Constructing List Homomorphisms for Parallelism

Sergei Gorlatch

University of Passau, D–94030 Passau, Germany.

E-mail: gorlatch@fmi.uni-passau.de

Abstract

List homomorphisms are functions which can be efficiently computed in parallel since they suit the divide-and-conquer paradigm ideally. We propose a simple approach to testing whether a function is a homomorphism and, if so, determining how it can be parallelized. The approach is based on analyzing two inherently sequential representations of functions based on *cons*- and *snoc*-lists. For some interesting functions which are not homomorphisms, e.g. the maximum segment sum problem, our method provides a systematic way of embedding into a homomorphism.

Keywords: Bird-Meertens formalism; Parallel program development.

1 Introduction

The properties of functional languages endow them with fundamental advantages for parallel programming, making possible software development via rigorous transformations and exploiting powerful data and control abstraction mechanisms. The practical problem is how to package these properties in order to ensure the efficiency and predictability of the program execution on parallel architectures. In this paper, we address the methodological aspect of this problem and concentrate on an important class of functions over lists, called homomorphisms, which are efficiently parallelizable.

A list function h is a *homomorphism* iff there exists a binary operation $\oplus$ such that, for all lists x and y:

$$h(x +\!\!\!+ y) = h(x) \oplus h(y) \tag{1}$$

where $+\!\!\!+$ is the list concatenation.

Intuitively this means that the value of h on the large list depends in a particular way (using combining operation $\oplus$) on the values of h applied to the pieces of the list. The computations of $h(x)$ and $h(y)$ are independent and can be carried out in parallel. This simple equation can be viewed as expressing the well-known divide-and-conquer paradigm of parallel programming.

Actually, homomorphisms can be defined on arbitrary composite types (trees, arrays, etc.), and this is one of the topics in the Bird-Meertens formalism (BMF) [1, 2]. Examples of homomorphisms are simple functions, such as *length*, which computes the length of the list, or *sum*, which, for a list of numbers, yields the sum of all its elements, and also functions that are more complicated and important in practice, such as e.g. *scan* (prefix sums) [3].

An important property of homomorphisms, which is one of the reasons of the interest in them, is Bird's First Homomorphism Theorem:

Theorem. Function h is a homomorphism iff it can be factored into the following composition of two functions:

$$h = red\,(\oplus) \;\circ\; map\,(f) \tag{2}$$

Both *map* and *red* are well known in functional programming (*red* stands for "reduction" and is often called *fold*). Again, the divide-and-conquer intuition allows to understand easily the right-hand side of (2) as, first, parallel application of function f to all elements of a list and second, combining the results using reduction with operation $\oplus$. Whereas *map* is totally parallel, the reduction can be computed in parallel on a tree-like structure, with the combining operation $\oplus$ applied in the nodes. For an arbitrary fixed processor number, this standard parallel implementation of a homomorphism often yields a linear scaled speed-up [4].

Therefore, the property of a function being a homomorphism is usually equal to the property of being well suitable for parallelization.

There are three main problems with homomorphisms:

- Testing condition (1) may be cumbersome.

- It would be desirable not only to test the homomorphic property but also to construct the homomorphism, i.e. for a given function h find the corresponding factorizing function f and combining operation $\oplus$ of (2), because this immediately gives "for free" a standard parallel implementation of h. For functions like *length* this construction is simple; however, already for the *scan* function it is not evident, and this even led to errors in published parallel algorithms and required a formal correctness proof [5].

- If a function is not a homomorphism, it can sometimes be converted to a homomorphism with some additional computations. Cole gives in [6, 7] several examples of such "almost-homomorphisms"; however, converting into a homomorphism is done using problem-specific *eureka* decisions.

Our goal is a systematic approach to expressing a function as a homomorphism or an almost-homomorphism aiming at its parallel implementation. The message of the paper is that this can be done using two well-known sequential functional techniques, namely *cons*- and *snoc*-lists.

2 Constructing Homomorphisms

We use one symbol ":" for two operations: *cons*, which attaches an element at the beginning of the list and *snoc*, which attaches the element at the list's end, relying on the context in distinguishing which of the two is meant, in that we use $a, b, \cdots$ for denoting elements and $x, y, xs, ys, \cdots$ for denoting lists. We restrict ourselves to finite non-empty lists.

List function h is called *leftwards* iff there exists a binary operation $\otimes$ such that for all elements a and lists y holds $h\,(a : y) = a \otimes h(y)$. Dually, function h is *rightwards* iff, for some $\odot$, $h\,(y : a) = h(y) \odot a$.

Note that operations $\otimes$ and $\odot$ may be arbitrarily complex, including for instance conditional branching, so many functions are either leftwards or rightwards or both. The importance of a function being both left- and rightwards is shown by the following theorem, which combines the so-called second and third homomorphism theorems. Both are attributed to Bird and considered folk theorems in the BMF-community. The former is simple to prove; the latter was proved by Meertens and presented systematically by Gibbons [8, 9].

Theorem. Function h is a homomorphism iff it is leftwards and rightwards.

