Constructing List Homomorphisms from Proofs

Yun-Yan Chi and Shin-Cheng Mu

IIS, Academia Sinica, Taiwan
{jaiyalas,scm}@iis.sinica.edu.tw

Abstract. The well-known third list homomorphism theorem states that if a function h is both an instance of *foldr* and *foldl*, it is a list homomorphism. Plenty of previous works devoted to constructing list homomorphisms, however, overlook the fact that proving h is both a *foldr* and a *foldl* is often the hardest part which, once done, already provides a useful hint about what the resulting list homomorphism could be. In this paper we propose a new approach: to construct a possible candidate of the associative operator and, at the same time, to transform a *proof* that h is both a *foldr* and a *foldl* to a proof that h is a list homomorphism. The effort constructing the proof is thus not wasted, and the resulting program is guaranteed to be correct.

1 Introduction

A function h on lists is called a *list homomorphism* [1] if it satisfies

$$h\,(xs +\!\!+ ys) = h\,xs \odot h\,ys, \tag{1}$$

for some associative operator $(\odot)$. We wish to identify list homomorphisms due to potential chances of parallelisation: to compute h, one may arbitrarily split the input list into $xs +\!\!+ ys$, compute $h\,xs$ and $h\,ys$ in parallel, and combine the results using $(\odot)$.

The well-known *third list-homomorphism theorem* [7] says that a function is a list homomorphism if it can be described as an instance of both *foldr* and *foldl*. That is, there exists $(\odot)$ satisfying (1) if

$$h = \mathit{foldr}\ (\triangleleft)\ e = \mathit{foldl}\ (\triangleright)\ e, \tag{2}$$

for some $(\triangleleft)$ and $(\triangleright)$. For example, $\mathit{sum} = \mathit{foldr}\ (+)\ 0 = \mathit{foldl}\ (+)\ 0$ and, indeed, there exists an $(\odot)$ such that $\mathit{sum}\,(xs +\!\!+ ys) = \mathit{sum}\,xs \odot \mathit{sum}\,ys$ — for this simple example, $(\odot)$ happens to be $(+)$ as well. The proof presented by Gibbons [7] showed that (1) can be satisfied by picking $x \odot y = h\ (h^{-1}\,x +\!\!+ h^{-1}\,y)$, where h^{-1} is a *weak inverse* of h, that is, one such that $h^{-1}\,(h\,x) = x$, which always exists if we assume a set-theoretic semantics.

One naturally wonders whether list homomorphisms can be mechanically constructed. Hu et al. [8] proposed to construct list homomorphisms by fusion with existing ones. Geser and Gorlatch [6] applied term rewriting techniques to construct a definition of $(\odot)$ from that of $(\triangleleft)$ and $(\triangleright)$. More recent

developments attempt to apply the third list-homomorphism theorem to mechanical construction of list homomorphisms. Morita et al. [10] proposed to automatically construct $(\odot)$ by picking some weak inverse h^{-1} and simplifying $h\ (h^{-1}\, x +\!\!+ h^{-1}\, y)$. For *sum*, one might pick $sum^{-1}\ x = [x]$, and the system simplifies $sum\ (sum^{-1}\, x +\!\!+ sum^{-1}\, y)$ to $x + y$. For the method to work, it is preferred that h^{-1} has a simple, non-recursive definition, such that $h\ (h^{-1}\, x +\!\!+ h^{-1}\, y)$ can be easily simplified. The method may even be generalised to trees [9].

Elegant as the approach is, when put into practice, however, one cannot help feeling that we have been solving the wrong problem. In all but the most simple cases, efforts are needed to prove (2), that the *foldr* and *foldl* definitions of h do define the same function. It occurs often that one of $(\triangleleft)$ or $(\triangleright)$ is picked as the definition of h, while the other is much harder to find. If the two definitions coincide so obviously that a proof is not necessary, like in the case of *sum*, the choice of $(\odot)$ is often equally trivial that a calculation/proof would be merely stating the obvious.

Once we have both $(\triangleleft)$ and $(\triangleright)$, the operator $(\odot)$ can often be constructed in an ad-hoc but effective manner: we may have a fairly good guess of $(\odot)$ by mixing fragments of code of $(\triangleleft)$ and $(\triangleright)$. We are still left with proving (1), but the proof often turns out to be very similar to that of (2). For a number of examples we fail to see the weak inverse approach applicable: we may have $(\odot)$ constructed, but cannot find any simple h^{-1} that "explains" its discovery.

For such problems, one may turn to the approach of Geser and Gorlatch [6]. An inherent disadvantage of term rewriting, however, is the lack of semantic concerns — having produced some $(\odot)$ offers no direct guarantee that it is correct. One would still like to have a proof of (1).

The way to go, we propose, is to transform the *proof* of (2), which we have to provide anyway, to a proof of (1), after assembling a possible definition of $(\odot)$ from that of $(\triangleleft)$ and $(\triangleright)$.

Program Construction: Syntax v.s. Semantics In program calculation one transforms a problem specification to a program through algebraic manipulation, thereby guaranteeing its correctness. The program and its correctness proof are developed at the same time.

During the process one is often encouraged to think formally, that is, to think in terms of the syntax rather than the semantics. Rather than focusing on the particular problem domain, the development of the program is ideally driven by syntatical guidelines such as achieving symmetry of expressions, matching the expression against certain forms, and exploiting algebraic properties such as a fusion theorem or the associativity of certain operators. The wish is to relieve programmers of the burden of the complexity in the problem domain through syntactical means — just like how we manipulate arithmetic expressions, using well-designed algebraic rules, without thinking what they "mean." As the slogan says, "let the symbols do the work" [5].

One of the main aims of researchers is therefore to develop convenient notations and theorems that apply generically to a wide range of problems. Such a style, however, could unfortunately have an unhealthy effect when taken to

the extreme. Plenty of works on program calculation claim to have discovered generic theorems that, once the problem specification is put into a certain form or shown to satisfy certain properties, can be applied formally and mechanically to construct an algorithm. Swept under the carpet, however, is the fact that molding the problem into the form or proving the required properties could be hard to carry out formally without domain specific, semantic knowledge. If it can be done, the programmer might have got hold of sufficient knowledge of the problem to just write up the program and, if still necessary at all, prove it correct afterwards.

