Generalising and Dualising the Third List-Homomorphism Theorem

Shin-Cheng Mu
Academia Sinica, Taiwan
scrm@iis.sinica.edu.tw

Akimasa Morihata
Tohoku University, Japan
morihata@riei.tohoku.ac.jp

Abstract
The third list-homomorphism theorem says that a function is a list homomorphism if it can be described as an instance of both a foldr and a foldl. We prove a dual theorem for unfolds and generalise both theorems to trees: if a function generating a list or tree can be described both as an unfoldr and an unfoldl, the list or tree can be generated from the middle, and a function that processes or builds a tree both upwards and downwards may independently process/build a subtree and its one-hole context. The point-free, relational formalism helps to reveal the beautiful symmetry hidden in the theorem.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Theory

Keywords program derivation, third list homomorphism theorem

1. Introduction
As multicore hardware has become standard in recent years, parallel programming rekindles as an important potential application of functional programming. The skeletal parallel programming [3] paradigm proposes the idea of developing parallel programs by combining parallel skeletons — functions that capture useful parallel programming patterns. Among the important parallel skeletons is list homomorphism [1], one that satisfies the equation

\[ h(xs + ys) = hxs \circ hys, \]

which says that to compute \( h \), one may arbitrarily split the input list into \( xs + ys \), compute \( h \) on them recursively in parallel, and combine the results using an associative operator \( \circ \).

A well-known third list-homomorphism theorem [6] says that a function is a list homomorphism if it can be described as an instance of both foldr and foldl. For example, since \( \text{sum} = \text{foldr } (+) \ 0 = \text{foldl } (+) \ 0 \), there exists some \( \circ \) such that \( \text{sum } (xs + ys) = \text{sum } xs \circ \text{sum } ys \), — for this simple example, \( \circ \) happens to be \( + \) as well. One naturally wonders whether \( \circ \) can be mechanically constructed. Such methods have been proposed [5, 9, 11], and even generalised to trees [10].

Less noticed, however, is that the theorem and its proof dualise very well to unfolds on lists. Consider the function \( \text{fromTo } (x, y) = [x, x + 1 \ldots y] \). One may imagine three possible implementations: generating the list from the left, from the right, and from some arbitrary point in the middle. Is it true that any function that can be defined as both an unfoldr and an unfoldl can be written as one that generates the list from the middle?

We show in this pearl that the answer is positive. This is not only of theoretical interest but could also have a practical impact. First, there are several efficient algorithms that are based on divide-and-conquer sequence generation, such as Quicksort. Moreover, the performance bottleneck in distributed parallel computing often lies in data distribution. Being able to generate the list anywhere allows us to distribute seeds of sublists and simultaneously generate from them, and thereby reduce communication costs and increase parallelism.

List homomorphisms and the third list-homomorphism theorem are reviewed in Section 2, before we present a dualised theorem in Section 3 and apply it, in Section 4, to examples including sorting and parallel scan. In Section 5, the results are further generalised to trees, before we conclude in Section 6.

2. The Third List-Homomorphism Theorem
As is well known, in the world of sets and total functions, the equations

\[
\begin{align*}
  h([]) &= e \\
  h(x:xs) &= f_a(x, hxs),
\end{align*}
\]

(1)

where \( e = b \) and \( f_a : (a \times b) \rightarrow b \), have a unique solution for \( h : [a] \rightarrow b \), denoted by \( \text{foldr } f_a e \). We deviate from the standard and let \( f_a \) be uncurried since it is more convenient in point-free style, where programs are described by function composition rather than application. In fact, we will also introduce uncurried constructor \( \text{cons } (x, xs) = x : xs \), and let \( (f \times g) (x, y) = (f x, g y) \), which satisfies a law \( (f \times g) (h \times k) = (f h \times g k) \), which we will refer to as “product functor”. Thus (1) can be written

\[ h \ast \text{cons } = f_a \ast (id \times h). \]

We define a variation of \( \text{foldr} \) that takes the base case as an extra argument,

\[
\begin{align*}
  \text{foldrr} &:: ((a \times b) \rightarrow b) \rightarrow ([a], b) \rightarrow b \\
  \text{foldrr } f_a ([], e) &= e \\
  \text{foldrr } f_a (x : xs, e) &= f_a (x, \text{foldrr } f_a (xs, e)),
\end{align*}
\]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP’11, September 19–21, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0865-6/11/09... $10.00
It can be seen as a “resumed” version of foldr (hence the suffix “r” in the name), that is, if \( h = \text{foldr} \ f \ e \), one can easily show by induction on \( xs \) that
\[
\begin{align*}
  h \ (xs + y) &= \text{foldr} \ f \ a \ (xs, h \ ys). \\
\end{align*}
\]
(2)

Let \( \text{cat} \ (xs, ys) = xs + ys \), (2) can be written point-free as
\[
\begin{align*}
  h \ * \ \text{cat} &= \text{foldr} \ f \ a \ (id \times h). \\
\end{align*}
\]
(3)

Symmetrically, let \( \text{snoc} \ (xs, x) = xs + [x] \). It is known that \( \text{foldl} \ f \ e \) is the unique solution for \( h : [a] \rightarrow b \) in
\[
\begin{align*}
  h \ [\ ] &= e \\
  h \ * \ \text{snoc} &= f \ * \ (h \times id),
\end{align*}
\]
where \( f : (b \times a) \rightarrow b \). Defining resumable \( \text{foldl} \) as
\[
\begin{align*}
  \text{foldlr} : ((b \times a) \rightarrow b) \rightarrow (b, [a]) \rightarrow b \\
  \text{foldlr} \ f \ e \ ([\ ] \rightarrow e) &= e \\
  \text{foldlr} \ f \ e \ (x, ys + [x]) &= f \ (\text{foldlr} \ f \ e \ (x, ys), x),
\end{align*}
\]
we have, if \( h = \text{foldl} \ f \ e \), that
\[
\begin{align*}
  h \ * \ \text{cat} &= \text{foldlr} \ f \ e \ (h \times id). \\
\end{align*}
\]
(4)