Gibbons says in [8] about this theorem with respect to the problem of constructing the homomorphism, i.e. finding a suitable operation $\oplus$ which satisfies (2) for a function which is say $\otimes$-leftwards and $\odot$-rightwards:

> "The existence of a suitable operation is guaranteed, but the theorem does not address the question of the existence – let alone the construction – of a direct and efficient way of computing it."

Solving this problem will be our first goal. We start by imposing an additional restriction on the leftwards and rightwards functions.

Function h is called *leftwards-homomorphic* iff there exists $\oplus$ such that, for an arbitrary element a and arbitrary list x, $h\,(a : x) = h\,([a]) \oplus h\,(x)$. The dual definition of a *rightwards-homomorphic* function is obvious.

Let us introduce function f such that $f\,(a) = h\,([a])$. There are unique leftwards-homomorphic and rightwards-homomorphic functions for given f and $\oplus$, which we denote by $lwh\,(f, \oplus)$ and $rwh\,(f, \oplus)$ correspondingly.

Evidently, every leftwards-homomorphic (rightwards-homomorphic) function is also leftwards (rightwards), but not always vice versa, e.g. :

$$
\begin{array}{rcl}
g\,[a] & = & |a| \\
g\,(a : x) & = & \texttt{if } a \leq g\,(x) \texttt{ then } |a + g\,(x)| \texttt{ else } |a - g\,(x)|
\end{array}
$$

Function g is clearly leftwards but not leftwards-homomorphic, since we cannot express $g\,(a : x)$ using only $|a|$ and $g\,(x)$.

To express the relation of *lwh* and *rwh* functions to homomorphisms, we first note that since operation $\oplus$ in (2) is necessarily associative on the range of h, a $\oplus$-homomorphic function is completely determined by its actions on singleton lists, so we may write $hom\,(f, \oplus)$ for the unique $\oplus$-homomorphic function h such that, for all a, $h\,([a]) = f\,(a)$.

Proposition 1. If function h is a homomorphism then it is both left- and rightwards-homomorphic with the same combining operation, i.e.:

$$h = hom\,(f, \oplus) \;\Rightarrow\; ((h = lwh\,(f, \oplus)) \wedge (h = rwh\,(f, \oplus))$$

The proposition is easily proved by specialization of (2) for two cases, namely when the first or the second of the concatenated lists is singleton.

We are interested in the other direction: do left- and/or right-homomorphy imply that the function is a homomorphism? Note that $\oplus$ in the definitions of left- and rightwards-homomorphic functions is not required to be associative; we introduce the following weakened definitions.

An operation $\oplus$ is *left-associative* on the range of h iff, for an arbitrary element a and lists x and y, $h\,([a]) \oplus (h\,(x) \oplus h\,(y)) \;=\; (h\,([a]) \oplus h\,(x)) \oplus h\,(y)$. Symmetrically, operation $\oplus$ can be *right-associative*.

Proposition 2. If function h is leftwards-homomorphic with operation $\oplus$ which is left-associative then h is a homomorphism. Dually, a rightwards-homomorphic function with a right-associative operation is a homomorphism.

Proof. We must show that for arbitrary lists x and y holds

$$h\,(x +\!\!+\, y) = h\,(x) \;\oplus\; h\,(y)$$

We prove this for the case $h = lwh\,(f, \oplus)$ by induction on the length of list x, for arbitrary y; the rightwards case is proved similarly.

Induction base. If $x = [a]$ then

$$\begin{array}{ll} & h\,([a] +\!\!+\, y) \\ = & \quad \{ \text{ list constructors } \} \\ & h\,(a : y) \\ = & \quad \{ \; h = lwh\,(f, \oplus) \; \} \\ & h\,([a]) \;\oplus\; h\,(y) \end{array}$$

Induction hypothesis: for a list x of length at most k, and an arbitrary list y:

$$h\,(x +\!\!+\, y) = h\,(x) \;\oplus\; h\,(y)$$

Induction step. List x of length $\leq k{+}1$ can be represented as follows: $x = a : xs$, where a is an element and xs is of length $\leq k$. Now we have:

$$\begin{array}{ll} & h\,(x +\!\!+\, y) \\ = & \quad \{ \; x = a : xs \; \} \end{array}$$

$$
\begin{array}{ll}
& h\,((a : xs) \mathbin{+\!\!+} y) \\
= & \quad \{ \text{ list constructors } \} \\
& h\,(a : (xs \mathbin{+\!\!+} y)) \\
= & \quad \{ \text{ } h \text{ is leftwards-homomorphic } \} \\
& h\,([a]) \,\oplus\, h\,(xs \mathbin{+\!\!+} y) \\
= & \quad \{ \text{ induction hypothesis } \} \\
& h\,([a]) \,\oplus\, (h\,(xs) \,\oplus\, h\,(y)) \\
= & \quad \{ \text{ left-associativity } \} \\
& (h\,([a]) \,\oplus\, h\,(xs)) \oplus h\,(y) \\
= & \quad \{ \text{ } h \text{ is leftwards-homomorphic } \} \\
& h\,([a] \mathbin{+\!\!+} xs) \oplus h\,(y) \\
= & \quad \{ \ [a] \mathbin{+\!\!+} xs = x \ \} \\
& h\,(x) \oplus h\,(y)
\end{array}
$$

End of proof

Proposition 2 suggests a possible way of finding the homomorphic representation of a function: construct a leftwards-homomorphic definition of the function on *cons*-lists and prove that the combining operation is left-associative. Another way is to use the (less traditional) rightwards *snoc*-representation.