This shall not be taken as a defect of the methodology. Instead, the value of program calculation is to separate the mechanical, routine process from those critical components that require insight into individual problems. Particularly when we face interesting problems, parts of the development have to be "seen" with the help of the programmer's semantical intuition of the problem, which is then formally proved afterwards. We wish, however, that the efforts doing the proof are not wasted.

Constructing list homomorphisms provides plenty of such examples. As we will see in the forthcoming sections, constructing $(\triangleright)$ from $(\triangleleft)$ in a purely formal manner could be rather hard. The programmer might find it much easier to speculate a possible candidate for $(\triangleright)$, which requires insight into the specific problem, and prove it correct afterwards. Once it is done, however, the construction of $(\odot)$ is relatively mechanical. Our wish is that the effort constructing and proving $(\triangleright)$ is not wasted — it can be used to guide the process finding $(\odot)$.

Contributions While plenty of previous work devoted to the construction of $(\odot)$ from the definitions of $(\triangleleft)$ and $(\triangleright)$, the novelty of our approach lies in expoiting the information in the proof of that $foldr\ (\triangleleft)\ e$ equals $foldl\ (\triangleright)\ e$. The effort proving (2) is thus not wasted. We find that this approach works for a number of problems that cannot be handled by previous approaches.

After giving a brief review of the concepts needed for this paper in Section 2, we demonstrate our method using three examples: the steep list problem, parallel scan, and a program returning the indexes of those elements in a list that satisfies a given predicate, in Section 3, 4, and 5, before we conclude in Section 6.

2 Preliminaries

We assume a set theoretic model for total functional programming, where a function $A \to B$ is a subset of $A \times B$ that is total (every value in A is mapped to something in B) and simple (every value in A is mapped to at most one value in B).

2.1 Folds and List Homomorphisms

As is well-known, given $e :: B$ and $(\triangleleft) :: A \to B \to B$, the following equations have a unique solution for $h :: [A] \to B$:

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

which is denoted $foldr\ (\triangleleft)\ e$. The *foldr*-fusion law is among the most important theorems one needs to know about *foldr*:

$$f \circ foldr\ (\triangleleft)\ e = foldr\ (\ll)\ (f\, e) \quad \Leftarrow \quad f\ (x \triangleleft z) = x \ll f\, z.$$

Not all functions are *foldr*s. Let $\langle h, k\rangle\, x = (h\, x, k\, x)$. For all function h taking a list as its input, $\langle h, id\rangle$ can always be defined in terms of a *foldr*. It is also often the case that $\langle h, k\rangle$ could be implemented, as a *foldr*, more efficiently than h alone. The technique of finding the right k to tuple with h is called *tupling* and is now a well-known functional programming technique [8].

Symmetrically, $foldl\ (\triangleright)\ e$ is the unique solution for h given the equations:

$$\begin{aligned} h\ [\,] &= e \\ h\ (xs +\!\!+ [x]) &= h\, xs \triangleright x, \end{aligned}$$

where $(\triangleright) :: B \to A \to B$. Like *foldr*, given h, we can often compute it faster in a *foldl* by tupling it with another function.

A function h is called a *list homomorphism* if it satisfies the following equations for some e, f, and $(\odot)$:

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

The equations imply that $(\odot)$ is associative with identity e. If the equations hold, we denote h by $hom\ (\odot)\ f\ e$. For any function h on lists, $\langle h, id\rangle$ is always a list homomorphism [4]. The equations form a terminating definition if xs and ys in the last clause are restricted to non-empty lists. When h is also defined as a *foldr* or a *foldl*, the cases for $h\,[\,]$ and $h\,[x]$ are determined and often omitted in this paper.

2.2 From Duality to Homomorphism

The *second duality theorem* of Bird [2, page 128] states that $foldr\ (\triangleleft)\ e = foldl\ (\triangleright)\ e$ if

$$z \triangleleft e = e \triangleright z \quad \wedge \quad (x \triangleleft y) \triangleright z = x \triangleleft (y \triangleright z). \tag{3}$$

When (3) holds, $(\triangleleft)$ and $(\triangleright)$ are said to *associate with each other*.[1] We present here a quick proof of the theorem. Let $h = foldr\ (\triangleleft)\ e$. To show that $h = foldl\ (\triangleright)\ e$

[1] By Bird [2]. One could argue that (3) is commutivity: $(\triangleright\, z) \circ (x \triangleleft) = (x \triangleleft) \circ (\triangleright\, z)$.

it is sufficient to prove that $h\,(xs +\!\!+ [z]) = h\,xs \triangleright z$, which, written point-free, is $h \circ (+\!\!+[z]) = (\triangleright\, z) \circ h$. We perform *foldr*-fusion on both sides:

$$
\begin{array}{ll}
 & h \circ (+\!\!+[z]) \\
= & \quad \{\ \textit{foldr}\text{-fusion, since } (+\!\!+[z]) = \textit{foldr}\,(:)\,[z]\ \} \\
 & \textit{foldr}\,(\triangleleft)\,(h\,[z]) \\
= & \quad \{\ \textit{foldr}\text{-fusion (backwards), see below}\ \} \\
 & (\triangleright\, z) \circ \textit{foldr}\ (\triangleleft)\ e \\
= & (\triangleright\, z) \circ h.
\end{array}
$$

The fusion conditions of the first *foldr*-fusion trivially holds, while those for the second fusion are $h\,[z] = (\triangleright\, z)\ e$ and $(\triangleright\, z)(x \triangleleft y) = x \triangleleft ((\triangleright\, z)\ y)$, which expands to (3).