A function \( h : [a] \rightarrow b \) is a list homomorphism if there exist \( e : b \), \( k : a \rightarrow b \), and \( f : (b \times b) \rightarrow b \) such that
\[
\begin{align*}
  h \ [\ ] &= e \\
  h \ [x] &= k \ x \\
  h \ (xs + ys) &= f \ (h \ xs, h \ ys).
\end{align*}
\]

In such a case we denote \( h \) by \( \text{hom} \ f \ k \ e \). The equations imply that \( f \) is associative, on the range of \( h \), with unit \( e \). To compute a list homomorphism \( h \), one may split the input list arbitrarily into two parts, recursively compute \( h \) on both parts, and combine the results using \( f \), implying a potential for parallel computation. If \( f \) and \( k \) are constant-time operations, a list homomorphism can be evaluated in time \( O(n/p + \log p) \), where \( n \) is the length of the input list and \( p \) the number of processors, and we get almost linear speedups with respect to \( p \).

Some famous theorems relate \( \text{foldr} \), \( \text{foldl} \) and list homomorphisms. Firstly, since the input list can be split arbitrarily, we may of course choose a biased split, reducing a list homomorphism to a \( \text{foldr} \) or a \( \text{foldl} \).

**Theorem 1** (the 2nd list-hom. theorem [1]). If \( h = \text{hom} \ f \ k \ e \) then \( h = \text{foldr} \ f \ e \) \( = \) \( \text{foldl} \ f \ e \), where \( f \ a \ (x, v) = f \ (k \ x, v) \) and \( f \ e \ (v, x) = f \ (v, k \ x) \).

Somewhat surprisingly, if a function can be computed both by a \( \text{foldr} \) and a \( \text{foldl} \), it is a list homomorphism.

**Theorem 2** (the 3rd list-hom. theorem [6]). \( h = \text{foldr} \ f \ e \) \( = \) \( \text{foldl} \ f \ e \) implies \( h = \text{hom} \ f \ k \ e \) for some \( f \) and \( k \).

**Proof.** The only possible choice for \( k \) is \( k \ x = h \ [x] \). The aim is to find \( f \) such that \( h \ * \ \text{cat} = f \ * \ (h \times h) \). A function \( f^{-1} \) is called a **right inverse** of \( f \) if for all \( y \) in the range of \( f \), we have \( f \ (f^{-1} \ y) = y \). Equivalently, \( f \ * \ f^{-1} \ * \ f = f \). In a set-theoretical model, a right inverse always exists but may not be unique.

While a semantical proof was given by Gibbons [6], we will provide a proof having a more equational flavour. We reason:
\[
\begin{align*}
  h \ * \ \text{cat} &= \{ \ (3) \ \\ \\
  \text{foldr} \ f \ a \ (id \times h) &= \{ \ h = h \ * \ h^{-1} \ * \ h, \ \text{product functor} \ \\ \\
  \text{foldr} \ f \ a \ (id \times h) &= \{ \ (3) \ \text{backwards, and (4)} \ \\ \\
  \text{foldl} \ f \ a \ (h \ * \ id) &= \{ \ h = h \ * \ h^{-1} \ * \ h, \ \text{product functor} \ \\ \\
  \text{foldl} \ f \ a \ (h \ * \ id) &= \{ \ (4) \ \text{backwards} \ \\ \\
  h \ * \ \text{cat} &= (h \ * \ h^{-1}) \ * \ (h \times h).
\end{align*}
\]

Thus the theorem holds if we pick \( f = h \ * \ \text{cat} = (h \ * \ h^{-1}) \).

**Theorem 2** in fact provides hints how to construct list homomorphisms. For example, since \( \text{sum} = \text{foldr} \ (+) \ 0 = \text{foldl} \ (+) \ 0 \), **Theorem 2** states that \( \text{sum} \) can be written as \( \text{sum} = \text{hom} \ f \ k \ o \) where \( k \ x = \text{sum} \ [x] \) and \( f \ (v, w) = \text{sum} \ (g \ v \ + \ g \ w) \) for any right inverse \( g \) of \( \text{sum} \). One may simply pick \( g \ x = [x] \), and \( f \ (v, w) \) simplifies to \( v + w \).

Readers might have noticed something odd in the proof: the property much talked about, that \( h \) being both a \( \text{foldr} \) and a \( \text{foldl} \), could be weakened — properties (3) and (4) were merely used to push \( h \) to the right. In fact, \( h \ * \ \text{cat} \) is never expanded in the proof. One thus wonders whether there is something more general waiting to be discovered, which is indeed what we will see in the following sections. The syntactical approach makes such generalisations much easier to spot.

