\left(\begin{array}{l}\texttt{foldl}\left(\begin{array}{l}\lambda(s_1,\ldots,s_m)\ i \rightarrow\\ \quad \texttt{let}\mathsf{TPAT}[\![pat_1]\!] = \mathsf{TPH}[\![exp_1]\!]\\ \qquad\vdots \qquad\qquad\qquad \vdots\\ \qquad \mathsf{TPAT}[\![pat_n]\!] = \mathsf{TPH}[\![exp_n]\!]\\ \quad \texttt{in}\ (v'_1,\ldots,v'_m)\end{array}\right)\\ \qquad (v_1,\ldots,v_m)\ range\end{array}\right)\ \texttt{of}
$$

$$
\quad \sim (v_1,\ldots,v_m)\ \texttt{->}\ exp_{fin}
$$

where

$$
\{(v_1,v'_1),\ldots,(v_m,v'_m)\} = \{(\texttt{v},\texttt{v'}) \mid \texttt{next v} \in \{pat_1,\ldots,pat_n\}\}
$$

$\mathsf{TPH}[\![\texttt{next v}]\!] = \texttt{v'}$ where $\mathsf{TPH}[\![exp]\!]$ is the identity translation for other expressions.

$\mathsf{TPAT}[\![\texttt{next v}]\!] = \texttt{v'}$ where $\mathsf{TPAT}[\![pat]\!]$ is the identity translation for other patterns.

Figure 2: Translating a *pH* for-loop

```
fact n
 =let ac = 1
  in for i<-[1..n] do
       next ac = ac*i
     finally ac
```

Figure 3: *pH*

```
fact n = f 1 [1..n]
 where
  f ac []     =ac
  f ac (i:is)=f (ac*i) is
```

Figure 4: Tail recursive

```
fact n =foldl (\ac i -> ac * i) 1 [1..n]
```

Figure 5: Factorial in terms of `foldl`

3 Parallelising simple loops

It is part of folklore that programs expressed as for-loops can be rewritten using tail recursion (for example, [12, 6]), although the functional programming community has concentrated on how the reverse translation can be used as an optimisation technique [19]. As an example of such a translation, the iterative factorial of figure 3 can be expressed as the function of figure 4. Unfortunately, replacing an inherently sequential for-loop with a sequential tail recursive function does not provide the right foundations for parallelisation. However it is interesting to note that the resulting tail recursive function is an instance of a fold-left computation over the loop range.

If $\ominus$ is a function that describes the computation that occurs at each iteration of a for-loop, then the definition of the factorial function can be rewritten using schema TPH of figure 2 as a fold-left of $\ominus$ (see figure 5). We term $\ominus$ the next-function of the for-loop. As we have already mentioned, a parallel fold has a $\mathcal{O}(\log N)$ time complexity, if the function being folded is $\mathcal{O}(1)$ and associative. Whenever $\ominus$ and the initial state of the for-loop form a monoid ($\ominus$ and x form a monoid if $\ominus$ is associative and has x as its left and right identity), then by the *first duality theorem* (eqn. 1) [4] the fold-left can be safely transformed into a fold-right, or more interestingly a parallel $\mathcal{O}(\log N)$ fold [7].

$$
\begin{aligned}
foldl\ (\ominus)\ x\ xs &\equiv foldr\ (\ominus)\ x\ xs \\
&\equiv foldPar\ (\ominus)\ x\ xs \\
&\quad \text{iff } \ominus \text{ and } x \text{ form a monoid} \\
&\quad \text{and } xs \text{ is finite.}
\end{aligned} \tag{1}
$$

If this theorem is used to transform a fold-left into a right or parallel fold, then the side condition requiring finite lists is ignored. The justification for this is that in all situations where a fold-left yields a non bottom value, a fold-right will give the same result. However, there may be situations where fold-left diverges and fold-right terminates. We regard this as a good thing, in the same way that we believe normal order reduction to be superior to applicative order reduction.

$pat \mapsto \texttt{next}\ var$

$exp \mapsto \texttt{for}\ pat\ \texttt{<-}\ exp\ \texttt{do}\ \{\text{pat=exp}\}^{+}\ \texttt{finally}\ exp$

$\quad \mid \texttt{next}\ var$

Figure 1: Proposed syntax extensions for *pH*

4 Relaxing associativity

When the next-function $\ominus$ is associative, the left argument used to represent the state of the loop, the right argument that represents the loop counter, and the result of the next-function that represents the successive state of the loop body, *all* *have the same type* (by the definition of associativity). This is rather unfortunate, as the numerical algorithms we intend to parallelise typically range over a subset of the integers, and the loop's state will be a mixture of floating point and integer values. In the situation where the body of the loop contains a single state (i.e., there is only one `next` binding in the body of the loop), a solution is to decompose $\ominus$ into a part that is specific to the computation of the loop-counter, and a remainder that is specific to the loop's state. If $\ominus$ has the structure $\lambda s\ i \rightarrow s\ \oplus\ f\ i$, where $\oplus$ and the initial value of the for-loop now form a monoid, then the part of the computation that is specific to the loop counter can be moved outside the fold-left computation by using the *fold-map fusion* law[1] (eqn. 2) [3].

$$\begin{aligned} &foldl\ (\lambda s\ i \rightarrow s \oplus f\ i)\ x \\ &\quad \equiv foldl\ (\oplus)\ x\ \circ\ map\ f \end{aligned} \tag{2}$$

Comparing this law to a conventional optimisation on loops, the inverse of loop fusion [1] is being performed in a scenario that would be detrimental to performance in a sequential environment as the overheads associated with evaluation of a loop will be incurred twice. Because of the differing complexities of fold and map in a data-parallel environment, the law is an optimisation technique.

