FUNCTIONAL PEARLS

The Third Homomorphism Theorem

Jeremy Gibbons
Department of Computer Science
University of Auckland
Private Bag 92019, Auckland, New Zealand.
Email: jeremy@cs.auckland.ac.nz

Abstract

The *Third Homomorphism Theorem* is a folk theorem of the constructive algorithmics community. It states that a function on lists that can be computed both from left to right and from right to left is necessarily a *list homomorphism*—it can be computed according to *any* parenthesization of the list.

We formalize and prove the theorem, and use it to improve an $O(n^2)$ sorting algorithm to $O(n \log n)$.

1 Introduction

List homomorphisms are those functions on finite lists that *promote* through list concatenation—that is, functions h for which there exists a binary operator $\odot$ such that, for all finite lists x and y,

$$h\ (x \mathbin{+\!\!+} y) \quad = \quad h\ x \odot h\ y$$

where '$+\!\!+$' denotes list concatenation. Such functions are ubiquitous in functional programming. Some examples of list homomorphisms are:

- the identity function id;
- the map function $map\ f$, which applies a given function f to every element of a list;
- the function $concat$, which concatenates a list of lists into a single long list;
- the function $head$, which returns the first element of a list;
- the function $length$, which returns the length of a list;
- the functions sum, min and all, which return the sum, the smallest and the boolean conjunction of the elements of a list, respectively.

However, there are also many useful list functions that are not list homomorphisms. One example is the function lsp, which returns the longest sorted prefix of a list. Knowing $lsp\ x$ and $lsp\ y$ is not enough to allow computation of $lsp\ (x \mathbin{+\!\!+} y)$. This function is a typical example of a *leftwards* function—one which can be computed from right to left. Dually, the *rightwards* functions can be computed from left to right.

One obvious relationship between homomorphisms and leftwards and rightwards functions is known as the Specialization Theorem (Bird, 1987): all homomorphisms are also leftwards and rightwards functions. In the Constructive Algorithmics community, this has become known as the 'Second Homomorphism Theorem'. (The 'First Homomorphism Theorem' states that a homomorphism can be factored into the composition of reduction—a homomorphism whose value on a singleton list is the sole element of that list—with a map, and conversely that any such composition is a homomorphism.)

The subject of this paper is another relationship between homomorphisms and leftwards and rightwards functions. This relationship is much less obvious, but is equally useful. It is the converse of the Specialization Theorem, and states that any function that is both leftwards and rightwards is also a homomorphism. This theorem is fairly well-known in the Constructive Algorithmics community, bearing the name 'The Third Homomorphism Theorem'. However, it has somewhat the status of a 'folk theorem' (Harel, 1980). It was conjectured by Richard Bird and proved by Lambert Meertens during a train journey across the Netherlands in 1989 (Meertens, 1995); the theorem has been published only in non-archival sources (Barnard *et al.*, 1991; Gibbons, 1993), and we feel that it deserves wider recognition.

In this paper we formalize and prove this theorem, and use it to derive 'mergesort' from 'insertsort'. The remainder of this paper is structured as follows. In Section 2, we introduce the necessary notation. In Section 3, we state the First and Second Homomorphism Theorems, for completeness' sake. Section 4 contains the main result of the paper, the Third Homomorphism Theorem. In Section 5, we use the theorem to derive mergesort from insertsort.

An earlier version of this paper appeared as (Gibbons, 1994).

2 Notation

In this section, we introduce the notation used in the rest of the paper.

Functions: Function application is denoted by juxtaposition, is tightest binding, and associates to the left. Function composition is written '$\circ$'.

