Generalising and Dualising the Third List-Homomorphism Theorem

Functional Pearl

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

Akimasa Morihata
Tohoku University, Japan
morihata@riec.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 can be described both as an *unfoldr* and an *unfoldl*, the list 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

$$h\,(xs \mathbin{+\!\!\!+} ys) = h\,xs \odot h\,ys,$$

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

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 $sum = foldr\ (+)\ 0 = foldl\ (+)\ 0$, there exists some $(\odot)$ such that $sum\,(xs \mathbin{+\!\!\!+} ys) = sum\,xs \odot sum\,ys$, — for this simple example, $(\odot)$ happens to be $(+)$ as well. One naturally wonders whether $(\odot)$ 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 $fromTo\ (x, y) = [x, x+1 \ldots y]$. 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

$$\begin{aligned} h\,[\,] &= e \\ h\,(x : xs) &= f_{\triangleleft}\,(x, h\,xs), \end{aligned} \tag{1}$$

where $e :: b$ and $f_{\triangleleft} :: (a \times b) \to b$, have a unique solution for $h :: [a] \to b$, denoted by $foldr\ f_{\triangleleft}\ e$. We deviate from the standard and let $f_{\triangleleft}$ 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 $cons\ (x, xs) = x : xs$, and let $(f \times g)\,(x, y) = (f\,x, g\,y)$, which satisfies a law $(f \times g) \circ (h \times k) = (f \circ h \times g \circ k)$, which we will refer to as "product functor". Thus (1) can be written

$$h \circ cons = f_{\triangleleft} \circ (id \times h).$$

We define a variation of *foldr* that takes the base case as an extra argument,

$$\begin{aligned} &foldrr :: ((a \times b) \to b) \to ([a], b) \to b \\ &foldrr\ f_{\triangleleft}\ ([\,], e) \quad\ = e \\ &foldrr\ f_{\triangleleft}\ (x : xs, e)\ = f_{\triangleleft}\ (x, foldrr\ f_{\triangleleft}\ (xs, e)), \end{aligned}$$

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 $h = \mathit{foldr}\ f_{\triangleleft}\ e$, one can easily show by induction on xs that

$$h\ (xs +\!\!+\ ys) = \mathit{foldrr}\ f_{\triangleleft}\ (xs, h\ ys). \qquad (2)$$

Let $\mathit{cat}\ (xs, ys) = xs +\!\!+\ ys$, (2) can be written point-free as

$$h \circ \mathit{cat} = \mathit{foldrr}\ f_{\triangleleft} \circ (id \times h). \qquad (3)$$

Symmetrically, let $\mathit{snoc}\ (xs, x) = xs +\!\!+ [x]$. It is known that $\mathit{foldl}\ f_{\triangleright}\ e$ is the unique solution for $h :: [a] \to b$ in

$$\begin{array}{ll} h\,[\,] & = e \\ h \circ \mathit{snoc} & = f_{\triangleright} \circ (h \times id), \end{array}$$

where $f_{\triangleright} :: (b \times a) \to b$. Defining resumable *foldl* as

$$\begin{array}{lcl} \mathit{foldlr} & :: & ((b \times a) \to b) \to (b, [a]) \to b \\ \mathit{foldlr}\ f_{\triangleright}\ (e, [\,]) & = & e \\ \mathit{foldlr}\ f_{\triangleright}\ (e, xs +\!\!+ [x]) & = & f_{\triangleright}\ (\mathit{foldlr}\ f_{\triangleright}\ (e, xs), x), \end{array}$$

we have, if $h = \mathit{foldl}\ f_{\triangleright}\ e$, that

$$h \circ \mathit{cat} = \mathit{foldlr}\ f_{\triangleright} \circ (h \times id). \qquad (4)$$

A function $h :: [a] \to b$ is a *list homomorphism* if there exist $e :: b$, $k :: a \to b$, and $f :: (b \times b) \to b$ such that

$$\begin{array}{ll} h\,[\,] & = e \\ h\,[x] & = k\ x \\ h\ (xs +\!\!+\ ys) & = f\ (h\ xs, h\ ys). \end{array}$$

In such a case we denote h by $\mathit{hom}\ f\ k\ e$. 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 *foldr*, *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 *foldr* or a *foldl*.

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

Somewhat surprisingly, if a function can be computed both by a *foldr* and a *foldl*, it is a list homomorphism.

Theorem 2 (the 3rd list-hom. theorem [6])**.** $h = \mathit{foldr}\ f_{\triangleleft}\ e = \mathit{foldl}\ f_{\triangleright}\ e$ implies $h = \mathit{hom}\ f\ k\ e$ 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 $h \circ \mathit{cat} = f \circ (h \times h)$. 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, $f \circ f^{-1} \circ f = f$. 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:

$$\begin{array}{cl} & h \circ \mathit{cat} \\ = & \{\ (3)\ \} \\ & \mathit{foldrr}\ f_{\triangleleft} \circ (id \times h) \\ = & \{\ h = h \circ h^{-1} \circ h, \text{product functor}\ \} \\ & \mathit{foldrr}\ f_{\triangleleft} \circ (id \times h) \circ (id \times h^{-1}) \circ (id \times h) \\ = & \{\ (3) \text{ backwards, and } (4)\ \} \\ & \mathit{foldlr}\ f_{\triangleright} \circ (h \times id) \circ (id \times h^{-1}) \circ (id \times h) \\ = & \{\ h = h \circ h^{-1} \circ h, \text{product functor}\ \} \\ & \mathit{foldlr}\ f_{\triangleright} \circ (h \times id) \circ (h^{-1} \times h^{-1}) \circ (h \times h) \\ = & \{\ (4) \text{ backwards}\ \} \\ & h \circ \mathit{cat} \circ (h^{-1} \times h^{-1}) \circ (h \times h). \end{array}$$