### 3. Dual of Third List-Homomorphism Theorem

The function \( \text{unfoldr} \), a dual of \( \text{foldr} \) that generates a list from left to right, may be defined as follows.¹
\[
\begin{align*}
\text{unfoldr} : (b \rightarrow (a, b)) \rightarrow (b \rightarrow \text{Bool}) \rightarrow b \rightarrow [a] \\
\text{unfoldr} \ g \ p \ v \\
  | \ (x, v') \leftarrow \ g \ v \times x : \text{unfoldr} \ g \ p \ v'.
\end{align*}
\]

Symmetrically, the function \( \text{unfoldl} \) is defined by
\[
\begin{align*}
\text{unfoldl} : (b \rightarrow (b, a)) \rightarrow (b \rightarrow \text{Bool}) \rightarrow b \rightarrow [a] \\
\text{unfoldl} \ g \ p \ v \\
  | \ (v', x) \leftarrow \ g \times v \ = \ \text{unfoldl} \ g \ p \ v' + \ [x].
\end{align*}
\]

Typically, unfolds are defined for coinductive, possibly infinite lists. Since we want the unfolded lists to have both a left end and a right end, and for another important technical reason to be mentioned later, our “unfolds” in this pearl return inductive, finite lists and require separate proofs that all successive applications of \( g \) and \( g \) eventually reaches some \( v \) for which \( p \ v \) is true. Due to space constraints, however, the proof of termination is usually treated informally.

Finally, we denote a function \( k : b \rightarrow [a] \) by \( \text{unhom} \ g \ f \ p \ q \) if it satisfies
\[
\begin{align*}
  k \ v \ | \ p \ v &= \{ \ [] \ \\
  q \ v &= \{ \ [f \ v] \ \\
  (v_1, v_2) \leftarrow \ g \ v \ = \ k \ v_1 \ + \ k \ v_2.
\end{align*}
\]

¹The “pattern guard” \( (x, v') \leftarrow \ g \ v \) matches the result of \( g \) in the guard.
We also demand that successive applications of $g$ eventually produce seeds for which either $p$ or $q$ is true, and thus $k$ generates finite lists.

Regrettably, Theorem 1 does not dualise to unfolds.

**Lemma 3.** There exists $h$ such that $h = \text{unhom} \ g \ f \ p \ q$ for which there exist neither $g_r$ satisfying $h = \text{unfdr} \ g_r \ p$ nor $g_s$ satisfying $h = \text{unfld} \ g_s \ p$.

**Proof.** Let $\exp :: \text{Int} \to [\text{Int}]$ generate a list of $1$’s of length $2^n$ for given $n$. It can be defined by $\text{unhom} \ g \ f \ p \ q$ where $f = 0$, $q = 0 \Rightarrow p = 0 \Rightarrow q$, and $g (1 \ast n) = (n, n)$, but not by an unfoldr or unfold. An intuitive reason is that there is not always an $m$ such that $2^m = 1 = 2^n$. One may use the theories of Gibbons et al. [7] for a proof.

Does Theorem 2 have an unfold counterpart? To establish that $k = \text{unhom} \ g \ f \ p' \ q'$ given $k = \text{unfdr} \ g_r \ p = \text{unfld} \ g_s \ p$, we can pick $p' = p$, $q = p \ast \text{sn} \ast g_r$, and $f = \text{fst} \ast g_s$. The challenge is to construct $g$. To do so, we introduce some concepts dual to those for folds. While unfoldr and unfoldl resume a partially computed fold, their duals pause during generation of lists. During the generation of a list there are many places where we may pause. The following functions unfoldrp and unfoldlp ("p" for pause) return, in a list, all immediate lists and seeds during list generation:

$\text{unfoldrp} \ g_r \ p \ v = \text{iter}_r (\{ [a], b \} \ p v)\ v\ v$ where $\text{iter}_r \ (x, v) \ | \ p v = \{ (x, v) \}$

\[ (x, v') \rightarrow g_r v = (x, v) : \text{iter}_r (x + [x], v') \] .

$\text{unfoldlp} \ g_s \ p \ v = \text{iter}_l (\{ [a], b \} \ p v)\ v\ v$ where $\text{iter}_l \ (v, x) \ | \ p v = \{ (v, x) \}$

\[ (v', x) \rightarrow g_s v = (v, x) : \text{iter}_l (v + [x], x + [v], x) \} \] .

We may think of them as returning the “traces” of unfolding. For example, unfoldrp $g_r \ p \ v_0$ yields the list $\{([a], v_0), ([x_1], v_1), ([x_1, x_2], v_2) \}$, etc., if $\{x_{i+1}, v_{i+1}\} = g_r v_i$. A crucial property relating unfoldr and unfoldlp is that if $k = \text{unfdr} \ g_r \ p$ we have

\[ \text{split} \ast k = \text{map} (id \times k) = \text{unfdr} \ g_r \ p \] ,

where $\text{split} \ : \{ [a], [a] \}$ returns all the “splits” of the given list (e.g. $\text{split} \ [1, 2] = \{([1, 1, 2], ([1], [2], ([1, 2], []))\}$). Similarly we have that

\[ \text{split} \ast k = \text{map} (k \times id) = \text{unfdr} \ g_r \ p \] ,

if $k = \text{unfdr} \ g_r \ p$. Notice how they roughly resemble the “converses” of (3) and (4); functions next to the composition $\circ$ are swapped; instead of $f \circ g$ we have $\text{split} \ast k$ on the left hand sides of the equalities; $(id \times k)$ in (5) is on the lefthand side of $(\circ)$, and is lifted to lists by $\text{map}$ due to the type, etc.

A relational perspective We could have proceeded from (5) and (6) to construct $g$ and thereby prove the dual theorem. However, we would like to take a bold approach and ask the readers to see paused unfolding as a relation. Functional programmers appear to regard relations as an arcane creation, which is an unfortunate misunderstanding. Dijkstra argued that, for program derivation, non-determinism should be the norm and deterministic a special case [4], and this pearl is indeed a case where one has to bring up relations to appreciate the nice duality between the theorems for folding and unfolding.