5 The leaky fold-left law

The aim of this section is to investigate the operational phenomenon of space-leaks [19, 22], and show how fold-left computations are particularly susceptible to leaking space. A collection of identities are used to highlight the leaky behaviour of fold-left. From these identities, a parallel fold is developed that is not just restricted to associative operators. Unfortunately fold computations parallelised in this way suffer from a similar problem to a space leak—a vectorisation leak. The purpose of this section is to therefore highlight some of the problems of parallel fold, and the techniques developed here form the foundations for the vectorisation based upon monads later in the paper.

Given the problem of finding the sum of a list of numbers, there is the opportunity to use a left or right fold, as the addition used in the sum is associative. A Scheme or ML programmer would probably side for a fold-left computation because of its tail recursive nature. A non-strict functional programmer has a dilemma! For each iteration of a fold-left, a closure is created in the accumulated parameter of the fold that represents the application of the folded function to the previous value of the accumulator. This closure remains unevaluated, growing whilst the spine of the folded list is unwound. Only when the end of the list is reached, and the closure's size is proportional to the length of the list, is the closure evaluated—this is the space leak. A clever compiler should be able to eliminate this 'dragging' if the folded function is known to be strict[2], because some *real* computation could occur in the accumulated parameter.

Putting such dilemmas to one side, what this detour into the operational characteristics of fold-left has exposed is yet another way of thinking of fold. As opposed to the normal space-leaking behaviour of fold-left being interwoven with the vagaries of a non-strict evaluation mechanism, it can be made a *feature* of fold-left such that an implementation leaks space in both a strict and non-strict language!

$$\begin{aligned} &foldl\ (\oplus)\ x\ xs \\ &\quad \equiv (foldr\ (\lambda s\ i \rightarrow\ \ (\lambda y \rightarrow y \oplus i) \circ s) \\ &\qquad\qquad id\ xs)\ x \end{aligned} \tag{3}$$

The *leaky fold-left law* (eqn. 3) gives a correspondence between fold-left and an explicitly leaky version of the function. As this definition is an instance of the fold-map fusion law, it can be transformed into the following identity[3]:

$$\begin{aligned} &foldl\ (\oplus)\ x\ xs \\ &\quad \equiv (foldr\ (\tilde{\circ})\ id\ (map\ (\tilde{\oplus})\ xs))\ x \end{aligned} \tag{4}$$

The program identity is now an instance of the first duality theorem (because $\tilde{\circ}$ and the identity function form a monoid), and the fold-right can be safely replaced by a left or parallel fold. The identity can be understood by decomposing the right-hand side into three parts:

1. $\tilde{\oplus}$ is partially applied to all the elements of the list being folded;
2. the resulting list of function values is folded using $\tilde{\circ}$. The effect is to create a closure that is equivalent to the function produced by unrolling $\tilde{\oplus}$ at runtime. e.g., given $[1, 2, 3]$, the closure $(\oplus\ 3) \circ (\oplus\ 2) \circ (\oplus\ 1)$ is created;

[1] There seems to be a strange anomaly that laws are used in the opposite direction when reasoning about parallel functions, compared to similar functions on lists. The names are borrowed from the list context, even though the term *fold-map fission* would be more appropriate in a data-parallel setting.

[2] or as is the case with Miranda, a strictness annotation is used that explicitly forces the evaluation in the accumulator.

[3] As in [2], $\tilde{\oplus}$ is used to represent the flipped version of the operator $\oplus$, i.e., $\lambda x\ y \rightarrow y \oplus x$

3. the unrolled closure is then applied to the initial state x, and only at this point can the real computation of all the $\oplus$'s commence.

What we have shown is that a for-loop is nothing more than syntactic sugar for a fold-left. By means of a series of program transformations, a fold-left can be transformed into a left or right fold of the compose function, and a map of the original function that we wanted to fold. In [8] we show how a $\mathcal{O}(1)$ map can be implemented, and because of the associativity of $\circ$, *a parallel fold can be used in the implementation of a for-loop regardless of the associativity of the function modelling the body of the loop.* Unfortunately, there is a rather large downside to this transformation. A program implemented in such a manner creates a closure with the same number of compositions as there are loop iterations. Evaluation of this closure has a linear complexity, therefore the complexity of the entire for-loop is $\mathcal{O}(N)$, and not the desired $\mathcal{O}(\log N)$. Fortunately, in a wide class of problems, monads can be used as a solution to this dilemma, and this is addressed in §7.