Lists: For the purposes of this paper, lists are finite sequences of elements, all of the same type. A list is either empty, a singleton, or the concatenation of two other lists. We write '$[\,]$' for the empty list, '$[a]$' for the singleton list with element a (and '$[\cdot]$' for the function taking a to $[a]$), and '$x \mathbin{+\!\!+} y$' for the concatenation of x and y. Concatenation is associative, and $[\,]$ is its unit. For example, the term $[a_1] \mathbin{+\!\!+} [a_2] \mathbin{+\!\!+} [a_3]$ denotes a list with three elements, often written in the abbreviated form $[a_1, a_2, a_3]$. We also write '$a : x$' for $[a] \mathbin{+\!\!+} x$; the operator '$:$' associates to the right.

Homomorphisms: For a binary operator $\odot$, the list function h is $\odot$-*homomorphic* iff, for all lists x and y,

$$h\ (x \mathbin{+\!\!+} y) \quad = \quad h\ x\ \odot\ h\ y$$

For example, the functions *length* and *sum* are both $+$-homomorphic, since

$$\begin{array}{rcl} sum\ (x \mathbin{+\!\!+} y) & = & sum\ x\ +\ sum\ y \\ length\ (x \mathbin{+\!\!+} y) & = & length\ x\ +\ length\ y \end{array}$$

Note that $\odot$ is necessarily associative on the range of h, because $\mathbin{+\!\!+}$ is associative. Moreover, $h\ [\,]$ is necessarily the unit of $\odot$ on the range of h (if it exists), because $[\,]$ is the unit of $\mathbin{+\!\!+}$. If $\odot$ has no unit, then $h\ [\,]$ is not defined. For example, *head* is $\ll$-homomorphic where $a \ll b = a$, but because $\ll$ has no unit, *head* $[\,]$ is undefined.

For associative operator $\odot$ with unit e, we write '$hom\ (\odot)\ f\ e$' for the (unique) $\odot$-homomorphic function h for which $h \circ [\cdot] = f$. For example,

$$\begin{array}{rcl} sum & = & hom\ (+)\ id\ 0 \\ length & = & length = hom\ (+)\ one\ 0 \end{array}$$

where $one\ a = 1$ for all a.

Leftwards and rightwards functions: The list function h is $\oplus$-*leftwards* for binary operator $\oplus$ iff, for all elements a and lists y,

$$h\ ([a] \mathbin{+\!\!+} y) \quad = \quad a \oplus h\ y$$

Here, $\oplus$ need not be associative. The (unique) $\oplus$-leftwards function h for which $h\ [\,] = e$ is written '$foldr\ (\oplus)\ e$'. For example, the function *lsp* referred to earlier is $\oplus$-leftwards where

$$\begin{array}{rcl} a \oplus [\,] & = & [a] \\ a \oplus (b : x) & = & \left\{ \begin{array}{ll} a : b : x, & \text{if } a \leq b \\ [a], & \text{otherwise} \end{array} \right. \end{array}$$

Since $lsp\ [\,] = [\,]$, we have $lsp = foldr\ (\oplus)\ [\,]$ with the above definition of $\oplus$.

Expanding the definition of a leftwards function reveals the significance of the name. For example:

$$foldr\ (\oplus)\ e\ [a_1, a_2, a_3] \quad = \quad a_1 \oplus (a_2 \oplus (a_3 \oplus e))$$

and so its computation proceeds from right to left. In general:

$$foldr\ (\oplus)\ e\ (x \mathbin{+\!\!+} y) \quad = \quad foldr\ (\oplus)\ (foldr\ (\oplus)\ e\ y)\ x \tag{1}$$

(The name '*foldr*' is unfortunate for a right-to-left computation, but it is well established.)

Symmetrically, the list function h is $\otimes$-*rightwards* for binary operator $\otimes$ iff, for all lists x and elements a,

$$h\ (x \mathbin{+\!\!+} [a]) \quad = \quad h\ x\ \otimes\ a$$

Again, the operator $\otimes$ need not be associative. We write '$foldl\ (\otimes)\ e$' for the unique $\otimes$-rightwards function h for which $h\ [\,] = e$. Expanding the definition reveals a left-to-right pattern of computation. For example:

$$foldl\ (\otimes)\ e\ [a_1, a_2, a_3] \quad = \quad ((e \otimes a_1) \otimes a_2) \otimes a_3$$

and in general:

$$foldl\ (\otimes)\ e\ (x +\!\!+\, y) \quad = \quad foldl\ (\otimes)\ (foldl\ (\otimes)\ e\ x)\ y \tag{2}$$

3 The First and Second Homomorphism Theorems

For the sake of completeness, we state without proof the First and Second Homomorphism Theorems.