Sometimes this simple method works, as the following example shows.

Example 1. Function *length* on a singleton list is: $length\,([a]) = 1$, so $f = one$, where $one\,(x) = 1$. The *cons*-definition is obvious:

$$length\,(a : x) = 1 + length\,(x) = f(a) + length\,(x)$$

According to the definition of *lwh*: $length = lwh\,(one, +)$. From associativity of $+$ follows immediately that $length = hom\,(one, +)$.

Example 2. A more complicated example is function *scan*, which for a list returns the list of its "running totals" with the associative operation $\odot$, e.g.:

$$scan\,(\odot)\,([a, b, c]) = [a, a \odot b, a \odot b \odot c]$$

For a singleton list: $scan\,([a]) = [a]$, so $f = [.]$.

In the *cons*-definition, we must apply $\odot$ to a with all elements of $scan\,(x)$ and position all the results after a in the resulting list:

$$scan\,(\odot)\,(a : x) = a \ : \ (map\,(a\,\odot)\,(scan\,(\odot)\,x)) \tag{3}$$

Here so-called sectioning is exploited in that we fix one argument of $\odot$ and obtain the unary function $(a\,\odot)$ which can be *map*ped.

The representation (3) does not match the format of *lwh* because a is used whereas only $f\ a$ is allowed. Since $f\ a = [a]$, there are different possibilities to express a via $f\ a$, e.g.: $a = fst(f\ a)$, where *fst* yields the first element of the list, $a = last(f\ a)$, or we could use $((f\ a)\mathbin{+\!\!+})$ for $(a\,:)$. So, we immediately obtain six possible expressions for $\oplus$, which must, in addition, be checked for left-associativity. The most disappointing is, however, that neither of these expressions define a left-associative operation!

Let us try to use rightwards-homomorphy. In the *snoc*-definition, we need to compute $scan\,(x)$ and append it on the right with the result of computing $\odot$ on a and the last element of $scan\,(x)$:

$$scan\,(\odot)\,(x : a) = scan\,(\odot)\,(x)\ :(last\,(scan\ \odot\ (x)) \odot a) \tag{4}$$

Alas, we run into a similar problem as for *cons*-lists: both obvious substitutions for a, namely $fst\,(f\ a)$ and $last\,(f\ a)$, lead to a non-associative operation, and thus we are still not able to express the *scan*-function as a homomorphism.

Proposition 1 shows a possible way out: for a homomorphic function, there exist *both* left- and rightwards-homomorphic representations with the *same* operation $\oplus$. In order to obtain this operation in our example, we must *match* the *cons*-representation (3) and the *snoc*-representation (4) as follows: both must be of the form $u \oplus v$, where $u = f\ a$ and $v = scan\,(\odot)\,x$ in (3), and $u = scan\,(\odot)\,x$ and $v = f\ a$ in (4). Our goal – the matching expressions – dictates how to replace the occurrences of a. The correctness of these replacements must be proven, so we write them down as *proof obligations* for arbitrary elements a, b and list y:

$$\begin{array}{rcl} a & = & last\,[a] \\ [a] & = & scan\,(\odot)\,[a] \\ a : y & = & [a] \mathbin{+\!\!+} y \\ y : (b \odot a) & = & y \mathbin{+\!\!+} (map\,(b\odot)\,[a]) \end{array}$$

All four equalities are easily proved. After replacement, we obtain:

$$\begin{array}{rcl} scan\,(\odot)\,(a : x) & = & scan\,(\odot)\,[a] \mathbin{+\!\!+} (map\,(last\,(scan\,(\odot)\,[a])\,\odot)\,(scan\,(\odot)\,x)) \\ scan\,(\odot)\,(x : a) & = & scan\,(\odot)\,(x) \mathbin{+\!\!+} (map\,(last\,(scan\ \odot\ (x))\,\odot)\,(scan\,(\odot)\,[a])) \end{array}$$

These expressions match perfectly and allow us to read out the desired operation:

$$u \oplus v = u \mathbin{+\!\!+} (map\,(last\,(u)\,\odot)\,(v))$$

After proving associativity of $\oplus$, we come to the conclusion that the *scan* function is the following homomorphism: $scan\,(\odot) = hom\,([.], \oplus)$.

That is, we have come to a new methodology for expressing a function as a homomorphism: define the function in both the *cons*- and the *snoc*-format and try to match the expressions in order to obtain the common operation. If the constructed operation is associative then the function is a homomorphism and we know its components f and $\oplus$ explicitly.