To show that h is a list homomorphism, we have to construct $(\odot)$ such that $h\,(xs +\!\!+ ys) = h\,xs \odot h\,ys$. The calculation is very similar to the one above. The equation can be written in point-free style as $h \circ (+\!\!+ys) = (\odot\, h\,ys) \circ h$. To find out conditions under which the equality holds, we perform *foldr*-fusion on both sides:

$$
\begin{array}{ll}
 & h \circ (+\!\!+ys) \\
= & \quad \{\ \textit{foldr}\text{-fusion, since } (+\!\!+ys) = \textit{foldr}\,(:)\,ys\ \} \\
 & \textit{foldr}\,(\triangleleft)\,(h\,ys) \\
= & \quad \{\ \textit{foldr}\text{-fusion (backwards), see below}\ \} \\
 & (\odot h\,ys) \circ \textit{foldr}\ (\triangleleft)\ e \\
= & (\odot h\,ys) \circ h,
\end{array}
$$

which follows a pattern similar to the calculation above. The conditions we need for the second fusion are $h\,ys = (\odot\, h\,ys)\ e$ and $(\odot\, h\,ys)\ (x \triangleleft y) = x \triangleleft ((\odot\, h\,ys)\ y)$, that is,

$$h\,ys = e \odot h\,ys \quad \wedge \quad (x \triangleleft y) \odot h\,ys = x \triangleleft (y \odot h\,ys). \tag{4}$$

Since $(\triangleleft)$ is in essence a special case of $(\odot)$, it is not surprising that a proof of (3) can be generalised to a proof of (4). In fact, inspecting the proof of (3) may give us useful hints what $(\odot)$ could be.

Our aim, therefore, is to come up with a definition of $(\odot)$, together with its correctness proof, given e, $(\triangleleft)$, $(\triangleright)$, and a proof of their associativity.

There is no reason to bias on one side, though. With a symmetric development, one can show that h is a list homomorphism if

$$h\,xs = h\,xs \odot e \quad \wedge \quad h\,xs \odot (y \triangleright z) = (h\,xs \odot y) \triangleright z. \tag{5}$$

Both directions will be handy in this paper.

3 The Steep List Problem

A list of numbers is said to be *steep* if it descends (or ascends, if read right-to-left) so rapidly that each number is larger than the sum of the numbers to its right. Formally:

$$\begin{array}{ll} steep & :: [Int] \to Bool \\ steep\ [\,] & = True \\ steep\ (x : xs) & = x > sum\ xs\ \wedge\ steep\ xs. \end{array}$$

The problem has been used as an introductory example to tupling: definition of *steep* above is a quadratic time program, while the function $steepsum = \langle steep, sum \rangle$ can be computed in linear time in a *foldr* [3].

Can we determinine the steepness of a list in a *foldl* and, thereby, in a list homomorphism? It turns out that *steep* does not carry enough information to be computed in a *foldl* and we have to generalise further. Let *cap xs*, the *capacity* of *xs*, be a (non-inclusive) upper-bound of values we can attach to the right of *xs* and still keep it steep. That is, for all y, $xs +\!\!+ [y]$ is steep if and only if $y < cap\ xs$. For example, $cap\ [9,5,3] = 1$, since $9 \not> 5 + 3 + x$ for $x > 1$; also, $cap\ [9,4,2] = 2$, since $4 \not> 2 + x$ for $x > 2$.

Definition of *cap* is generalised from that of *steep*:

$$\begin{array}{ll} cap & :: [Int] \to Int \\ cap\ [\,] & = \infty \\ cap\ (x : xs) & = (x - sum\ xs) \downarrow cap\ xs, \end{array}$$

where $(\downarrow)$ returns the minimum of its two arguments. Some intuition for the inductive case: for $x : xs +\!\!+ [y]$ to be steep, we need $x > sum\ xs + y$, and thus the "margin" $x - sum\ xs$ is an upper-bound for y. Furthermore, $xs +\!\!+ [y]$ must be steep as well. Therefore, inductively, *cap xs* is another bound for y. By formalising the argument one easily comes up with an inductive proof that $steep\ xs \equiv cap\ xs > 0$.

Since *sum* and *cap* are both *foldr*s, so is $capsum\ xs = (cap\ xs, sum\ xs)$:

$$\begin{array}{ll} capsum & :: [Int] \to (Int, Int) \\ capsum\ [\,] & = (\infty, 0) \\ capsum\ (x : xs) & = \mathbf{let}\ (c_2, s_2) = capsum\ xs\ \mathbf{in}\ ((x - s_2) \downarrow c_2, x + s_2). \end{array}$$

Can *cap* be computed in a *foldl*? As we will see in Section 3.1, it is rather difficult to construct, purely by formal calculation, a *foldl* definition of *cap* from the *foldr* definition (and vice versa). By thinking semantically, one might guess that

$$cap\ (xs +\!\!+ [z]) = (cap\ xs - z) \downarrow z.$$

The rationale is that $cap\ (xs +\!\!+ [z])$ is the minimum of two upper-bounds: firstly, having z on the right-end lowers the upper-bound *cap xs* by z, and secondly, z itself is a upper-bound. We may thus also define *capsum* as a *foldl*:

$$\begin{array}{ll} capsum\ [\,] & = (\infty, 0) \\ capsum\ (xs +\!\!+ [z]) & = \mathbf{let}\ (c_1, s_1) = capsum\ xs\ \mathbf{in}\ ((c_1 - z) \downarrow z, s_1 + z). \end{array}$$

Without a proof, however, one cannot be confident that the two definitions of *cap* (and thus *capsum*) do define the same function. Once the proof of their equivalence is done, however, we would be taking an unnecessarily long route if we abandon all the efforts above, start from scratch, and try to construct $(c_1, s_1) \odot (c_2, s_2) = capsum\ (capsum^{-1}\,(c_1, s_1) +\!\!+ capsum^{-1}\,(c_2, s_2))$ — for this example, in fact, we have failed to come up with a simple $capsum^{-1}$.