Thus the theorem holds if we pick $f = h \circ \mathit{cat} \circ (h^{-1} \times h^{-1})$. □

Theorem 2 in fact provides hints how to construct list homomorphisms. For example, since $\mathit{sum} = \mathit{foldr}\ (+)\ 0 = \mathit{foldl}\ (+)\ 0$, Theorem 2 states that sum can be written as $\mathit{sum} = \mathit{hom}\ f\ k\ 0$ where $k\,x = \mathit{sum}\,[x] = x$ and $f\ (v, w) = \mathit{sum}\ (g\ v +\!\!+\ g\ w)$ for any right inverse g of 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 *foldr* and a *foldl*, could be weakened — properties (3) and (4) were merely used to push h to the right. In fact, $h \circ \mathit{cat}$ 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 *unfoldr*, a dual of *foldr* that generates a list from left to right, may be defined as follows. [1]

$$\begin{array}{lcl} \mathit{unfoldr} & :: & (b \to (a, b)) \to (b \to \mathit{Bool}) \to b \to [a] \\ \mathit{unfoldr}\ g_{\triangleright}\ p\ v & | & p\ v \qquad\qquad = [\,] \\ & | & (x, v') \leftarrow g_{\triangleright}\ v\ = x : \mathit{unfoldr}\ g_{\triangleright}\ p\ v'. \end{array}$$

Symmetrically, the function *unfoldl* is defined by

$$\begin{array}{lcl} \mathit{unfoldl} & :: & (b \to (b, a)) \to (b \to \mathit{Bool}) \to b \to [a] \\ \mathit{unfoldl}\ g_{\triangleleft}\ p\ v & | & p\ v \qquad\qquad = [\,] \\ & | & (v', x) \leftarrow g_{\triangleleft}\ v\ = \mathit{unfoldl}\ g_{\triangleleft}\ p\ v' +\!\!+ [x]. \end{array}$$

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 $g_{\triangleright}$ and $g_{\triangleleft}$ 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 $k :: b \to [a]$ by $\mathit{unhom}\ g\ f\ p\ q$ if it satisfies

$$\begin{array}{lll} k\ v & |\ p\ v & = [\,] \\ & |\ q\ v & = [f\ v] \\ & |\ (v_1, v_2) \leftarrow g\ v & = k\ v_1 +\!\!+\ k\ v_2. \end{array}$$

[1] The "pattern guard" $(x, v') \leftarrow g_{\triangleright}\ v$ matches the result of $g_{\triangleright}$ 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 $h = \mathit{unhom}\ g\ f\ p\ q$ for which there exist neither $g_{\triangleright}$ satisfying $h = \mathit{unfoldr}\ g_{\triangleright}\ p$ nor $g_{\triangleleft}$ satisfying $h = \mathit{unfoldl}\ g_{\triangleleft}\ p$.

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

Does Theorem 2 have an unfold counterpart? To establish that $k = \mathit{unhom}\ g\ f\ p'\ q$ given $k = \mathit{unfoldr}\ g_{\triangleright}\ p = \mathit{unfoldl}\ g_{\triangleleft}\ p$, we can pick $p' = p$, $q = p \circ \mathit{snd} \circ g_{\triangleright}$, and $f = \mathit{fst} \circ g_{\triangleright}$. The challenge is to construct g. To do so, we introduce some concepts dual to those for folds. While foldrr and foldlr 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 $\mathit{unfoldrp}$ and $\mathit{unfoldlp}$ ("p" for pause) return, in a list, all intermediate lists and seeds during list generation:

```
$\mathit{unfoldrp} :: (b \to (a, b)) \to (b \to \mathit{Bool}) \to b \to [([a], b)]$
$\mathit{unfoldrp}\ g_{\triangleright}\ p\ v = \mathit{iter}_{\triangleright}\ ([\,], v)$   where
  $\mathit{iter}_{\triangleright}\ (xs, v) \mid p\ v = [(xs, v)]$
                 $\mid (x, v') \leftarrow g_{\triangleright}\ v = (xs, v) : \mathit{iter}_{\triangleright}\ (xs +\!\!+ [x], v')$,

$\mathit{unfoldlp} :: (b \to (b, a)) \to (b \to \mathit{Bool}) \to b \to [(b, [a])]$
$\mathit{unfoldlp}\ g_{\triangleleft}\ p\ v = \mathit{iter}_{\triangleleft}\ (v, [\,])$   where
   $\mathit{iter}_{\triangleleft}\ (v, xs) \mid p\ v = [(v, xs)]$
                 $\mid (v', x) \leftarrow g_{\triangleleft}\ v = \mathit{iter}_{\triangleleft}\ (v', x : xs) +\!\!+ [(v, xs)]$.
```

We may think of them as returning the "traces" of unfolding. For example, $\mathit{unfoldrp}\ g_{\triangleright}\ p\ v_0$ yields the list $[([\,], v_0), ([x_1], v_1), ([x_1, x_2], v_2) \ldots]$, etc, if $(x_{i+1}, v_{i+1}) = g_{\triangleright}\ v_i$.

