

# Parallel Haskell: The vectorisation monad

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

## Abstract

It has long been known that some of the most common uses of `for` and `while`-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 **parallel Haskell** [27] (figure 1) as a vehicle in which imperative functional programs will be expressed. Surprisingly, incorporating imperative features into a purely functional language has become an active area of research within the functional programming community [30, 24, 36, 20]. One of the techniques gaining widespread acceptance as a model for imperative functional programming is monads [38, 37, 26]. 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 multi-processor platform, or by a parallel implementation of graph reduction such as that offered by the GRIP [12] machine.

**Keywords:** Data parallelism; Monads; Bird-Meertens Formalism; Program transformation; Category theory; Vectorisation; Imperative functional programming.

## 1 Background

“*‘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 [22].

The work presented in this paper uses a model of parallel computation based upon the Bird Meertens Formalism [6]. To make things simple, all parallelism is expressed in terms of a  $\mathcal{O}(1)$  complexity *map* function, and  $\mathcal{O}(\log N)$  complexity *fold* and *scan* functions which both require an associative operator. 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 our earlier paper [15] 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 (i.e. no pattern matching on lists), it is possible to give the impression of using lists, whilst actually implementing these list like objects on-top of the *infinite* PODs of DPHaskell [15, 14]—for more information on how PODs are implemented, and how the fabled complexities are achieved on a SIMD machine, see [16, 17].

---

\*Email: {Jon.Hill,keithc}@dcs.qmw.ac.uk

## 2 Parallelising simple loops

It is part of folklore that programs expressed as for-loops can be re-written using tail recursion (for example, [21, 13]), although the the functional programming community has concentrated on the reverse translation as an optimisation technique [35, 29]. As an example of such a translation, the iterative definition of the factorial function of figure 2 can be expressed as the tail recursive function of figure 3. Unfortunately, such a translation does not provide the right foundations for the parallelisation of for-loops—replacing an inherently sequential for-loop with a sequential tail recursive function doesn't get us very far. What is interesting however, is the resulting tail recursive function is nothing more than 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 can be rewritten using translation scheme TPH of figure 5 as a fold-left of  $\ominus$  (see figure 4). We term  $\ominus$  the next-function of the for-loop. As we have already mentioned, a parallel fold on PODs can be implemented with  $\mathcal{O}(\log N)$ , *if the function being folded has a  $\mathcal{O}(1)$  and is known to be 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* [5] the fold-left can be safely transformed into a fold-right, or more interestingly a parallel fold as the initial value of the fold-left can be dropped.

$$\begin{aligned} \text{foldl } (\ominus) \ x \ xs &\equiv \text{foldr } (\ominus) \ x \ xs && \text{Identity 1: "First} \\ &\equiv \text{foldPod } (\ominus) \ xs && \text{duality theorem"} \\ &\text{iff } \ominus \text{ and } x \text{ form a monoid; and } xs \text{ is finite.} \end{aligned}$$

In the rest of the program identities, if this theorem is used to transform a fold-left into a right or parallel fold, then the side condition requiring finite lists/PODs is ignored. The justification for this is that in all situations that a fold-left computation yields a non bottom value, a fold-right will give the same result. However, there may be situations where an otherwise diverging fold-left computation produces a result when a fold-right is used. It is taken for granted that this is a good thing, in the same way that we believe normal order reduction to be superior to applicative order reduction.

## 3 Does it really have to be associative?

Parallelising for-loops is rather bland when the next-function  $\ominus$  is associative, because the types of 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 to be the same*. This is rather unfortunate, as the iterative numerical algorithms we intend to parallelise typically have a range over a subset of the integers, and the loops state will be a mixture of floating point and integer values. In the situation where the body of the loop contains a single state, 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 loops 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<sup>1</sup> [4].

<sup>1</sup> 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

---

(pattern)	<i>pat</i>	↦	<b>next</b> <i>var</i>	Loop variable
(expression)	<i>exp</i>	↦	<b>while</b> <i>exp</i> <b>do</b> { <i>pat</i> = <i>exp</i> [;]} <sup>+</sup> <b>finally</b> <i>exp</i>	While-loop
			<b>for</b> <i>pat</i> <- <i>exp</i> <b>do</b> { <i>pat</i> = <i>exp</i> [;]} <sup>+</sup> <b>finally</b> <i>exp</i>	For-loop
			<b>next</b> <i>var</i>	Loop variable

---

Figure 1: Proposed syntax *extensions* for *pH*

---

---

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

Figure 2: Factorial in *pH*

```
>fact n = f 1 1
>   where f acc i
>         | i >= n   =acc
>         | otherwise =f (acc*i) (i+1)
```

Figure 3: A tail recursive factorial in Haskell

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

Figure 4: Factorial in terms of `foldl`


---


$$\text{foldl } (\lambda s i \rightarrow s \oplus f i) x \equiv \text{foldl } (\oplus) x \circ \text{map } f$$

*Identity 2: "fold-map fusion law"*

An added bonus of this transformation is that an otherwise unparallelisable for-loop due to the lack of an associative next-function, can be converted into a parallelisable form. For example, the for-loop shown in the left of figure 6 can be transformed into the definition shown on the right in which an associative operator has been exposed to the fold.

Comparing this law to a conventional optimisation on loops, the inverse of loop fusion [1] is being performed in a scenario that would normally be detrimental to performance in a sequential environment as the overheads associated with evaluation of a loop will be incurred twice. However, because of the differing complexities of fold and map in a data-parallel language, the law is an optimisation technique as  $f$  will only be evaluated once in the transformed program, compared to  $\log N$  times if the fold-map fusion law were not applied.

## 4 A warning to the wise...

One of the things that becomes apparent from the everyday use of a non-strict language is the unpleasant operational phenomenon of space-leaks [29, 34]. Any functional programmer worth his or her salt knows that fold-left computations have a rather nasty space-leak associated with them. To illustrate this problem, if we wanted to find 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 seasoned Scheme [33] or ML [11] 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 with the previous value of the accumulator. Unfortunately 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 closures 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

term *fold-map fission* would be more appropriate in a data-parallel setting.

---


$$\text{TPH} \left[ \left[ \begin{array}{l} \text{for } i \leftarrow \text{range do} \\ \text{pat}_1 = \text{exp}_1 \\ \vdots \\ \text{pat}_n = \text{exp}_n \\ \text{finally } \text{exp}_{fin} \end{array} \right] \right] = \text{case } \left( \text{foldl} \left( \begin{array}{l} \lambda (s_1, \dots, s_m) i \rightarrow \\ \text{let TPAT}[\text{pat}_1] = \text{TPH}[\text{exp}_1] \\ \vdots \\ \text{TPAT}[\text{pat}_n] = \text{TPH}[\text{exp}_n] \\ \text{in } (v'_1, \dots, v'_m) \\ (v_1, \dots, v_m) \text{ range} \\ \sim (v_1, \dots, v_m) \rightarrow \text{exp}_{fin} \end{array} \right) \right) \text{ of}$$

where  $\{(v_1, v'_1), \dots, (v_m, v'_m)\} = \{(v, v') \mid \text{next } v \in \{\text{pat}_1, \dots, \text{pat}_n\}\}$

$\text{TPH}[\text{next var}] = \text{var}'$  where  $\text{TPH}[\text{exp}]$  is the identity translation for other expressions.

$\text{TPAT}[\text{next var}] = \text{var}'$  where  $\text{TPAT}[\text{pat}]$  is the identity translation for other patterns.

Figure 5: Translating a *pH* for-loop into a fold-left

---

```

>approxSquare n
> = let acc = 0.0
>   in for i <- [1..n] do
>     next acc=acc + 1.0/fromInteger i
>     finally f

```

```

>approxSquare n
> = foldl
>   (+) 0.0
>   (map (\i->1.0/fromInteger i)
>    [1..n])

```

---

Figure 6: Transforming a non-associative loop into a form that is parallelisable

“dragging” if the folded function is known to be strict<sup>2</sup>, because some *real* computation could occur in the accumulated parameter. But should we rely on clever compilers?

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!

$$\text{foldl } (\oplus) x xs \equiv \text{foldr } (\lambda s i \rightarrow (\lambda y \rightarrow y \oplus i) \circ s) (\lambda y \rightarrow y) xs x \quad \text{Identity 3: "Leaky fold-left law"}$$

The *leaky fold-left law* 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<sup>3</sup>:

$$\text{foldl } (\oplus) x xs \equiv \text{foldr } (\delta) (\lambda y \rightarrow y) (\text{map } (\tilde{\oplus}) xs) x \quad \text{Identity 4}$$

The program identity is now an instance of the first duality theorem (because  $\delta$  and the identity function form a monoid), and the fold-right can be safely replaced by a left or parallel fold. Before continuing it may be appropriate to expand on this rather bizarre program identity. 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 are folded together using  $\delta$ . The effect of this fold is to create a closure that is equivalent to the function produced by unrolling  $\tilde{\oplus}$  at runtime. For example, given the list  $[1, 2, 3]$ , the closure  $(\oplus 3) \circ (\oplus 2) \circ (\oplus 1)$  is created;
3. The unrolled closure is then applied to the initial state of the accumulator of the original fold, 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 fold of the compose function, and a map of the original function that we wanted to fold. In [16] we show how map can be implemented with a  $\mathcal{O}(1)$ , 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  $\mathcal{O}(\log N)$ . It would seem that the exercise is of theoretical interest only—or is it? All will be revealed in section 8 on monads!