It would be tempting to deduce that, if a function is both *lwh* and *rwh* with the same operation, then it is a homomorphism, i.e. to drop the test for associativity. The following simple counter example (courtesy of J. Gibbons) shows that this is not the case. We can define the identity function id on lists as $lwh\,([.], \oplus)$ and $rwh\,([.], \oplus)$, with the following operation $\oplus$:

$$u \oplus v = [head\,u] \mathbin{+\!\!+} init\,v \mathbin{+\!\!+} tail\,u \mathbin{+\!\!+} [last\,v]$$

where functions $tail$ and $init$ yield the list without its first or last element, correspondingly. Although id is both right- and leftwards homomorphic with the same $\oplus$, it is clearly not a homomorphism with such operation.

The considerations in the next section show that our methodology is even more useful in the case of almost-homomorphisms, where the method based on the *cons*- or *snoc*-representation alone simply does not work.

3 Almost-Homomorphisms

In this section, we deal with the situation that a function is not a homomorphism. Many practical non-homomorphic functions are still convertible into so-called *almost-homomorphisms*, which are usually a composition of a homomorphism and some adjusting function. If the adjusting function is simple then the conversion to a well-parallelizable homomorphism is feasible.

It is well known that every function can be represented as follows: for a non-homomorphic function h, consider a new function g such that $g\,(x) = (x, h\,(x))$. The tuple-function g is homomorphic:

$$g\,(x \mathbin{+\!\!+} y) = (x \mathbin{+\!\!+} y\;,\; h\,(x \mathbin{+\!\!+} y)) = g\,(x) \oplus g\,(y)$$

where $(x, u) \oplus (y, v) = (x \mathbin{+\!\!+} y\;,\; h\,(x \mathbin{+\!\!+} y))$.

Evidently, $h = snd \circ g$, where snd yields the second component of a tuple. This seems to provide an amazingly simple way of computing every function in parallel as a homomorphism, followed by a simple projection. A closer look at the operation $\oplus$ reveals the snag: it does not make use of the previously computed values u and v and instead computes function h from scratch!

However, there are also examples where a conversion to a "true" homomorphism exists. Cole reports several case studies [6, 7]; his method consists of constructing a homomorphism as a tuple of functions, where the original function is one of the components. The main difficulty is to guess which *auxiliary functions* must be included in a tuple and to prove that the tuple is indeed a homomorphism, i.e. to find a corresponding associative operation. Usually, this requires a lot of ingenuity from the program developer. Cole says in [6]:

> "It is of interest to ask how easily the resulting algorithms might have been derived in a more strictly formal setting."

We will show that our approach with *snoc*- and *cons*-lists allows to construct tuple-homomorphisms systematically.

Example 3. We consider an example using the original notation from [6]: the *maximum segment sum* (*mss*) problem. Given a list of integers, function *mss* finds the contiguous segment of the list whose members have the largest sum among all such segments. For simplicity, we only consider the problem of returning the value of this sum. For example,

$$mss\,[\,2,-4,2,-1,6,-3\,] = 7$$

where the result is contributed by the segment $[2,-1,6]$. The empty segment is defined to have sum 0, so the result is always non-negative.

Let us try to express function *mss* in a usual functional style over *cons*-lists. For some element a and list x, it may well be the case that $mss\,(a : x) = (mss\,[a]) \uparrow (mss\,x)$, where operation $\uparrow$ returns the larger of its two integer arguments. But we must not overlook the possibility that the true segment of interest includes both a and some (initial) segment of x. Therefore, we introduce an auxiliary function *mis* which yields the sum of the *maximum initial segment*. We must add the definition of *mis* on *cons*-lists (no new auxiliary functions are required for that), and thus we obtain the following definition of *mss*:

$$\begin{aligned} mss\,(a : x) &= mss\,[a] \uparrow mss\,x \uparrow (a + mis\,x) \\ mis\,(a : x) &= mis\,[a] \uparrow (a + mis\,x) \end{aligned}$$

Now, according to our methodology, we define *mss* on *snoc*-lists. The construction is the same as for *cons*-lists and leads to another auxiliary function, *mcs*, which yields the sum of the *maximum concluding segment* of a list:

$$\begin{aligned} mss\,(x : a) &= mss\,x \uparrow (mcs\,x + a) \uparrow mss\,[a] \\ mcs\,(x : a) &= (mcs\,x + a) \uparrow mcs\,[a] \end{aligned}$$

The example demonstrates how some auxiliary functions come into play, but we do not see a homomorphism yet. We proceed now with some general considerations for tuples of functions, and then come back to our *mss* example.

We consider tuples of functions over lists. The notion of homomorphism can be extended straightforwardly to them: tuple $h = (h_1, h_2, \cdots, h_n)$ is a homomorphism, iff there exists such $\oplus$ that $(h_1, h_2, \cdots, h_n)\,(x +\!\!+\, y) \;=\; (h_1, h_2, \cdots, h_n)\,x \oplus (h_1, h_2, \cdots, h_n)\,y$. Operator $\oplus$ takes two n-tuples as arguments and yields a resulting n-tuple, where every element is computed using possibly all $2n$ argument values. We use our usual notation also for tuple homomorphisms: $h = hom\,(f, \oplus)$.