A crucial property relating $\mathit{unfoldr}$ and $\mathit{unfoldrp}$ is that if $k = \mathit{unfoldr}\ g_{\triangleright}\ p$ we have

$$\mathit{splits} \circ k = \mathit{map}\ (\mathit{id} \times k) \circ \mathit{unfoldrp}\ g_{\triangleright}\ p, \tag{5}$$

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

$$\mathit{splits} \circ k = \mathit{map}\ (k \times \mathit{id}) \circ \mathit{unfoldlp}\ g_{\triangleleft}\ p, \tag{6}$$

if $k = \mathit{unfoldl}\ g_{\triangleleft}\ p$. Notice how they roughly resemble the "converses" of (3) and (4): functions next to the composition $(\circ)$ are swapped; instead of $f \circ \mathit{cat}$ we have $\mathit{splits} \circ k$ on the left hand sides of the equalities; $(\mathit{id} \times k)$ in (5) is on the lefthand side of $(\circ)$, and is lifted to lists by 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 determinism 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 $f :: a \to b$ 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 ys. Composition of relations is defined by

$$(z, x) \in (R \circ S) \equiv (\exists y :: (z, y) \in R \wedge (y, x) \in S).$$

Given $R :: a \to b$, its *converse* $R^{\circ} :: b \to a$ is defined by $(x, y) \in R^{\circ} \equiv (y, x) \in R$. We have $(R^{\circ})^{\circ} = R$, $\mathit{id}^{\circ} = \mathit{id}$, and that converse distributes covariantly over product, and contravariantly over composition:

$$\begin{aligned}(R \times S)^{\circ} &= (R^{\circ} \times S^{\circ}),\\ (R \circ S)^{\circ} &= S^{\circ} \circ R^{\circ}.\end{aligned}$$

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 $\mathit{mem} :: [a] \to a$ relate a list to any of its members, and let

$$\begin{aligned}\mathit{unfr}\ g_{\triangleright}\ p &= \mathit{mem} \circ \mathit{unfoldrp}\ g_{\triangleright}\ p,\\ \mathit{unfl}\ g_{\triangleleft}\ p &= \mathit{mem} \circ \mathit{unfoldlp}\ g_{\triangleleft}\ p.\end{aligned}$$

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

$$\mathit{cat}^{\circ} \circ k = (\mathit{id} \times k) \circ \mathit{unfr}\ g_{\triangleright}\ p. \tag{7}$$

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

$$\mathit{cat}^{\circ} \circ k = (k \times \mathit{id}) \circ \mathit{unfl}\ g_{\triangleleft}\ p. \tag{8}$$

Recall that our aim is to construct g. It suffices to find a g such that

$$\mathit{cat}^{\circ} \circ k = (k \times k) \circ g.$$

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

Notice that by applying the converse operator $(\_)^{\circ}$ 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 formulae:

$$\begin{aligned}k^{\circ} \circ \mathit{cat} &= (\mathit{unfr}\ g_{\triangleright}\ p)^{\circ} \circ (\mathit{id} \times k^{\circ}),\\ k^{\circ} \circ \mathit{cat} &= (\mathit{unfl}\ g_{\triangleleft}\ p)^{\circ} \circ (k^{\circ} \times \mathit{id}).\end{aligned}$$

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 $R \circ \mathit{cat} = S_{\triangleright} \circ (\mathit{id} \times R) = S_{\triangleleft} \circ (R \times \mathit{id})$ and $R \circ R' \circ R = R$ for some R', then $R \circ \mathit{cat} = R \circ \mathit{cat} \circ (R^{\circ} \times R^{\circ}) \circ (R \times R)$.

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 $k = \mathit{unhom}\ g\ f\ p\ q$ for some g, f, and q, then $k = \mathit{unfoldr}\ g_{\triangleright}\ p = \mathit{unfoldl}\ g_{\triangleleft}\ p$.

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

$$\mathit{cat}^{\circ} \circ k = (k \times k) \circ (k^{\circ} \times k^{\circ}) \circ \mathit{cat}^{\circ} \circ k,$$

thus g can be any functional subset of $(k^{\circ} \times k^{\circ}) \circ \mathit{cat}^{\circ} \circ k$. □

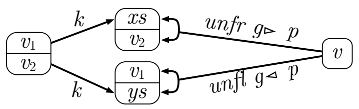

Figure 1. Depicting $((k^{\circ} \times id) \circ unfr\ g_{\rhd}\ p) \cap ((id \times k^{\circ}) \circ unfl\ g_{\lhd}\ p)$.

