Compact Fusion

Catherine Hope and Graham Hutton
School of Computer Science and IT, University of Nottingham,
Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, UK
{*cvh,gmh*}*@cs.nott.ac.uk*

Abstract

There are many advantages to writing functional programs in a compositional style, such as clarity and modularity. However, the intermediate data structures produced may mean that the resulting program is inefficient in terms of space. These may be removed using deforestation techniques, but whether the space performance is actually improved depends upon the structures being consumed in the same order that they are produced. In this paper we explore this problem for the case when the intermediate structure is a list, and present a solution. We then formalise the space behaviour of our solution by means of program transformation techniques and the use of abstract machines.

Keywords: hylomorphism, space, fold, abstract machine

1. INTRODUCTION

Hylomorphisms [1] represent a common programming pattern of using an intermediate data structure, that is first built and then collapsed, to give a result. More formally, it is the composition of an unfold and a fold: the unfold uses a seed value to generate a data structure and the fold takes this structure and collapses it in some way. The space efficiency of this composition may be improved by applying fusion techniques to eliminate the intermediate data structure. However, whether the space performance is actually improved depends on the fold being able to consume elements as they are generated. If this is not the case, then the result is the creation of the whole structure before any folding evaluation can take place, and the intermediate structure still effectively exists in the fused function.

Here we will illustrate this problem with some examples and show how using an accumulating fold, *fold-left*, will improve the space performance. We then show how to formalise these space results, by using abstract machines to expose the underlying data structures, which can then be measured. The contributions are i) a new hylomorphism theorem, that captures the idea of consuming elements as they are generated, and ii) the process of producing space results. To achieve the second contribution, we derive an abstract machine using program transformation techniques. Once we have such a machine we can produce a high-level function that measures space usage. All our examples are given in Haskell [2].

2. HYLOMORPHISMS

We will consider hylomorphisms where the intermediate data structure is a list; that is, the unfold function generates a list from a seed value, and the fold then consumes this list.

2.1. Unfold

The unfold function builds a list from an initial seed value. It takes three additional arguments: a predicate, p, to determine when to stop generating list elements, and two other functions, hd and tl, to make the head of the list and to modify the seed value to pass to the recursive call, and generate the rest of the list:

$$
\begin{aligned}
&unfold && :: (a \to Bool) \to (a \to b) \to (a \to a) \to a \to [b] \\
&unfold\ p\ hd\ tl = f \\
&\quad \textbf{where}\ f\ x = \textbf{if}\ p\ x\ \textbf{then}\ [\,]\ \textbf{else}\ hd\ x : f\ (tl\ x)
\end{aligned}
$$

The resulting list is therefore of the form:

$$unfold\ p\ hd\ tl\ x = [hd\ x, hd\ (tl\ x), hd\ (tl\ (tl\ x)), ...]$$

For example, we can define a function $downFrom$ using $unfold$, which takes a natural number n and produces a list of all the numbers from n down to 1, where id and $pred$ are the identity and predecessor functions:

$$downFrom = unfold\ (\equiv 0)\ id\ pred$$

Applying $downFrom$ to the number 3 produces evaluation trace A in figure 1. We can use the shape of the trace to informally measure the space requirements in evaluation of the expression. The expression size can be estimated by counting constructor symbols and the space requirements for evaluation of an expression is given by the maximum expression size generated during evaluation, since space may be re-used at each step of evaluation. As we can see in the trace, the expression size reaches its maximum when the list has been completely generated, producing a list of length equal to the argument to $downFrom$. Evaluating $downFrom$ therefore requires additional space proportional to its argument, and so has linear space requirements.

2.2. Fold-right

The standard fold operator for lists [3] takes two arguments, a binary operator $(\oplus)$ and value v, replacing every list constructor $(:)$ with $(\oplus)$ and v in place of the empty list $[\,]$. It is defined as follows:

$$
\begin{aligned}
&foldr && :: (b \to c \to c) \to c \to [b] \to c \\
&foldr\ (\oplus)\ v = g \\
&\quad \textbf{where}\ g\ [\,] && = v \\
&\qquad\quad g\ (x : xs) && = x \oplus g\ xs
\end{aligned}
$$

For example, a list $[a, b, c]$ would be folded as:

$$foldr\ (\oplus)\ v\ [a, b, c] = a \oplus (b \oplus (c \oplus v))$$

Calculating the product of a list of numbers can be expressed by folding the multiplication operator over the list, and substituting the unit of multiplication in the empty list case:

$$product = foldr\ (*)\ 1$$

Applying $product$ to the list $[3, 2, 1]$ gives evaluation trace B shown in figure 1, and takes space proportional to the length of the list. This fold is called fold-right because, as shown in the trace, after replacing each $(:)$ with $(*)$, the application brackets to the right.

2.3. Hylomorphisms

A hylomorphism is the composition of a unfold with a fold, and is defined as follows:

$$hylor\ p\ hd\ tl\ (\oplus)\ v = foldr\ (\oplus)\ v \circ unfold\ p\ hd\ tl$$

We use the name $hylor$ for this function, rather than the standard $hylo$, to emphasise that it is specified in terms of fold-right. Within the definition for $hylor$, a list is generated by the unfold function and passed to the fold, which consumes it. However, the well-known hylo theorem [1] states that the two functions may be fused together to eliminate this intermediate data structure.

The hylomorphism theorem for lists is:

FIGURE 1: Evaluation traces for $downFrom$ and $product$

A)

$$
\begin{array}{ll}
 & downFrom\ 3 \\
= & f\ 3 \\
= & 3 : f\ 2 \\
= & 3 : 2 : f\ 1 \\
= & 3 : 2 : 1 : f\ 0 \\
= & 3 : 2 : 1 : [\,]
\end{array}
$$

B)

$$
\begin{array}{ll}
 & product\ (3 : 2 : 1 : [\,]) \\
= & g\ (3 : 2 : 1 : [\,]) \\
= & 3 * g\ (2 : 1 : [\,]) \\
= & 3 * (2 * g\ (1 : [\,])) \\
= & 3 * (2 * (1 * g\ [\,])) \\
= & 3 * (2 * (1 * 1)) \\
= & 3 * (2 * 1) \\
= & 3 * 2 \\
= & 6
\end{array}
$$

$$
\begin{array}{l}
hylor :: (a \to Bool) \to (a \to b) \to (a \to a) \to (b \to c \to c) \to c \to a \to c \\
hylor\ p\ hd\ tl\ (\oplus)\ v = h \\
\quad \textbf{where}\ h\ x = \textbf{if}\ p\ x\ \textbf{then}\ v\ \textbf{else}\ hd\ x \oplus h\ (tl\ x)
\end{array}
$$

Now we will look at an example hylomorphism and see how the space performance is affected by applying this theorem.