6 Example: inverse sine

The program identities developed so far are used in the parallelisation of a numerical problem, the evaluation of a series that approximates the trigonometric function $\sin^{-1} x$ to any desired accuracy (equation 5). For a given accuracy, the idea is to convert a sequential algorithm that performs a linear number of expansions of the inverse sine series, into an algorithm with a logarithmic number of parallel expansions.

$$\begin{aligned}\sin^{-1} x &= x + \frac{x^3}{2\cdot 3} + \frac{1\cdot 3\cdot x^5}{2\cdot 4\cdot 5} + \frac{1\cdot 3\cdot 5\cdot x^5}{2\cdot 4\cdot 6\cdot 7} + \ldots \\ &= \sum_{i=0}^{\infty} \frac{x^{2i+1}\prod_{j=1}^{i}(2j-1)}{(\prod_{j=1}^{i} 2j)\cdot(2i+1)} \end{aligned} \tag{5}$$

Although there is little point in parallelising this algorithm as the series rapidly converges, it forms an interesting case study. A naive $\mathcal{O}(N^2)$ sequential algorithm leads to a naive $\mathcal{O}(N \log N)$ parallel algorithm. The naive sequential algorithm may be converted using standard imperative style program transformations into a $\mathcal{O}(N)$ algorithm. Unfortunately the improved sequential algorithm fails to be parallelised with the identities developed so far. This is remedied by a collection of additional laws, and a $\mathcal{O}(\log N)$ parallel algorithm is finally derived.

```
sinI :: Float -> Int -> Float
sinI x n
 = let s = 0.0
   in for i <- [0..n] do
        top= x ^(2*i+1) *toFloat (product
               (map (\j->2*j-1) [1..i]))
        bot= (2*i + 1) * product
               (map (\j -> 2*j) [1..i])
        next s = s + (top / toFloat bot)
      finally s
```

Step 0: Original Program

The function `sinI` (Step 0) defines an implementation that performs a fixed number of expansions of the series of equation 5 in terms of a *pH* for-loop. The recursive let-bindings `top` and `bot` in the body of the for-loop have a structure that mimics the numerator and denominator of equation 5. As the body is an instance of the fold-map fusion law, the function can be transformed using the relationship of equation 6.

$$\begin{aligned} &\lambda s\ i \rightarrow s \ \oplus\ f\ i \\ &\equiv \lambda s\ i \rightarrow s + \frac{x^{2i+1}\prod_{j=1}^{i}(2j-1)}{(\prod_{j=1}^{i} 2j)\cdot(2i+1)} \\ &\text{iff}\ s \oplus t\ = s + t \\ &f\ i\ \ = \frac{x^{2i+1}\prod_{j=1}^{i}(2j-1)}{(\prod_{j=1}^{i} 2j)\cdot(2i+1)} \end{aligned} \tag{6}$$

Because floating-point addition and `0.0` form a monoid[4], the first duality theorem can be used to transform the fold-left into a parallel fold.

```
sinI x n = foldl (+) 0.0 (map f [0..n])
f i=let top= x^(2*i+1) * toFloat (product
               (map (\j->2*j-1) [1..i]))
        bot=(2*i+1)*
              product (map (\j->2*j) [1..i])
    in top / toFloat bot
```

Step 1: TPH and fold-map fusion

This implementation of inverse sine, perhaps surprisingly, is not $\mathcal{O}(\log N)$. The original function can be thought of as a nested $\mathcal{O}(N^2)$ for-loop, as the computations of both the numerator and denominator are $\mathcal{O}(N)$. The derived parallel algorithm has an outer loop with $\log N$ parallel iterations, each of which contains a $\mathcal{O}(N)$ product, resulting in a $\mathcal{O}(N \log N)$ parallel algorithm. It would be expected that if the inner loop were parallelised in a similar manner to the outer loop, then an $\mathcal{O}(\log^2 N)$ algorithm could be derived. This highlights a fundamental problem with these program identities, and the data-parallel model of computation we assume—parallelisation can only occur at a single level in

[4] Floating point addition is not associative because of rounding errors. However, like HPF's [21] fold functions we are willing to pay the price of potential instabilities in this numerical algorithm, and assume addition is associative.

a program[5].

Close examination of the series reveals each term to be similar to its predecessor. By taking advantage of this similarity, a common portion of the expression can be retained from one iteration of the loop to the next, and the linear complexity associated with the product can be eliminated. Starting with the original *pH* definition `sinI` (Step 0), a conventional imperative-style program transformation can be performed to produce an improved $\mathcal{O}(N)$ *sequential* algorithm.

```
sinI x n
 =let pow = x; s = 0.0; top = 1; bot = 1
  in for i <- [0..n] do
       next pow=pow * x * x
       next s  =s+((pow* toFloat top) /
                   (toFloat ((2*i+1)*bot)))
       next top=top * (2 * (i+1) -1)
       next bot=bot * (2 * (i+1))
     finally s
```

Step 2: Imperative munging

A cursory investigation of the types of the loop-range and the body of the loop reveal that none of the program identities developed so far are applicable as they rely upon an associative operator, which by definition must have a type of the form $\alpha \rightarrow \alpha \rightarrow \alpha$. The integer type used as the loop-counter and the tuple of integer and floating point numbers used in the body of the loop means that no associative next-function exists for loops of this form as the types don't 'fit' together.