## 5 Example: Testing if a number is prime

The program identities developed so far are used in the parallelisation of a simple function that determines if a natural number is prime. Looking at the definition of `isPrime:Step 0` below, the

---

<sup>2</sup>or as is the case with Miranda, a strictness annotation is used that explicitly forces the evaluation in the accumulator.

<sup>3</sup>As in [6],  $\tilde{\oplus}$  is used to represent the flipped version of the operator  $\oplus$ , i.e.,  $\lambda x y \rightarrow y \oplus x$

first step in the parallelisation process is to determine if the body of the for-loop can be expressed by an associative function. If the body is found to be associative, then by the first duality theorem parallelisation of the for-loop would be complete as it could be implemented by a parallel fold.

```
> isPrime :: Int -> Bool
> isPrime n = let s = True
>             in for i <- [2 .. truncate (sqrt (fromInteger n))] do
>                 next s = if n `rem` i == 0
>                         then False
>                         else s
>             finally t
```

Step 0: The Start

Unfortunately, the conditional in the body of the loop is rather off-putting. Faced with simple arithmetic operations such as addition and multiplication, the associativity of the body is quite simple to determine. However, what is the associativity of an expression that contains a conditional? This issue is side-stepped, as in this context the conditional can be converted into the conjunction of logical operators whose associativity is easy to determine. Using translation scheme TPOD, in conjunction with the equivalence  $\text{if } x \text{ then False else } y \equiv \neg x \wedge y$ , the definition of `isPrime:Step 0` is transformed into the following:

```
> isPrime n
> = foldl (\s i -> not (n `rem` i == 0) && s) True
>       [2 .. truncate (sqrt (fromInteger n))]
```

Step 1: Conditional removal and TPOD

The function used to model the body of the loop now has the structure required by the fold-map fusion law, as shown by equation 1.

$$\lambda s i \rightarrow \neg(n \text{ rem } i = 0) \wedge s \equiv \lambda s i \rightarrow s \oplus f i \quad \text{iff} \quad \begin{array}{l} s \oplus t = t \wedge s \\ f i = \neg(n \text{ rem } i = 0) \end{array} \quad (1)$$

Applying the program identity transforms `isPrime:Step 1` into a form in which the monoid of the flipped `&&` operator and boolean `True` are used in the fold. This fulfils the side conditions of the first duality theorem, and the definition of fold-left can be replaced by a parallel fold, completing the parallelisation of the function. Contrasting the complexities of the original and the transformed function, to test if the number  $N$  is prime, the *pH* version of `isPrime` has a  $\mathcal{O}(\sqrt{N})$  complexity, compared to the  $\mathcal{O}(\log \sqrt{N})$  complexity of the transformed function.

```
> isPrime n
> = foldl (\s t -> t && s) True
>       (map (\i -> n `rem` i /= 0)
>         [2 .. truncate (sqrt (fromInteger n))])
```

Parallelsed with Step 2: help from the first duality theorem

## 6 Example: Calculating the inverse sine

Let's turn to another numerical problem, the evaluation of a series that approximates the trigonometric function  $\sin^{-1} x$  to any desired accuracy (equation 2). For a given accuracy, the motivation behind parallelising the problem 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.

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

Although there is little point in parallelising this algorithm as the series rapidly converges<sup>4</sup>, it forms an interesting case study because a naive sequential algorithm with a  $\mathcal{O}(N^2)$  leads to the derivation of naive parallel algorithm with  $\mathcal{O}(N \log N)$ . The naive sequential algorithm is then converted using standard imperative-style program transformations into an algorithm with  $\mathcal{O}(N)$ . Unfortunately the improved sequential algorithm fails to be parallelised with the identities developed so-far. This is remedied by a collection of new laws, and a parallel algorithm is finally derived with a  $\mathcal{O}(\log N)$ .

```
> 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 2 in terms of a *pH* for-loop. Notice 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 2. Removing the clutter from the loop reveals the body to be nothing more than an instance of the fold-map fusion law. The relationship between the desired lambda expression  $\lambda s i \rightarrow s \oplus f i$  and the function used to model the body of the loop is shown in equation 3.

$$\begin{aligned} \lambda s i \rightarrow s \oplus f i &\equiv \lambda s i \rightarrow s + \frac{x^{2i+1} \prod_{j=1}^i (2j-1)}{(\prod_{j=1}^i 2j) \cdot (2i+1)} \\ \text{iff } s \oplus t &= s + t \\ f i &= \frac{x^{2i+1} \prod_{j=1}^i (2j-1)}{(\prod_{j=1}^i 2j) \cdot (2i+1)} \end{aligned} \quad (3)$$

Applying the law transforms the sine function into the form shown below. Because floating-point addition and 0.0 form a monoid, 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])
>   where 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
```

TPOD and  
Step 1: fold-map fu-  
sion

If the rhetoric of the previous sections were to be believed, then the sine function would have a logarithmic complexity—it doesn't! The original function can be thought of as a nested for-loop with  $\mathcal{O}(N^2)$  as the computations of both the numerator and denominator have a  $\mathcal{O}(N)$ . As the derived parallel algorithm has a  $\mathcal{O}(N \log N)$ , it would be expected that if the inner loop were parallelised in a similar manner to the outer loop, then an algorithm with a  $\mathcal{O}(\log^2 N)$  could be derived. This highlights a fundamental problem with these program identities, DPHaskell, and vectorisation in general—parallelisation can only occur at a single level of a program, although some languages like Blleloch's NESL [7] 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.

<sup>4</sup>Only ten iterations of the series are required before the denominator overflows 32-bit integer arithmetic.

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 *sequential* algorithm with a  $\mathcal{O}(N)$ .

```
> sinI :: Float -> Int -> Float
> 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.

## 6.1 Program identities for multiple states in a loop

One solution to this 'fitting' 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], an optimisation technique traditionally used in imperative languages. The idea behind the optimisation 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 (these are 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 loops state outwards.

Here we use a generalised scheme that abstracts part of the state outside the loop, regardless if 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—our good old friend `scan` accomplishes this! All the values of the states in the scanned list/POD 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, which part(s) of the original loops state should be abstracted out? Remember that we are trying to make the types of the loop range and the state of the body of the loop the same. This balancing act can be achieved by abstracting just those parts which make the zipped range, and the new state in the body of the loop the same. Unfortunately, we have to be aware of the following:

- Care has to be taken that the variables of any states do not become free. A topological sort of the variables is required to ensure strongly connected groups of states are lifted out as a whole;
- Remember after applying the fold-map fusion law the type of the body of the loop may change;

I think an example is required! Looking back at step 2 in the transformation of the inverse sine function, the type of the states in the body of the loop is `(Float, Float, Int, Int)`, and the loop range is an `Int`. By abstracting the float and integer associated with the calculations `pow` and `top`

respectively, the types of the remaining state in the body of the loop, and the new ‘zipped’ loop range almost balance.

```
> sinI :: Float -> Int -> Float
> sinI x n
>   = let s      = 0.0; bot  = 1
>       pows = scanl (\pow i -> pow * x * x) x [0..n]
>       tops = scanl (\top i -> top * (2 * (i+1) - 1)) 1 [0..n]
>       in for (i,pow,top) <- (zip3 [0..n] pows tops) do
>         next s      = s + ((pow * toFloat top) /
>                             (toFloat ((2*i + 1) * bot)))
>         next bot    = bot * (2 * (i+1))
>       finally s
```

Step 3: Eureka scan

Unfortunately, the function modelling the body of the loop does not have the form  $\lambda s \ i \rightarrow s \oplus f \ i$  required by the fold-map fusion law, as `bot` used in the state `s` is associated with the index computation that will be abstracted out of the loop when the fold-map fusion law is applied. In general, if the body of the loop contains any dependencies between the states, then there is the possibility the fold-map fusion law cannot be applied. The solution to this new problem is to remove the offending piece of state from the body of the loop. Abstracting `bot` out of the loop in a similar manner to `pow` and `top` produces:

```
> sinI :: Float -> Int -> Float
> sinI x n
>   = let s      = 0.0
>       pows = scanl (\pow i -> pow * x * x) x [0..n]
>       tops = scanl (\top i -> top * (2 * (i+1) - 1)) 1 [0..n]
>       bots = scanl (\bot i -> bot * (2 * (i+1))) 1 [0..n]
>       in for (i,pow,top,bot) <- (zip4 [0..n] pows tops bots) 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 a form required by the fold-map fusion law, equation 4 defines the relationship between the body of the loop and the function required by the law.