***Calculating** g* The expression $(k^{\circ} \times k^{\circ}) \circ cat^{\circ} \circ k$, however, is not easy to simplify. To calculate g we often use a lemma that, if $k = unfoldr\ g_{\rhd}\ p = unfoldl\ g_{\lhd}\ p$, we can refine $(k^{\circ} \times k^{\circ}) \circ cat^{\circ} \circ k$ to

$$((k^{\circ} \times id) \circ unfr\ g_{\rhd}\ p) \cap ((id \times k^{\circ}) \circ unfl\ g_{\lhd}\ p).$$

For readers who are not familiar with intersection and converses, Figure 1 offers some intuition. Given a seed v, $unfr\ g_{\rhd}\ p$ and $unfl\ g_{\lhd}\ p$ non-deterministically pause somewhere and respectively generate (xs, v_2) and (v_1, ys). The intersection means that they stop when they "meet in the middle", that is, when $k\ v_1 = xs$ and $k\ v_2 = ys$. The new pair of seeds, a possible result of g, is (v_1, v_2).

The lemma has a functional formulation that may be more friendly to readers. Writing the list membership predicate $elem :: a \to [a] \to Bool$ in the Haskell Standard Prelude in infix position as $(\in)$, we have,

Lemma 6. Assume that $k = unfoldr\ g_{\rhd}\ p = unfoldl\ g_{\lhd}\ p$. A function g is a subset of $(k^{\circ} \times k^{\circ}) \circ cat^{\circ} \circ k$ if, for all v, $g\ v = (v_1, v_2)$ where v_1 and v_2 satisfy $(k\ v_1, v_2) \in unfoldrp\ g_{\rhd}\ p\ v$ and $(v_1, k\ v_2) \in unfoldlp\ g_{\lhd}\ p$.

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 $\uplus$. One may come up with two definitions of selection sort: $sort = unfoldr\ g_{\rhd}\ p = unfoldl\ g_{\lhd}\ p$, where $p = null$ and

$$\begin{aligned} g_{\rhd}\ zs \mid zs \neq \varnothing, x \leftarrow min\ zs &= (x, zs - \{x\}), \\ g_{\lhd}\ zs \mid zs \neq \varnothing, x \leftarrow max\ zs &= (zs - \{x\}, x). \end{aligned}$$

Let $xs \mathrel{\dot{\leq}} ys \equiv (\forall x \in xs, y \in ys :: x \leq y)$. One can see that

$$\begin{aligned} (sort\ xs, ys) &\in unfoldrp\ g_{\rhd}\ p\ zs, \\ (xs, sort\ ys) &\in unfoldlp\ g_{\lhd}\ p\ zs, \end{aligned}$$

if $xs \uplus ys = zs$ and $xs \mathrel{\dot{\leq}} ys$. We show only the proof of the first membership, for which we show that

$$(ws + sort\ xs, ys) \in iter_{\rhd}\ (ws, zs) \Leftarrow xs \uplus ys = zs \wedge xs \mathrel{\dot{\leq}} ys. \quad (9)$$

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

By Lemma 6 we may thus choose $g\ zs = (xs, ys)$ for any $xs \uplus ys = zs \wedge xs \mathrel{\dot{\leq}} ys$. That is, we split the set zs 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,

$$sort\ (xs \uplus ys) \mid xs \mathrel{\dot{\leq}} ys \ = \ sort\ xs + sort\ ys.$$

$$\begin{aligned} & (ws + sort\ xs, ys) \in iter_{\rhd}\ (ws, zs) \\ \equiv &\quad \{\ zs \neq \varnothing, \text{let } z = min\ zs\ \} \\ & (ws + sort\ xs, ys) = (ws, zs) \vee \\ & (ws + sort\ xs, ys) \in iter_{\rhd}\ (ws + [z], zs - \{z\}) \\ \Leftarrow &\quad \{\ \text{for non-empty } xs, \text{let } x = min\ xs\ \} \\ & (xs = \varnothing \wedge ys = zs) \vee \\ & (ws + [x] + sort\ (xs - \{x\}), ys) \in iter_{\rhd}\ (ws + [z], zs - \{z\}) \\ \Leftarrow &\quad \{\ \text{induction}\ \} \\ & (xs = \varnothing \wedge ys = zs) \vee \\ & ((xs - \{x\}) \uplus ys = zs - \{z\} \wedge xs \mathrel{\dot{\leq}} ys \wedge x = z) \\ \equiv &\ xs \uplus ys = zs \wedge xs \mathrel{\dot{\leq}} ys. \end{aligned}$$

$$\begin{aligned} & (ws + s\ (e, xs), (e \oplus r\ xs, ys)) \in iter_{\rhd}\ (ws, (e, z : zs)) \\ \equiv &\ (ws + s\ (e, xs), (e \oplus r\ xs, ys)) = (ws, (e, z : zs)) \vee \\ & (ws + s\ (e, xs), (e \oplus r\ xs, ys)) \in iter_{\rhd}\ (ws + [e], (e \oplus z, zs)) \\ \Leftarrow &\quad \{\ \text{let } xs = x : xs' \text{ for the non-empty case}\ \} \\ & (xs = [\,] \wedge ys = z : zs) \vee \\ & (ws + [e] + s\ (e \oplus x, xs'), (e \oplus x \oplus r\ xs', ys)) \in \\ & \qquad iter_{\rhd}\ (ws + [e], (e \oplus z, zs)) \\ \Leftarrow &\quad \{\ \text{induction}\ \} \\ & (xs = [\,] \wedge ys = z : zs)\ \vee (xs' + ys = zs \wedge x = z) \\ \equiv &\ xs + ys = z : zs. \end{aligned}$$

Figure 2. Proofs of properties (9) and (10). In the latter proof, $scan$ and $reduce$ are respectively abbreviated to s and r.