One solution to this problem is to abstract part of the state of the body of the loop outwards. A way of achieving this is to perform *induction-variable elimination* [1]. The idea is to infer if any of the changes to a subset of the states in the body of the loop occur in a lock-step manner (the induction variables). When there are two or more induction variables in a loop, it may be possible to remove all but one of them by abstracting part of the loop's state outwards. We use a generalised scheme that abstracts part of the state outside the loop, whether the state is an induction variable or not. The trick is to split the loop in two, but ensure that the abstracted loop remembers all of its intermediary states—a parallel $\mathcal{O}(\log n)$ scan accomplishes this. All the values of the states in the scanned list are then 'zipped' together with the original loop-range so that each of the values can be used by an expression inside the remaining part of the fold-left.

Although the idea is quite simple, we need to identify which part of the original loops state should be abstracted out. We are trying to make the types of the loop range and the state of the body of the loop the same. This is achieved by abstracting just those parts which make the zipped range, and the new state in the body of the loop the same; we have to be aware that variables may become free. A topological sort of the variables is used to ensure strongly connected groups of states are lifted out as a whole. However, after applying the fold-map fusion law the type of the body of the loop may change. For example, looking back at step 2, the type of the states in the body of the loop is `(Float, Float, Int, Int)`, and the loop range is an `Int`. Abstracting all but the state `s` out of the loop produces a zipped loop range that is susceptible to further simplification.

```
sinI x n
 =let
    s =0.0
    ps=scanl (\p i->p*x*x) x [0..n]
    ts=scanl (\t i->t*(2*(i+1)-1)) 1 [0..n]
    bs=scanl (\b i->b*(2*(i+1))) 1 [0..n]
   in for (i,pow,top,bot) <-
            (zip4 [0..n] ps ts bs) do
        next s = s+((pow * toFloat top) /
                    (toFloat ((2*i+1)*bot)))
      finally s
```

Step 4: All out

The body of the loop now has the form required by the fold-map fusion law. Equation 7 defines the relationship between the body of the loop and the function required by the law.

$$\begin{aligned} &\lambda s\ i \rightarrow s + \tfrac{pow \times top}{(2\times i+1)*bot} \\ &\quad \equiv \lambda s\ i \rightarrow s\ \oplus\ f\ i \\ &\quad \text{iff}\ s \oplus t\ = s + t \\ &\quad\quad f\ i\quad = \tfrac{pow\times top}{(2\times i+1)*bot} \end{aligned} \tag{7}$$

Applying the law transforms the fold-left into a form where the first duality theorem is applicable. Parallelisation isn't finished as the scans introduced by moving part of the state outside the loop need to be transformed into a parallelisable form. Luckily, the first duality theorem and the fold-map fusion law are applicable to scan computations as well as folds. However, the first use of scan poses a problem as the identity of multiplication is not used as the starting value of the scan. Program identity 8 is taken from Bird & Wadler [4, page 125], and follows from the definition of fold-left. This identity forms the basis of the analogous identity 9 on scans.

$$foldl\ (\otimes)\ (x \oplus y) \equiv (x\oplus) \circ foldl\ (\otimes)\ y \tag{8}$$

$$scanl\ (\otimes)\ (x \oplus y) \equiv map\ (x\oplus) \circ scanl\ (\otimes)\ y \tag{9}$$

[5] Some languages like Blelloch's NESL [5] are based around such forms of nested parallelism. Opposed to making a choice at which level parallelisation should occur in a nested expression, we take the conventional approach in such situations of flattening the computation into a single loop.

The first use of scan in `sinI` (Step 4) is an instance of program identity 9, where the expression $x \times 1.0$ has the form $x \oplus y$. As 1.0 and $\times$ form a monoid, identity 9 and the fold-map fusion law can be used to transform the function into the form shown in `sinI` (Step 5), which completes the parallelisation of inverse sine. The original version of the problem is $\mathcal{O}(N^2)$, but can be trivially made linear. This improved linear algorithm can then be transformed into a parallel $\mathcal{O}(\log N)$ algorithm.

```
sinI x n
 = foldl (+) 0.0
         (zipWith4 f [0..n] pows tops bots)
 where
   pows =map (x*)
             (scanl (*) 1.0
                    (map (\i->x*x) [0..n]))
   tops =scanl (*) 1
                (map (\i->2*(i+1)-1) [0..n])
   bots =scanl (*) 1
                (map (\i->2*(i+1))   [0..n])
   f i pow top bot =(pow * toFloat top) /
                    (toFloat ((2*i+1)*bot))
```

Step 5: Finished

7 Category theory and monads

Although the program identities used in the prior sections are straightforward, as was seen to be the case in the inverse sine example, the application of the identities in even a relatively simple example can be troublesome. Instead of writing programs which are inferred to be vectorisable, the aim of the rest of the paper is to provide a 'sub-programming' language in which all well-formed and well-typed programs are vectorisable. In the next section a model inspired by monads is presented that enables the program identities developed so far to be incorporated into an object that looks like a monad (for more worked examples of the program identities, and an extended introduction to category theory see [9]).