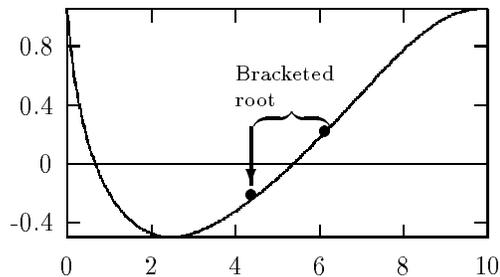
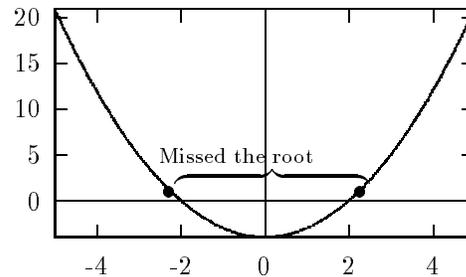
$$\begin{aligned} \lambda s \ i \rightarrow s + \frac{pow \times top}{(2 \times i + 1) * bot} &\equiv \lambda s \ i \rightarrow s \oplus f \ i \\ \text{iff } s \oplus t &= s + t \\ f \ i &= \frac{pow \times top}{(2 \times i + 1) * bot} \end{aligned} \quad (4)$$

Applying the law transforms the fold-left into a form where the first duality theorem is applicable. However, 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. Fortunately the first duality theorem and the fold-map fusion law are applicable to scan computations as well as folds. 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 5 is taken from Bird & Wadler [5, page 125], and follows from the definition of fold-left. This identity forms the basis of the analogous identity 6 on scans which can be used to solve the problem at hand.

$$foldl (\otimes) (x \oplus y) \equiv (x \oplus) \circ foldl (\otimes) y \quad \text{Identity 5}$$

$$scanl (\otimes) (x \oplus y) \equiv map (x \oplus) \circ scanl (\otimes) y \quad \text{Identity 6}$$

The first use of scan in `sinI:Step 4` is an instance of program identity 6, where the expression  $x \times 1.0$  has the form  $x \oplus y$ . As `1.0` and `*` form a monoid, identity 6 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 has a  $\mathcal{O}(N^2)$ , the transformed

Figure 7:  $f(x) = \tan((\cos \sqrt{x})^2) - 0.5$ Figure 8:  $f(x) = x^2 - 4$ 

program a  $\mathcal{O}(\log N)$ . If  $N = 1000$ , then the empirical difference is approximately one million compared to ten—not even an implementation in a functional language would have a constant of proportionality that outweighed the improved complexity measure!

```
> sinI :: Float -> Int -> Float
> sinI x n
> = let 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])
>     in foldl (+) 0.0 (zipWith4 f [0..n] pows tops bots)
>     where
>     f i pow top bot = (pow * toFloat top) / (toFloat ((2*i+1) * bot))
```

Step 5: Finished

## 7 Example: Bracketing the roots of a polynomial

The objective of our last numerical example is to parallelise a small, but non-trivial FORTRAN subroutine taken from a numerical algorithms book [32]. As will become apparent, the subroutine has an obviously parallel implementation, which the program identities successfully transform the original FORTRAN program into. A secondary objective of the exercise is to address the unresolved problem of dependencies between variables in a loop that can cause the program identities to fail.

Given the polynomial  $f(x)$ , the FORTRAN subroutine shown below “brackets” the roots of the polynomial within a specified interval. For example, given the polynomial  $f(x) = \cos(x)$  and an interval from zero to three, a root of the polynomial is  $\frac{\pi}{2}$ , and the bracket subroutine would find a pair of numbers that straddle this root.

The bracketing algorithm works by sub-dividing the interval in which bracketed roots are to be searched, and calculates the polynomial at successive sub-divisions in the interval. If the values of the polynomial at adjacent sub-divisions have a different sign, then as can be seen from figure 7 the root must lie between the two points. However, the technique is not fool proof, as can be seen from figure 8. A root can easily be missed if the size of the sub-divisions is too coarse. The technique lends itself to a data-parallel implementation as the cost of evaluating thousands of sub-divisions will *probably*<sup>5</sup> be the same as evaluating hundreds—the size of the sub-divisions can therefore be made very small. The problem with this approach is that it may be a little over the top, as with many polynomials a coarse approach may be sufficient. As [32] points out, Hammings motto “*the purpose of computing is insight, not numbers*” is particularly apt in the area of finding roots—parallel algorithms seem to rely too heavily on brute force, rather than insight into the characteristics of a particular problem.

<sup>5</sup>In the same sense as the Carlsberg lager adverts, in practice its stretching the truth!

```

SUBROUTINE Bracket(Fun,lower,upper,interval,resLows,resUps,noRes)
C fun                the polynomial to be bracketed;
C lower & upper      the interval to be searched;
C interval           the number of subdivisions in the interval;
C noRes, resLows & resUps size and result arrays of bracketed pairs;
  noRes = 0
  x     = lower
  dx    = (upper - lower) / interval
  fp    = fun(x)
DO 10 i=1,interval
  x = x + dx
  fc = Fun(x)
  IF (fc * fp .LT. 0) THEN
    noRes = noRes + 1
    resLows(n) = x - dx
    resUps(n) = x
  ENDIF
  fp = fc
10 CONTINUE
  RETURN
END

```

Step 0: FORTRAN

The first step in parallelising the subroutine `bracket:Step 0` is the conversion into *pH*. Two interesting features of the FORTRAN implementation are:

- Two arrays are used to collate the resulting bracketed roots of the polynomial. As the size of this array is unknown before the algorithm starts, book keeping is required (the variable `n`) so that the resulting arrays are filled in ascending order, and the size of the final array is returned to the calling subroutine;
- The adjacent pairs of the polynomial are calculated by a “dragging” technique, where two variables are used to represent adjacent values of the polynomial to be tested for bracketing.

Both of these features have a natural and elegant implementation in *pH*. The result array can be eliminated, and a list of bracketed points can be used that is more natural for the problem. Also, during an iteration of a *pH* for-loop it is possible to access both the value of a variable on entry and exit from a single loop iteration. This is highlighted in `bracket:Step 1` where the expression `fp` represents the old value of the polynomial at a given sub-division, and the expression `next fp` represents the value at the adjacent sub-division:

```

> bracket :: (Float -> Float) -- Function to be bracketed
>         -> Float           -- Lower bound of interval
>         -> Float           -- Upper bound of interval
>         -> Int             -- Number of subdivisions of interval
>         -> [(Float,Float)] -- List of intervals
> bracket fn lower upper interval
>   = let x   = lower
>       dx   = (upper - lower) / fromInteger interval
>       fp   = fn x
>       res  = []
>   in for i <- [1..interval] do
>     next x  = x + dx
>     next fp = fn (next x)
>     next res = if (fp * next fp < 0)
>                 then (x, next x):res
>                 else res
>   finally res

```

Step1 :  $pH$ 

The first step in the parallelisation process is to reduce the number of states used in the body of the loop. The body of `bracket:Step1` contains three state variables, we arbitrarily choose `x` for transformation:

```

> bracket fn lower upper interval
>   = let dx = (upper - lower) / fromInteger interval
>       fp  = fn lower
>       res = []
>       xs  = scanl (\x i -> x + dx) lower [1..interval]
>   in for (i,x) <- zip [1..interval] xs do
>     next fp = fn (next x)
>     next res = if (fp * next fp < 0)
>                 then (x, next x):res
>                 else res
>   finally res

```

Step 2: abstract `x`

Straight away we hit a problem. The body of the loop contains the expression `next x`, which has become free as all the values of `x` from the zipped range represent the value of the variable on *entry* to each iteration of the loop, and not the *next* value of `x` that is required. This strange dependency between a single variable stems from  $pH$ 's pedigree in the single assignment language Id [28]. The aim of these transformations is to implement a for-loop by a mixture of fold and scan operations. One of the characteristics of parallel scan is that *during* the scanning process, all the intermediary states of the scan don't really exist. This means that the consecutive values of a variable such as `x` cannot be accessed in a parallel implementation based upon scan.

However, this problem can be solved after part of the state has been abstracted outside the loop. The first thing to notice, is that all abstracted states will be "zipped" back together with the range of the for-loop. If the range is finite, then it *doesn't* matter if the data-structure abstracted outside the loop is infinite (because of the semantics of `zip`). By removing the upper bounds of the list being scanned, scan performs a potentially infinite computation. If `xs` represents all the values of `x` during each iteration of the loop, then the tail of `xs` will represent all the values of `next x` during the same iterations of the loop. Any problems arising from inconsistencies during the last iteration of the loop are eliminated by using an infinite list/POD. In the context of PODs, the tail of a POD has an obvious parallel implementation as a rightwards-shift.

```

> bracket fn lower upper interval
> = let dx = (upper - lower) / fromInteger interval
>     fp = fn lower
>     res = []
>     xs = scanl (\x i -> x + dx) lower [1..]
>     in for (i,x,next_x) <- zip [1..interval] xs (tail xs) do
>         next fp = fn next_x
>         next res = if (fp * next fp < 0)
>                     then (x, next_x):res
>                     else res
>     finally res

```

Step 3: remove `x`'s  
dependency

Next, `fp` and `next fp` are abstracted from the loop in a similar manner to `x`. Because `fp` is dependent on `next x`, when `fp` is abstracted outside the loop, all the values of `next x` are used as an argument to the scan computation that creates the `fp`'s.

```

> bracket fn lower upper interval
> = let dx = (upper - lower) / fromInteger interval
>     res = []
>     xs = scanl (\x i -> x + dx) lower [1..]
>     fps = scanl (\fp (i,x) -> fn x) (fn lower) (zip [1..] xs')
>     in for (i,x,next_x,fp,next_fp) <- zip [1..interval] xs
>                                     (tail xs) fps (tail fps) do
>         next res = if (fp * next_fp < 0)
>                     then (x, next_x):res
>                     else res
>     finally res

```

Step 4: Abstract `fp`  
and `next fp`

Using the relationship  $x : xs \equiv [x] \ddagger xs$ , the body of the for-loop is an instance of the fold-map fusion law. The relationship between the lambda expression  $\lambda s i \rightarrow s \oplus f i$  required by the law and the function used to model the body of the loop is shown in equation 5.