Despite being valid, the equation does not form a definition — as a program $sort$ might not terminate since, for example, xs could be empty and the size of ys equals that of zs. For this example, one may come up with a terminating definition by enforcing that the neither xs nor ys is empty. We thus have $sort = unhom\ g\ f\ p\ q$ where $g\ zs = (xs, ys)$ for some non-empty xs and ys such that $xs \uplus ys = zs \wedge xs \mathrel{\dot{\leq}} ys$, $f\ \{x\} = x$, $p\ xs \equiv xs = \varnothing$, and $q\ xs$ holds if xs is singleton. By "unfolding" $sort$ by one step we come up with the definition

$$\begin{aligned} & sort\ \varnothing && = [\,] \\ & sort\ (xs \uplus \{x\} \uplus ys) && \mid\ xs \mathrel{\dot{\leq}} \{x\} \mathrel{\dot{\leq}} ys \\ & && = sort\ xs + [x] + sort\ ys. \end{aligned}$$

4.2 Parallel Scan

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

$$\begin{aligned} & scanl\ (\oplus)\ \iota_{\oplus}\ (xs + ys) \\ & \quad \mid\ xs' + [x] \leftarrow scanl\ (\oplus)\ \iota_{\oplus}\ xs \\ & \quad = \ xs' + [x] + map\ (x\oplus)\ (scanl\ (\oplus)\ \iota_{\oplus}\ ys). \end{aligned}$$

In an actual implementation, however, one would like to avoid having to perform $map\ (x\oplus)$. 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 $scan$ discards the right-most element of

[2] What we derive here, however, is the toy Quicksort well-known among functional programmers. It is arguable that the essence of real Quicksort is the algorithm for partition, which is not addressed here.

the input list:

$$\begin{array}{lcl} \mathit{scan}\ (e, [\,]) &=& [\,] \\ \mathit{scan}\ (e, x : xs) &=& e : \mathit{scan}\ (e \oplus x, xs). \end{array}$$

For example, $\mathit{scan}\ (e, [1,2,3]) = [e, e \oplus 1, e \oplus 1 \oplus 2]$. It is not hard to show that

$$\mathit{scan}\ (e, xs +\!\!+ [x]) \;=\; \mathit{scan}\ (e, xs) +\!\!+ [e \oplus \mathit{reduce}\ xs],$$

where $\mathit{reduce} = \mathit{hom}\ (\oplus)\ \mathit{id}\ \iota_\oplus$. We thus have $\mathit{scan} = \mathit{unfoldr}\ g_\triangleright\ p = \mathit{unfoldl}\ g_\triangleleft\ p$, where $p = \mathit{null} \circ \mathit{snd}$ and

$$\begin{array}{lcl} g_\triangleright\ (e, x : xs) &=& (e, (e \oplus x, xs)), \\ g_\triangleleft\ (e, xs +\!\!+ [\_]) &=& ((e, xs), e \oplus \mathit{reduce}\ xs), \end{array}$$

where the domains of $g_\triangleright$ and $g_\triangleleft$ are pairs whose second components are non-empty.

To construct g, we show that for $xs +\!\!+ ys = zs$ we have

$$(\mathit{scan}\ (e, xs), (e \oplus \mathit{reduce}\ xs, ys)) \in \mathit{unfoldrp}\ g_\triangleright\ p\ (e, zs),$$
$$((e, xs), \mathit{scan}\ (e \oplus \mathit{reduce}\ xs, ys)) \in \mathit{unfoldlp}\ g_\triangleleft\ p\ (e, zs).$$

Again we prove only the first property, for which we need to prove a slight generalisation,

$$(ws +\!\!+ \mathit{scan}\ (e, xs), (e \oplus \mathit{reduce}\ xs, ys)) \in \mathit{iter}_\triangleright\ (ws, (e, zs)), \tag{10}$$

if $xs +\!\!+ ys = zs$. The proof is an uninteresting induction on zs, and the inductive case is shown in Figure 2.

Thus we pick $g\ (e, zs) = ((e, xs), (e \oplus \mathit{reduce}\ xs, ys))$ for some $xs +\!\!+ ys = zs$. For termination we want xs and ys to be both non-empty, which gives rise to the definition

$$\begin{array}{lcl} \mathit{scan}\ (e, [\,]) &=& [\,] \\ \mathit{scan}\ (e, [x]) &=& [e] \\ \mathit{scan}\ (e, xs +\!\!+ ys) &|& xs \neq [\,] \wedge ys \neq [\,] \\ &=& \mathit{scan}\ (e, xs) +\!\!+ \mathit{scan}\ (e \oplus \mathit{reduce}\ xs, ys). \end{array}$$

This is not yet an efficient implementation. To avoid repeated calls to *reduce*, one typically performs a *tupling*. Let

$$\mathit{sr}\ (e, xs) \;=\; (\mathit{scan}\ (e, xs), \mathit{reduce}\ xs).$$

One may calculate a definition for sr:

$$\begin{array}{lcl} \mathit{sr}\ (e, [\,]) &=& ([\,], \iota_\oplus) \\ \mathit{sr}\ (e, [x]) &=& ([e], x) \\ \mathit{sr}\ (e, xs +\!\!+ ys) &|& xs \neq [\,] \wedge ys \neq [\,] \\ &=& \mathbf{let}\ (s_1, r_1) = \mathit{sr}\ (e, xs) \\ && \quad\ (s_2, r_2) = \mathit{sr}\ (e \oplus r_1, ys) \\ && \mathbf{in}\ (s_1 +\!\!+ s_2, r_1 \oplus r_2). \end{array}$$

However, the second call to sr in the $xs +\!\!+ ys$ case demands the value of r_1, which is a result of the first call to sr. This prevents the two calls to sr from being executed in parallel.