For this pearl we need only a one-paragraph introduction to relations. A function $f :: a \rightarrow b$ is a subset of $(b \times a)$ where $(y, x) \in f$ means that $x$ is mapped to $y$ by $f$, or $y$ is a result to which $f$ maps the argument $x$, with the constraint that $(y, x) \in f$ and $(y, x) \in f$ implies $y = y'$. The constraint is relaxed for relations, allowing $x$ to be mapped to multiple $y$. Composition of relations is defined by

\[ (x, y) \in (R \times S) \equiv (3y : (y, x) \in R \land (y, x) \in S) \] .

Given $R :: a \rightarrow b$, its converse $R' :: b \rightarrow a$ is defined by $(x, y) \in R' \equiv (y, x) \in R$. We have $(R')^0 = R$, $id^0 = id$, and that converse distributes covariantly over product, and contravariantly over composition:

\[ (R \times S)^0 = (R^0 \times S^0) \]

\[ (R \times S)^0 = R^0 \times S^0 \]

In this pearl we view unfolds as the relational converse of folds, which is another reason why we restrict ourselves to finite lists — our “unfolds” are actually converses of folds in disguise, and an unhom is the relational converse of a list homomorphism.

Let the relation $\text{mem} :: [a] \rightarrow a$ relate a list to any of its members, and let

\[ \text{unfr} \ g_r \ p = \text{mem} \ast \text{unfr} \ g_r \ p \]

\[ \text{unfr} \ g_r \ p = \text{mem} \ast \text{unfr} \ g_r \ p \]

While unfoldr and unfoldlp generate the entire trace, unfr and unfl map the input seed to one arbitrary intermediate state. That is, $\text{unfr} \ g_r \ p$ maps $v_0$ to each of $\{ ([1], v_0), ([x_1], v_1), ([x_1, x_2], v_2), \ldots \}$, etc., if $\{x_{i+1}, v_{i+1}\} = g_r v_i$. The equality (5) can be expressed relationally — if $k = \text{unfr} \ g_r \ p$ we have

\[ \text{cat} \ast k = (id \times k) \ast \text{unfr} \ g_r \ p \] ,

Both sides relate the input seed, say, $v$, to a $pair$ of lists. On the left hand side, the list returned by $k \ast v$ is arbitrarily split into $(x, ys)$. The equation says that $xs \ast ys = k \ast v$ if and only if $xs$ can be generated by $\text{unfr} \ g_r \ p$ and $ys$ can be generated by $k$ starting from where $\text{unfr} \ g_r \ p$ left off. For $k = \text{unfr} \ g_r \ p$, a symmetric property holds:

\[ \text{cat} \ast k = (id \times k) \ast \text{unfl} \ g_r \ p \] ,

Recall that our aim is to construct $g$. It suffices to find a $g$ such that $\text{cat} \ast k = (k \ast k) \ast g$.

That is, $k \ast v = xs \ast ys$ if and only if $xs$ and $ys$ can be generated from seeds produced by $g$.

Notice that by applying the converse operator $(\ast)$ to both sides of (7) and (8), we get equations that are almost (3) and (4) apart from having converses of $k$ and unfolds in the formulæ:

\[ k \ast \text{cat} = (\text{unfr} \ g_r \ p) \ast (id \times k^0) \]

\[ k \ast \text{cat} = (\text{unfl} \ g_r \ p) \ast (k^0 \times id) \]

The proof of Theorem 2, however, proceeds the same way even if the components are not functions! In the realm of relations, part of the proof of Theorem 2 can be generalised to:

**Theorem 4.** If $R \ast \text{cat} = S_o \ast (id \times R) = S_o \ast (R \times id)$ and $R' \ast R = R$ for some $R'$, then $R \ast \text{cat} = R \ast \text{cat} \ast (R' \times R^0) \ast (R \times R)$.

**Proof.** The same as that of Theorem 2. We will prove a more general Theorem 7 later.

The desired dual theorem thus follows:

**Corollary 5.** If $k = \text{unhom} \ g f p q$ for some $g$, $f$, and $q$, then $k = \text{unfdr} \ g_r \ p = \text{unfld} \ g_s \ p$.

**Proof.** We have talked about $f$ and $q$, and now we aim to find $g$ such that $\text{cat} \ast k = (k \ast k) \ast g$. Using (7) and (8) as antecedents of Theorem 4, we get

\[ \text{cat} \ast k = (k \ast k) \ast (k^0 \times k^0) \ast \text{cat} \ast k \]

thus $g$ can be any functional subset of $(k^0 \times k^0) \ast \text{cat} \ast k$.
Calculating \( g \) The expression \((k^2 \times id) \circ unfoldr \_ \circ p \) \cap ((id \times k^2) \times unfoldl \_ \circ p)\), however, is not easy to simplify. To calculate \( g \) we often use a lemma that, if \( k = unfoldr \_ \circ p = unfoldl \_ \circ p \), we can refine \((k^2 \times id) \circ cat^2 \circ k\) to

\[
((k^2 \times id) \circ unfoldr \_ \circ p) \cap ((id \times k^2) \times unfoldl \_ \circ p).
\]

For readers who are not familiar with intersection and converses, Figure 1 offers some intuition. Given a seed \( v = \langle v_1, v_2 \rangle \) where \( v_1 \) and \( v_2 \) satisfy \((k \times v_1) \in unfoldlp \_ \circ p v \) and \((v_1 \times v_2) \in unfoldrp \_ \circ p\).