2.3.1. Example: factorial

The factorial of a natural number, n, can be calculated by taking the product of a list from n down to 1. We can therefore express the factorial function as the composition of the two functions $product$ and $downFrom$:

$$fact = product \circ downFrom$$

This composition is a hylomorphism, since the $downFrom$ function is an unfold and $product$ is a fold, and so we can apply the $hylor$ theorem, inlining the $pred$ function, to give the following fused program:

$$
\begin{array}{l}
fact = h \\
\quad \textbf{where}\ h\ x = \textbf{if}\ x \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ x * h\ (x - 1)
\end{array}
$$

The purpose of fusing the program is to eliminate the creation of the intermediate list. In this case the input and output are both integers, but a list is built in the process, so potentially we could perform the multiplication after each element of the list is generated and achieve evaluation in a constant amount of space. However, the unwinding of the fused definition of factorial, given in trace A of figure 2, shows that this isn't the case. The trace shows that all of the list elements do have to be generated before any multiplication evaluation can occur. Although there is not an explicit list, the structure is still there, with the list constructor replaced by the multiplication operator. Multiplication evaluation can only occur once the unfold has finished producing list elements, and the structure is then collapsed from the right.

The maximum expression size produced in the factorial example occurs when the list has been completely generated. Therefore, the amount of space required in evaluating the factorial of a number is directly proportional to that number, so it is linear and not the constant desired.

2.3.2. Impedance mismatch

The problem is the impedance mismatch[1] between unfold and fold-right; the former generates the list elements in left-to-right order, but the latter consumes them in right-to-left order. The $hylor$

[1]The term impedance mismatch comes from electrical engineering and is used in systems analysis to describe a system that cannot efficiently accommodate the output from another system.

theorem eliminates the overhead of constructing/destructing the intermediate list, but retains the impedance mismatch and hence gives poor space performance.

FIGURE 2: Evaluation traces for $fact$ and $factl$

A)

$$
\begin{array}{ll}
 & fact\ 3 \\
= & h\ 3 \\
= & 3 * (h\ 2) \\
= & 3 * (2 * (h\ 1)) \\
= & 3 * (2 * (1 * (h\ 0))) \\
= & 3 * (2 * (1 * 1)) \\
= & 3 * (2 * 1) \\
= & 3 * 2 \\
= & 6
\end{array}
$$

B)

$$
\begin{array}{ll}
 & factl\ 3 \\
= & h\ 1\ 3 \\
= & h\ (1 * 3)\ 2 \\
= & h\ 3\ 2 \\
= & h\ (3 * 2)\ 1 \\
= & h\ 6\ 1 \\
= & h\ (6 * 1)\ 0 \\
= & h\ 6\ 0 \\
= & 6
\end{array}
$$

2.4. Fold-left

An alternative way to fold a list is to bracket the operator from the left:

$$foldl\ (\oplus)\ v\ [a, b, c] = (((v \oplus a) \oplus b) \oplus c)$$

This version, called fold-left, uses an accumulator that is returned in the empty list case, and, in the non-empty case, combined with the head of the list, using the operator, and then the updated accumulator is passed to fold the tail of the list. The definition for fold-left is:

$$
\begin{array}{l}
foldl\ \ \ :: (b \to a \to b) \to b \to [a] \to b \\
foldl\ (\oplus)\ v = g\ v \\
\quad \textbf{where}\ g\ a\ [\,]\ \ \ \ \ \ = a \\
\qquad\qquad g\ a\ (x : xs) = g\ (a \oplus x)\ xs
\end{array}
$$

2.4.1. Duality

A well known duality property [4] is that when the operator $(\oplus)$ is associative and has the element e as its unit, $foldr$ and $foldl$ always give the same result. In fact, the opposite result also holds, giving the following equivalence:

$$
\begin{array}{l}
x \oplus (y \oplus z) = (x \oplus y) \oplus z \\
x \oplus e = e \oplus x = x
\end{array}
\quad \Leftrightarrow \quad foldr\ (\oplus)\ e\ xs = foldl\ (\oplus)\ e\ xs
$$

In the case of the product function, $(*)$ is associative and has 1 as its unit, so it can be re-expressed using fold-left:

$$productl = foldl\ (*)\ 1$$

Under Haskell's lazy evaluation strategy, the outermost redex is chosen to be evaluated first, so the recursive call is evaluated before the accumulator expression. This is illustrated in evaluation trace A of figure 3. To force evaluation of the multiplication first we can introduce a strictness annotation, $\$!$. In the expression, $f\ \$!\ x$, the strictness annotation will ensure that x is evaluated first, though only enough to check that it is not undefined (head-normal form), before $f\ \ x$ is evaluated [4]. Fold-left can be modified using the strictness operator as so:

$$
\begin{array}{l}
foldl'\ (\oplus)\ v = g\ v \\
\quad \textbf{where}\ g\ a\ [\,]\ \ \ \ \ \ = a \\
\qquad\qquad g\ a\ (x : xs) = (g\ \$!\ (a \oplus x))\ xs
\end{array}
$$

Re-expressing product using $foldl'$ now means it is evaluated as in trace B in figure 3, with the evaluation of the multiplication now occurring before the recursive call.