Let us consider *definitions* of tuples of functions as systems of recursive equations of the form:

$$h_i\,z = E_i\,(h_1, h_2, \cdots, h_n, z)\,,\;\; i = 1, 2, \cdots, n$$

where z is an expression over list constructors, say $(a : x)$, expressions E_i make use of the values of h_i on z and/or its parts, i.e. a and x. Such definitions are usual in contemporary functional languages like ML. Depending on the constructor used in z, we distinguish *cons*-, *snoc*- and $+\!\!+$-definitions.

A tuple of functions is called *cons-complete* if it has a closed *cons*-definition, i.e. no auxiliary functions, other than those already in the tuple are necessary. Similarly, tuples may be defined to be *snoc*- and $+\!\!+$-complete. In Example 3, we have seen two tuples: (mss, mis) is *cons*-complete, (mss, mcs) is *snoc*-complete.

A tuple $(h_1, h_2, \cdots, h_n)$ is called *leftwards-homomorphic*, or just *lwh*-tuple, iff it is *cons*-complete and can be defined in the form:

$$(h_1, h_2, \cdots, h_n)\,(a : x) = (h_1\,[a], h_2\,[a], \cdots, h_n\,[a]) \;\oplus\; (h_1\,x, h_2\,x, \cdots, h_n\,x)$$

where $\oplus$ is an associative operation on tuples. It is easy to see that *lwh*-tuples are a natural generalization of *lwh*-functions from the previous section, so we use the same notation. Dually, we define *rwh*-tuples.

We can generalize also the properties of left- and right-associativity and reformulate Propositions 1 and 2 for the case of tuples.

Proposition 3. If a tuple of functions is leftwards- (rightwards-) homomorphic with a left- (right-) associative operation then it is homomorphic with the same operation. If a tuple is homomorphic with operation $\oplus$ then there exist both leftwards- and rightwards-homomorphic representations of it with $\oplus$.

This proposition leads the way to a homomorphism: we must construct a tuple of functions which is both *lwh* and *rwh* with the same operation $\oplus$. Therefore, all functions arising in the *cons*- and *snoc*-definitions of the original function must belong to this tuple. For arbitrary two tuples, t_1 and t_2, we define a *union tuple*, written $t_1 \cup t_2$, as the tuple consisting of functions from both tuples without repetitions.

Example 3 (continued). In our example, the union tuple is obviously as follows: $(mss, mis) \cup (mss, mcs) = (mss, mis, mcs)$. To be a homomorphism, the new 3-tuple must be both *cons*- and *snoc*-complete.

Let us try to find a *cons*-definition of the union-tuple in our example. For functions *mss* and *mis*, which have been already defined over *cons*-lists, we can simply borrow their definitions. It remains to define *mcs* – and this seems impossible if we are only allowed to use the functions provided in the tuple, i.e. *mss*, *mis*, *mcs*. The point is that the concluding segment of $(a : x)$ may be the whole list, and thus we need to know the sum of elements of x which no (combination) of the three available functions can yield. So, tuple (mss, mis, mcs) is not *cons*-complete, and we have to introduce one more auxiliary function *ts* (for *total sum*).

The new tuple (mss, mis, mcs, ts) has the following complete *cons*-representation:

$$\begin{aligned}
mss\,(a : x) &= mss\,[a] \uparrow (a + mis\,x) \uparrow mss\,x \\
mis\,(a : x) &= mis\,[a] \uparrow (a + mis\,x) \\
mcs\,(a : x) &= (a + ts\,x) \uparrow mcs\,x \\
ts\,(a : x) &= ts\,[a] + ts\,x
\end{aligned}$$

It turns out that our new 4-tuple is also *snoc*-complete:

$$\begin{aligned}
mss\,(x:a) &= mss\,x \uparrow (mcs\,x + a) \uparrow mss\,[a] \\
mis\,(x:a) &= mis\,x \uparrow (ts\,x + a) \\
mcs\,(x:a) &= (mcs\,x + ts\,[a]) \uparrow mcs\,[a] \\
ts\,(x:a) &= ts\,x + ts\,[a]
\end{aligned}$$

Function f determines the result of the tuple on singleton list:

$$f\,(a) = (mss, mis, mcs, ts)\,[a] = (a \uparrow 0\,,\ a \uparrow 0\,,\ a \uparrow 0\,,\ a)$$

We must match the *cons*- and the *snoc*-representation for each of four functions. We see immediately that the two definitions for function ts are *lwh* and *rwh*, furthermore they match perfectly. For other functions in the tuple, we must replace a in their definitions using (a part of) tuple $(mss\,[a], mis\,[a], mcs\,[a], ts\,[a])$ The straightforward replacement for a would be $ts\,[a]$, which is equal to a by definition of $f\,(a)$, but the definitions do not match in this case!

To achieve the necessary match in *mss*, we need to replace a by $mcs\,[a]$ in the leftwards and by $mis\,[a]$ in the rightwards representation. The methodology thus urges us to prove the following:

$$\begin{aligned}
(a + mis\,x) \uparrow mss\,x &= (mcs\,[a] + mis\,x) \uparrow mss\,x \\
mss\,x \uparrow (a + mis\,x) &= mss\,x \uparrow (mcs\,[a] + mis\,x)
\end{aligned}$$

which is in fact true. A similar check is also required for functions *mcs* and *mis*, where some occurrences of a are replaced by $ts\,[a]$ (which is obviously correct) and other by $mis\,[a]$ (this has to be proved).