From now on we will use Lemma 6 and restrict ourselves to functions when we calculate \( g \).

4. Generating Sequences from Middle

In this section we look at two examples using the dual theorem for list generation.

4.1 Quicksort

While Gibbons [6] demonstrated how to derive merge sort from insertion sort using the third list homomorphism theorem, we use the dual theorem to derive quicksort from selection sort.²

We regard the input as a set and denote disjoint union by \( y \). One may come up with two definitions of selection sort: \( sort = unfoldr \_ \circ p = unfoldl \_ \circ p \), where \( p = null \) and

\[
g_\circ p \circ s z xs = \begin{cases} \emptyset & s z \neq \emptyset, x = \min \{x, z \setminus \{x\}\} \\ x & \emptyset \neq \emptyset, x = \max \{z \setminus \{x\}, x\}. \end{cases}
\]

Let \( xs \leq ys \equiv (\forall x \in xs, y \in ys : x \leq y) \). One can see that

\[
(sort \circ xxs, ys) \in unfoldr \_ \circ p zxs, \\
(zs, sort \circ ys) \in unfoldlp \_ \circ p zxs,
\]

if \( xzs \cup ys = zs \) and \( xzs \leq ys \). We show only the proof of the first membership, for which we introduce

\[
(ws + sort \circ zs, ys) \in iter \circ (ws, zxs) \equiv ws \cup ys = ws \cup zs \leq ys. \tag{9}
\]

Property (9) can be proved by induction on the size of \( zs \). For \( zs = \emptyset \) the property trivially holds. The nonempty case is shown in Figure 2.

By Lemma 6 we may thus choose \( zs = (ws, ys) \) for any \( zxs \cup ys = xzs \wedge xzs \leq ys \). That is, we split the set \( zxs \) into two, such that all elements in one set are no larger than any element in the other set, and sort them recursively. That gives rise to the equation,

\[
\text{sort} (xs \cup ys) \mid xzs \leq ys = \text{sort} zxs \circ \text{sort} ys.
\]

Despite being valid, the equation does not form a definition — as a program \text{sort} might not terminate since, for example, \( xs \) could be empty and the size of \( yz \) equals that of \( xzs \). For this example, one may come up with a terminating definition by enforcing that the neither \( xzs \) or \( yz \) is empty. We thus have \( \text{sort} = unhom \circ g \circ p \circ p \) where \( g \circ z xs = (ys, zxs, ws) \) for some \( non-empty \) \( xzs \) and \( yz \) such that \( xzs \cup yz = xzs \cup yz \leq yz \), \( f \{x\} = x \), \( p \circ p \equiv y = \emptyset \), and \( q \circ xzs \) holds if \( yz \) is singleton. By "unfolding" \text{sort} by one step we come up with the definition

\[
\text{sort} \emptyset = [\{} \\
\text{sort} (xs \cup \{x\} \cup yz) \mid xzs \leq yz = \text{sort} zxs \circ \text{sort} yz.
\]

4.2 Parallel Scan

It is known that the Haskell Prelude function \text{scanl} \circ (\oplus) \circ e, when \( \oplus \) is associative and \( e = \text{tag} \), the unit of \( \oplus \), is both \text{foldl} and \text{foldr}. Geser and Gorlatch [5] in fact showed how the following list homomorphism can be derived using the third list homomorphism theorem:

\[
\text{scanl} \circ (\oplus) \circ (\text{id} \times \{x\} \cup yz) \mid xzs \leq yz \leq ys = xzs \times [x] \times \text{map} (x \circ \text{tag}) (\text{scanl} \circ (\oplus) \circ yz).
\]

In an actual implementation, however, one would like to avoid having to perform \text{map} (\{x\}). Here we demonstrate that an attention to unfolds leads to a faster program.

For a concise presentation we again consider a slightly different variation. The following \text{scan} discards the right-most element of
the input list:
\[
\begin{align*}
\text{scan} (e, []) &= [] \\
\text{scan} (e, x : xs) &= e : \text{scan} (e \oplus x, xs).
\end{align*}
\]
For example, \(\text{scan} (e, [1, 2, 3]) = \{ e, e \oplus 1, e \oplus 1 \oplus 2 \}\). It is not hard to show that
\[
\text{scan} (e, xs + [x]) = \text{scan} (e, xs) + [e \oplus \text{reduce} xs],
\]
where \(\text{reduce} = \text{hom} (\oplus) \) \(\text{id} \) \(\text{f} \). We thus have \(\text{scan} = \text{unfoldr} g_p = \text{unfoldl} g_p \), \(p = \text{null} \oplus \text{snd} \) and
\[
\begin{align*}
g_x (e, x : xs) &= (e, (e \oplus x, xs)), \\
g_x (e, xs + []) &= ((e, xs), \oplus \text{reduce} xs),
\end{align*}
\]
where the domains of \(g_x\) and \(g_e\) are pairs whose second components are non-empty.

To construct \(g\), we show that for \(xs + ys = zs\) we have
\[
(\text{scan} (e, xs), (e \oplus \text{reduce} xs, ys)) \in \text{iter}_e (ws, (e, zs)).
\]
(10)
if \(xs + ys = zs\). The proof is an uninteresting induction on \(zs\), and the inductive case is shown in Figure 2.