Instead, it turns out that the homomorphic definition of *capsum* is assembled from parts of its *foldr* and *foldl* definitions:

$$\begin{aligned} capsum\ (xs +\!\!+ ys) = \ & \mathbf{let}\ \{\ (c_1, s_1) = capsum\ xs;\ (c_2, s_2) = capsum\ ys\ \} \\ & \mathbf{in}\ ((c_1 - s_2) \downarrow c_2, s_1 + s_2). \end{aligned}$$

Still, one needs a proof that the definition above does coincide with the *foldr* definition. The proof highly resembles the proof of the associativity. One thus wonders whether it is possible to somehow reuse the definitions and proofs, which will be done in Section 3.2.

3.1 Constructing $(\triangleright)$ Formally

In the opening of this section we have argued, semantically, that $capsum = foldr\,(\triangleleft)\,(\infty, 0) = foldl\,(\triangleright)\,(\infty, 0)$, where

$$\begin{aligned} x \triangleleft (c_2, s_2) &= ((x - s_2) \downarrow c_2, x + s_2), \\ (c_1, s_1) \triangleright z &= ((c_1 - z) \downarrow z, s_1 + z). \end{aligned}$$

Before we proceed with constructing the homomorphic definition of *capsum*, we show in this section that it is hard to construct $(\triangleright)$ from $(\triangleleft)$ in a purely syntactical manner. This echos our observation in the next subsection that the proof of $(x \triangleleft y) \triangleright z = x \triangleleft (y \triangleright z)$, which has to be done if $(\triangleright)$ is not derived, already provides plenty of information necessary to construct $(\odot)$. Finding $(\triangleright)$ is thus where all the hard work is.

Here is an attempt. We assume that $(c, s) \triangleright z = (f_1\ c\ s\ z, f_2\ c\ s\ z)$ for some f_1 and f_2, which we shall try to construct such that $z \triangleleft (\infty, 0) = (\infty, 0) \triangleright z$ and $(x \triangleleft (c, s)) \triangleright z = x \triangleleft ((c, s) \triangleright z)$. We start from the left-hand side of the latter:

$$\begin{aligned} & (x \triangleleft (c, s)) \triangleright z \\ = \quad & \{\ \text{definition of } (\triangleleft)\ \} \\ & ((x - s) \downarrow c, x + s) \triangleright z \\ = \quad & \{\ \text{definition of } (\triangleright)\ \} \\ & (f_1\ ((x - s) \downarrow c)\ (x + s)\ z, f_2\ ((x - s) \downarrow c)\ (x + s)\ z). \end{aligned}$$

Now that we are stuck, we expand the right-hand side:

$$\begin{aligned} & x \triangleleft ((c, s) \triangleright z) \\ = \quad & \{\ \text{definition of } (\triangleright)\ \} \\ & x \triangleleft (f_1\ c\ s\ z, f_2\ c\ s\ z) \end{aligned}$$

$$
\begin{array}{ll}
= & \{\ \text{definition of } (\triangleleft)\ \} \\
& ((x - f_2\ c\ s\ z) \downarrow f_1\ c\ s\ z, x + f_2\ c\ s\ z).
\end{array}
$$

We shall try to somehow unify the two resulting expressions.

The simplest choice of f_2 such that $f_2\ ((x-s)\downarrow c)\ (x+s)\ z = x + f_2\ c\ s\ z$ would be $f_2\ c\ s\ z = s$, which unfortunately fails the requirement that $z \triangleleft (\infty, 0) = (\infty, 0) \triangleright z$, which expands to

$$(z, z) = (f_1\ \infty\ 0\ z, f_2\ \infty\ 0\ z). \tag{6}$$

It may be the next logical choice to try $f_2\ c\ s\ z = s + z$, which turns out to be correct. Having found f_2, we shall then unify

$$f_1\ ((x-s)\downarrow c)\ (x+s)\ z \quad \text{and} \quad (x - (s+z)) \downarrow f_1\ c\ s\ z.$$

Which is no easy task. The simplest choice is $f_1\ c\ s\ z = c - z$, which unifies the two expressions,

$$
\begin{aligned}
f_1\ ((x-s)\downarrow c)\ (x+s)\ z &= ((x-s)\downarrow c) - z \\
&= (x - s - z) \downarrow (c - z) = (x - (s+z)) \downarrow f_1\ c\ s\ z,
\end{aligned}
$$

but again turns out to be wrong because it fails the requirement in (6) that $f_1\ \infty\ 0\ z = z$. It takes some creativity to come up with $f_1\ c\ s\ z = (c \downarrow z) - z$, which is best explained by the semantical view in the beginning of this section.

3.2 Computing Capacity by a List Homomorphism

Return to the problem of constructing *capsum* as a list homomorphism. From the discussion in Section 2.2, to prove that $capsum = foldr\ (\triangleleft)\ (\infty, 0) = foldl\ (\triangleright)\ (\infty, 0)$ we need to prove that

$$z \triangleleft (\infty, 0) = (\infty, 0) \triangleright z \quad \wedge \quad (x \triangleleft y) \triangleright z = x \triangleleft (y \triangleright z),$$

where $x \triangleleft (c_2, s_2) = ((x - s_2) \downarrow c_2, x + s_2)$ and $(c_1, s_1) \triangleright z = ((c_1 - z) \downarrow z, s_1 + z)$. While we have have seen in Section 3.1 that it is hard to construct $(\triangleright)$ in a purely syntactical manner, once we have somehow conjectured $(\triangleright)$, the proof of its correctness is routine. To show that $(\triangleleft)$ and $(\triangleright)$ associate, we reason:

$$
\begin{array}{ll}
& (x \triangleleft (c, s)) \triangleright z \\
= & \{\ \text{definition of } (\triangleleft)\ \} \\
& ((x - s) \downarrow c, x + s) \triangleright z \\
= & \{\ \text{definition of } (\triangleright)\ \} \\
& ((((x - s) \downarrow c) - z) \downarrow z, x + s + z) \\
= & \{\ (-z) \text{ distributes over } (\downarrow)\ \} \\
& (((x - s - z) \downarrow (c - z)) \downarrow z, x + s + z) \\
= & \{\ \text{arithmetics}\ \}
\end{array}
$$

$$
\begin{array}{ll}
 & (((x - (s + z)) \downarrow ((c - z) \downarrow z), x + s + z) \\
= & \quad \{ \text{ definition of } (\triangleleft) \ \} \\
 & x \triangleleft ((c - z) \downarrow z, s + z) \\
= & \quad \{ \text{ definition of } (\triangleright) \ \} \\
 & x \triangleleft ((c, s) \triangleright z).
\end{array}
$$

The proof will be referred to as the "proof of associativity". The aim now is to construct a definition of $(\odot)$ and generalise the proof above to a proof of

$$(c_2, s_2) = (\infty, 0) \odot (c_2, s_2) \quad \wedge \quad (x \triangleleft y) \odot (c_2, s_2) = x \triangleleft (y \odot (c_2, s_2)),$$

for all (c_1, s_1) in the range of *capsum*.