In order to demonstrate that, after these replacements, the representations are matched, we write them for each function under each other:

$$\begin{aligned}
mss\,(a:x) &= mss\,[a] \uparrow (mcs\,[a] + mis\,x) \uparrow mss\,x \\
mss\,(x:a) &= mss\,x\ \uparrow (mcs\,x + mis\,[a]) \uparrow mss\,[a] \\[4pt]
mis\,(a:x) &= mis\,[a] \uparrow (ts\,[a] + mis\,x) \\
mis\,(x:a) &= mis\,x\ \uparrow (ts\,x + mis\,[a]) \\[4pt]
mcs\,(a:x) &= (mcs\,[a] + ts\,x) \uparrow mcs\,x \\
mcs\,(x:a) &= (mcs\,x + ts\,[a]) \uparrow mcs\,[a] \\[4pt]
ts\,(a:x) &= ts\,[a] + ts\,x \\
ts\,(x:a) &= ts\,x + ts\,[a]
\end{aligned}$$

Therefore, our tuple is the homomorphism: $(mss, mis, mcs, ts) = hom\,(f, \oplus)$, where f has been introduced before and $\oplus$ is read out directly from the matched representations as follows:

$$
\begin{array}{ll}
& (mss\,x, mis\,x, mcs\,x, ts\,x) \;\oplus\; (mss\,y, mis\,y, mcs\,y, ts\,y) \\
= & (\,(mss\,x) \uparrow (mss\,y) \uparrow (mcs\,x + mis\,y), mis\,x \uparrow (ts\,x + mis\,y)\,), \\
& (mcs\,x + ts\,y) \uparrow (mcs\,y), (ts\,x + ts\,y)
\end{array}
$$

The associativity of $\oplus$ is easily proved.

The target function mss is now computable as follows:

$$
mss \;=\; fst \circ red\,(\oplus) \circ map\,(f)
$$

where fst yields the first component of a tuple.

This homomorphic representation is parallelizable in a standard way on a tree-like structure of processors. Since both function f and operation $\oplus$ require constant time, the total time complexity of this, obtained "for free", parallel homomorphic algorithm is $O(\log n)$. The processor number can be reduced to $O(n/\log n)$ employing the standard trick of simulating lower levels of the tree sequentially based on Brent's theorem [4]. Therefore, the algorithm is both time and cost optimal.

4 Nested Parallelism

In this section, we apply our methodology to the example from [7] which demonstrates so-called *nested parallelism*. The idea is that, if the operation $\oplus$ in a homomorphism is an expensive one, then it is sometimes possible to parallelize it using the same methodology, i.e. by detecting/embedding into a homomorphism.

We consider the problem of determining whether the brackets of an arbitrary number of types are correctly matched in a given string. Such languages are sometimes called "input-driven" and are known to be the most complicated subclass of context-free languages which are parsable by optimal parallel algorithms [10]. There exists a straightforward linear-time sequential algorithm which maintains a stack during scanning the input. Opening brackets are pushed, and closing brackets are matched with the current stack top. Failure is indicated by a mismatch, or by an empty stack at the end of scanning. Obviously the use of a stack appears unpromising for parallelizing the algorithm.

The idea of the algorithm is to exclude all matching brackets and then to test the remaining list for emptiness. If *exmatch* denotes the function which yields the unmatched brackets in a given list then the solution is given by $isempty \circ exmatch$.

Let us consider how function *exmatch* can be computed on *cons*- and *snoc*-lists in a homomorphic way. For simplicity, we assume that the input string (list) consists exclusively of brackets. Using predicate *match* for comparing two brackets, the *cons*- and *snoc*-representations look as follows:

$$\begin{array}{rcl} \mathit{exmatch}\,[a] & = & [a] \\ \mathit{exmatch}\,(a : x) & = & \texttt{if}\ \mathit{match}\,(a, \mathit{head}\,(\mathit{exmatch}\,x)) \\ & & \texttt{then}\ \mathit{exmatch}\,(\mathit{tail}\,(\mathit{exmatch}\,x))\ \texttt{else}\ (a : (\mathit{exmatch}\,x)) \\ \mathit{exmatch}\,(x : a) & = & \texttt{if}\ \mathit{match}\,(\mathit{last}\,(\mathit{exmatch}\,x), a) \\ & & \texttt{then}\ \mathit{exmatch}\,(\mathit{init}\,(\mathit{exmatch}\,x))\ \texttt{else}\ ((\mathit{exmatch}\,x) : a) \end{array}$$

where x is non-empty.