FIGURE 3: Evaluation traces for $productl$ and $productl'$

A)		B)	
	$productl\ (3:2:1:[])$		$productl'\ (3:2:1:[])$
$=$	$g\ 1\ (3:2:1:[])$	$=$	$g\ 1\ (3:2:1:[])$
$=$	$g\ (1*3)\ (2:1:[])$	$=$	$g\ (1*3)\ (2:1:[])$
$=$	$g\ ((1*3)*2)\ (1:[])$	$=$	$g\ 3\ (2:1:[])$
$=$	$g\ (((1*3)*2)*1)\ []$	$=$	$g\ (3*2)\ (1:[])$
$=$	$((1*3)*2)*1$	$=$	$g\ 6\ (1:[])$
$=$	$(3*2)*1$	$=$	$g\ (6*1)\ []$
$=$	$6*1$	$=$	$g\ 6\ []$
$=$	6	$=$	6

2.5. Left hylomorphism

The corresponding hylomorphism theorem for fold-left is:

$$hylol\ p\ hd\ tl\ (\oplus)\ v = h\ v$$
$$\quad \textbf{where}\ h\ a\ x = \textbf{if}\ p\ x\ \textbf{then}\ a\ \textbf{else}\ (h\ \$!\ (a \oplus hd\ x))\ (tl\ x)$$

Although straightforward, to the best of our knowledge, this operator has not been considered before.

2.5.1. Proof of left hylomorphism theorem

Structural induction cannot be used to prove that this definition satisfies the specification above, because there is nothing to do induction over; we do not know the structure of the seed value to the unfold. There is also no structured result to do co-induction over. However, because both $foldl'$ and $unfold$ are defined as fixpoints and therefore $hylol$ is a composition of two fixpoints, we can apply the "total fusion" [5] theorem. This states that a function that is the composition of two fixpoints, is related by:

$$\frac{fa \circ gb = h(a \circ b)}{\mu f \circ \mu g = \mu h}$$

We can prove the total fusion theorem using fixpoint induction [6].

$$\frac{P\bot \qquad \forall x.Px \Rightarrow P(fx)}{P(\mu f)}$$

The assumptions here are that types are complete partial orders (CPOs), which are sets with a partial-ordering $\sqsubseteq$, a least element $\bot$, and limits of all non-empty chains, and programs are continuous functions, functions between CPOs that preserve the partial-order and limit structure.

$$\begin{array}{ll} & \mu f \circ \mu g = \mu h \\ \Leftrightarrow & \quad \{\ \text{define } P(a,b,c) = a \circ b \equiv c\ \} \\ & P(\mu f, \mu g, \mu h) \\ \Leftarrow & \quad \{\ \text{fixpoint induction}\ \} \\ & P(\bot,\bot,\bot) \wedge \forall a,b,c.P(a,b,c) \Rightarrow P(fa,gb,hc) \end{array}$$

Showing that the first conjunct is satisfied is trivial ($\bot \circ \bot \equiv \bot$), so we proceed straight to verifying the second conjunct:

$$
\begin{array}{ll}
 & P(fa, gb, hc) \\
\Leftrightarrow & \quad \{ \text{ definition of } P \} \\
 & fa \circ gb = hc \\
\Leftarrow & \quad \{ \text{ assumption: } P(a,b,c), \text{ i.e. } a \circ b \equiv c \} \\
 & fa \circ gb = h(a \circ b)
\end{array}
$$

This completes the proof, apart from showing that the predicate P is admissible (preserves limits of chains), which is immediate from the fact that any equality between continuous functions can be shown to be admissible, and that the composition of any two continuous functions is continuous.

To apply total fusion first we need to re-express $unfold$, $foldl'$ and $hylol$ in terms of least fixpoints:

$$
\begin{array}{lll}
unfold\mu\ p\ hd\ tl & = \mu\ unfold' & \\
\quad \textbf{where}\ unfold'\ g & = \lambda x \rightarrow \textbf{if}\ p\ x\ \textbf{then}\ [\,]\ \textbf{else}\ hd\ x : g\ (tl\ x) & \\
foldl'\mu\ (\oplus)\ a & = \mu\ foldl'' & \\
\quad \textbf{where}\ foldl''\ g & = \lambda xs\ a \rightarrow \textbf{case}\ xs\ \textbf{of} & \\
 & \qquad [\,] \rightarrow a & \\
 & \qquad y : ys \rightarrow g\ ys\ \$!\ (a \oplus y) & \\
hylol\mu\ p\ hd\ tl\ (\oplus)\ v & = \mu\ hylol' & \\
\quad \textbf{where}\ hylol'\ h & = \lambda x\ a \rightarrow\ \textbf{if}\ p\ x\ \textbf{then}\ a\ \textbf{else}\ h\ (tl\ x)\ (a \oplus hd\ x) &
\end{array}
$$

The list and accumulator arguments have been swapped over in the $foldl''$ and $hylol'$ functions, so the list is now the first argument. This is to make it easier to compose the fold-left and unfold in the proof, in that the result of the unfold (a list) is the first argument to the fold-left.

We can now prove the $hylol$ theorem:

$$
\begin{array}{ll}
 & foldl'\ (\oplus)\ v \circ unfold\ p\ hd\ tl = hylol\ p\ hd\ tl\ (\oplus)\ v \\
\Leftrightarrow & \quad \{ \text{ definition of the functions } \} \\
 & \mu\ foldl'' \circ \mu\ unfold' = \mu\ hylol' \\
\Leftarrow & \quad \{ \text{ total fusion } \} \\
 & foldl''\ b \circ unfold'\ c = hylol'\ (b \circ c)
\end{array}
$$

The final equation can be verified as follows:

$$
\begin{array}{ll}
 & (foldl''\ b \circ unfold'\ c)\ x\ a \\
= & \quad \{ \text{ definition of } \circ \} \\
 & foldl''\ b\ (unfold'\ c\ x)\ a \\
= & \quad \{ \text{ definition of } unfold' \} \\
 & foldl''\ b\ (\textbf{if}\ p\ x\ \textbf{then}\ [\,]\ \textbf{else}\ hd\ x : c\ (tl\ x))\ a \\
= & \quad \{ \text{ definition of } foldl' \} \\
 & \textbf{if}\ p\ x\ \textbf{then}\ a\ \textbf{else}\ b\ (c\ (tl\ x))\ \$!\ (a \oplus hd\ x) \\
= & \quad \{ \text{ definition of } \circ \} \\
 & \textbf{if}\ p\ x\ \textbf{then}\ a\ \textbf{else}\ ((b \circ c)\ (tl\ x))\ \$!\ (a \oplus hd\ x) \\
= & \quad \{ \text{ definition of } hylol' \} \\
 & hylol'\ (b \circ c)\ x\ a
\end{array}
$$

The functions $unfold'$, $foldl''$ and $hylol'$ are only locally defined above and so contain free variables, but we use them for clarity.