$$\begin{aligned}
 \lambda s i \rightarrow s \oplus f i &\equiv \lambda s (i, x, x_{next}, fp, fp_{next}) \rightarrow \begin{array}{l} \text{if } (fp \times fp_{next} \geq 0.0) \\ \text{then } [] \ddagger s \\ \text{else } [(x, x_{next})] \ddagger s \end{array} \\
 \text{iff } s \oplus t &= t \ddagger s \\
 f (i, x, x_{next}, fp, fp_{next}) &= \begin{array}{l} \text{if } (fp \times fp_{next} \geq 0.0) \\ \text{then } [] \\ \text{else } [(x, x_{next})] \end{array}
 \end{aligned} \tag{5}$$

Applying the law transforms the function into `bracket:Step 5`. Because  $\ddagger$  and  $[]$  form a monoid, then by the first duality theorem the fold-left can be safely transformed into a parallel fold. It should be noted that whenever this monoid is used in this way inside a fold computation, a flattening operation is being performed. As Hill shows in his thesis [17], a logarithmic complexity flattening operation expressed as a parallel fold is only possible when the size of the resulting list is independent of the data-structure being flattened—i.e., the inner list contains lots of empty lists. This highlights the distinction between lists/PODs which has been glossed over so far. If the fold of this monoid is applied to a parallel data-structure, should the same kind of parallel data-structure be returned by the flattening operation, or should a list be returned? In this particular scenario it doesn't matter. A fold of this monoid could be used, or a parallel flattening operation based upon communication could be used which accumulates its results within a POD and not a list. This kind of flattening is guaranteed to have a  $\mathcal{O}(\log N)$  (see [17] for more details).

```

> bracket fn lower upper interval
>   = let dx = (upper - lower) / fromInteger interval
>       xs = scanl (\x i -> x + dx) lower [1..]
>       fps = scanl (\fp (i,x) -> fn x) (fn lower) (zip [1..] xs')
>       in foldl (flip (++)) [] (map gn (zip5 [1..interval] xs xs' fps fps'))
>   where gn (i,x,next_x,fp,next_fp) = if (fp * next_fp >= 0.0)
>       then []
>       else [(x,next_x)]

```

Step 5: TPOD and fold-map fusion
----------------------------------

All that remains is to transform the scans abstracted from the loop into a parallelisable form. The scan associated with `xs` is transformed using program identity 6, whereas the second scan can be eliminated altogether. By using the fold-map fusion law in conjunction with the lambda expression  $\lambda s i \rightarrow f i$ , the scan will put the identity element at the start of the resulting list, and the list being scanned will be copied to the tail—program identity 7. The scan can therefore be replaced by a map expression, completing parallelisation.

$$\begin{aligned} \text{scanl } (\lambda s i \rightarrow f i) x &\equiv \text{scanl } (\lambda s t \rightarrow t) x \circ \text{map } f && \text{Identity 7: Useless} \\ &\equiv (x :) \circ \text{map } f && \text{Scan} \end{aligned}$$

Comparing the original and transformed functions, the FORTRAN version of the bracket algorithm has a  $\mathcal{O}(N)$  (where  $N$  is the number of sub-divisions to be made), compared to the  $\mathcal{O}(\log N)$  of the transformed function.

```

> bracket fn lower upper interval
>   = let x = lower
>       dx = (upper - lower) / fromInteger interval
>       fp = fn x
>       res = []
>       xs = map (lower+) (scanl (\x i -> x + dx) 0.0) [1..]
>       fps = (fn x) : map (\(i,x) -> fn x) (zip [1..] (tail xs))
>       in foldl (flip (++)) [] (map gn (zip5 [1..interval] xs xs' fps fps'))
>   where gn (i,x,next_x,fp,next_fp) = if (fp * next_fp >= 0.0)
>       then []
>       else [(x,next_x)]

```

Finished: tidy up scans
-------------------------

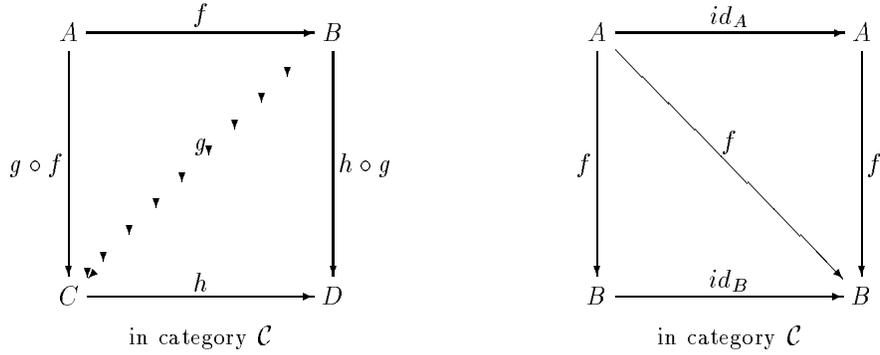
## 8 The vectorisation monad

In the remainder of this paper we tie up the loose ends of §4 “*A warning to the wise*” which developed the *leaky fold-left law*. 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. The main contribution of this work is that the model provides a straight jacket in much the same way as monads do, such that all well-typed programs expressed using the model can be trivially vectorised. Before introducing the model, some basic category theory, and a category theoretic description of monads is given. This description stems from the desire for a understanding of Moggi’s work on computational  $\lambda$ -calculus and Monads [26, 25], and owes much to the collection of category theory books [31, 2, 9, 23, 3].

## 9 Basic category theory and monads

Category theory is concerned with the observation that many of the properties from algebra can be simplified by a presentation in terms of diagrams containing arrows—it’s just pictures! A category  $\mathcal{C}$  comprises of a collection of objects and a collection of morphisms (also called arrows). If the morphism  $f$  has a domain  $A$  and a co-domain  $B$ , then the morphism is written as  $f : A \rightarrow B$  or

$A \xrightarrow{f} B$ . The collection of all morphisms in the category  $\mathcal{C}$  with a domain  $A$  and co-domain  $B$  is called a *hom-set* written  $\mathcal{C}(A, B)$ . The composition of morphisms must obey the associative law such that for any morphism  $f : A \rightarrow B$ ,  $g : B \rightarrow C$ , and  $h : C \rightarrow D$  then  $h \circ (g \circ f) \equiv (h \circ g) \circ f$  must hold. Another requirement of a category is that for each object in the category  $\mathcal{C}$  (written  $Obj(\mathcal{C})$ ), there exists an identity morphism  $id_A$  in the hom-set  $\mathcal{C}(A, A)$  such that for any morphism  $f : A \rightarrow B$ ,  $f \circ id_A \equiv f$  and  $id_B \circ f \equiv f$ . These two requirements of a category provides an introduction to our first pair of commuting diagrams, which provide a pictorial representation of the relationships between objects and morphisms *within a single category*. The following diagrams express the identity and associative laws of the morphisms  $f$ ,  $g$ , and  $h$  in the category  $\mathcal{C}$ :



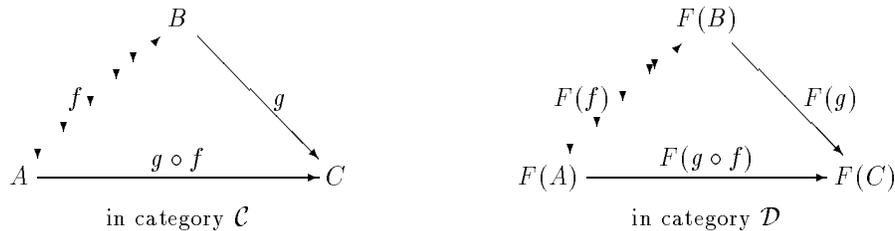
Category theory has a potentially important application to data-parallel programming. As associativity of morphism composition is one of the building blocks of a category, then morphisms can be used as the higher order arguments to either parallel fold or scan.

### 9.1 Functors: mapping between categories

Given two categories  $\mathcal{C}$  and  $\mathcal{D}$ , a *functor*  $F : \mathcal{C} \rightarrow \mathcal{D}$  is a mapping between the two categories and consists of an object mapping and an morphism mapping. The object mapping  $F_{obj}$  defines a relationship between each object in the category  $\mathcal{C}$ , and objects in the category  $\mathcal{D}$ , written as  $F_{obj} : Obj(\mathcal{C}) \rightarrow Obj(\mathcal{D})$ . By convention, if  $A$  is an object in  $\mathcal{C}$ , then  $F_{obj}(A)$  is the corresponding element in category  $\mathcal{D}$ . The morphism mapping is an analogous relationship between the morphisms of two categories. Given each morphism in the hom-set  $\mathcal{C}(A, B)$ , the morphism mapping  $F_{mor}$ , defines a relationship with the morphisms in the hom-set  $\mathcal{D}(F_{obj}(A), F_{obj}(B))$ , written as  $F_{mor} : \mathcal{C}(A, B) \rightarrow \mathcal{D}(F_{obj}(A), F_{obj}(B))$ . A functor is required to preserve the structure of the compositions of morphisms such that given the identity morphism  $id_A$ , and two morphisms  $f$  and  $g$  in the category  $\mathcal{C}$ , the following equations hold:

$$\begin{aligned}
 F_{mor}(id_A) &= id_{F_{obj}(A)} \\
 F_{mor}(f \circ g) &= F_{mor}(f) \circ F_{mor}(g)
 \end{aligned}$$