Instead, we compute scan in two phases: all the r's are first computed and cached, which are then used in the second phase to compute scan. For that we need a data structure storing the r's. Consider the following binary tree, with a function val extracting the value at the root,

$$\begin{array}{l} \mathbf{data}\ \mathit{Tree}\ a = \mathsf{L}\ a \mid \mathsf{N}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a), \\ \mathit{val}\ (\mathsf{L}\ n) = n \\ \mathit{val}\ (\mathsf{N}\ \_\ n\ \_) = n. \end{array}$$

The following function builds a tree out of a non-empty list, with the invariant that $\mathit{val}\ (\mathit{build}\ xs) = \mathit{reduce}\ xs$:

$$\begin{array}{lcl} \mathit{build}\ [x] &=& \mathsf{L}\ x \\ \mathit{build}\ (xs +\!\!+ ys) &|& xs \neq [\,] \wedge ys \neq [\,] \\ &=& \mathbf{let}\ t = \mathit{build}\ xs;\ u = \mathit{build}\ ys \\ && \mathbf{in}\ \mathsf{N}\ t\ (\mathit{val}\ t \oplus \mathit{val}\ u)\ u. \end{array}$$

The key to construct an efficient implementation of scan is to perform 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

$$\mathit{scan}\ (e, xs) = f\ (e, \mathit{build}\ xs).$$

The singleton case is easy: $f\ (e, \mathsf{L}\ x) = [e]$. For inputs of length at least two, we calculate (for non-empty xs and ys):

$$\begin{array}{cl} & \mathit{scan}\ (e, xs +\!\!+ ys) \\ = & \mathit{scan}\ (e, xs) +\!\!+ \mathit{scan}\ (e \oplus \mathit{reduce}\ xs, ys) \\ = & \quad \{ \text{ since } \mathit{val}\ (\mathit{build}\ xs) = \mathit{reduce}\ xs \ \} \\ & \mathit{scan}\ (e, xs) +\!\!+ \mathit{scan}\ (e \oplus \mathit{val}\ (\mathit{build}\ xs), ys) \\ = & \quad \{ \text{ induction: } \mathit{scan}\ (e, xs) = f\ (e, \mathit{build}\ xs)\ \} \\ & f\ (e, \mathit{build}\ xs) +\!\!+ f\ (e \oplus \mathit{val}\ (\mathit{build}\ xs), \mathit{build}\ ys) \\ = & \quad \{ \text{ let } \mathsf{N}\ t\ v\ u = \mathit{build}\ (xs +\!\!+ ys)\ \} \\ & f\ (e, t) +\!\!+ f\ (e \oplus \mathit{val}\ t, u) \\ = & \quad \{ \text{ let } f\ (e, \mathsf{N}\ t\ \_\ u) = f\ (e, t) +\!\!+ f\ (e \oplus \mathit{val}\ t,\ u)\ \} \\ & f\ (e, \mathit{build}\ (xs +\!\!+ ys)). \end{array}$$

We rename f to acc since it accumulates the result:

$$\begin{array}{lcl} \mathit{scan}\ (e, [\,]) &=& [\,] \\ \mathit{scan}\ (e, xs) &=& \mathit{acc}\ (e, \mathit{build}\ xs), \\ \mathit{acc}\ (e, \mathsf{L}\ x) &=& [e] \\ \mathit{acc}\ (e, \mathsf{N}\ t\ \_\ u) &=& \mathit{acc}\ (e, t) +\!\!+ \mathit{acc}\ (e \oplus \mathit{val}\ t,\ u). \end{array}$$

By constructing a balanced binary tree, the evaluation can be performed in $\mathrm{O}(\log n)$ time given a sufficient number of processors, or in $\mathrm{O}(n/p + \log p)$ time if $p \ll n$ where p is the number of processors. We have in fact reconstructed a well-known efficient implementation of 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 $R \circ \mathit{cat}$ in Theorem 4 is superficial, and the theorem can still be generalised.

Theorem 7. $R = U \circ (\mathit{id} \times S) = V \circ (T \times \mathit{id})$, $S \circ S^\circ \circ S = S$, and $T \circ T^\circ \circ T = T$ imply $R = R \circ (T^\circ \times S^\circ) \circ (T \times S)$.