Definition 3.1
A function of the form $hom\ (\odot)\ id\ e$ for some $\odot$ is called a *reduction*.

Definition 3.2
For given f, the function $hom\ (+\!\!+)\ ([\cdot] \circ f)\ [\,]$ is written '$map\ f$' and called a *map*.

Theorem 3.3 (First Homomorphism Theorem)
Every homomorphism can be written as the composition of a reduction and a map:

$$hom\ (\odot)\ f\ e \quad = \quad hom\ (\odot)\ id\ e\ \circ\ map\ f$$

Conversely, every such composition is a homomorphism.

Theorem 3.4 (Second Homomorphism Theorem, or Specialization Theorem)
Every homomorphism is both a leftwards and a rightwards function. That is, if $\odot$ is associative, then

$$\begin{array}{rcll} hom\ (\odot)\ f\ e & = & foldr\ (\oplus)\ e & \quad \text{where } a \oplus s = f\ a \odot s \\ & = & foldl\ (\otimes)\ e & \quad \text{where } r \otimes a = r \odot f\ a \end{array}$$

4 The Third Homomorphism Theorem

This section contains the main result of the paper, the statement and proof of the Third Homomorphism Theorem.

Theorem 4.1 (Third Homomorphism Theorem)
If h is leftwards and rightwards, then h is a homomorphism.

In fact, we will show that h is $\odot$-homomorphic where

$$t \odot u \quad = \quad h\ (g\ t +\!\!+\ g\ u)$$

for some g such that $h \circ g \circ h = h$. Such a g exists, as the following lemma shows.

Lemma 4.2
For every computable total function h with enumerable domain, there is a computable (but possibly partial) function g such that $h \circ g \circ h = h$.

Proof
Here is one suitable definition of g. To compute $g\ t$ for some t, simply enumerate the domain of h and return the first x such that $h\ x = t$. If t is in the range of h, then this process terminates. $\square$

The proof of the Third Homomorphism Theorem relies on the following lemma:

Lemma 4.3
The list function h is a homomorphism iff the implication

$$h\ v = h\ x \wedge h\ w = h\ y \quad \Rightarrow \quad h\ (v +\!\!+\ w) = h\ (x +\!\!+\ y) \tag{3}$$

holds for all lists v, w, x, y.

(We note in passing an interesting corollary to Lemma 4.3: any injective function is homomorphic.)

Proof
The 'only if' is obvious: if h is a homomorphism, then there is a $\oplus$ such that $h\ (x +\!\!+\ y) = h\ x \oplus h\ y$ for all lists x and y. Now consider the 'if' part.

Assume that h satisfies (3). Choose a g such that $h \circ g \circ h = h$, and define operator $\odot$ by the equation

$$t \odot u \quad = \quad h\ (g\ t +\!\!+\ g\ u)$$

(as in the statement of the Third Homomorphism Theorem). We show that h is $\odot$-homomorphic.

Because of the way we chose g, $h\ x = h\ (g\ (h\ x))$ and $h\ y = h\ (g\ (h\ y))$, and so, by (3) (with $v = g\ (h\ x)$ and $w = g\ (h\ y)$), we have

$$\begin{aligned} h\ (x +\!\!+\ y) \quad &= \quad h\ (g\ (h\ x) +\!\!+\ g\ (h\ y)) \\ &= \quad h\ x \odot h\ y \end{aligned}$$

$\square$

We now prove the Third Homomorphism Theorem.

Proof
We show that, if h is leftwards and rightwards, then h satisfies (3).