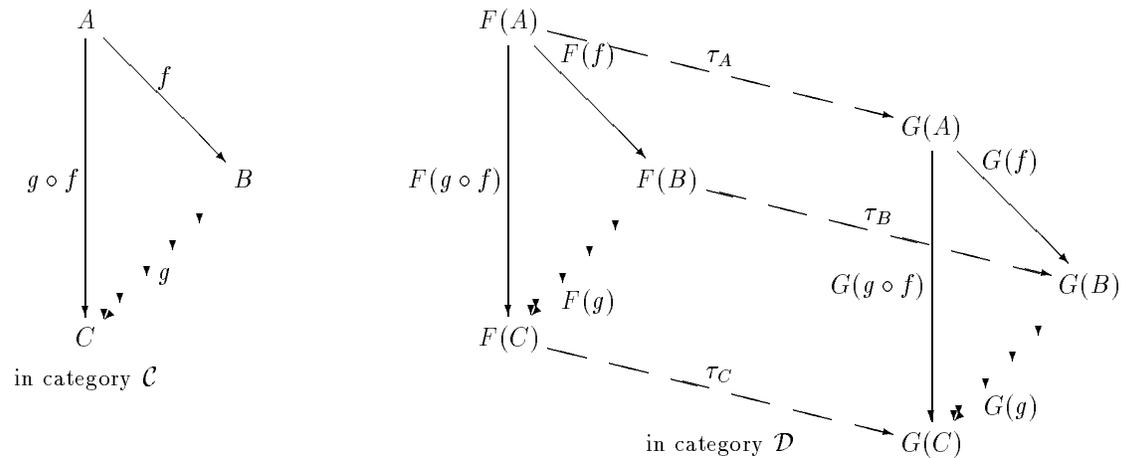
Category theorists enjoy omitting unnecessary syntactic baggage from their notation. By convention, the subscripts *obj* and *mor* can be dropped from a functor as it is always clear from the context whether an object or morphism mapping is being described. The commuting diagram on the left below defines the associativity of composition for the morphisms  $f$  and  $g$  in the category  $\mathcal{C}$ ; the commuting diagram on the right defines the corresponding relationship between the morphisms  $F(f)$  and  $F(g)$  in category  $\mathcal{D}$ :



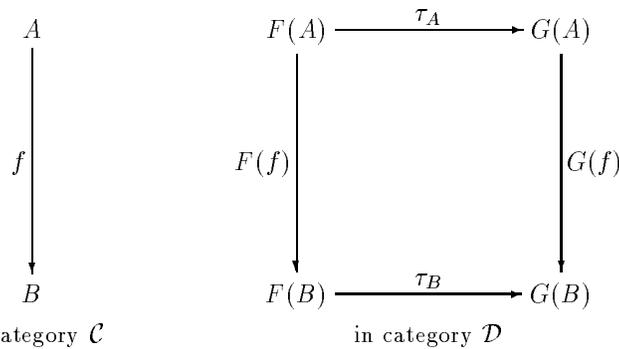
Like morphisms, functors can be composed, such that for any two functors  $F : \mathcal{A} \rightarrow \mathcal{B}$  and  $G : \mathcal{B} \rightarrow \mathcal{C}$ , then the composition  $G \circ F$  is a functor that maps objects and morphisms in the category  $\mathcal{A}$  to corresponding elements in  $\mathcal{C}$ . By convention, if  $A$  is an object in the category  $\mathcal{A}$ , then the corresponding element in category  $\mathcal{C}$  is written as  $GF(A)$  in favour of the more obvious  $G(F(A))$ .

### 9.2 Natural transformations: sliding between categories

Given two functors  $F, G : \mathcal{C} \rightarrow \mathcal{D}$  which form two mappings between the same category, a *natural transformation* between  $F$  and  $G$  is a structure preserving translation from one functor to another. The natural transformation  $\tau$  from  $F$  to  $G$  can be thought of as a way of “sliding” the picture defining the functor  $F$  onto that of  $G$  (the left commuting diagram is in the category  $\mathcal{C}$ , and the sliding commuting diagram on the right is in category  $\mathcal{D}$ ):



Concentrating on a single morphism  $f : A \rightarrow B$ , then a natural transformation  $\tau$  from  $F$  to  $G$ , written  $\tau : F \rightarrow G$  can be defined by assigning to each object  $A$  in  $Obj(\mathcal{C})$  a *morphism*  $\tau_A : F(A) \rightarrow G(A)$ , such that for every morphism  $f$  in the hom-set  $\mathcal{C}(A, B)$ , the following diagram commutes:



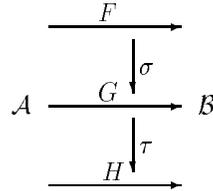
The morphisms  $\tau_A$  and  $\tau_B$  in the diagram above are called the *components* of the natural transformation  $\tau$ . A common pictorial representation of a natural transformation is a diagram of the form:

$$\begin{array}{ccc}
 & \xrightarrow{F} & \\
 \mathcal{C} & \downarrow \tau & \mathcal{D} \\
 & \xrightarrow{G} &
 \end{array}$$

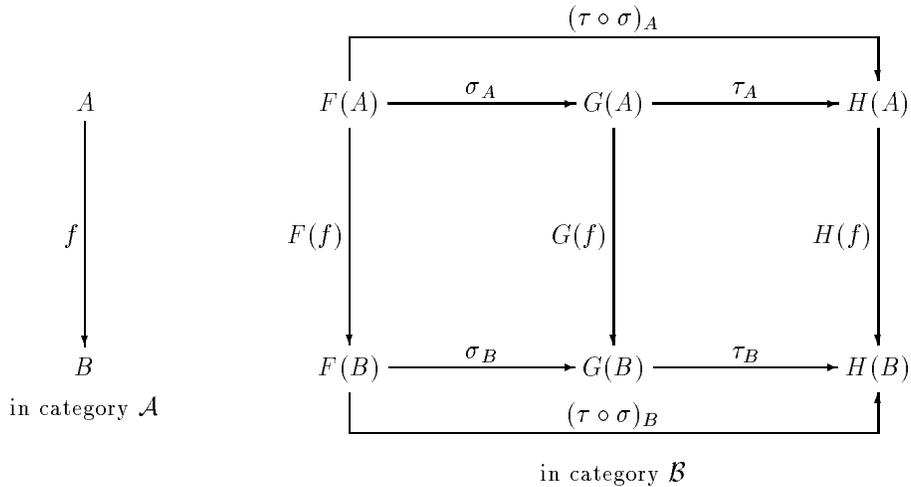
, which can be read as the two functors  $F, G : \mathcal{C} \rightarrow \mathcal{D}$ , with the natural transformation  $\tau$  that provides a “translation” between the two.

### 9.3 Composing natural transformations

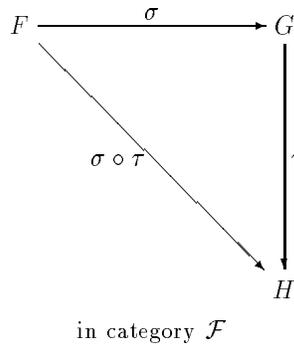
Natural transformations can be composed either “vertically” or “horizontally”. Given two natural transformations  $\tau$  and  $\sigma$ , the diagram below shows a vertical composition:



For every object  $A$  in the category  $\mathcal{A}$ , then by the definition of a natural transformation of the previous section, the following commuting diagram in the category  $\mathcal{B}$  can be drawn (remember that  $\sigma_B$  and  $\tau_B$  are the *morphisms* that are the components of the natural transformation):

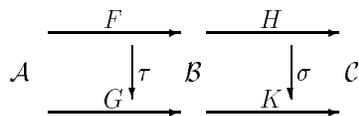


The commuting diagram on the right above can be used as the basis for a new category  $\mathcal{F}$  that has functors as its objects, and natural transformations as its morphisms—the *functor category*. The prior commuting diagram can now be written as a commuting diagram in  $\mathcal{F}$ :



Therefore given two natural transformations,  $\tau$  and  $\sigma$ , the composition of the two  $\tau \circ \sigma$  also forms a natural transformation between the functor  $F$  and  $H$ , written as  $\sigma \circ \tau : F \rightarrow H$ .

Next we consider the horizontal composition of  $\tau$  and  $\sigma$ , expressed by the following natural transformation diagram:



For each object in the category  $\mathcal{A}$  we consider the corresponding objects in the category  $\mathcal{C}$ . By the definition of a functor and a natural transformation, given an object  $A$  in category  $\mathcal{A}$ , then *one of the* corresponding objects in category  $\mathcal{C}$  is  $HF(A)$  (by composition of the functors on the top line of the natural transformation diagram); another object in category  $\mathcal{C}$  is  $HG(A)$ . The morphism that describes the relationship between these two objects in  $\mathcal{C}$  is  $H(\tau_A)$ , i.e., the morphism  $\tau_A$  which is the component of the natural transformation  $\tau$ , is used to select the morphism mapping from the functor  $H$ . Another interesting morphism in the diagram is between the objects  $HG(A)$  and  $KG(A)$  in the category  $\mathcal{C}$ . Applying the functor  $G$  to the object  $A$  in category  $\mathcal{A}$  gives the object  $G(A)$  in category  $\mathcal{B}$ . This object is then used to select the component of the natural transformation  $\sigma$  that describes the morphism  $\sigma_{G(A)}$  between  $HG(A)$  and  $KG(A)$ . Applying similar techniques to each of the remaining paths in the natural transformation diagram, gives the following commuting diagram in category  $\mathcal{C}$ :

$$\begin{array}{ccc}
 HF(A) & \xrightarrow{H(\tau_A)} & HG(A) \\
 \sigma_{F(A)} \downarrow & & \downarrow \sigma_{G(A)} \\
 FK(A) & \xrightarrow{K(\tau_A)} & KG(A)
 \end{array}$$

in category  $\mathcal{C}$

In a similar vein to the treatment of horizontal compositions, a new category  $\mathcal{F}$  can be defined with functors as its objects and natural transformations as its morphisms. If  $id_{\mathcal{C}} : \mathcal{C} \rightarrow \mathcal{C}$  is the identity functor for the category  $\mathcal{C}$  and  $1_{\mathcal{C}} : id_{\mathcal{C}} \rightarrow id_{\mathcal{C}}$  the identity natural transformation of that functor to itself, then by the composition of natural transformations  $1_{\mathcal{D}} \circ \tau \equiv \tau \equiv \tau \circ 1_{\mathcal{C}}$ . Now for the utterly revolting part—if  $F$  is a functor, then it is also convenient to use the symbol  $F$  to represent the identity natural transformation from  $F$  to  $F$ , i.e.,  $F : F \rightarrow F$ . If  $G, H$ , and  $K$  are also identity natural transformations from the appropriate functors, then the following commuting diagram in  $\mathcal{F}$  can be given which induces four new natural transformations  $H\tau : HF \rightarrow HG$ ,  $\sigma G : HG \rightarrow KG$ ,  $K\tau : FK \rightarrow HK$ , and  $\sigma F : HF \rightarrow FK$ .