Since $(\odot)$ is a generalisation of $(\triangleright)$, we start with replacing the occurrences of z in $(\triangleright)$ by metavariables. Our first guess is

$$(c_1, s_1) \odot (c_2, s_2) = ((c_1 - X_1) \downarrow X_2, s_1 + X_3).$$

To prove that $(x \triangleleft y) \odot (c_2, s_2) = x \triangleleft (y \odot (c_2, s_2))$ we copycat the steps in the proof of associativity. Starting from $(x \triangleleft (c, s)) \odot (c_1, s_1)$, we reason:

$$
\begin{array}{ll}
 & (x \triangleleft (c, s)) \odot (c_2, s_2) \\
= & \quad \{ \text{ definition of } (\triangleleft) \ \} \\
 & ((x - s) \downarrow c, x + s) \odot (c_2, s_2) \\
= & \quad \{ \text{ definition of } (\odot) \ \} \\
 & ((((x - s) \downarrow c) - X_1) \downarrow X_2, x + s + X_3) \\
= & \quad \{ \ (-X_1) \text{ distributes over } (\downarrow) \ \} \\
 & (((x - s - X_1) \downarrow (c - X_1)) \downarrow X_2, x + s + X_3) \\
= & \quad \{ \text{ arithmetics } \} \\
 & (((x - (s + X_1)) \downarrow ((c - X_1) \downarrow X_2), x + s + X_3).
\end{array}
$$

In the next step we are supposed to fold back the definition of $(\triangleleft)$. To be able to do so, however, $(s + X_1)$ and $(s + X_3)$ have to be the same term and we thus have to unify X_1 and X_3. The last two steps go:

$$
\begin{array}{ll}
 & (((x - (s + X_1)) \downarrow ((c - X_1) \downarrow X_2), x + s + X_1) \\
= & \quad \{ \text{ definition of } (\triangleleft) \ \} \\
 & x \triangleleft ((c - X_1) \downarrow X_2, s + X_1) \\
= & \quad \{ \text{ definition of } (\odot) \ \} \\
 & x \triangleleft ((c, s) \odot (c_2, s_2))
\end{array}
$$

Thus the definition of $(\odot)$ is now refined to

$$(c_1, s_1) \odot (c_2, s_2) = ((c_1 - X_1) \downarrow X_2, s_1 + X_1),$$

for some X_1 and X_2. Notice that any choice of X_1 and X_2 would allow the proof to go through. In a sense, we have exploited all information from the proof above; it could tell us no more about X_1 and X_2!

To figure out X_1 and X_2 we turn to the base case, where to have to show that $(c_2, s_2) = (\infty, 0) \odot (c_2, s_2)$. Expanding the right-hand side, we get

$$(c_2, s_2) = ((\infty - X_1) \downarrow X_2, 0 + X_1).$$

An obvious choice would be $X_1 = s_2$ and $X_2 = c_2$. We have thus discovered that $(c_1, s_1) \odot (c_2, s_2) = ((c_1 - s_2) \downarrow c_2, s_1 + s_2)$. This $(\odot)$ has got to be correct, because we have the proof already!

As a remark, similar principles can be applied to derive a list-homomorphic solution of the maximum segment sum problem. We will need a four-tuple whose components respectively store the maximum segment sum, prefix sum, suffix sum, and the total sum. The calculation is tedious, but not essentially harder.

4 Parallel Scan

Constructing the list-homomorphic definition of *scanr* is dealt with in Hu et al. [8] and Geser and Gorlatch [6], but notably not in Morita et al [10]. In this section we present our solution. The function $scanr\ (\oplus)\ e$ applies $foldr\ (\oplus)\ e$ to every tail of its input list. It is well known that $scanr\,(\oplus)\,e = foldr\,(\triangleleft)\,[e] = foldl\,(\triangleright)\,[e]$ if $(\oplus)$ is associative with unit e, where

$$x \triangleleft ys = (x \oplus head\ ys) : ys,$$
$$ys \triangleright z = map\ (\oplus z)\ ys +\!\!+ [e].$$

The proof of associativity of $(\triangleleft)$ and $(\triangleright)$ is given below, where ys, being in the range of $scanr\,(\oplus)\,e$, is a non-empty list.

$$\begin{array}{ll}
 & (x \triangleleft ys) \triangleright z \\
= & \quad \{\ \text{definition of } (\triangleleft)\ \} \\
 & ((x \oplus head\ ys) : ys) \triangleright z \\
= & \quad \{\ \text{definition of } (\triangleright)\ \} \\
 & map\ (\oplus z)\ ((x \oplus head\ ys) : ys) +\!\!+ [e] \\
= & \quad \{\ \text{definition of } map\ \} \\
 & ((x \oplus head\ ys) \oplus z) : map\ (\oplus z)\ ys +\!\!+ [e] \\
= & \quad \{\ \text{associativity of } \oplus\ \} \\
 & (x \oplus (head\ ys \oplus z)) : map\ (\oplus z)\ ys +\!\!+ [e] \\
= & \quad \{\ f(head\ xs) = head\ (map\ f\ xs)\ \} \\
 & (x \oplus head\ (map\ (\oplus z)\ ys)) : map\ (\oplus z)\ ys +\!\!+ [e] \\
= & \quad \{\ head\ xs = head\ (xs +\!\!+ ys) \text{ if } xs \text{ non-empty}\ \} \\
 & (x \oplus head\ (map\ (\oplus z)\ ys +\!\!+ [e])) : map\ (\oplus z)\ ys +\!\!+ [e]
\end{array}$$

$$
\begin{array}{ll}
= & \{\ \text{definition of } (\triangleleft)\ \} \\
& x \triangleleft (map\ (\oplus z)\ ys +\!\!+ [e]) \\
= & \{\ \text{definition of } (\triangleright)\ \} \\
& x \triangleleft (ys \triangleright z).
\end{array}
$$