Suppose that $h = \mathit{foldr}\ (\oplus)\ e = \mathit{foldl}\ (\otimes)\ e$, and that $h\ v = h\ x$ and $h\ w = h\ y$. Then:

$$
\begin{array}{ll}
 & h\ (v +\!\!+\ w) \\
= & \quad \{\ \text{treating } h \text{ as a leftwards function}\ \} \\
 & \mathit{foldr}\ (\oplus)\ e\ (v +\!\!+\ w) \\
= & \quad \{\ (1)\ \} \\
 & \mathit{foldr}\ (\oplus)\ (\mathit{foldr}\ (\oplus)\ e\ w)\ v \\
= & \quad \{\ \text{since } h\ w = h\ y\ \} \\
 & \mathit{foldr}\ (\oplus)\ (\mathit{foldr}\ (\oplus)\ e\ y)\ v \\
= & \quad \{\ (1)\ \} \\
 & \mathit{foldr}\ (\oplus)\ e\ (v +\!\!+\ y) \\
= & \quad \{\ \text{treating } h \text{ as a leftwards function}\ \} \\
 & h\ (v +\!\!+\ y) \\
= & \quad \{\ \text{symmetrically, treating } h \text{ as a rightwards function}\ \} \\
 & h\ (x +\!\!+\ y)
\end{array}
$$

Hence, by Lemma 4.3, h is a homomorphism. $\square$

5 Application: sorting

We now use the Third Homomorphism Theorem to derive the $O(n \log n)$ sorting algorithm 'mergesort' from the $O(n^2)$ 'insertsort'. (In fact, the Third Homomorphism Theorem yields only an inefficient homomorphic sorting algorithm; we have to do a little more work to derive mergesort itself.)

The function *sort*, which sorts a list, is leftwards, since it can be written

$$\mathit{sort} \quad = \quad \mathit{foldr}\ \mathit{ins}\ [\,]$$

where

$$
\begin{array}{rcl}
\mathit{ins}\ a\ [\,] & = & [a] \\
\mathit{ins}\ a\ (b : x) & = & \begin{cases} a : b : x, & \text{if } a \leq b \\ b : (\mathit{ins}\ a\ x), & \text{otherwise} \end{cases}
\end{array}
$$

This is just traditional 'insertsort', and takes $O(n^2)$ time to sort n elements.

The same function is also rightwards, since it can be written as a 'backwards insertsort':

$$\mathit{sort} \quad = \quad \mathit{foldl}\ \mathit{ins}'\ [\,] \tag{4}$$

where

$$\mathit{ins}'\ x\ a \quad = \quad \mathit{ins}\ a\ x$$

The Third Homomorphism Theorem concludes that $sort$ is therefore homomorphic. The homomorphism constructed by the proof is $hom\ (\odot)\ [\cdot]\ [\,]$ where

$$u \odot v \quad = \quad sort\ (unsort\ u \mathbin{+\!\!+} unsort\ v)$$

for some function $unsort$ such that $sort \circ unsort \circ sort = sort$, that is, which permutes the elements of a list.

We pick $unsort = id$ for simplicity, giving

$$u \odot v \quad = \quad sort\ (u \mathbin{+\!\!+} v) \tag{5}$$

This gives us a homomorphic method of sorting, but clearly it is very inefficient. To sort $x \mathbin{+\!\!+} y$, we sort x and y (yielding u and v), concatenate u and v, and then (presumably using some other sorting method, such as insertsort) sort the result $u \mathbin{+\!\!+} v$. However, we can improve this algorithm, by capitalizing on the fact that—in the context of evaluating $hom\ (\odot)\ [\cdot]\ [\,]$—the arguments u and v to $\odot$ will be sorted. This improvement takes us directly to the traditional 'mergesort' algorithm, which is $O(n \log n)$.