$$\begin{array}{ccc}
 HF & \xrightarrow{H\tau} & HG \\
 \sigma F \downarrow & & \downarrow \sigma G \\
 FK & \xrightarrow{K\tau} & KG
 \end{array}$$

in category  $\mathcal{F}$

By the commutativity of the diagram, the following equivalence of the compositions of the morphisms of  $\mathcal{F}$  (i.e., natural transformations) hold,  $(\sigma \circ G) \circ (H \circ \tau) \equiv (K \circ \tau) \circ (\sigma \circ F)$ . ♠ **ToDo:** *Should this be •? ♠*

## 9.4 Monads in category theory

The principle underlying Moggi's work on monads and the computational lambda 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.

In category theory, a monad over the category  $\mathcal{C}$  is the triple  $(T, \eta, \mu)$ , where  $T$  is the *endofunctor*<sup>6</sup>  $T : \mathcal{C} \rightarrow \mathcal{C}$ , and  $\eta$  and  $\mu$  are two natural transformations. As already mentioned, the convention for writing the composition of functors  $T \circ T$  is  $TT$ , which is simplified further here to the exponentiation notation  $T^2$ . Using this abbreviation, the natural transformations of the monad are defined as  $\eta : id_{\mathcal{C}} \rightarrow T$  and  $\mu : T^2 \rightarrow T$ :

$$\begin{array}{ccc}
 \xrightarrow{id_{\mathcal{C}}} & & \xrightarrow{T^2} \\
 \mathcal{C} & \downarrow \eta & \mathcal{C} \\
 \xrightarrow{T} & & \xrightarrow{T}
 \end{array}$$

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  $\tau$ , to a corresponding set of objects  $T(Obj(\mathcal{C}))$  which are to be interpreted as the set of computations of type  $\tau$ . The natural transformation  $\eta$  can be thought of as an operator that includes values into a computation; the natural transformation  $\mu$  "flattens" a computation of computations into a single computation.

For  $T, \eta, \mu$  to be termed a monad, three laws called the *associative law of a monad*, and the *left and right identity laws* must hold. Rather than giving the laws and attempting to justify them, the laws are presented in a step-by-step manner by considering various compositions of the monads components  $T, \eta$ , and  $\mu$ .

We first consider the associative law by investigating the characteristics of the flattening natural transformation  $\mu$ . For each object  $A$  in the category  $\mathcal{C}$  we consider the corresponding elements in the category  $\mathcal{C}$  after applying a combination of the morphism mappings of the endofunctors  $T$  and  $T^2$ , and the components of the natural transformation  $\mu$ . The diagram on the left below defines a left combination of the endofunctor  $T$  with the natural transformation  $\mu$ . By the definition of a functor and a natural transformation, given an object  $A$  in category  $\mathcal{C}$ , then *one of the* corresponding objects in category  $\mathcal{C}$  is  $T^2T(A)$  (by composition of the functors on the top line of the natural transformation diagram); another object in category  $\mathcal{C}$  is  $TT(A)$ . The morphism that describes the relationship between these two objects in  $\mathcal{C}$  is  $\mu_{T(A)}$ , i.e., the object  $T(A)$  from the object mapping of the functor  $T$ , is used to select the component of the natural transformation  $\mu$ . The diagram on the right shows this relationship pictorially:

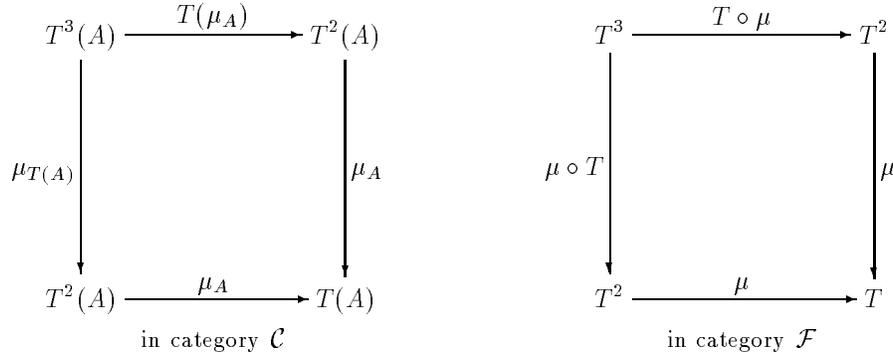
$$\begin{array}{ccc}
 \xrightarrow{T^2} & & \xrightarrow{\mu_{T(A)}} \\
 \mathcal{C} \xrightarrow{T} \mathcal{C} & \downarrow \mu & T^2T(A) \xrightarrow{\mu_{T(A)}} TT(A) \\
 \xrightarrow{T} & & \text{in category } \mathcal{C}
 \end{array}$$

A similar technique can be used in the right-wards combination of the endofunctor  $T$  with the natural transformation  $\mu$ :

$$\begin{array}{ccc}
 \xrightarrow{T^2} & & \xrightarrow{T(\mu_A)} \\
 \mathcal{C} & \downarrow \mu & \mathcal{C} \xrightarrow{T} \mathcal{C} \\
 \xrightarrow{T} & & TT^2(A) \xrightarrow{T(\mu_A)} TT(A) \\
 & & \text{in category } \mathcal{C}
 \end{array}$$

If we now define a new category  $\mathcal{F}$  with functors as its objects, and natural transformations as its morphisms, then the commuting diagram on the left below which combines the morphisms of the prior diagrams can be used as the basis for the diagram on the right—the morphism  $T$  on the right is the identity natural transformation from the endofunctor  $T$  to  $T$ :

<sup>6</sup>i.e., inside functor—a functor with a mapping to and from the same category.

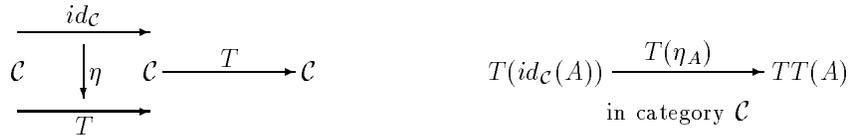


By the commutativity of the diagram,  $\mu \circ (T \circ \mu) \equiv \mu \circ (\mu \circ T)$ , which is termed the *associative law of the monad*.

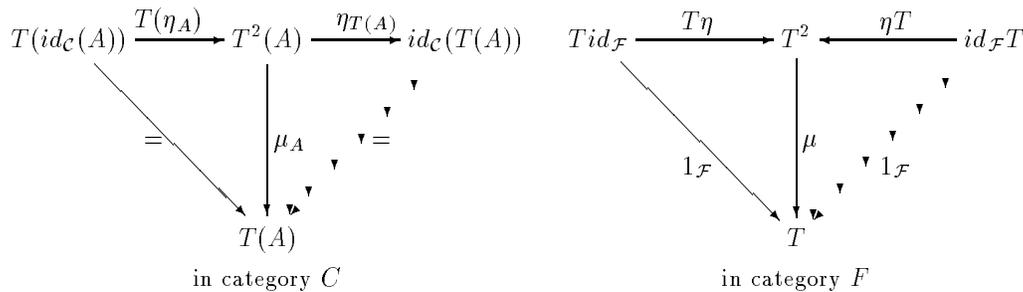
Next we consider the *left and right identity laws* by investigating the characteristics of the injective operator  $\eta$ , in a similar manner to the treatment of  $\mu$ . For each object  $A$  in the category  $\mathcal{C}$  we consider the corresponding elements in the category  $\mathcal{C}$  after applying  $\eta$  and the endofunctor  $T$  in a leftwards manner:



, a similar technique can be applied to a right-wards combination of the endofunctor  $T$  with the natural transformation  $\eta$ :



If we now define a new category  $\mathcal{F}$  with functors as its objects, and natural transformations as its morphisms, then the commuting diagram on the left below which combines the morphisms of the prior diagrams can be used as the basis of the diagram on the right—the morphism  $T$  on the right is the identity natural transformation from the endofunctor  $T$  to  $T$ :

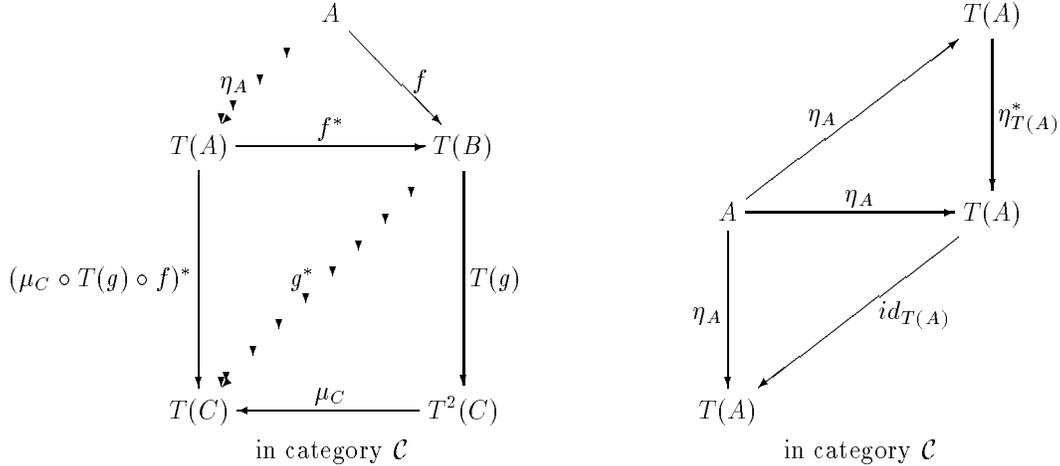


By the commutativity of the diagram,  $\mu \circ (\eta \circ T) \equiv 1_{\mathcal{F}} \equiv \mu \circ (T \circ \eta)$ , which is termed the *left and right unit laws of a monad*.