We aim to construct $(\odot)$ and generalise the proof above to a proof of

$$zs = [e] \odot zs \quad \wedge \quad (x \triangleleft ys) \odot zs = x \triangleleft (ys \odot zs),$$

where zs is also an non-empty list. One possible candidate of $(\odot)$ is obtained by replacing the sole occurrence of z in $(\triangleright)$ by a metavariable:

$$ys \odot zs = map\ (\oplus X_1)\ ys +\!\!+ [e].$$

We may proceed with it and show that it indeed satisfies every step in the proof, that is, for such a choice it is true that $(x \triangleleft ys) \odot zs = x \triangleleft (ys \odot zs)$, whatever X_1 is. This candidate fails, however, when we consider the base case $zs = [e] \odot zs$, which expands to $zs = X_1 : [e]$ and restricts zs to lists having exactly two elements. Apparently our $(\odot)$ is not general enough.

To allow zs to be of arbitrary length, one possibility is to generalise $[e]$ to X_2 — an instance of the common technique to generalise occurrences of constants to metavariables. The proof goes:

$$
\begin{array}{ll}
& (x \triangleleft ys) \odot zs \\
= & \{\ \text{definition of } (\triangleleft)\ \} \\
& ((x \oplus head\ ys) : ys) \odot zs \\
= & \{\ \text{conjecture: } ys \odot zs = map\ (\oplus X_1)\ ys +\!\!+ X_2\ \} \\
& map\ (\oplus X_1)\ ((x \oplus head\ ys) : ys) +\!\!+ X_2 \\
= & \{\ \text{definition of } map\ \} \\
& ((x \oplus head\ ys) \oplus X_1) : map\ (\oplus X_1)\ ys +\!\!+ X_2 \\
= & \{\ \text{associativity of } (\oplus)\ \} \\
& (x \oplus (head\ ys \oplus X_1)) : map\ (\oplus X_1)\ ys +\!\!+ X_2 \\
= & \{\ f\ (head\ xs) = head\ (map\ f\ xs)\ \} \\
& (x \oplus head\ (map\ (\oplus X_1)\ ys)) : map\ (\oplus X_1)\ ys +\!\!+ X_2 \\
= & \{\ head\ xs = head\ (xs +\!\!+ ys), \text{ if } xs \text{ non-empty}\ \} \\
& (x \oplus head\ (map\ (\oplus X_1)\ ys) +\!\!+ X_2) : map\ (\oplus X_1)\ ys +\!\!+ X_2 \\
= & \{\ \text{definition of } (\triangleleft)\ \} \\
& x \triangleleft (map\ (\oplus X_1)\ ys +\!\!+ X_2) \\
= & \{\ \text{definition of } (\odot)\ \} \\
& x \triangleleft (ys \odot zs).
\end{array}
$$

Thus another possible choice is $ys \odot zs = map\ (\oplus X_1)\ ys +\!\!+ X_2$, which also allows the proof of $(x \triangleleft ys) \odot zs = x \triangleleft (ys \odot zs)$ to go through. The base case $zs = [e] \odot zs$,

this time, expands to

$$zs = map\ (\oplus X_1)\ [e] +\!\!+ X_2 = X_1 : X_2.$$

We may pick $X_1 = head\ zs$ and $X_2 = tail\ zs$, and have thus discovered $(\odot)$,

$$ys \odot zs = map\ (\oplus\ head\ zs)\ ys +\!\!+ tail\ zs.$$

5 Reasoning With Conditionals

Our last example is chosen to demonstrate calculation involving conditional expressions, and the symmetric property that h is a list homomorphism if (5) holds, which is repeated here,

$$h\,xs = h\,xs \odot e \quad \wedge \quad h\,xs \odot (y \triangleright z) = (h\,xs \odot y) \triangleright z.$$

Given a predicate $p :: a \to Bool$, the function $pos\ p :: [a] \to [Int]$ returns the indexes of elements in the input list that satisfy p. For example, $pos\ (\geq 5)\ [6,8,4,2,0,10] = [0,1,5]$. For a definition,

$$\begin{array}{ll} pos & :: (a \to Bool) \to [a] \to [Int] \\ pos\ p\ [\,] & = [\,] \\ pos\ p\ (x : xs) & = \textbf{let}\ ys = pos\ p\ xs \\ & \quad \textbf{in if}\ p\ x\ \textbf{then}\ 0 : map\ (1+)\ ys\ \textbf{else}\ map\ (1+)\ ys. \end{array}$$

To compute pos in a $foldl$, we have to tuple it with $length$. Let $poslen = \langle pos, length \rangle$, we claim that $poslen = foldr\,(\triangleleft)\,([\,],0) = foldl\,(\triangleright)\,([\,],0)$, where

$$\begin{array}{l} x \triangleleft (ys, n) = (\textbf{if}\ p\ x\ \textbf{then}\ 0 : map\ (1+)\ ys\ \textbf{else}\ map\ (1+)\ ys, 1+n), \\ (ys, n) \triangleright z = (\textbf{if}\ p\ z\ \textbf{then}\ ys +\!\!+ [n]\ \textbf{else}\ ys, n+1). \end{array}$$

When we construct a proof of (4) in Section 3.2, the step where we fold back the definition of $(\triangleleft)$, due to repeated occurrences of s_2, is the step that triggered unification of metavariables. For $poslen$, we could construct $(\odot)$ and a proof of (4), but more guesswork is involved. Since $(\triangleright)$ for $poslen$ shares one more variable, n, in both components of the pair, it could be, and indeed is, easier to try the other direction — to generalise the proof of associativity to a proof of (5). That way we will be folding $(\triangleright)$ instead of $(\triangleleft)$, which possibly allows more unification to happen.