Thus we pick \(g (e, xs) = ((e, xs), (e \oplus \text{reduce} xs, ys))\) for some \(xs + ys = zs\). For termination we want \(xs\) and \(ys\) to be both non-empty, which gives rise to the definition
\[
\begin{align*}
\text{scan} (e, []) &= [] \\
\text{scan} (e, [x]) &= [e] \\
\text{scan} (e, xs + ys) &= \text{scan} (e, xs) + \text{scan} (e \oplus \text{reduce} xs, ys).
\end{align*}
\]
This is not yet an efficient implementation. To avoid repeated calls to \(\text{reduce}\), one typically performs a \textit{tupling}. Let
\[
\text{sr} (e, xs) = (\text{scan} (e, xs), \text{reduce} xs).
\]

One may calculate a definition for \(\text{sr}\):
\[
\begin{align*}
\text{sr} (e, []) &= ([], \text{id}) \\
\text{sr} (e, [x]) &= ([e], x) \\
\text{sr} (e, xs + ys) &= \text{let} \ (s_1, r_1) = \text{sr} (e, xs) \\
&s_2, r_2 = \text{sr} (e \oplus r_1, ys) \\
&\text{in} \ (s_1 + s_2, r_1 + r_2).
\end{align*}
\]
However, the second call to \(\text{sr}\) in the \(xs + ys\) case demands the value of \(r_1\), which is a result of the first call to \(\text{sr}\). This prevents the two calls to \(\text{sr}\) from being executed in parallel.

Instead, we compute \(\text{scan}\) in two phases: all the \(r\)’s are first computed and cached, which are then used in the second phase to compute \(\text{scan}\). For that we need a data structure storing the \(r\)’s.

Consider the following binary tree, with a function \(\text{val}\) extracting the value at the root.

\[
\begin{align*}
\text{data} \ &\text{Tree} \ a = \text{L} \ a \ | \ \text{N} (\text{Tree} \ a) \ a \ (\text{Tree} \ a), \\
\text{val} \ (\text{L} \ n) &= n \\
\text{val} \ (\text{N} \ n \ n) &= n.
\end{align*}
\]

The following function builds a tree out of a non-empty list, with the invariant that \(\text{val} (\text{build} \ xs) = \text{reduce} \ xs\):
\[
\begin{align*}
\text{build} \ [x] &= \text{L} \ x \\
\text{build} \ (xs + ys) &= \text{let} \ t = \text{build} \ xs; \ u = \text{build} \ ys \\
&\text{in} \ \text{N} (\text{val} \ t \oplus \text{val} \ u) \ u.
\end{align*}
\]

The key to construct an efficient implementation of \(\text{scan}\) is to perform \(\text{build}\) in a separate phase and use only the results stored in the tree. That is, we wish to construct some function \(f\) such that
\[
\text{scan} (e, xs) = f (e, \text{build} \ xs),
\]
The singleton case is easy: \(f (e, L \ x) = [e]\). For inputs of length at least two, we calculate (for non-empty \(xs\) and \(ys\)):
\[
\begin{align*}
\text{scan} (e, xs + ys) &= \text{scan} (e, xs) + \text{scan} (e \oplus \text{reduce} \ xs, ys) \\
&= \{ \text{since} \ \text{val} (\text{build} \ xs) = \text{reduce} \ xs \} \\
\text{scan} (e, xs) + \text{scan} (e \oplus \text{val} (\text{build} \ xs), ys) \\
&= \{ \text{induction: scan} (e, xs) = f (e, \text{build} \ xs) \} \\
f (e, \text{build} \ xs) + f (e \oplus \text{val} (\text{build} \ xs), \text{build} \ ys) \\
&= \{ \ \text{let} \ L \ t \ u = \text{build} \ (xs + ys) \} \\
f (e, t) + f (e \oplus \text{val} \ t, u) \\
&= \{ \ \text{let} \ f (e, L \ t \ u) = f (e, t) + f (e \oplus \text{val} \ t, u) \} \\
f (e, \text{build} \ (xs + ys)).
\end{align*}
\]

We rename \(f\) to \(--\text{acc}\) since it accumulates the result:
\[
\begin{align*}
\text{scan} (e, []) &= [] \\
\text{scan} (e, [x]) &= \text{acc} (e, \text{build} \ xs), \\
\text{acc} (e, L \ x) &= [e] \\
\text{acc} (e, N \ t \ u) &= \text{acc} (e, t) \oplus \text{acc} (e \oplus \text{val} \ t, u).
\end{align*}
\]
By constructing a balanced binary tree, the evaluation can be performed in \(O (\log n)\) time given a sufficient number of processors, or in \(O (n/p + \log p)\) time if \(p \ll n\) where \(p\) is the number of processors. We have in fact reconstructed a well-known efficient implementation of \(\text{scan}\) recorded by, for example, Blelloch [2].

### 5. Generalising to Trees

As mentioned earlier, when looking at the proof of Theorem 2 in detail, one notices that further generalisation is possible. Indeed, the presence of \(R \cdot \text{cat}\) in Theorem 4 is superficial, and the theorem can still be generalised.

**Theorem 7.** \(R = U \cdot (\text{id} \times S) = V \cdot (T \times \text{id})\), \(S \cdot S^o \cdot S = S\), and \(T \cdot T^o \cdot T = T\) imply \(R = R \cdot (T^o \times S^o) \cdot (T \times S)\).