Suppose first that u is sorted, that is, that $u = sort\ u$. Then

$$\begin{array}{ll}
& u \odot v \\
= & \qquad \{\ (5)\ \} \\
& sort\ (u \mathbin{+\!\!+} v) \\
= & \qquad \{\ (4)\ \} \\
& foldl\ ins'\ [\,]\ (u \mathbin{+\!\!+} v) \\
= & \qquad \{\ (2)\ \} \\
& foldl\ ins'\ (foldl\ ins'\ [\,]\ u)\ v \\
= & \qquad \{\ (4)\ \} \\
& foldl\ ins'\ (sort\ u)\ v \\
= & \qquad \{\ u \text{ is sorted }\} \\
& foldl\ ins'\ u\ v \\
= & \qquad \{\ \text{let } merge = foldl\ ins'\ \} \\
& merge\ u\ v
\end{array}$$

We have picked a suggestive name in the last step, but it is justified by the observation that

$$\begin{array}{rcl}
merge\ u\ [\,] & = & foldl\ ins'\ u\ [\,] \\
& = & u
\end{array}$$

and

$$\begin{array}{rcl}
merge\ u\ (b : v) & = & foldl\ ins'\ u\ (b : v) \\
& = & foldl\ ins'\ (ins'\ u\ b)\ v \\
& = & merge\ (ins'\ u\ b)\ v
\end{array}$$

This is a straightforward method of merging two lists, the first one already sorted, to produce a sorted list. Note, however, that it takes quadratic time, and so computing *hom merge* $[\cdot]$ $[\,]$ also takes quadratic time†. We can make a further improvement by assuming that the second argument to *merge* is also sorted.

We use the following lemma, which is easily proved by induction. We write '$a \leq v$' to denote that $a \leq b$ for every element b of list v.

Lemma 5.1
If $a \leq x$ and $a \leq y$, then

$$\mathit{foldl}\ \mathit{ins}'\ (a : x)\ y \quad = \quad a : \mathit{foldl}\ \mathit{ins}'\ x\ y$$

Suppose that v is sorted. Then

$$\begin{array}{ll} & \mathit{merge}\ [\,]\ v \\ = & \qquad \{ \text{ definition of } \mathit{merge} \ \} \\ & \mathit{foldl}\ \mathit{ins}'\ [\,]\ v \\ = & \qquad \{ \ (4) \ \} \\ & \mathit{sort}\ v \\ = & \qquad \{ \ v \text{ is sorted } \} \\ & v \end{array}$$

Now suppose that $a : u$ and $b : v$ are sorted. Then

$$\begin{array}{ll} & \mathit{merge}\ (a : u)\ (b : v) \\ = & \qquad \{ \text{ definition of } \mathit{merge} \ \} \\ & \mathit{foldl}\ \mathit{ins}'\ (a : u)\ (b : v) \\ = & \qquad \{ \text{ defining property of } \mathit{foldl} \ \} \\ & \mathit{foldl}\ \mathit{ins}'\ (\mathit{ins}'\ (a : u)\ b)\ v \end{array}$$

We now consider the cases $a < b$ and $a \geq b$ separately.

Case $a < b$**:** Since $a : u$ and $b : v$ are sorted, we have $a \leq u$ and $a \leq v$; hence $a \leq \mathit{ins}'\ u\ b$ also. Then

$$\begin{array}{ll} & \mathit{foldl}\ \mathit{ins}'\ (\mathit{ins}'\ (a : u)\ b)\ v \\ = & \qquad \{ \ \mathit{ins}';\ a < b \ \} \\ & \mathit{foldl}\ \mathit{ins}'\ (a : \mathit{ins}'\ u\ b)\ v \\ = & \qquad \{ \text{ Lemma 5.1 } \} \\ & a : \mathit{foldl}\ \mathit{ins}'\ (\mathit{ins}'\ u\ b)\ v \\ = & \qquad \{ \text{ defining property of } \mathit{foldl} \ \} \\ & a : \mathit{foldl}\ \mathit{ins}'\ u\ (b : v) \\ = & \qquad \{ \text{ definition of } \mathit{merge} \ \} \\ & a : \mathit{merge}\ u\ (b : v) \end{array}$$