That $z \triangleleft ([\,],0) = ([\,],0) \triangleright z$ can be easily verified. The proof that $(\triangleleft)$ and $(\triangleright)$ associate is given below. To adapt to (5), the proof starts from a different side. For brevity we use the Bird-Meertens notation denoting $map\ f$ by f^*, and denote $\textbf{if}\ p\ \textbf{then}\ e_1\ \textbf{else}\ e_2$ by $\langle\!\langle p \to e_1, e_2 \rangle\!\rangle$.

$$\begin{array}{ll} & x \triangleleft ((ys, n) \triangleright z) \\ = & \quad \{\ \text{definition of } (\triangleright)\ \} \\ & x \triangleleft (\langle\!\langle p\,z \to ys +\!\!+ [n],\ ys \rangle\!\rangle, n+1) \end{array}$$

$$
\begin{array}{ll}
= & \{\ \text{definition of } (\triangleleft)\ \} \\
& (\langle\!\langle p\,x \to 0 : (1+)^*\ \langle\!\langle p\,z \to\ ys +\!\!+ [n],\ ys\rangle\!\rangle, \\
& \qquad\quad (1+)^*\ \langle\!\langle p\,z \to\ ys +\!\!+ [n],\ ys\rangle\!\rangle\rangle\!\rangle, \\
& \quad 1+n+1) \\
= & \{\ \text{transposition of nested-}\textbf{if}\ \} \\
& (\langle\!\langle p\,z \to \langle\!\langle p\,x \to\ (0:)\circ(1+)^*,\ (1+)^*\rangle\!\rangle\ (ys +\!\!+ [n]), \\
& \qquad\quad \langle\!\langle p\,x \to\ (0:)\circ(1+)^*,\ (1+)^*\rangle\!\rangle\ ys\rangle\!\rangle, \\
& \quad 1+n+1) \\
= & \{\ \langle\!\langle b \to f_1\ a, f_2\ a\rangle\!\rangle = \langle\!\langle b \to f_1, f_2\rangle\!\rangle\ a\ \} \\
& (\langle\!\langle p\,z \to \langle\!\langle p\,x \to\ 0 : ((1+)^*\ ys),\ (1+)^*\ ys\rangle\!\rangle +\!\!+ [1+n], \\
& \qquad\quad \langle\!\langle p\,x \to\ 0 : ((1+)^*\ ys),\ (1+)^*\ ys\rangle\!\rangle\rangle\!\rangle, \\
& \quad 1+n+1) \\
= & \{\ \text{definition of } (\triangleright)\ \} \\
& (\langle\!\langle p\,x \to\ 0 : (1+)^*\ ys,\ (1+)^*\ ys\rangle\!\rangle, 1+n) \triangleright z \\
= & \{\ \text{definition of } (\triangleleft)\ \} \\
& (x \triangleleft (ys, n)) \triangleright z.
\end{array}
$$

The rule "transposition of nested-**if**" states that

$$
\begin{array}{ll}
& \langle\!\langle p \to\ \langle\!\langle q \to\ f_1,\ f_2\rangle\!\rangle\ x_1,\ \langle\!\langle q \to\ f_1,\ f_2\rangle\!\rangle\ x_2\rangle\!\rangle \\
= & \langle\!\langle q \to\ f_1\ \langle\!\langle p \to\ x_1,\ x_2\rangle\!\rangle,\ f_2\ \langle\!\langle p \to\ x_1,\ x_2\rangle\!\rangle\rangle\!\rangle.
\end{array}
$$

The next step is to construct a definition of $(\odot)$ together with a proof of $h\,xs \odot (y \triangleright z) = (h\,xs \odot y) \triangleright z$ from the proof of associativity. A possible candidate of $(\odot)$ is generalised from the definition of $(\triangleleft)$:

$$
(ys_1, n_1) \odot (ys_2, n_2) = (\langle\!\langle X_1 \to\ X_2 +\!\!+ (X_3+)^*\ ys_2,\ (X_4+)^*\ ys_2\rangle\!\rangle, X_5 + n).
$$

Applying the lessons learnt in previous sections, we replace subterms involving x to metavariables, and replace constants by metavariables in a way that allows flexibility in length.

Now we try to construct a proof of $(x \triangleleft ys) \odot zs = x \triangleleft (ys \odot zs)$. We follow the steps of the proof of associativity:

$$
\begin{array}{ll}
& (ys_1, n_1) \odot ((ys, n) \triangleright z) \\
= & \{\ \text{definition of } (\triangleright)\ \} \\
& (ys_1, n_1) \odot (\langle\!\langle p\,z \to\ ys +\!\!+ [n],\ ys\rangle\!\rangle, n+1) \\
= & \{\ \text{definition of } (\odot)\ \} \\
& (\langle\!\langle X_1 \to X_2 +\!\!+ (X_3+)^*\ \langle\!\langle p\,z \to\ ys +\!\!+ [n],\ ys\rangle\!\rangle, \\
& \qquad\quad (X_4+)^*\ \langle\!\langle p\,z \to\ ys +\!\!+ [n],\ ys\rangle\!\rangle\rangle\!\rangle, \\
& \quad X_5 + n + 1) \\
= & \{\ \text{transposition of nested-}\textbf{if}\ \}
\end{array}
$$

$$
\begin{array}{l}
(\langle\!\langle p\, z \to \langle\!\langle X_1 \to (X_2 +\!\!+) \circ (X_3+)^*,\ (X_4+)^* \rangle\!\rangle\ (ys +\!\!+ [n]),\\
\qquad\qquad \langle\!\langle X_1 \to (X_2 +\!\!+) \circ (X_3+)^*,\ (X_4+)^* \rangle\!\rangle\ ys \rangle\!\rangle,\\
\quad X_5 + n + 1),
\end{array}
$$