### 9.5 The Kleisli Star

Given the monad  $(T, \eta, \mu)$  in category  $\mathcal{C}$ , consider for each morphism  $f : A \rightarrow T(B)$  a new morphism  $f^* : T(A) \rightarrow T(B)$ , where  $\_*$  is the “extension” operator that lifts the domain  $A$  of the morphism  $f : A \rightarrow T(B)$  to a computation  $T(A)$ . In the context of Moggi’s work on computations,

$f$  is a function from values to computations, whereas  $f^*$  is a function from computations to computations. The expression  $g^* \circ f$ , where  $f : A \rightarrow T(B)$  and  $g : B \rightarrow T(C)$  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 monad  $(T, \eta, \mu)$  and the extension operator can be used to create the following commuting diagrams in  $\mathcal{C}$ :



The Kleisli triple  $(T_{obj}, \eta, *)$  can be constructed from the monad  $(T, \eta, \mu)$ , where  $T_{obj}$  is the restriction of the endofunctor  $T$  to objects;  $(g : B \rightarrow T(C))^* \equiv \mu_C \circ T(g)$  from the commutativity of the diagram above left. Kleisli triples can be thought of as a different syntactic presentation of a monad, as there is a one-to-one correspondence between a Kleisli triple and a monad (see [23, page 143] for a proof). In a similar vein to the monad laws, the following laws can be constructed from the commuting diagrams above:

- Left Unit:*  $\eta_{T(A)} \equiv id_{T(A)}$
- Right Unit:*  $f^* \circ \eta_A \equiv f$
- Associativity of computations:*  $g^* \circ f^* \equiv (g^* \circ f)^*$

### 9.6 Monads in functional programming

Wadler [37] adapted Moggi's ideas of using monads to structure the semantics of computations into a tool for structuring functional programs. Given a monad  $(T, \eta, \mu)$ , the functor  $T$  and the two natural transformations are modelled in a functional programming language by a type constructor<sup>7</sup>, a function *map* parameterised on the type constructor  $M$  (see [8] for parametrising *map* in this way), and two polymorphic functions.

Given a functor  $T : \mathcal{C} \rightarrow \mathcal{D}$ , a morphism  $f : A \rightarrow B$ , and the set of object  $A$  in category  $\mathcal{C}$ , the corresponding elements in the category  $\mathcal{D}$  will be the objects  $T(A)$  and the morphisms  $T(f) : T(A) \rightarrow T(B)$ . The object mapping part of a functor is represented in a functional language by a type-constructor. For example, if an object  $A$  has the type  $\alpha$ , then the object  $T(A)$  has the type  $M \alpha$ ; where  $M$  is a type constructor that represents a "computation". The morphism mapping of the functor is modelled by a function analogous to *map*:

$$map :: (\alpha \rightarrow \beta) \rightarrow (M \alpha \rightarrow M \beta)$$

The corresponding morphism of  $f$  in the category  $\mathcal{D}$  is  $T(f)$  which is modelled in a functional language by the curried function application *map f*.

Finally, a natural transformation can be thought of as a family of arrows from each object in a category (the components of a natural transformation), to objects in another category. A natural

<sup>7</sup>The type constructor  $M$  on lists is normally written as  $M \alpha \equiv [\alpha]$ .

transformation is therefore *similar* to a polymorphic function, and as a consequence  $\eta$  and  $\mu$  are written as the polymorphic functions *unit* and *join* of type:

$$\begin{aligned} \text{unit} &:: \alpha \rightarrow M \alpha \\ \text{join} &:: M (M \alpha) \rightarrow M \alpha \end{aligned}$$

In summary, given a category theory monad  $(T, \eta, \mu)$ , a functional programming monad is represented by the quadruple  $(M, \text{map}, \text{unit}, \text{join})$ , which must obey the monad laws of section 9.4 which are re-expressed as:

$$\begin{aligned} \text{Left Unit:} & & \text{join} \circ \text{unit} & \equiv \text{id} \\ \text{Right Unit:} & & \text{join} \circ \text{map unit} & \equiv \text{id} \\ \text{Associativity of computations:} & & \text{join} \circ \text{map join} & \equiv \text{join} \circ \text{join} \end{aligned}$$

## 9.7 How people really use monads in functional programming

In Wadler's more recent papers [38, 30], and with the onset of the application of monads as a method of incorporating imperative features into a purely functional language, the current use of monads bears a closer resemblance to Kleisli triples. The triple  $(M, \text{unit}, \star)$  forms a monad when the following laws hold:

$$\begin{aligned} \text{Left Unit:} & & \text{unit } a \star \lambda b \rightarrow n & \equiv n[a/b] \\ \text{Right Unit:} & & m \star \lambda b \rightarrow \text{unit } b & \equiv m \\ \text{Associativity of computations:} & & m \star (\lambda a \rightarrow n \star \lambda b \rightarrow m) & \equiv (m \star \lambda a \rightarrow n) \star \lambda b \rightarrow m \end{aligned}$$

As with the monads in the previous section,  $M$  is a type-constructor such that an expression of type  $M \alpha$  can be thought of as a computation delivering an object of type  $\alpha$  as a result. *unit* is the inclusion function of the monad (same as  $\eta$ ) of type  $\alpha \rightarrow M \alpha$  that converts a value  $v$  of type  $\alpha$  into a computation that does nothing but deliver the value  $v$  as a result. The expression  $f \star g$  is the same as  $g^* \circ f$  of the Kleisli triple, and has the type  $\star :: M\alpha \rightarrow (\alpha \rightarrow M\beta) \rightarrow M\beta$ .

## 10 Creating a `printf` vectorisation monad

A model of stream based output can be defined in terms of Moggi's side effecting monad [25, example 3.6, page 11]. In a simplified scenario where the output stream is a list of characters (a Landin-stream [21]), the monad  $(IO, \text{unit}, \text{bind})$  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'
>
```

Step 0: Character stream based IO monad
---

The monad operations are augmented with a `printf` computation that outputs its string argument as a “side-effect” onto the output stream and delivers `()` as a result of the function. In the context of category theory, given the monad  $(T, \eta, \mu)$  in the category  $\mathcal{C}$ , functions like `printf` 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 they require “inside knowledge” of the representation of the monad:

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

Step 0 <sub>p</sub> : printing into the IO 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. By hiding the state inside the monad compared to being visible in terms of variables such as `next v` in a *pH* loop means that it is possible to straight-jacket any interactions with the state such that the *first duality theorem* and the *fold-map fusion laws* can be satisfied 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 the loop counter by side-effecting the output stream:

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

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  $\lambda s i \rightarrow s \text{ 'bind' } (\lambda () \rightarrow \text{bodyFn } i)$  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  $k^{\text{th}}$  iteration of the loop, the argument `'s'` encapsulates the state of all the previous  $k - 1$  iterations of the loop; (5) The initial value 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 = printf "Hello World " 'bind' (\ ()->
>     printf (show n)       'bind' (\ ()->
>     printf "\n"           'bind' (\ ()->
>     unit ()
>     )))
```

Loop 2: The loop expressed as a fold-left
---

The problem with the translation shown in `helloWorlds:loop 2` is the function used in the fold isn't associative. The remainder of this chapter ensures the operations used by the fold are associative by converting the monad  $(IO, \text{unit}, \text{bind})$  into the monoid  $(IO', \oplus, \oplus_{id})$ . The salient features of this transformation is the original program undergoes a minor syntactic change, and each occurrence of `unit` and `bind` in the program are replaced by  $\oplus$  and  $\oplus_{id}$ . If the transformation from monad to monoid is successful, the original monad is termed a *vectorisation monad*—the remarkable property of this transformed monad is that a compiler no longer needs to infer if vectorisation of a well-typed for-loop is possible, it always is!