Proof.

$$\begin{array}{rcl} R &=& U \circ (\mathit{id} \times S) \\ &=& U \circ (\mathit{id} \times S) \circ (\mathit{id} \times S^\circ) \circ (\mathit{id} \times S) \\ &=& R \circ (\mathit{id} \times S^\circ) \circ (\mathit{id} \times S) \\ &=& V \circ (T \times \mathit{id}) \circ (\mathit{id} \times S^\circ) \circ (\mathit{id} \times S) \\ &=& V \circ (T \times \mathit{id}) \circ (T^\circ \times S^\circ) \circ (T \times S) \\ &=& R \circ (T^\circ \times S^\circ) \circ (T \times S). \end{array}$$

□

However, the conclusion of the theorem, that $R = R \circ (T^\circ \times S^\circ) \circ (T \times S)$, 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 $R = k^\circ \circ \mathit{cat}$ and $S = T = k^\circ$. 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 *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

$$\begin{aligned}&\mathbf{type}\ Cxt\ a = [Z\ a],\\ &\mathbf{data}\ Z\ a \ = \mathsf{Nl}\ a\ (Tree\ a)\ \mid\ \mathsf{Nr}\ (Tree\ a)\ a,\end{aligned}$$

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

$$\begin{aligned}&fill\ ([\,], t) &&= t\\ &fill\ (\mathsf{Nl}\ x\ u : xs, t) &&= fill\ (xs, \mathsf{N}\ t\ x\ u)\\ &fill\ (\mathsf{Nr}\ t\ x : xs, u) &&= fill\ (xs, \mathsf{N}\ t\ x\ u).\end{aligned}$$

For example, consider the tree

$$t = \mathsf{N}\,(\mathsf{N}\,(\mathsf{L}\,1)\,2\,(\mathsf{L}\,3))\,4\,(\mathsf{N}\,(\mathsf{N}\,(\mathsf{L}\,5)\,6\,(\mathsf{L}\,7))\,8\,(\mathsf{L}\,9)).$$

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

$$cx = [\mathsf{Nl}\,8\,(\mathsf{L}\,9), \mathsf{Nr}\,(\mathsf{N}\,(\mathsf{L}\,1)\,2\,(\mathsf{L}\,3))\,4],$$

with $fill\ (cx, u) = t$.

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

$$f \circ fill = f \circ fill \circ (f'^{\circ} \times f^{\circ}) \circ (f' \times f),$$

if there exist U and V such that

$$f \circ fill\ = U \circ (id \times f)\ \wedge \qquad (11)$$

$$f \circ fill\ = V \circ (f' \times id). \qquad (12)$$

Equation (11) basically says that f is *outwards* — $f\ (fill\ (cx, u))$ can be computed from cx and $f\ u$, while (12) says that f is also *inwards* — $f\ (fill\ (cx, u))$ 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 $g_\downarrow :: b \to (b, a, b)$ is relatively familiar:

$$\begin{aligned}&unf_\downarrow\ ::\ (b \to (b,a,b), b \to a, b \to Bool) \to b \to Tree\ a\\ &unf_\downarrow\ (fs@(g_\downarrow, f_\downarrow, p))\ v\\ &\quad \mid p\ v \qquad\qquad\qquad = \mathsf{L}\ (f_\downarrow\ v)\\ &\quad \mid (v_1, x, v_2) \leftarrow g_\downarrow\ v\ =\ \mathsf{N}\ (unf_\downarrow\ fs\ v_1)\ x\ (unf_\downarrow\ fs\ v_2).\end{aligned}$$

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

$$\begin{aligned}&unfp_\downarrow\ ::\ (b \to (b,a,b), b \to a, b \to Bool) \to b \to [(Cxt\ a, b)]\\ &unfp_\downarrow\ (fs@(g_\downarrow, f_\downarrow, p))\ v\ =\ iter_\downarrow\ ([\,], v),\quad \mathbf{where}\\ &iter_\downarrow\ (xs, v) \mid p\ v = [(xs, v)]\\ &\qquad\qquad \mid (v_1, x, v_2) \leftarrow g_\downarrow\ v =\\ &\qquad\qquad\quad (xs, v) : iter_\downarrow\ (\mathsf{Nl}\ x\ (unf_\downarrow\ fs\ v_2) : xs, v_1) +\!\!\!+\\ &\qquad\qquad\qquad\quad iter_\downarrow\ (\mathsf{Nr}\ (unf_\downarrow\ fs\ v_1)\ x : xs, v_2).\end{aligned}$$

With the definition we have, for $k = unf_\downarrow\ (g_\downarrow, f_\downarrow, p)$, that

$$fill^{\circ} \circ k = (id \times k) \circ mem \circ unfp_\downarrow\ (g_\downarrow, f_\downarrow, p). \qquad (13)$$

By generating a tree "from the middle" we mean to find $g :: b \to (b, b)$ and $k' :: b \to Cxt\ a$ such that

$$\begin{aligned}&k\ v \mid p\ v &&= \mathsf{L}\ (f_\downarrow\ v)\\ &\quad \mid (v_\uparrow, v_\downarrow) \leftarrow g\ v &&= fill\ (k'\ v_\uparrow)\ (k\ v_\downarrow).\end{aligned}$$

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

$$fill^{\circ} \circ k = (k' \times k) \circ g.$$

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 $unfp_\uparrow\ (g_\uparrow, f_\uparrow) :: b \to [(b, Tree\ a)]$ that satisfies, for some k',

$$fill^{\circ} \circ k = (k' \times id) \circ mem \circ unfp_\uparrow\ (g_\uparrow, f_\uparrow), \qquad (14)$$