2.5.2. Example: left factorial

The factorial function can be re-expressed using the fold-left version of the product function:

$$factl = productl \circ downFrom$$

Applying the left-hylomorphism theorem gives the following fused definition:

$$
\begin{aligned}
&factl = h\ 1 \\
&\quad \mathbf{where}\ h\ a\ x = \mathbf{if}\ x \equiv 0\ \mathbf{then}\ a\ \mathbf{else}\ (h\ \$!\ (a * x))\ (x - 1)
\end{aligned}
$$

The resulting trace (B in figure 2) shows that the multiplication evaluation now happens as soon as the list elements are generated. The shape of the evaluation trace is different, because the evaluation now occurs in constant space; only the additional space to hold the accumulator is required.

2.6. Calculating an accumulator version

It is interesting to consider whether a function produced from the $hylor$ theorem can be turned into a space efficient version by calculation. In general, an accumulator version f' can be calculated, with an appropriate $\otimes$, for a function f using the specification:

$$f'\ a\ x = a \otimes f\ x$$

In the factorial example, we can attempt to calculate an accumulating version:

$$facta\ a\ x = a * fact\ x$$

The proof would proceed directly as:

$$
\begin{aligned}
& facta\ a\ x \\
=\quad & \{\ \text{specification}\ \} \\
& a * fact\ x \\
=\quad & \{\ \text{definition of } fact\ \} \\
& a * (\mathbf{if}\ x \equiv 0\ \mathbf{then}\ 1\ \mathbf{else}\ x * fact\ (x - 1)) \\
=\quad & \{\ \text{distribute through if-expression}\ \} \\
& \mathbf{if}\ x \equiv 0\ \mathbf{then}\ a\ \mathbf{else}\ a * (x * fact\ (x - 1)) \\
=\quad & \{\ *\ \text{is associative}\ \} \\
& \mathbf{if}\ x \equiv 0\ \mathbf{then}\ a\ \mathbf{else}\ (a * x) * fact\ (x - 1)
\end{aligned}
$$

The next step would be to substitute $facta\ a\ (x - 1)$ for $(a * x) * fact\ (x - 1)$, but we cannot do this because there is no induction hypothesis. One could be created for this specific case by induction on natural numbers, but not for the general case of functions produced using the $hylor$ rule. It is therefore not possible to produce an accumulator version in the general case from $hylor$, but this can instead be done by applying the $hylol$ theorem instead.

2.7. Strictness

The space performance of the original hylomorphism definition may in some cases still be constant. This occurs when the fold operator is non-strict in its second argument; it does not require the value of it to produce a result.

Example: prime

We can naively define a function that tests if a number is prime by creating a list from two up to the integer argument and checking to see that none of the list elements are divisors.

$$
\begin{aligned}
upto\ n &= unfold\ (\equiv n)\ id\ (+1)\ 2 \\
nodivisors\ n &= foldr\ (\lambda x\ xs \rightarrow n\ `mod`\ x \not\equiv 0 \wedge xs)\ True \\
prime\ n &= (nodivisors\ n \circ upto)\ n
\end{aligned}
$$

Applying the $hylor$ theorem, gives the fused function:

$$
\begin{aligned}
&prime\ n = h\ 2 \\
&\quad \mathbf{where}\ h\ x = \mathbf{if}\ (x \equiv n)\ \mathbf{then}\ True\ \mathbf{else}\ n\ `mod`\ x \not\equiv 0 \wedge h\ (x + 1)
\end{aligned}
$$

In Haskell, the conjunction function $\wedge$ is strict on its first argument, and non-strict in its second:

$$\begin{aligned} &\mathit{False} \wedge x = \mathit{False} \\ &\mathit{True} \wedge x = x \end{aligned}$$

Using this definition of $\wedge$, the evaluation trace for $\mathit{prime}\ 9$ is:

$$\begin{aligned} & \mathit{prime}\ 9 \\ = \quad & h\ 2 \\ = \quad & \mathit{False} \wedge h\ 3 \\ = \quad & h\ 3 \\ = \quad & \mathit{True} \wedge h\ 4 \\ = \quad & \mathit{True} \end{aligned}$$

The resulting trace has constant space requirements, because $\wedge$ can be evaluated solely based on the value of its first argument. If the conjunction was implemented differently, so that it was strict in both its arguments, then evaluation would occur as in the previous examples. The fold-left version of this function still has constant space requirements, though the time requirements are worse if the number isn't prime, because the fold-left always has a tail-recursive call, it can never exploit the laziness of the $\wedge$ if the first argument evaluates to False.

3. FORMALISING

We now seek to formalise the space performance results of the previous section. Inspired by our earlier work on measuring time performance [7], the approach here is to first transform the function whose space performance we wish to measure into an abstract machine that makes explicit how evaluation proceeds. This technique has been developed by Danvy *et al* [8] and has been applied in a calculational way by Hutton and Wright [9]. We then label the transitions of the machine with explicit space information, and reverse the transformation process to obtain a high-level function that measures the space behaviour of the original function. In the remainder of this section we show how this proceeds for the particular case of the hylor function.

3.1. Abstract machines

Let us start with the definition of the hylor function:

$$\begin{aligned} &\mathit{hylor}\ p\ \mathit{hd}\ \mathit{tl}\ (\oplus)\ v = f \\ &\quad \mathbf{where}\ f\ x = \mathbf{if}\ p\ x\ \mathbf{then}\ v\ \mathbf{else}\ \mathit{hd}\ x \oplus f\ (\mathit{tl}\ x) \end{aligned}$$

The first step in the process of obtaining an abstract machine that implements this function is to make the control flow explicit, by transforming the function into continuation-passing style [10], giving the following result:

$$\begin{aligned} &\mathit{hylorCPS}\ p\ \mathit{hd}\ \mathit{tl}\ (\oplus)\ v\ x = h\ x\ \mathit{id} \\ &\quad \mathbf{where}\ h\ x\ c = \mathbf{if}\ p\ x\ \mathbf{then}\ c\ v\ \mathbf{else}\ h\ (\mathit{tl}\ x)\ (\lambda z \rightarrow c\ (\mathit{hd}\ x \oplus z)) \end{aligned}$$