Unification happens in the next two steps. In the first step, to move ys into the conditional while leaving n outside, X_3 and X_4 must be unified. In the second step, X_5 and X_3 are unified to allow the definition of $(\triangleright)$ to fold:

$$
\begin{array}{ll}
 & (\langle\!\langle p\, z \to \langle\!\langle X_1 \to (X_2 +\!\!+) \circ (X_3+)^*,\ (X_4+)^* \rangle\!\rangle\ (ys +\!\!+ [n]),\\
 & \qquad\qquad \langle\!\langle X_1 \to (X_2 +\!\!+) \circ (X_3+)^*,\ (X_3+)^* \rangle\!\rangle\ ys \rangle\!\rangle,\\
 & \quad X_3 + n + 1)\\
= & \quad \{\ \langle\!\langle b \to f_1\ a, f_2\ a \rangle\!\rangle = \langle\!\langle b \to f_1, f_2 \rangle\!\rangle\ a\ \}\\
 & (\langle\!\langle p\, z \to \langle\!\langle X_1 \to X_2 +\!\!+ ((X_3+)^*\ ys),\ (X_3+)^*\ ys \rangle\!\rangle +\!\!+ [X_3 + n],\\
 & \qquad\qquad \langle\!\langle X_1 \to X_2 +\!\!+ ((X_3+)^*\ ys),\ (X_3+)^*\ ys \rangle\!\rangle \rangle\!\rangle,\\
 & \quad X_3 + n + 1)\\
= & \quad \{\ \text{definition of } (\triangleright)\ \}\\
 & (\langle\!\langle X_1 \to X_2 +\!\!+ (X_3+)^*\ ys,\ (X_3+)^*\ ys \rangle\!\rangle, X_3 + n) \triangleright z\\
= & \quad \{\ \text{definition of } (\odot)\ \}\\
 & ((ys_1, n_1) \odot (ys, n)) \triangleright z.
\end{array}
$$

Therefore, one possible candidate of $(\odot)$ is

$$
(ys_1, n_1) \odot (ys_2, n_2) = (\langle\!\langle X_1 \to X_2 +\!\!+ (X_3+)^*\ ys_2,\ (X_3+)^*\ ys_2 \rangle\!\rangle, X_3 + n_2),
$$

which satisfies $(x \triangleleft ys) \odot zs = x \triangleleft (ys \odot zs)$ for any choice of X_1, X_2, and X_3.

Now we turn to the base case to figure out the rest of the metavariables. Expanding the base case,

$$
(ys_1, n_1) = (ys_1, n_1) \odot ([\,], 0) = (\langle\!\langle X_1 \to X_2, [\,] \rangle\!\rangle, X_3),
$$

one bold choice is letting $X_1 = \mathit{true}$, $X_2 = ys_1$, and $X_3 = n_1$. This completes our search for $(\odot)$:

$$
(ys_1, n_1) \odot (ys_2, n_2) = (ys_1 +\!\!+ (n_1+)^*\ ys_2,\ n_1 + n_2).
$$

6 Conclusions

Previous discussions on constructing list homomorphisms using the third list homomorphism theorem often overlook the fact that, for even slightly non-trivial problems, it is not an easy task to construct one of $(\triangleleft)$ and $(\triangleright)$, given another, in a constructive, formal manner, and thus an associativity proof is often needed. The $(\odot)$ operator, on the other hand, is a generalisation of $(\triangleleft)$ and $(\triangleright)$, whose proof of correctness also generalises from the proof of associativity. It is thus a waste throwing the proof away.

We have proposed and demonstrated a novel approach to constructing $(\odot)$. Starting with a trivial generalisation of either $(\triangleleft)$ or $(\triangleright)$, we exploit the constraint enforced by the proof of associativity to refine $(\odot)$. Once we have constructed $(\odot)$, we have its correctness proof too. It also explains the phenomena that in practice, $(\odot)$ often consists of fragments of code from $(\triangleleft)$ and $(\triangleright)$ — it can be constructed by generalising from one of them before being refined by another.

Acknowledgements The authors would like to thank Zhenjiang Hu and Akimasa Morihata for discussion on the subject matter and, in particular, for suggesting the reference to work by Geser and Gorlatch. We would also like to thank the anonymous referees for their useful comments.

References

1. Bird, R.S.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design, pp. 3–42. No. 36 in NATO ASI Series F, Springer-Verlag (1987)
2. Bird, R.S.: Introduction to Functional Programming using Haskell. Prentice Hall (1998)
3. Bird, R.S., de Moor, O.: Algebra of Programming. International Series in Computer Science, Prentice Hall (1997)
4. Cole, M.: Parallel programming, list homomorphisms and the maximum segment sum problems. Tech. Rep. CSR-25-93, Department of Computer Science, University of Edinburgh (1993)
5. Dijkstra, E.W.: The next fifty years. Tech. Rep. EWD1243, Eindhoven University of Technology (2004)
6. Geser, A., Gorlatch, S.: Parallelizing functional programs by generalization. Journal of Functional Programming 9(6), 649–673 (1999)
7. Gibbons, J.: The third homomorphism theorem. Journal of Functional Programming 6(4), 657–665 (1996)
8. Hu, Z., Iwasaki, H., Takeichi, M.: Construction of list homomorphisms via tupling and fusion. In: Penczek, W., Szalas, A. (eds.) International Symposium on Mathematical Foundations of Computer Science. pp. 407–418. No. 1113 in Lecture Notes in Computer Science, Springer-Verlag (1996)
9. Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In: Shao, Z., Pierce, B.C. (eds.) Symposium on Principles of Programming Languages. pp. 177–185. ACM Press, Savannah, GA, USA (jan 2009)
10. Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs. In: Ferrante, J., McKinley, K.S. (eds.) Programming Language Design and Implementation. pp. 146–155. ACM Press (2007)