† because $\sum_{i=0}^{\log n} 2^i (\frac{n}{2^i})^2 \ \simeq\ 2n^2$

Case $a \geq b$: Since $a : u$ and $b : v$ are sorted, we have $b \leq a : u$ and $b \leq v$. Then

$$\begin{array}{ll} & foldl\ ins'\ (ins'\ (a : u)\ b)\ v \\ = & \qquad \{\ ins';\ a \geq b\ \} \\ & foldl\ ins'\ (b : a : u)\ v \\ = & \qquad \{\ \text{Lemma 5.1}\ \} \\ & b : foldl\ ins'\ (a : u)\ v \\ = & \qquad \{\ \text{definition of } merge\ \} \\ & b : merge\ (a : u)\ v \end{array}$$

We have just derived the following characterization of *merge*, when both of its arguments are sorted:

$$\begin{array}{rcl} merge\ [\,]\ v & = & v \\ merge\ u\ [\,] & = & u \\ merge\ (a : u)\ (b : v) & = & \left\{ \begin{array}{ll} a : merge\ u\ (b : v), & \text{if } a < b \\ b : merge\ (a : u)\ v, & \text{otherwise} \end{array} \right. \end{array}$$

which is the standard way of merging two sorted lists (except that the comparison is usually '$a \leq b$' rather than '$a < b$'). This version of *merge* takes linear time, and yields the well-known mergesort algorithm, which is $O(n \log n)$ when the list is decomposed in a balanced fashion. Green and Barstow (1978) describe a similar derivation of *merge* and mergesort.

6 Conclusion

To summarize, we have presented and proved Bird and Meertens' Third Homomorphism Theorem, stating that any function on lists that can be computed both from left to right and from right to left is necessarily a list homomorphism. We gave an example of its use—deriving 'mergesort' from 'insertsort'—illustrating that the theorem does not usually give an efficient characterization of the homomorphism; further development must be done to produce this.

Further applications of the Third Homomorphism Theorem are given by Barnard *et al.* (1991), Gibbons (1993), and Gorlatch (1995).

Acknowledgements

Murray Cole, Rod Downey, Sergei Gorlatch, Lindsay Groves, Lambert Meertens, the participants of the *Computing—the Australian Theory Seminar* in Sydney in December 1994, and especially Richard Bird have all made comments to improve the presentation and content of this paper. Thanks are also due to Sue Gibbons, for her energetic red pen. The research reported here has been partially supported by University of Auckland Research Committee grant number 3414013.

References

Barnard, D. T., Schmeiser, J. P. and Skillicorn, D. B. 1991. Deriving associative operators for language recognition. *Bulletin of the European Association for Theoretical Computer Science*, 43: pp. 131–139.

Bird, R. S. 1987. An introduction to the theory of lists. In M. Broy (editor), *Logic of Programming and Calculi of Discrete Design*, pp. 3–42. Springer-Verlag. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.

Gibbons, J. 1993. Computing downwards accumulations on trees quickly. In G. Gupta, G. Mohay, and R. Topor (editors), *16th Australian Computer Science Conference*, pp. 685–691, Brisbane. Available by anonymous `ftp` as `out/jeremy/papers/quickly.ps.Z` on `ftp.cs.auckland.ac.nz`.

Gibbons, J. 1994. The Third Homomorphism Theorem. In C. Barry Jay (editor), *Computing: The Australian Theory Seminar*. University of Technology, Sydney.

Gorlatch, S. 1995. Constructing List Homomorphisms. Technical Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau.

Green, C. and Barstow, B. 1978. On program synthesis knowledge. *Artificial Intelligence*, 10: pp. 241–279.

Harel, D. 1980. On folk theorems. *Communications of the ACM*, 23(7): pp. 379–389.

Meertens, L. G. L. T. 1995. Personal communication.