The next step is to replace the use of continuations by an explicit stack data structure, by applying the technique of defunctionalization [10], which results in the following definition:

$$\begin{aligned} &\mathbf{data}\ \mathit{Stack}\ a && = \mathit{TOP} \mid \mathit{PUSH}\ a\ (\mathit{Stack}\ a) \\ &\mathit{hyloMach}\ p\ \mathit{hd}\ \mathit{tl}\ (\oplus)\ v\ x && = h\ x\ \mathit{TOP} \\ &\quad \mathbf{where}\ h\ x\ c && = \mathbf{if}\ p\ x \\ &&& \quad\ \mathbf{then}\ \mathit{exec}\ c\ v \\ &&& \quad\ \mathbf{else}\ h\ (\mathit{tl}\ x)\ (\mathit{PUSH}\ (\mathit{hd}\ x \oplus)\ c) \\ &\qquad \mathit{exec}\ \mathit{TOP}\ v && = v \\ &\qquad \mathit{exec}\ (\mathit{PUSH}\ \mathit{yop}\ c)\ z && = \mathit{exec}\ c\ (\mathit{yop}\ z) \end{aligned}$$

We can now rewrite this function in the form of transition rules for an abstract machine with two states—the state (x, c) corresponds to evaluating an expression using the function call $h\ x\ c$, and $\langle c, v\rangle$ to executing a stack using the function call $exec\ c\ v$:

$$\begin{array}{lcl}(x, c) & \rightarrow & \mathbf{if}\ p\ x\ \mathbf{then}\ \langle c, v\rangle\ \mathbf{else}\ (tl\ x, PUSH\ (hd\ x\oplus)\ c)\\ \langle TOP, v\rangle & \rightarrow & v\\ \langle PUSH\ yop\ c, z\rangle & \rightarrow & \langle c, yop\ z\rangle\end{array}$$

Finally, we also specify the evaluation order of the $\mathbf{else}$ branch within these rules, by introducing explicit let bindings with strict semantics:

$$\begin{array}{lcl}(x, c) & \rightarrow & \mathbf{if}\ p\ x\\ & & \quad\mathbf{then}\ \langle c, v\rangle\\ & & \quad\mathbf{else\ let}\ t = tl\ x\\ & & \qquad\mathbf{in\ let}\ y = hd\ x\\ & & \qquad\quad\mathbf{in\ let}\ yop = (\oplus)\ y\\ & & \qquad\qquad\mathbf{in}\ (t, PUSH\ yop\ c)\\ \langle TOP, v\rangle & \rightarrow & v\\ \langle PUSH\ yop\ c, z\rangle & \rightarrow & \langle c, yop\ z\rangle\end{array}$$

Further details of this approach to transforming a function to an abstract machine can be found in [8, 9].

3.2. Memory management

To keep track of the space usage a memory manager data structure is introduced, consisting of a pair of non-negative integers:

$$\mathbf{type}\ MemMng = (Int, Int)$$

The first component of the pair is the amount of memory that has been explicitly freed at the current point, and the second is the amount that has been explicitly allocated:

$$\begin{array}{ll}freed & = fst\\ allocated & = snd\end{array}$$

As we shall see, both parts are necessary to capture an accurate space model, in that memory freed by earlier evaluation may be re-used by a later on. Two functions are defined on the manager to allocate and free memory, $alloc$ and $free$. To free some memory, the amount to be freed is simply added to the free memory integer, and is then available to use in later allocation requests:

$$\begin{array}{ll}free & :: Int \rightarrow MemMng \rightarrow MemMng\\ free\ n\ (f, a) & = (f + n, a)\end{array}$$

When allocating memory, the request is first satisfied using the pool of free memory that is currently available, by subtracting the amount from the free memory integer until it is zero, with the difference then added to the allocated memory integer:

$$\begin{array}{ll}alloc & :: Int \rightarrow MemMng \rightarrow MemMng\\ alloc\ n\ (f, a) & = (f \dot{-} n, a + (n \dot{-} f))\end{array}$$

For simplicity we assume an infinite amount of memory, and hence allocation requests are always successful. The auxiliary subtraction function, $x \dot{-} y$, is defined as the maximum of $x - y$ and 0, thereby ensuring that the result is never negative:

$$\begin{array}{ll}(\dot{-}) & :: Int \rightarrow Int \rightarrow Int\\ x \dot{-} y & = x - y \uparrow 0\end{array}$$

For the purposes of later proofs, we will exploit the following properties for these functions, which can easily be proved from the above definitions:

$$\begin{aligned} \mathit{free}\ n \circ \mathit{free}\ m &= \mathit{free}\ (m+n) && (1)\\ \mathit{alloc}\ n \circ \mathit{alloc}\ m &= \mathit{alloc}\ (m+n) && (2)\\ \mathit{alloc}\ n \circ \mathit{free}\ n &= \mathit{id} && (3)\\ \mathit{allocated} \circ \mathit{free}\ n &= \mathit{allocated} && (4) \end{aligned}$$

The first and second properties express that repeated occurrences of free or alloc may be accumulated. The third states that an alloc immediately followed by a free of the same amount has no effect, since the allocation can use up the previously freed amount. Finally, the last property expresses that freeing memory does not affect the amount allocated.

3.3. Space costs

For the purposes of assigning space costs we use the notation x_s to denote the space requirements for evaluating x. In the case when x is a piece of data, this will be a non-negative integer representing the size of that data, which we measure by simply counting constructors. For example, the cost of the stack data structure is defined recursively as follows:

$$\begin{aligned} \mathit{TOP}_s &= 1\\ (\mathit{PUSH}\ x\ c)_s &= 1 + x_s + c_s \end{aligned}$$

In the case of a function f of a single argument, f_s will be a function that takes this argument along with a memory manager, and returns a modified memory manager that reflects the cost of this application. For example, the cost of applying the tail function on lists can be expressed as follows:

$$\mathit{tail}_s\ (x : xs) = \mathit{free}\ (1 + x_s)$$

Functions with multiple arguments can be treated in the same way by exploiting currying, resulting in a function of n arguments having n unary cost functions.