The principle underlying Moggi's[16] work on monads and the computational λ-calculus is the distinction between simple data-valued functions and functions that perform computations. A data-valued function is one in which the value returned by the function is determined solely by the values of its arguments. In contrast, a function that performs a *computation* can encompass ideas such as side-effects or non-determinism, which implicitly produce more results as a consequence of an application of the function than the result explicitly returned.

Given the objects A in the category $\mathcal{C}$, and the endofunctor[6] $T : \mathcal{C} \to \mathcal{C}$, Moggi's work on monads views the endofunctor T as a mapping between all the objects $Obj(\mathcal{C})$ of the category $\mathcal{C}$ which are to be viewed as the set of all values of type τ, to a corresponding set of objects $T(Obj(\mathcal{C}))$ which are to be interpreted as the set of computations of type τ.

Now consider for each morphism $f : A \to T(B)$ a new morphism $f^* : T(A) \to T(B)$, where $\_^*$ is the "extension" operator that lifts the domain A of the morphism f to a computation $T(A)$. In the context of computations, f is a function from values to computations, whereas f^* is a function from computations to computations. The expression $g^* \circ f$, where $f : A \to T(B)$ and $g : B \to T(C)$ is interpreted as applying f to some value a to produce some computation $f\ a$; this computation is evaluated to produce some value b, and g is applied to b to produce a computation as a result.

The Kleisli triple $(T_{obj}, \eta, \_^*)$ is defined as the restriction of the endofunctor T to objects, the extension operator $\_^*$, and a natural transformation $\eta : id_{\mathcal{C}} \to T$, where $id_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ is the identity functor for category $\mathcal{C}$. In the context of computations, η can be thought of as an operator that includes values into a computation. For the triple to be well formed, the following laws are required to hold; a pictorial presentation is given in the commuting diagrams below, where $h : A \to T(A)$.

$$
\begin{array}{lll}
\text{Left Unit:} & f^* \circ \eta_A & \equiv f \\
\text{Right Unit:} & \eta_A^* \circ h & \equiv id_{T(A)} \circ h \\
 & \eta_A^* & \equiv id_{T(A)} \\
\text{Associativity:} & (g^* \circ (f^* \circ h)) & \equiv (g^* \circ f)^* \circ h \\
 & g^* \circ f^* & \equiv (g^* \circ f)^* \\
 & \multicolumn{2}{l}{\text{by assoc. of } \circ \text{, and eliding } h.}
\end{array}
$$

$$
\begin{tikzcd}
A \arrow[d, "\eta_A"'] \arrow[dr, "f"] & \\
T(A) \arrow[r, "f^*"] & T(B)
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "\eta_A"] \arrow[d, "\eta_A"'] \arrow[dr, "\eta_A"] & T(A) \arrow[d, "\eta_A^*"] \\
T(A) \arrow[r, "id_{T(A)}"] & T(A)
\end{tikzcd}
$$

in category $\mathcal{C}$ in category $\mathcal{C}$

The Kleisli triple can be thought of as a different syntactic presentation of a classical category theory monad, as there is a one-to-one correspondence between a Kleisli triple and a monad (see [15, page 143] for a proof).

The use of monads in the functional programming community bears a closer resemblance to Kleisli triples, than classical monads. Wadler [24] adapted Moggi's ideas of using monads to struc-

[6] i.e., a functor with a mapping to and from the same category.

ture the semantics of computations into a tool for structuring functional programs. The Kleisli triple $(T_{obj}, \eta, \_^*)$ can be defined in a functional language by the Wadler-ised triple $(M, unit, \star)$, where M is a type constructor, $\star$ is a function of type $M\alpha \rightarrow (\alpha \rightarrow M\beta) \rightarrow M\beta$, and $f \star g$ is the same as $g^* \circ f$ of the Kleisli triple. The natural transformation η can be modelled as a polymorphic function as it can be thought of as a family of morphisms from each object in a category to objects in another category (the components of the natural transformation). A natural transformation is therefore *similar* to a polymorphic function, and as a consequence η is written as the polymorphic function *unit* of type $\alpha \rightarrow M\ \alpha$. The laws required by the Kleisli triple can now be recast as:

Left Unit:	$unit\ a \star (\lambda b \rightarrow n) \equiv n[a/b]$
Right Unit:	$m \star (\lambda b \rightarrow unit\ b) \equiv m$
Associativity:	$m \star ((\lambda a \rightarrow n) \star (\lambda b \rightarrow o))$ $\equiv (m \star (\lambda a \rightarrow n)) \star (\lambda b \rightarrow o)$

8 A vectorisation monad

A model of stream based output can be defined in terms of Moggi's side-effecting monad [16, example 3.6, page 11]. In a simplified scenario where the output stream is a list of characters (a Landin-stream [12]), the monad $(IO, unit, bind)$ is used, where `bind` is the Haskell identifier that represents $\star$ of the previous section:

```
type IO a = String -> (a,String)

unit :: a -> IO a
unit x = \s -> (x,s)

bind :: IO a -> (a -> IO b) -> IO b
l `bind` r = \s -> let (res,s') = l s
                   in r res s'

print :: String -> IO ()
print str = \s -> ( (), s ++ str)
```

Step 0: Character stream based IO monad

The monad operations are augmented with a `print` computation that outputs its string argument as a 'side-effect' onto the output stream and delivers `()`. In the context of category theory, given the Kleisli triple $(T, \eta, \_^*)$ in the category $\mathcal{C}$, functions like `print` are the morphisms $f : A \rightarrow T(B)$, the set of all such morphisms is the hom-set $\mathcal{C}(A, T(B))$. If the monad is to be interpreted as an abstract data-type, then this hom-set of morphisms forms an interface to the type as it requires 'inside knowledge' of the representation of the monad.