**Proof.**
\[
\begin{align*}
R &= U \cdot (\text{id} \times S) \\
&= U \cdot (\text{id} \times S) \cdot (\text{id} \times S) \\
&= R \cdot (\text{id} \times S^o) \cdot (\text{id} \times S) \\
&= V \cdot (T \times \text{id}) \cdot (\text{id} \times S) \\
&= V \cdot (T \times \text{id}) \cdot (T^o \times S^o) \\
&= R \cdot (T^o \times S^o) \cdot (T \times S).
\end{align*}
\]

However, the conclusion of the theorem, that \(R = R \cdot (T^o \times S^o) \cdot (T \times S)\), does not have much structure hinting at how this theorem can be useful. The use-cases we found are when \(S\) is a sub-expression of \(R\) (for example, to prove Corollary 5, we used \(R = k^o \cdot \text{cat}\) and \(S = T = k^o\). We do not require \(S = T\) in general), and we use Theorem 7 to establish recursive equations about \(S\), hoping to construct a terminating definition of \(S\).

#### 5.1 Third Homomorphism Theorem on Trees Revisited

Consider the \text{Tree} datatype defined in Section 4.2 and assume that we wish to efficiently compute a function on trees by distributing the work to multiple processors. One slight annoyance is that splitting a tree into two at an arbitrary point yields not two trees, but
one tree and a context containing a single hole that can be filled by a tree. The concept of contexts of a datatype was proposed by Huet [8] as the zipper. In particular, the context for Tree can be modelled by

\[
\text{type Cxt } a = \left[ Z_{a} \right], \\
\text{data } Z_{a} = \text{Nil } a (\text{Tree } a) \mid \text{Nr } (\text{Tree } a) a,
\]

with a function \( \text{fill} : (\text{Cxt } a, \text{Tree } a) \to \text{Tree } a \) that fills the hole of the context by a tree:

\[
\text{fill} ([], t) = t \\
\text{fill} (\text{Nil } x u ; xs, t) = \text{fill} (xs, \text{Nt } t u)
\]

For example, consider the tree

\[
t = (\text{N (N (L 1) 2 (L 3))}) 4 (\text{N (N (L 5) 6 (L 7))}) 8 (L 9))
\]

What remains after taking out the subtree \( u = \text{N (L 5)} 6 (L 7) \) is

\[
xz = (\text{Nil } 8 (L 9), \text{Nr (N (L 1) 2 (L 3))}) 4,
\]

with \( \text{fill} (xz, u) = t \).

To parallelise a function \( f : \text{Tree } a \to b \), we must have a variation \( f' : \text{Cxt } a \to b \) defined on contexts. Instantiating \( R, S, \) and \( T \) in Theorem 7 respectively to \( f \), \( \text{fill} \), and \( f' \), we see that \( f t \) can be computed in terms of \( f' \) \( cx \) and \( f u \),

\[
f \circ \text{fill} = f \circ \text{fill} \circ (f' \times f') \circ (f' \times f),
\]

if there exist \( U \) and \( V \) such that

\[
f \circ \text{fill} = (id \times f) \wedge \tag{11}
\]

\[
f \circ \text{fill} = V \circ (f' \times id). \tag{12}
\]

Equation (11) basically says that \( f \) is outwards — \( f (\text{fill} (cx, u)) \) can be computed from \( cx \) and \( f u \), while (12) says that \( f \) is also inwards — \( f (\text{fill} (cx, u)) \) can be computed from \( f' \) \( cx \) and \( u \). This is another way to view the previous work of Morihata et al. [10].

5.2 Generating Trees from the Middle

One naturally wonders whether the result can also be dualised to generating trees rather than consuming them. Again the answer is yes: if a tree can be generated both upwards and downwards, it can be generated “from the middle.” We will formalise what we mean below.

Let \( b \) be the type of seeds. Unfolding a tree downwards from the root using a function \( g_1 : b \to (b, a, b) \) is relatively familiar:

\[
\begin{align*}
\text{unf}_1 & : (b \to (b, a, b)) \to a \to b \to \text{Tree } a \\
\text{unf}_1 & (fs @ (g_1, f_1, p)) = v \\
\text{p} & v = L (f_1 v) \\
\text{v} & \sim 1 = v \circ (v_1, x, v_2) \sim g_1 v = \text{Nt } s(v_1, s f_1 v) x (v_1, s f_2 v).
\end{align*}
\]

As in the case of lists, our unfolds are actually converts of folds in disguise and are well-defined only if they terminate and produce finite trees. We consider paused versions of unfolding, which yields \( (\text{Cxt } a, b) \), a context and a seed. The function \( \text{unfp}_1 \) returns a list containing all such pairs of contexts and seeds:

\[
\begin{align*}
\text{unfp}_1 & : (b \to (b, a, b)) \to b \to [(\text{Cxt } a, b)] \\
\text{unfp}_1 & (fs @ (g_1, f_1, p)) v = \text{iter}_1 ([], v), \quad \text{where} \\
\text{iter}_1 & (xs, v) | p v = [[(xs, v)] \\
& \sim (v_1, x, v_2) \sim g_1 v = \text{Nt } x (\text{unfp}_1 s f_2 v) x (v_1, s f_1 v) \sim \text{iter}_1 (\text{Nr } (\text{unfp}_1 s f_1 v) x : xs, v_2).\end{align*}
\]

With the definition we have, for \( k = \text{unf}_1 (g_1, f_1, p) \), that

\[
\text{fill} \times k = (id \times k) \times \text{mem} = \text{unfp}_1 (g_1, f_1, p). \tag{13}
\]

By generating a tree “from the middle” we mean to find \( g : b \to (b, b) \) and \( k' : b \to \text{Cxt } a \) such that

\[
k v \mid p v \sim L (f_1 v) \\
\sim (v_1, v_2) \sim g v \circ \text{fill} (k' v_1) k v_2.
\]

That is, the tree returned by \( k \), if not a leaf, can be split into a context and a subtree that can be generated separately from the two seeds returned by \( g \). It suffices for \( g \) and \( k' \) to satisfy

\[
\text{fill} \times k = (k' \times k) \times g.
\]

Generating a tree “upwards” intuitively means to start from a leaf and find the path back to the root. With application of Theorem 7 in mind, we want to come up with a function \( \text{unfp}_1 (g_2, f_2) : b \to [(b, \text{Tree } a)] \) that satisfies, for some \( k' \),

\[
\text{fill} \times k = (k' \times id) \times \text{mem} = \text{unfp}_1 (g_2, f_2), \tag{14}
\]

thus each of (13) and (14) matches one antecedent of Theorem 7.