3.4. Transition costs

To add space information to the abstract machine, a way of instrumenting each transition with its cost is required. The space requirements are added using an accumulator, so that it remains an abstract machine. The accumulator is a memory manager and is updated according to the structure of the transition. For a basic transition of the form $x \to y$ we can perform an update operation $\mathit{update}\ x_s\ y_s$, when provided with the sizes of the data structures on the left and right-hand of the transition (before and after the transition occurs). The update captures the idea that as much space is-used as possible. First the space occupied by structures in x that don't occur in y is freed, allowing it to be re-used, and then the space for additional structures, that appear only in y, is allocated. We can defined the update function as:

$$\begin{aligned} \mathit{update} &:: \mathit{Int} \to \mathit{Int} \to \mathit{MemMng} \to \mathit{MemMng}\\ \mathit{update}\ x_s\ y_s &= \mathit{alloc}\ (y_s \dot{-} x_s) \circ \mathit{free}\ (x_s \dot{-} y_s) \end{aligned}$$

There are two special cases to consider, when transitions are of the structure $\mathbf{let}$ or $\mathbf{if}$. For transitions of the form $x \to \mathbf{let}\ y = f\ x'\ \mathbf{in}\ z$, initially the space for the argument x' is allocated, then the space requirements of the function f applied to x' is performed, and finally an update occurs, with the sizes of the left hand (which now includes the new bound data y) and right-hand-side, $\mathit{update}\ (x_s + y_s)\ z_s$. Altogether this occurs as:

$$\mathit{update}\ (x_s + y_s)\ z_s \circ f_s\ x' \circ \mathit{alloc}\ {x'}_s$$

Similarly in the $\mathbf{if}$ case, the space cost of performing the transition $x \to \mathbf{if}\ p\ x'\ \mathbf{then}\ y\ \mathbf{else}\ z$ first allocates the space for x', then applies the cost of applying the function p to x'. If the predicate $p\ x'$

evaluates to $True$ then an update occurs with the size of the left-hand-side $x_s + True_s$ and right-hand-side y_s, and if it is $False$ then the size of the left-hand-side is $x_s + False_s$ and right-hand-side z_s.

$$(\mathbf{if}\ p\ x'\ \mathbf{then}\ update\ (x_s + True_s)\ y_s\ \mathbf{else}\ update\ (x_s + False_s)\ z_s) \circ p_s\ x' \circ alloc\ x'_s$$

In the new machine each argument is paired with its space cost, as defined in the previous section. For example x is replaced by (x, x_s). The resulting machine, which has also been simplified by inlining the definition of $update$ and applying the properties in section 3.2, is given below:

$$\begin{aligned}
&spaceMach\ (p, p_s)\ (hd, hd_s)\ (tl, tl_s)\ ((\oplus), (\oplus_{s1}), (\oplus_{s2}))\ v\ (x, x_s)\ m = h\ x\ (alloc\ 1\ m)\ TOP\\
&\quad \mathbf{where}\ h\ x\ m\ c = \mathbf{if}\ p\ x\\
&\qquad\qquad \mathbf{then}\ exec\ c\ v\ ((free\ (x_s + 1) \circ p_s\ x \circ alloc\ (x_s))\ m)\\
&\qquad\qquad \mathbf{else\ let}\ t = tl\ x\\
&\qquad\qquad\qquad\qquad y = hd\ x\\
&\qquad\qquad\qquad\qquad yop = (\oplus)\ y\\
&\qquad\qquad\qquad \mathbf{in}\ h\ t\ ((alloc\ 1 \circ free\ x_s \circ (\oplus_{s1})\ y \circ hd_s\ x \circ alloc\ x_s \circ tl_s\ x \circ\\
&\qquad\qquad\qquad\qquad\qquad alloc\ x_s \circ free\ 1 \circ p_s\ x \circ alloc\ x_s)\ m)\ (PUSH\ yop\ c)\\
&\quad exec\ TOP\ v\ m \qquad\quad = free\ 1\ m\\
&\quad exec\ (PUSH\ yop\ c)\ z\ m = exec\ c\ (yop\ z)\ ((free\ 1 \circ (\oplus_{s2})\ z)\ m)
\end{aligned}$$

The next step is to perform the same program transformations, but in the reverse order, to produce a high-level function that measures the space from the abstract machine. After refunctionalizing the continuation and transforming from CPS, the following accumulator version is produced:

$$\begin{aligned}
&spacer\ (p, p_s)\ (hd, hd_s)\ (tl, tl_s)\ ((\oplus), (\oplus_{s1}), (\oplus_{s2}))\ v\ (x, x_s) = free\ 1 \circ h\ x \circ alloc\ 1\\
&\quad \mathbf{where}\ h\ x = \mathbf{if}\ p\ x\\
&\qquad\qquad \mathbf{then}\ free\ (x_s + 1) \circ p_s\ x \circ alloc\ x_s\\
&\qquad\qquad \mathbf{else\ let}\ t = tl\ x\\
&\qquad\qquad\qquad\qquad z = hylor\ p\ hd\ tl\ (\oplus)\ v\ t\\
&\qquad\qquad\qquad \mathbf{in}\ free\ 1 \circ (\oplus_{s2})\ z \circ h\ t \circ alloc\ 1 \circ free\ x_s \circ (\oplus_{s1})\ (hd\ x) \circ hd_s\ x \circ\\
&\qquad\qquad\qquad\qquad alloc\ x_s \circ tl_s\ x \circ alloc\ x_s \circ free\ 1 \circ p_s\ x \circ alloc\ x_s
\end{aligned}$$

In the next section, we will use this derived function to prove the space properties of the $factorial$ example function.

3.5. Example: factorial space

We can analyse the space performance of the factorial function by first producing space requirements functions for the primitive functions: equivalence to zero, multiplication and predecessor functions. This is done simply by taking the difference in size between the input and output, for example, if we define the size of an integer to be one unit of space, then the multiplication function will free one unit of space, since it takes two integers as arguments and the result is one integer.

Applying the $spacer$ function and inlining the primitive space functions gives the following result:

$$\begin{aligned}
&spaceFact\ x \quad = free\ 1 \circ f\ x \circ alloc\ 1\\
&\quad \mathbf{where}\ f\ x = \mathbf{if}\ (x \equiv 0)\ \mathbf{then}\ free\ 2 \circ alloc\ 1\ \mathbf{else}\ free\ 2 \circ f\ (x - 1) \circ alloc\ 2
\end{aligned}$$

The resulting function shows how, for each recursive call, two units need to be allocated before the call, which are then released afterwards.