A monadic for-loop is defined to have the structure `for` *ctr* `<-` *range* `do` *body*, where *body* is a computation of type `IO ()`. The result returned by a monadic loop expression will always be `()`, but the state of the for-loop is changed by side-effects during successive iterations of the loop. The state of a *pH* for-loop is represented by `next` bindings. In contrast, a monadic loop hides the state, which makes it possible to strait-jacket any interactions with the state such that the *first duality theorem* and the *fold-map fusion laws* can be satisfied, therefore making vectorisation possible. The function `helloWorlds` (Loop 1) is an example of a monadic for-loop. The function takes a numeric argument n, and performs n iterations of a for-loop printing the string `"Hello World"` followed by printing the value of the loop counter by side-effecting the output stream:

```
helloWorlds :: Int -> IO ()
helloWorlds n
  = for i <- [1..n] do
      print "Hello World "  `bind` (\ ()->
      print (show i)                )
```

Loop 1: An example for-loop

In a similar vein to the translation of a *pH* for-loop, a monadic loop is translated into a fold-left computation. The translation is relatively straight forward: (1) the body of the loop is modelled by a function that is parameterised on the loop counter; (2) a fold-left of the function $\lambda s\ i \rightarrow (s$ `` `bind` `` $(\lambda() \rightarrow bodyFn\ i))$ is performed where $bodyFn$ is bound by the function that represents the body of the original loop; (3) during the first iteration of the loop, the 's' argument of the folded lambda expression represents a computation that encapsulates the state on entry to the loop; (4) during the k^{th} iteration of the loop, the argument 's' encapsulates the state of all the previous $k-1$ iterations; (5) the initial state of the for-loop is a 'do-nothing' computation represented by `unit ()`.

```
helloWorlds :: Int -> IO ()
helloWorlds n
  = foldl (\s i->s `bind` (\ ()->bodyFn i))
          (unit ()) [1..n]
  where
    bodyFn :: Int -> IO ()
    bodyFn i
      = print "Hello World " `bind` (\ ()->
        print (show i)              )
```

Loop 2: The loop expressed as a fold-left

8.1 From monads to monoids

The flaw in the translation of a monadic for-loop into a fold-left is that the monad operation *bind* used as the folded function has the type $M\ \alpha \rightarrow (\alpha \rightarrow M\ \beta) \rightarrow M\ \alpha$ and is therefore not associative. One solution to this problem is not to use monads, but use a structure that is very similar, and can be used to achieve the same operational behaviour—re-enter monoids!

As each monad operation in the running example returns the same value `()`, and there is never a situation in which anything other than the computation delivering `()` can be returned (i.e., the dreaded bottom[7]), then all the `()`'s can be elided from the program. Removing the parameterisation from the type constructor `IO`, and changing the monad operations accordingly produces:

```
type IO = String -> String

unit :: IO
unit = \s -> s

bind :: IO -> IO -> IO
l `bind` r = r . l

print :: String -> IO
print str = \s -> s ++ str
```

Step 1: Character stream IO monoid and `print`

The reason why this transformation is possible is the extension operator $\_^*$ of the Kleisli triple isn't needed in this example. As $f \star g$ is syntactic sugar for $g^* \circ f$, we can note that all the functions used with the monad belong to the hom-set $\mathcal{C}(A, T(\{()\}))$. Because we aren't interested in the result of f (because we know it is $()$), then there is no need to use the Kleisli star to lift the argument of the function g so that it picks up the known object `()`. Given the Wadler-ised triple $(T, unit, \lambda f\ g \rightarrow f\ \tilde{\circ}\ g^*)$, we create the monoid $(T', \tilde{\circ}, id)$, where T' is a perturbation of the type T which is no longer parameterised.

This monoid is very similar to the monoid $(ShowS, \circ, id)$ that is used for printing in Haskell [11]. This output monoid is used to ensure a $\mathcal{O}(1)$ when printing data-structures such as trees, and not the quadratic complexity usually associated with using $+\!\!+$ as the compositional printing operator (see [10]). The monoid used here utilises the opposite behaviour as `print` injects the string to be printed onto the end of the output stream—the complexity of the `helloWorlds` function is therefore $\mathcal{O}(N^2)$ where N is the number of iterations of the for-loop. Using the monoid shown in `IO` (Step 1), the definition of `helloWorlds` can be transformed into:

```
helloWorlds :: Int -> IO
helloWorlds n
  = for i <- [1..n] do
      print "Hello World "  `bind`
      print (show i)
```

Loop 3: Using the a monoid

This can be translated into a fold-left computation in which the lambda expression ($\lambda s\ i \rightarrow s$ `` `bind` `` $bodyFn\ i$) is folded down all the values of the loop range. As this lambda expression is now an instance of the lambda expression required by the *fold map fusion law*, then `helloWorlds` (Loop 3) can be transformed into:

```
helloWorlds :: Int -> IO
helloWorlds n
  = foldl bind unit (map bodyFn [1..n])
    where
      bodyFn i =print "Hello World " `bind`
                print (show i)
```

Loop 4: fold-left and the fold-map fusion law.