To grow a tree upwards from a leaf, we use a function \( g_2 \) having type \( \text{Tree}_a \to [(b, (a, b)) \to \text{Lt } a \to \text{Rt } b] \). If \( g_2 v \) is empty, we have reached the root. If it contains \((v', \text{Lt } x, \text{Rt } v_2)\), we go up by using the current tree as the left child, \( v_1 \), the seed for the right child, and \( v_2 \) the seed going up. Similarly with \( \text{Rt} \). We could have grown a \( \text{Tree } a \). To reuse the function later, however, we grow a context instead. Define \( \text{cxtp}_1, g_1 : b \to [(b, \text{Cxt } a)] \) that generates all (seed, context) pairs when one goes upwards:

\[
\text{cxtp}_1 g_1 v = \text{iter}_1 ([v, []]), \quad \text{where} \\
\text{iter}_1 (v, cx) \sim (v, cx) \sim y \sim (v', v) \sim g_1 v, v \sim \text{iter}_1 (v', cx \times [up v]), \\
\sim (\text{Lt } x, v_1) \sim \text{Nil } k v_1, v \sim \text{Rt } (v_1, x) \sim \text{Nr } (k v_1, v x).
\]

To generate all the splits, we need to be able to start from any leaf. Thus we let \( f_1 : b \to [(b, a)] \) return the list of values on each leaf, paired with a seed to go up with. We may then define \( \text{unfp}_1 \) by

\[
\begin{align*}
\text{unfp}_1 & : (G_1 a b, b \to [(b, a)]) \to b \to [(b, \text{Tree } a)] \\
\text{unfp}_1 & (g_1, f_1) v = [[v'] \circ \text{fill} (\text{cxtp}_1 (v_1, x)), \\
& (v', x) \sim f_1 v, (v', v) \sim \text{cxtp}_1 g_1 v'.
\end{align*}
\]

While \( \text{cxtp}_1 \) and \( \text{unfp}_1 \) return the entire history of upwards tree generation, their non-pausing version, \( \text{cxtp}_1 \) and \( \text{unf}_1 \), keep only completed contexts and trees (those which have reached the root):

\[
\text{cxtp}_1 g_1 v = \text{cxtp}_1 (v', v) \sim \text{cxtp}_1 g_1 v, v \sim \text{null } (p v).
\]

Equation (14) is satisfied if \( k = \text{mem} = \text{unf}_1 (g_1, f_1, p) \) is a function (that is, all routes from leaves to the root yields the same tree), and \( k' = \text{mem} \times \text{cxtp}_1 g_1 v \).

With (13) and (14), we therefore obtain from Theorem 7 that

\[
\text{fill} \times k = (k' \times k) \times (k' \times k) \times \text{fill} \times k,
\]

if \( k = \text{unf}_1 (g_1, f_1, p) = \text{mem} = \text{unf}_1 (g_1, f_1, p) \) and \( k' = \text{mem} \times \text{cxtp}_1 g_1 v \). To compute \( k \) “from the middle”, we may pick \( g \) to be a subset of \((k' \times k') \times \text{fill} \times k\).

6. Conclusions

By formulating the third list homomorphism theorem in point-free, relational style, we have dualised the theorem to unfolds, as well as generalised the theorem to both folds and unfolds for trees. While the original theorem establishes a connection between insertion sort and merge sort, the dual theorem shows a similar connection between selection sort and quicksort. We have also derived an efficient parallel algorithm for scan based on unfolds. To the best of the authors’ knowledge, while there has been many studies
on parallel programming based on structural recursion, none have considered the dual — list generation in the form of unhom.

The theorem generalises nicely to trees: if a function processes or generates a tree both downwards and upwards, it may process or generate the tree from the middle. Finally, we have also presented an example that shows how a relational view may shed new light on an old topic by revealing its hidden symmetry. The authors believe that relational methods deserve to be appreciated more among functional programmers.

As a remark, in practice, for both list and tree generation one might need different types of seeds for left/right or inwards/outwards unfolding. That is, we have $k = \text{unfoldr} \ g_0 \ p \circ i_1 = \text{unfoldl} \ g_0 \ p \circ i_2$ or $k = \text{unf↓} \ f_1 \ p \circ i_1 = \text{unf↑} \ f_1 \ p \circ i_2$, for some $i_1$ and $i_2$ that initialise the seeds. This is also covered by Theorem 7. We leave it to the readers to work out the details.

Acknowledgements The authors would like to thank Jeremy Gibbons, José Nuno Oliveira, and Janis Voigtländer for comments on earlier drafts, and to the anonymous referees for their valuable comments. This pearl was inspired by a question proposed by Zhenjiang Hu, who also cooperated with the authors throughout the development of this paper and made plenty of important technical contributions.

References