To prove that the space requirements are linear we can form a specification that says, if we free n units of space initially and execute the space requirements function, then the allocated amount of memory will be unchanged. This means there was no need to request more memory, since the pool of n units of free memory was sufficient for evaluation. If we can prove this specification, then we can say that the function executes in n units of memory.

We can prove that the space requirements for the factorial function, $spaceFact\ x$, is linear, by showing that it executes in $2 * x$ units of additional space. This is achieved by proving the following specification:

$$allocated \circ spaceFact\ x \circ free\ (2 * x) = allocated$$

Proof by induction over natural number x:

Case: 0

$$\begin{array}{ll} & allocated \circ spaceFact\ 1 \circ free\ 2 \\ = & \quad \{\ \text{definition of } spaceFact\ \} \\ & allocated \circ free\ 1 \circ f\ 1 \circ alloc\ 1 \circ free\ 2 \\ = & \quad \{\ \text{definition of } f\ \} \\ & allocated \circ free\ 1 \circ free\ 2 \circ alloc\ 1 \circ alloc\ 1 \circ free\ 2 \\ = & \quad \{\ \text{property 3: } alloc\ n \circ free\ n = id\ \} \\ & allocated \circ free\ 1 \circ free\ 2 \\ = & \quad \{\ \text{property 1: } free\ n \circ free\ m = free\ (m + n)\ \} \\ & allocated \circ free\ 3 \\ = & \quad \{\ \text{property 4: } allocated \circ free\ n = allocated\ \} \\ & allocated \end{array}$$

Case: $x + 1$

$$\begin{array}{ll} & allocated \circ spaceFact\ (x + 1) \circ free\ (2 + 2 * x) \\ = & \quad \{\ \text{definition of } spaceFact\ \} \\ & allocated \circ free\ 1 \circ f\ (x + 1) \circ alloc\ 1 \circ free\ (2 + 2 * x) \\ = & \quad \{\ \text{definition of } f\ \} \\ & allocated \circ free\ 1 \circ free\ 2 \circ f\ x \circ alloc\ 2 \circ alloc\ 1 \circ free\ (2 + 2 * x) \\ = & \quad \{\ \text{property 3: } alloc\ n \circ free\ n = id\ \} \\ & allocated \circ free\ 1 \circ free\ 2 \circ f\ x \circ alloc\ 1 \circ free\ 2 * x \\ = & \quad \{\ \text{definition of } spaceFact\ \} \\ & allocated \circ free\ 2 \circ spaceFact \circ free\ 2 * x \\ = & \quad \{\ \text{induction hypothesis}\ \} \\ & allocated \end{array}$$

We can perform a similar calculation for the left factorial version, using the left-hylo function. The function $hylol$ is already an abstract machine, since it is tail-recursive and first order, so we can directly annotate the function in the same way with the space rules, to produce the following function:

$$\begin{array}{l} spacel\ (p, p_s)\ (hd, hd_s)\ (tl, tl_s)\ ((\oplus), (\oplus_{s1}), (\oplus_{s2}))\ v\ (x, x_s) = h\ v\ x \\ \quad \mathbf{where}\ h\ a\ x = \mathbf{if}\ p\ x \\ \qquad\qquad\qquad\qquad \mathbf{then}\ free\ (x_s + 1) \circ p_s\ x \circ alloc\ x_s \\ \qquad\qquad\qquad\qquad \mathbf{else}\ h\ (a \oplus hd\ x)\ (tl\ x) \circ tl_s\ x \circ (\oplus_{s2})\ a \circ (\oplus_{s1})\ (hd\ x) \circ \\ \qquad\qquad\qquad\qquad\qquad\qquad hd_s\ x \circ alloc\ x_s \circ free\ 1 \circ p_s\ x \circ alloc\ x_s \end{array}$$

Applying the space requirements function, gives the following function for the left factorial:

$$\begin{array}{ll} spaceFactl & = h \\ \quad \mathbf{where}\ h\ x = \mathbf{if}\ x \equiv 0\ \mathbf{then}\ free\ 2 \circ alloc\ 1\ \mathbf{else}\ h\ (x - 1) \circ free\ 2 \circ alloc\ 2 \end{array}$$

In this function, for each recursive call, two units of space are allocated but then freed before the call, so there is no overhead. We can prove that the additional space for evaluating $spaceFactl\ x$ is two units, using the specification:

$$allocated \circ spaceFactl\ x \circ free\ 2 = allocated$$

Proof by induction over natural number x:

Case: 0

$$
\begin{array}{ll}
 & allocated \circ spaceFactl\ 1 \circ free\ 2 \\
= & \quad \{\ \text{definition of } spaceFactl\ \} \\
 & allocated \circ free\ 2 \circ alloc\ 1 \circ free\ 2 \\
= & \quad \{\ \text{property 3: } alloc\ n \circ free\ n = id\ \} \\
 & allocated \circ free\ 2 \circ free\ 1 \\
= & \quad \{\ \text{property 1: } free\ n \circ free\ m = free\ (m + n)\ \} \\
 & allocated \circ free\ 3 \\
= & \quad \{\ \text{property 4: } allocated \circ free\ n = allocated\ \} \\
 & allocated
\end{array}
$$

Case: $x + 1$

$$
\begin{array}{ll}
 & allocated \circ spaceFactl\ (x + 1) \circ free\ 2 \\
= & \quad \{\ \text{definition of } spaceFactl\ \} \\
 & allocated \circ spaceFactl\ x \circ free\ 2 \circ alloc\ 2 \circ free\ 2 \\
= & \quad \{\ \text{property 3: } alloc\ n \circ free\ n = id\ \} \\
 & allocated \circ spaceFactl\ x \circ free\ 2 \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
 & allocated
\end{array}
$$

3.6. Example: converting to binary

Calculating the binary representation of a natural number, as a list of zeros and ones, can be expressed as an unfold. The head of the list is created by taking the argument modulo two, and tail is computed by repeatedly dividing the current number by two, until we reach zero.