It would seem that vectorisation is complete as `bind` and `unit` form a monoid. Unfolding the definitions of `bind` and `unit` into `helloWorlds` (Loop 4) reveals the fold-left to be nothing more than an instance of the *leaky fold-left law*, rendering vectorisation futile—a further transformation is required!

8.2 An historical aside

A monoid of this form (step 1) is the essence of the state monad. Monads are typically equated with single-threadedness, and are therefore used as a technique for incorporating imperative features into a purely functional language. Category theory monads have little to do with single-threadedness; it is the sequencing imposed by composition that ensures single-threadedness. In a Wadler-ised monad this is a consequence of bundling the Kleisli star and flipped compose into the `bind` operator. There is nothing new in this connection. Peter Landin in his Algol 60 paper [12] used functional composition to model semi-colon. Semi-colon can be thought of as a state transforming operator that threads the state of the machine throughout a program. The work of Peyton-Jones and Wadler [20] has turned full circle back to Landin's earlier work as their use of Moggi's sequencing monad enables real side-effects to be incorporated into monad operations such as `print`. This is similar to Landin's implementation of his sharing machine where the *assignandhold* function can side-effect the store of the sharing machine because of the sequencing imposed by functional composition. Landin defined that "*Imperatives are treated as null-list producing functions*"[8]. The *assignandhold* imperative is subtly different in that it enables Algol's compound statements to be handled. The function takes a store location and a value as its argument, and performs the assignment to

[7] A non-terminating function whose type is `IO ()` is different from a *computation* that returns $\perp$. The distinction arises because of the lifted type used in the implementation of the monad. Because $\perp$ is different from a tuple containing $\perp$, then all we are guaranteeing is that the 'result' part of the tuple will never be $\perp$.

[8] In Landin's paper, `()` is the syntactic representation of the empty list and not the unit.

the store of the sharing machine, returning the value assigned as a result of the function. Because Landin assumed applicative order reduction, the $\mathbf{K}-$combinator[9] was used to return `()`, and the imperative was evaluated as a side effect by the unused argument of the $\mathbf{K}-$combinator. Statements are formed by wrapping such an imperative in a lambda expression that takes `()` as an argument. Two consecutive Algol-60 assignments would be encoded in the lambda calculus as:

Algol 60	Lambda Calculus
`x:= 2;`	$((\lambda() \rightarrow \mathbf{K}\ ()\ (assignandhold\ x\ 2))\ \tilde{\circ}$
`x:= -3;`	$(\lambda() \rightarrow \mathbf{K}\ ()\ (assignandhold\ x\ (-3))))\ ()$

By using a lambda with `()` as its parameter, `()` can be thought of as the "state of the world" that is threaded throughout a program by functional composition.

8.3 Making the leaky fold-left law work

In a parallel implementation of a loop, what the previous sections have taught us is that the fold-map fusion law and the first duality theorem are crucial in the transformation into a vectorisable fold-left.

Given the monoid $(M', bind, unit)$, and the set of functions in the hom-set $\mathcal{C}(A, M(\{()\})$ (in the running example, this set of functions is the singleton set containing `print`), then we require that there exists a function g of type $\alpha \rightarrow M'$ that models *all* the functions in the hom-set.

The composition of instances of the function g can be used to create new functions that can be used in the body of the loop—e.g., $g\ v_1$ `` `bind` ``$g\ v_2$. To successfully vectorise a loop where the body is a computation created from the compositions of the function g, then the amalgamated function must be an instance of the lambda expression $\lambda s\ i \rightarrow s \oplus f\ i$ required by the fold-map fusion law. We define g to be $g\ v = \lambda s \rightarrow s \oplus f\ v$ where the operator $\oplus$ and f are functions specific to the definition of the hom-set $\mathcal{C}(A, M(\{()\}))$, and $\oplus$ is associative. The result of unfolding the definition of `bind` into the composition of two instances of g produces:

$$
\begin{array}{lll}
 & g\ v_1\ \text{`bind`}\ g\ v_2 & \\
\Rightarrow & g\ v_2\ \circ\ g\ v_1 & \text{unfolding bind} \\
\Rightarrow & (\lambda s \rightarrow s \oplus f\ v_2) \circ & \\
 & (\lambda s \rightarrow s \oplus f\ v_1) & \text{unfolding } g \\
\Rightarrow & \lambda s \rightarrow (s \oplus f\ v_1) \oplus f\ v_2 & \text{definition of } \circ \\
\Rightarrow & \lambda s \rightarrow s \oplus (f\ v_1 \oplus f\ v_2) & \text{assoc. of } \oplus
\end{array}
$$

[9] $\mathbf{K} = \lambda x\ y \rightarrow x$

As can be seen from the last transformation above, because of the associativity of $\oplus$, any combination of the compositions of g produces a function that is also an instance of the lambda expression required by the fold-map fusion law. Using the definition of g, the monoid $(M', bind, unit)$ is converted into $(M'', \oplus, \oplus_{id})$, and the set of monad operations modelled by g is replaced by f.