Since $\mathit{init}\,[a] = \mathit{tail}\,[a] = [\,]$ and $\mathit{head}\,[a] = \mathit{last}\,[a] = a$, and noting that $[a] = \mathit{exmatch}\,[a]$, we come to the following matching expressions:

$$\begin{array}{rcl} \mathit{exmatch}\,(a : x) & = & \texttt{if}\ \mathit{match}\,(\mathit{last}\,[a], \mathit{head}\,(\mathit{exmatch}\,x)) \\ & & \texttt{then}\ \mathit{exmatch}\,(\mathit{init}\,[a] +\!\!+\ \mathit{tail}\,(\mathit{exmatch}\,x))\ \texttt{else}\ ([a] +\!\!+\ (\mathit{exmatch}\,x)) \\ \mathit{exmatch}\,(x : a) & = & \texttt{if}\ \mathit{match}\,(\mathit{last}\,(\mathit{exmatch}\,x), \mathit{head}\,[a]) \\ & & \texttt{then}\ \mathit{exmatch}\,(\mathit{init}\,(\mathit{exmatch}\,x) +\!\!+\ \mathit{tail}\,[a])\ \texttt{else}\ ((\mathit{exmatch}\,x) +\!\!+\ [a]) \end{array}$$

and from them to the conclusion that $\mathit{exmatch} = \mathit{hom}\,(f, \oplus)$, where $f\ a = [a]$ and the provably associative operation $\oplus$ is as follows:

$$u \oplus v \quad = \quad \texttt{if}\ \mathit{match}\,(\mathit{last}\,u, \mathit{head}\,v)\ \texttt{then}\ \mathit{exmatch}\,(\mathit{init}\,u +\!\!+\ \mathit{tail}\,v)\ \texttt{else}\ (u +\!\!+\ v)$$

Expression of $\oplus$ can be simplified:

$$\begin{array}{ll} & \mathit{exmatch}\,(\mathit{init}\,u +\!\!+\ \mathit{tail}\,v) \\ = & \quad \{\ \mathit{exmatch} = \mathit{hom}\,(f, \oplus)\ \} \\ & \mathit{exmatch}\,(\mathit{init}\,u) \oplus \mathit{exmatch}\,(\mathit{tail}\,v) \\ = & \quad \{\ u \text{ and } v \text{ are results of } \mathit{exmatch}\ \} \\ & \mathit{init}\,u \oplus \mathit{tail}\,v \end{array}$$

yielding the following:

$$u \oplus v \quad = \quad \texttt{if}\ \mathit{match}\,(\mathit{last}\,u, \mathit{head}\,v)\ \texttt{then}\ (\mathit{init}\,u \oplus \mathit{tail}\,v)\ \texttt{else}\ (u +\!\!+\ v)$$

This operation, applied to a pair (u, v), does the same as the composition:

$$(\mathit{combine} \circ \mathit{dropmatches})\,(\mathit{reverse}\,u, v)$$

which was defined in [7]. Function *dropmatches* takes a pair of lists with the reversed first argument and yields a pair of lists which are then concatenated by *combine* so that the first list is reversed again. To allow the comparison of our results with those by Cole, let us take his definition of *dropmatches*:

$$\begin{array}{rcl} \mathit{dropmatches}\,(u, v) & = & \texttt{if}\ \mathit{match}\,(\mathit{head}\,u, \mathit{head}\,v) \\ & & \texttt{then}\ \mathit{dropmatches}\,(\mathit{tail}\,u, \mathit{tail}\,v)\ \texttt{else}\ (u, v) \end{array}$$

which, not surprisingly, is also recursive as our $\oplus$ is. Since a recursive function may be expensive to compute, let us try to express *dropmatches* as a homomorphism. First we massage it from a function on pairs of lists to a function on lists of pairs; following [7], we do not change the name of the function. We assume that *dropmatches* accepts a list of pairs, where the first element is from u and the second is from v. If u and v are of different lengths, some special symbol is used to supplement the smaller list. Function *dropmatches* yields a list of pairs, with leading matching pairs removed.

The important observation for expressing *dropmatches* as leftwards and rightwards homomorphic function is that we need an auxiliary function. Indeed, if it happens in $dropmatches\,((a,b):x)$ that a and b are not matched then the result is just the input list, i.e. we need to know not only $dropmatches\,x$ but also x itself. The necessary auxiliary function is *original* which is the identity on lists of pairs. Now, the tuple (*dropmatches*, *original*) is complete and is defined in the following matching format:

$$\begin{aligned} dropmatches\,((a,b):x) &= (dropmatches\,[(a,b)]) \mathbin{+\!\!+} (\,\mathtt{if}\ (dropmatches\,[(a,b)] = [])\\ &\quad\ \mathtt{then}\ dropmatches\,x\ \mathtt{else}\ original\,x\,)\\ dropmatches\,(x:(a,b)) &= (dropmatches\,x) \mathbin{+\!\!+} (\,\mathtt{if}\ (dropmatches\,x = [])\\ &\quad\ \mathtt{then}\ dropmatches\,([(a,b)])\ \mathtt{else}\ original\,([(a,b)])\,) \end{aligned}$$

$$\begin{aligned} original\,((a,b):x) &= original\,[(a,b)] \mathbin{+\!\!+} original\,x\\ original\,(x:(a,b)) &= original\,x \mathbin{+\!\!+} original\,[(a,b)] \end{aligned}$$