$$toBinSigRight = unfold\ (\equiv 0)\ (`mod`2)\ (`div`2)$$

This produces a list where the most significant bit is on the right, which can then be reversed to give the standard representation. Reversing a list can be written as a fold-right:

$$reverse' = foldr\ (\lambda x\ xs \rightarrow xs +\!\!+ [x])\ [\,]$$

The composition of these two definitions yields a function that calculates the standard binary representation of a natural number as a list of bits, with the most significant bit on the left:

$$toBin = reverse' \circ toBinSigRight$$

For example, $toBin\ 10 = [1, 0, 1, 0]$. Applying the $hylor$ theorem gives the fused definition:

$$
\begin{array}{l}
toBin = h \\
\quad \textbf{where}\ h\ x = \textbf{if}\ x \equiv 0\ \textbf{then}\ [\,]\ \textbf{else}\ h\ (x\ `div`\ 2) +\!\!+ [x\ `mod`\ 2]
\end{array}
$$

It is not possible for evaluation to occur in constant space, because the result creates a list of non-constant size, but we can look at the additional space required aside from the result. Applying the space function, yields the requirements function:

$$
\begin{array}{l}
spaceToBin\ x = free\ 1 \circ h\ x \circ alloc\ 1 \\
\quad \textbf{where}\ h\ x = \textbf{if}\ x \equiv 0\ \textbf{then}\ free\ 2 \circ alloc\ 1\ \textbf{else}\ free\ 2 \circ h\ (x\ `div`\ 2) \circ alloc\ 4
\end{array}
$$

From this, we can prove the minimum additional space required for evaluation is $6 + 4 * \log_2 n$. The size of the result is a list sized $1 + 2 * \log_2 n$ (one unit for each 0 and 1 and list-constructor plus an additional unit to hold the empty list). Not including this space, the evaluation requires an additional $5+2*\log_2 n$ units. We can compare this to the accumulating version by re-expressing the reverse function using fold-left. Another fold duality theorem [4] states that a fold-right operation

may be rewritten as a fold-left if their operators associate with each other, and the empty list value, v, is the right and left unit for the fold-right and fold-left operator respectively:

$$\begin{array}{l} x \oplus (y \otimes z) = (x \oplus y) \otimes z \\ x \oplus v = v \otimes x \end{array} \quad \Rightarrow \quad foldr\ (\oplus)\ v\ xs = foldl\ (\otimes)\ v\ xs$$

Since $(z : y) +\!\!+ [x] = z : (y +\!\!+ [x])$ and $[\,] +\!\!+ [x] = x : [\,]$, we can apply this theorem to get:

$$foldr\ (\lambda x\ xs \rightarrow xs +\!\!+ [x])\ [\,] = foldl\ (flip\ (:))\ [\,]$$

If we redefine the $toBin$ function and apply the $hylol$ rule, we end up with the following definition:

$$\begin{array}{l} toBinl = g\ [\,] \\ \quad \textbf{where}\ g\ a\ x = \textbf{if}\ x \equiv 0\ \textbf{then}\ a\ \textbf{else}\ (g\ \$!\ (x\ `mod`\ 2 : a))\ (x\ `div`\ 2) \end{array}$$

The left-hylo space function can then be applied to get the space requirements function:

$$\begin{array}{l} spaceToBinl = h \\ \quad \textbf{where}\ h\ x = \textbf{if}\ x \equiv 0\ \textbf{then}\ free\ 3 \circ alloc\ 2\ \textbf{else}\ h\ (x\ `div`\ 2) \circ alloc\ 2 \end{array}$$

This function gives the space requirements $4 + 2 * \log_2 n$, but not including the space that the result is occupying, it only requires a constant 3 units for evaluation.

The results of the two space requirements functions show that the right-hylo version requires additional space proportional to the size of the result, whereas the left version only requires a constant amount of additional space.

4. CONCLUSION AND FURTHER WORK

The aim of applying fusion theorems, such as the hylomorphism theorem, is to eliminate the intermediate data structure produced. However, we have shown that this is only achieved if the generating function produces elements in the *same order* as they are consumed. The examples given illustrate this impedance mismatch and show how an accumulator version, using fold-left, is often a solution. The accumulator is then able to evaluate the elements generated in-place, rather than waiting until the end, giving improved space performance.

The space results may be observed informally by looking at evaluation traces, but we can get more concrete space measures by using program transformation techniques to derive the underlying abstract machine. At this level we can measure data structures that were not visible at the original function level. The machine can then be instrumented with space usage and then the transformations reversed to get a resulting space requirements function. This can then be used to prove the space performance.

Applying this technique to more general structures is not so simple, since fold-left cannot be generalised as fold-right can. There are more restrictive functions that can be generalised, such as *crush* [11], where the structure is first flattened to a list and then folded. The same idea, of using an accumulating fold, may be applied to improve the space usage. How to extend this approach to other structures would be an interesting topic for further work.

The authors would like to thank Andres Löh and the anonymous referees for their thorough comments, which have greatly improved both the content and presentation of the paper.

REFERENCES

[1] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, number 523 in LNCS. Springer-Verlag, 1991.

[2] S. Peyton Jones. Haskell 98 language and libraries: The revised report. Technical report.
[3] Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.
[4] Richard Bird. *Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice-Hall, second edition, 1998.
[5] Erik Meijer. *Calculating Compilers*. PhD thesis, 1992.
[6] Jeremy Gibbons and Graham Hutton. Proof Methods for Corecursive Programs. *Fundamenta Informaticae Special Issue on Program Transformation*, 66(4):353–366, 2005.
[7] Catherine Hope and Graham Hutton. Accurate Step Counting. In *Proceedings of the 17th International Workshop on the Implementation and Application of Functional Languages*, Dublin, Ireland, September 2005. To appear in the volume of selected papers from IFL 2005, to be published by Springer.
[8] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. Technical report, 2003. RS-03-13.
[9] Graham Hutton and Joel Wright. Calculating an Exceptional Machine. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming volume 5*. Intellect, February 2006. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004.
[10] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher Order Symbol. Comput.*, 11(4):363–397, 1998.
[11] Lambert Meertens. Calculate polytypically! In *Proceedings 8th Int. Symp. on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany, 24–27 Sept 1996*, volume 1140, pages 1–16. Springer-Verlag, 1996.