This transformation is repeatedly applied to the monoid, until M'' is a non-functional type, or the process fails. The relationship between the monad used before and after this transformation is a *monoid homomorphism*—i.e., if hom is the monoid homomorphism, then the following holds:

$$
\begin{array}{ll}
hom(unit) & \equiv \oplus_{Id} \\
hom(x\,\text{`bind`}\,y) & \equiv hom(x) \oplus hom(y)
\end{array}
$$

The monoid (Step 1) in the running example only has one operation in the associated hom-set. This function `print` can be coerced into the form required by g as shown in eqn 10.

$$
\begin{array}{ll}
\texttt{print} & \equiv g \\
\lambda str\ s \rightarrow s +\!\!+\ str & \equiv \lambda v\ s \rightarrow s \oplus f\ v \\
 & \text{iff } x \oplus y\ = x +\!\!+\ y \\
 & \quad f\ v\ \ \ = v
\end{array}
\tag{10}
$$

From the equation, the monoid $(String, +\!\!+, [])$ can be used as the new definitions of *unit* and *bind*, and `print` becomes the identity function, completing vectorisation.

```
type IO = String

unit :: IO
unit = []

bind :: IO -> IO -> IO
l `bind` r = l ++ r

print :: String -> IO
print str = str
```

Finished: the final monoid

The function `helloWorlds` (Loop 4) can be left syntactically unchanged, and unlike the result of §5, because the monoid contains a non-functional type (i.e., *String*), the fold can be truly implemented in parallel.

9 Conclusions

What this paper has shown is that a small collection of laws can be used to transform imperative for-loops into a form that can be implemented in terms of fold and scan. Instead of a compiler transforming a for-loop using these laws, we have used monads to develop a constrained programming language in which only vectorisable programs can be expressed. All that is required of a compiler is that it performs a minor syntactic translation of a for-loop into the vectorisation monad. This operates together with an ordinary type inference algorithm that is used to determine if the loop is vectorisable. The monad guarantees that a loop can be vectorised, and a proof for the monad can be given once, *independently of the compiler*.

10 Acknowledgements

We would like to thank Richard Howarth, Paul Boca, and the referees for their numerous comments on the paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, techniques and tools.* Addison-Wesley, 1986.

[2] R. S. Bird. An introduction of the theory of lists. Technical Report PRG-56, Oxford University, 1986.

[3] R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

[4] R. S. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall International, 1988.

[5] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CS-93-129, Carnegie Mellon Univ., April 1993.

[6] P. Henderson. *Functional Programming: Application and implementation.* Prentice-Hall International, 1980.

[7] J. M. D. Hill. The *aim* is laziness in a data-parallel language. In *Glasgow functional programming workshop*, Workshops in computing. Springer-Verlag, 1993. Available by FTP from ftp.dcs.qmw.ac.uk in /pub/cpc/jon_hill/aimDpLaziness.ps.

[8] J. M. D. Hill. Vectorizing a non-strict functional language for a data-parallel "Spineless (not so) Tagless G-Machine". In *Proc. of the 5th international workshop on the implementation of functional languages*, Nijmegen, Holland, September 1993. Available by FTP.

[9] J. M. D. Hill and K. M. Clarke. Parallel Haskell: The vectorization monad. Technical Report 658, QMW CS, December 1993. Available by FTP.

[10] P. Hudak and J.H. Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, May 1992.

[11] P. Hudak, S. L. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *SIGPLAN Notices*, March 1992.

[12] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Communications of the ACM*, 8(2):89–101, February 1965.

[13] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[14] J. Launchbury. Lazy imperative programming. In *Proc. ACM Sigplan workshop on State in Programming Languages*, pages 46–56, June 1993.

[15] S. Mac-Lane. *Categories for the working mathematician.* Springer-Verlag, 1971.

[16] E. Moggi. Computational lambda-calculus and monads. In *IEEE symposium on Logic in computer science*, 1989.

[17] E. Moggi. An abstract view of programming languages. Technical Report LFCS-90-113, Univ. of Edinburgh, April 1990.

[18] R . S. Nikhil, Arvind, and J. Hicks. pH language proposal (preliminary). Sept 1993.

[19] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, 1987.

[20] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *ACM POPL*, 1993.

[21] Rice University, Houston, Texas. *High Performance Fortran Language Specification*, May 1993. Version 1.0.

[22] J. Sparud. Fixing some spaces leaks without a garbage collector. In *FPCA*, 1993.

[23] Jean-Pierre Talpin. *Aspects théoriques et praticques de l'inférence de type et d'effets.* PhD thesis, L'Ecole nationale supérieure des mines de Paris, May 1993. (In English).

[24] P. Wadler. Comprehending monads. In *ACM Conference on Lisp and functional programming*, pages 61–78, June 1990.