Therefore, $(dropmatches, original) = hom\,(g, \otimes)$, where

$$\begin{aligned} g\,[(a,b)] &= (\,\mathtt{if}\ match\,(a,b)\ \mathtt{then}\ []\ \mathtt{else}\ [(a,b)]\ ,\ [(a,b)])\\ (u,u') \otimes (v,v') &= (u \mathbin{+\!\!+} (\mathtt{if}\ u = []\ \mathtt{then}\ v\ \mathtt{else}\ v')\ ,\ u' \mathbin{+\!\!+} v') \end{aligned}$$

The parallel implementation of *dropmatches* is then expressed by:

$$dropmatches = fst \circ red\,(\otimes) \circ\ map\,(g)$$

i.e. it is of logarithmic time complexity. Since the computation of *dropmatches* takes place in every application of $\oplus$ in the homomorphism *exmatch*, the total complexity of the nested homomorphic algorithm for bracket matching is $O(\log^2 n)$.

5 Conclusions and Future Work

Our methodology for constructing a parallelizable homomorphic representation for functions over lists can be summarized as follows:

1. Define the function in a sequential way using *cons*- and *snoc*-lists, introducing auxiliary functions if necessary and writing down definitions for them.

2. In case of auxiliary functions, i.e. tuples of functions, construct the union of tuples and make the new tuple *cons*- and *snoc*-complete, possibly by introducing additional auxiliary functions.

3. Make the definitions of the function (and of every auxiliary function if any) leftwards- and rightwards-homomorphic so that they match; prove facts necessary for that (including the associativity of the operation).

If the third step is successful then our function is a homomorphism or, in the case of tuple, an almost-homomorphism. The matching definitions provide explicitly the format of the homomorphism and thus yield immediately a parallel algorithm which has a standard form: *map* followed by a reduction. This format is efficiently implementable on a wide variety of parallel systems, but is not always the best possible [11].

Our approach can be compared with the method advocated in [12] since we consider the same examples. On simple examples like *length*, we actually do the same: take the *cons*-representation and try to guess from it the form of the operation in case of $+\!\!\!+$ -lists, after which its associativity has to be proved. The difference becomes visible on more complicated examples: in [12], the *scan* function requires rather involved calculations and intuition to come to the expression for $\oplus$; here the advantages of the systematic "*cons*+*snoc*" approach are evident. For almost-homomorphisms, the method of [12] does not work at all.

In [13], the third homomorphism theorem is applied for two language recognition problems: the existence of a leftwards and rightwards algorithms is used as an evidence that a homomorphic algorithm exists. The algorithm itself, in particular its associative combining operation, has to be developed in some way. In comparison, our approach systematically derives the desired operation.

Our methodology facilitates a systematic development of parallel solutions for almost-homomorphisms which were derived in [6, 7] using intuition about the necessary auxiliary functions. Similar derivation for the maximum segment sum problem was apparently first done by Smith [14], applying a strategy of divide-and-conquer. His approach consists of fixing the decomposition operator (e.g. an inverse of $+\!\!\!+$) and constructing the composing operator (analog of our $\oplus$) by exploiting the suitable mathematical properties, specific to the problem. Our approach is more systematic and less prone to errors. In our future work we are going to study the two-dimensional *mss*-problem where, according to [14], the homomorphic tuple consists of eleven functions and thus corresponding manipulations must be cumbersome (Smith does not present them).

Another question requiring further study is finding a condition which, together with leftwards- and rightwards-homomorphy, guarantees that the function is a homomorphism and which is easier to prove than associativity.

Methodologically, an important feature of our approach is that it is based on sequential thinking: the developer is required to syntactically match two sequential functional programs, on *cons*- and on *snoc*-lists. Considerations involving data and control dependences, which are usual in parallelization techniques, are completely avoided.

6 Acknowledgment

I am grateful to Murray Cole, Alfons Geser, Jeremy Gibbons and Lambert Meertens for discussing this manuscript.

References

[1] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, NATO ASO Series F: Computer and Systems Sciences. Vol. 55, pages 151–216. Springer Verlag, 1988.

[2] D. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

[3] G. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[4] M. J. Quinn. *Parallel Computing.* McGraw-Hill, Inc., 1994.

[5] J. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, 1994.

[6] M. Cole. Parallel programming, list homomorphisms and the maximum sum problem. Technical Report CSR–25–93, University of Edinburgh, May 1993.

[7] M. Cole. List homomorphic parallel algorithms for bracket matching. Technical report, University of Edinburgh, 1993.

[8] J. Gibbons. The third homomorphism theorem. Technical report, Univ. of Auckland, 1994.

[9] J. Gibbons. The third homomorphism theorem. *J. Fun. Programming.* To appear.

[10] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[11] D. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.

[12] Z. Grant-Duff and P. Harrison. Parallelism via homomorphisms. *Parallel Processing Letters.* To appear.

[13] D. Barnard, J. Schmeiser, and D. Skillicorn. Deriving associative operators for language recognition. *Bulletin of EATCS*, 43:131–139, 1991.

[14] D. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, (8):213–229, 1987.