## 10.1 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, or never will be associative due to its type. 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 behaviours—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<sup>8</sup>), then all the `()`'s can be elided from the program. Removing the parameterisation from the type constructor `IO`, and changing the monad operations according produces:

```
> type IO = String -> String
> infixr 9 'bind'
>
> unit :: IO
> unit = \s -> s
>
> bind :: IO -> IO -> IO
> l 'bind' r = r . l
>
> printf :: String -> IO
> printf str = \s -> s ++ str
```

Character	stream
Step 1:based IO monoid and	<code>printf</code>

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  $g^* \circ f$ , what we have done is noticed 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  $(T, unit, \lambda f g \rightarrow f \tilde{\circ} g^*)$ , then 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  $(ShowS, \circ, id)$  used for printing in Haskell [19]. In Haskell, the monoid is used to ensure printing has 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 (for a more detailed discussion on the benefits of this monoid, see [18]). The monoid used here is rather “brain dead” as it utilises the opposite behaviour as `printf` 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
>     printf "Hello World " 'bind'
>     printf (show n)      'bind'
>     printf "\n"         'bind'
>     unit ()
```

Loop 3: Using the a monoid
----------------------------

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

<sup>8</sup>A non-terminating function which returns an object of type `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 than a tuple containing  $\perp$ , then all we are guarenteeing is that the ‘result’ part of the tuple will never be  $\perp$ .

`helloWorlds:Loop 3` can be transformed into:

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

Translating to a fold-  
 Loop 4: left, and applying the  
 fold-map fusion law.

It would seem the 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!

### 10.1.1 An historical aside

As an aside, a monoid of this form 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 [21] 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 [30] 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 `printf`. This technique is similar to Landin’s implementation of the sharing machine where the *assignandhold* function can side-effect the store of the sharing machine because of the sequencing imposed by functional composition. Peter defined that “*Imperatives are treated as null-list producing functions*”<sup>9</sup>. The imperative *assignandhold* 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 the store of the sharing machine, returning the value assigned as a result of the function. Because Peter assumed applicative order reduction, the **K**-combinator<sup>10</sup> of Curry & Feys [10] was used to return `()`, and the imperative was evaluated as a side effect by the unused argument of the **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
<code>x := 2;</code>	$((\lambda() \rightarrow \mathbf{K} () (assignandhold\ x\ 2)) \delta$
<code>x := 1+3;</code>	$(\lambda() \rightarrow \mathbf{K} () (assignandhold\ x\ (1+3)))) ()$

By using a lambda expression with `()` as its parameter, `()` can be thought of as the “state of the world” that is threaded throughout a program by functional composition—a technique Peyton Jones and Wadler [30] re-used 28 years later.

## 10.2 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.

<sup>9</sup>In Landin’s paper, `()` is the syntactic representation of the empty list and not the unit.

<sup>10</sup> $\mathbf{K} = \lambda x\ y \rightarrow x$

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 `printf`), then we require that there exists a function  $g$  of type  $\alpha \rightarrow M'$  that encapsulates *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:

$$\begin{aligned} & g\ v_1\ 'bind'\ g\ v_2 \\ \Rightarrow & g\ v_2 \circ g\ v_1 && \text{unfolding } bind \\ \Rightarrow & (\lambda s \rightarrow s \oplus f\ v_2) \circ (\lambda s \rightarrow s \oplus f\ v_1) && \text{unfolding } g \\ \Rightarrow & \lambda s \rightarrow (s \oplus f\ v_1) \oplus f\ v_2 && \text{by the definition of } \circ \\ \Rightarrow & \lambda s \rightarrow s \oplus (f\ v_1 \oplus f\ v_2) && \text{by the associativity of } \oplus \end{aligned}$$

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 encapsulated 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 monoid used before and after this transformation is a *monoid homomorphism*—i.e., if  $hom$  is the monoid homomorphism, then the following holds:

$$\begin{aligned} hom(unit) & \equiv \oplus_{Id} \\ hom(x\ 'bind'\ y) & \equiv hom(x) \oplus hom(y) \end{aligned}$$

### 10.3 Transforming the output monoid

The monoid :Step 1 in the running example only has one operation in the associated hom-set. This function `printf` can be coerced into the form required by  $g$  as shown in the equation:

$$\begin{aligned} printf & \equiv g \\ \lambda str\ s \rightarrow s ++ str & \equiv \lambda v\ s \rightarrow s \oplus f\ v \\ & \text{iff } x \oplus y = x ++ y \\ & \quad f\ v = v \end{aligned}$$

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

```
> type IO = String
>
> unit :: IO
> unit = []
>
> bind :: IO -> IO -> IO
> l 'bind' r = l ++ r
>
> printf :: String -> IO
> printf str = str
```

Step 0: Character stream based IO monad

The function `helloWorlds:Loop 4` can be left syntactically unchanged and the fold-left can be implemented in `parallel`<sup>11</sup>.

## 11 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, which incidentally may fail, what we have done is used monads to develop a constrained programming language in which only vectorisable programs can be expressed. All that is required of a compiler is to perform a minor syntactic translation of a for-loop into the vectorisation monad, and an ordinary type inference algorithm is used to determine if the loop is vectorisable. As a consequence of this, there is no need for the compiler to guarantee the vectorisation properties, the monad defines that a loop can be vectorised, and a proof for the monad can be given once, *independently of the compiler*. All that is left for the vectorising compiler to do is to vectorise the *known* operations defined by the `bind` operation of the monad.

## References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, techniques and tools*. Addison-Wesley, 1986.
- [2] Andrea Asperti and Giuseppe Longo. *Categories, Types and Structures*. The MIT Press, 1991.
- [3] M. Barr and C. Wells. *Toposes, Triples and Theories*. Number 278 in Comprehensive studies in mathematics. Springer-Verlag, 1985.
- [4] R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.
- [5] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.
- [6] Richard S. Bird. An introduction of the theory of lists. Technical Report PRG-56, Oxford University Computing Laboratory, 1986.
- [7] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.
- [8] K. Chen, P. Hudak, and M. Odersky. Parametric type classes. In *Lisp and functional programming*, 1992. Enables things like map to be overloaded on vectors, lists, etc..
- [9] P. M. Cohn. *Universal Algebra*. Harper & Row, 1965.
- [10] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland Pub. Co., Amsterdam, 1958.
- [11] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *ACM Symposium on Principles of Programming Languages*, pages 119–130, 1978.
- [12] K. Hammond and S. L. Peyton Jones. Profiling scheduling strategies on the GRIP parallel reducer. Technical report, Glasgow University, 1991.
- [13] Peter Henderson. *Functional Programming: Application and implementation*. Prentice-Hall international, 1980.

---

<sup>11</sup> The use of `+` is alarming, and in practice makes parallelism futile—but thats another matter! See section ??.

- [14] Jonathan M. D. Hill. Data Parallel Haskell: Mixing old and new glue. Technical Report 611, QMW CS, December 1992. Available by FTP from ftp.dcs.qmw.ac.uk in /pub/cpc/jon\_hill/dpGlue.ps.
- [15] Jonathan M. D. Hill. The *aim* is laziness in a data-parallel language. In K. Hammond and J. T. O'Donnell, editors, *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.
- [16] Jonathan M. D. Hill. Vectorizing a non-strict functional language for a data-parallel “Spineless (not so) Tagless G-Machine”. In *Proc. of the 5<sup>th</sup> international workshop on the implementation of functional languages*, Nijmegen, Holland, September 1993. Available by FTP from ftp.dcs.qmw.ac.uk in /pub/cpc/jon\_hill/vectorizeNonStrict.ps.
- [17] Jonathan M. D. Hill. *Data-parallel lazy functional programming*. PhD thesis, Dept of Computer Science, Queen Mary & Westfield College, University of London, June 1994.
- [18] P. Hudak and J.H. Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, May 1992.
- [19] 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.
- [20] Paul Hudak. Mutable abstract datatypes. Technical Report 914, Yale University, Department of Computer Science, December 1992.
- [21] P. J. Landin. A correspondence between ALGOL 60 and Church’s lambda notation. *Communications of the ACM*, 8(2):89–101, February 1965. Part 2 in CACM Vol 8(2) 1965, pages 158–165.
- [22] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.
- [23] Saunders Mac Lane. *Categories for the working mathematician*. Springer-Verlag, 1971.
- [24] John Launchbury. Lazy imperative programming. In *Proc ACM Sigplan workshop on State in Programming Languages*, pages 46–56, June 1993.
- [25] E. Moggi. Computational lambda-calculus and monads. In *IEEE symposium on Logic in computer science*, 1989. University of Edinburgh LFCS technical report ECS-LFCS-88-66 is an extended version of the paper.
- [26] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, April 1990.
- [27] R . S. Nikhil, Arvind, and James Hicks. pH language proposal (preliminary). September 1993.
- [28] R.S. Nikhil. Id-Nouveau (version 88.0) reference manual. Technical report, MIT Laboratory for Computer Science, Cambridge, Mass., mar 1988.
- [29] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, 1987.
- [30] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages*, 1993.
- [31] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

June 5, 1994 (12)

- [32] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes: The art of scientific computing*. Cambridge University Press, 1986.
- [33] J. Rees and W. Clinger (eds.). The revised<sup>3</sup> report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.
- [34] Jan Sparud. Fixing some spaces leaks without a garbage collector. In *Functional Programming and Computer Architecture*, 1993.
- [35] G. L. Steele. Lambda — the ultimate goto.
- [36] Jean-Pierre Talpin. *Aspects théoriques et pratiques de l'inférence de type et d'effets*. PhD thesis, L'Ecole nationale supérieure des mines de Paris, May 1993. (Thesis in English).
- [37] Philip Wadler. Comprehending monads. In *ACM Conference on Lisp and functional programming*, ACM Conferences, pages 61–78. ACM, June 1990.
- [38] Philip Wadler. The essence of functional programming. In *ACM Symposium on Principles of Programming Languages*, January 1992.

June 5, 1994 (12)