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

To grow a tree upwards from a leaf, we use a function $g_\uparrow$ having type $G_\uparrow\ a\ b = b \to [(b, (a, b) + (b, a))]$ (where $\mathbf{data}\ a + b = \mathsf{Lt}\ a \mid \mathsf{Rt}\ b$). If $g_\uparrow\ v$ is empty, we have reached the root. If it contains $(v', \mathsf{Lt}\ (x, v_r))$, we go up by using the current tree as the left child, v_r the seed for the right child, and v' the seed going up. Similarly with Rt. We could have grown a $Tree\ a$. To reuse the function later, however, we grow a context instead. Define $cxtp_\uparrow\ g_\uparrow :: b \to [(b, Cxt\ a)]$ that generates all (seed, context) pairs when one goes upwards:

$$\begin{aligned}&cxtp_\uparrow\ g_\uparrow\ v = iter_\uparrow\ (v, [\,]), \mathbf{where}\\ &\ iter_\uparrow\ (v, cx) = (v, cx) : [y \mid (v', lr) \leftarrow g_\uparrow\ v,\\ &\qquad\qquad\qquad\qquad y \leftarrow iter_\uparrow\ (v', cx +\!\!\!+ [up\ lr])],\\ &\ up\ (\mathsf{Lt}\ (x, v_r)) = \mathsf{Nl}\ x\ (k\ v_r)\\ &\ up\ (\mathsf{Rt}\ (v_l, x)) = \mathsf{Nr}\ (k\ v_l)\ x.\end{aligned}$$

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

$$\begin{aligned}&unfp_\uparrow\ ::\ (G_\uparrow\ a\ b, b \to [(b,a)]) \to b \to [(b, Tree\ a)]\\ &unfp_\uparrow\ (g_\uparrow, f_\uparrow)\ v = [(v'', fill\ (cx, \mathsf{L}\ x)) \mid\\ &\qquad\qquad\qquad (v', x) \leftarrow f_\uparrow\ v,\ (v'', cx) \leftarrow cxtp_\uparrow\ g_\uparrow\ v'].\end{aligned}$$

While $cxtp_\uparrow$ and $unfp_\uparrow$ return the entire history of upwards tree generation, their non-pausing version, $cxt_\uparrow$ and $unf_\uparrow$, keep only completed contexts and trees (those which have reached the root):

$$\begin{aligned}&cxt_\uparrow\ g_\uparrow\ v &&= [cx \mid (v', cx) \leftarrow cxtp_\uparrow\ g_\uparrow\ v, null\ (p\ v')],\\ &unf_\uparrow\ (g_\uparrow, f_\uparrow)\ v &&= [t \mid (v', t) \leftarrow unfp_\uparrow\ (g_\uparrow, f_\uparrow)\ v, null\ (p\ v')].\end{aligned}$$

Equation (14) is satisfiable if $k = mem \circ unf_\uparrow\ (g_\uparrow, f_\uparrow)$ is a function (that is, all routes from leaves to the root yields the same tree), and $k' = mem \circ cxt_\uparrow\ g_\uparrow\ v$.

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

$$fill^{\circ} \circ k = (k' \times k) \circ (k'^{\circ} \times k^{\circ}) \circ fill^{\circ} \circ k,$$

if $k = unf_\downarrow\ (g_\downarrow, f_\downarrow, p) = mem \circ unf_\uparrow\ (g_\uparrow, f_\uparrow)$ and $k' = mem \circ cxt_\uparrow\ g_\uparrow\ v$. To compute k "from the middle", we may pick g to be a subset of $(k'^{\circ} \times k^{\circ}) \circ fill^{\circ} \circ k$.

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 $k = unfoldr\ g_{\triangleright}\ p \circ i_1 = unfoldl\ g_{\triangleleft}\ p \circ i_2$ or $k = unf_{\downarrow}\ fs_{\downarrow} \circ i_1 = unf_{\uparrow}\ fs_{\uparrow} \circ i_2$, 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

[1] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer-Verlag, New York, NY, USA, 1987. *Technical Monograph PRG*-56.

[2] G. E. Blelloch. Prefix sums and their applications. In J. E. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[3] M. I. Cole. *Algorithmic Skeletons: Structural Management of Parallel Computation*. MIT Press, 1989.

[4] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1976.

[5] A. Geser and S. Gorlatch. Parallelizing functional programs by generalization. *Journal of Functional Programming*, 9(6):649–673, 1999.

[6] J. Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657–665, 1996.

[7] J. Gibbons, G. Hutton, and T. Altenkirch. When is a function a fold or an unfold? *Electr. Notes Theor. Comput. Sci.*, 44(1), 2001.

[8] G. P. Huet. The zipper. *Journal of Functional Programming*, 7(5): 549–554, 1997.

[9] A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Program parallelization by candidate generation and conformity testing. *IPSJ Transaction on Programming*, 2(2):132–143, 2009. (In Japanese).

[10] A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. The third homomorphism theorem on trees: Downward & upward lead to divide-and-conquer. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, Georgia, USA, January 21-23, 2009*, pages 177–185. ACM, 2009.

[11] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 146–155. ACM, 2007.