# Parallelizing Functional Programs by Term Rewriting *

Alfons Geser[1] and Sergei Gorlatch[2]

[1]Wilhelm-Schickard-Institut für Informatik, University, Sand 13, D-72076 Tübingen

[2]University of Passau, D–94030 Passau, Germany

**Abstract**

List homomorphisms are functions that can be computed in parallel using the divide-and-conquer paradigm. We study the problem of finding a homomorphic representation of a given function, based on the Bird-Meertens theory of lists. A previous work proved that to each pair of *leftward* and *rightward* sequential representations of a function, based on `cons`- and `snoc`-lists, respectively, there is also a representation as a homomorphism. Our contribution is a *mechanizable method* to extract the homomorphism representation from a pair of sequential representations. The method is decomposed to a generalization problem and an inductive claim, both solvable by term rewriting techniques. To solve the former we present a sound generalization procedure which yields the required representation, and terminates under reasonable assumptions. We illustrate the method and the procedure by the parallelization of the *scan*-function (parallel prefix). The inductive claim is provable automatically.

**Keywords:** Program derivation; Term rewriting; Bird-Meertens formalism; Functional programming; Divide-and-Conquer; Generalization (Anti-unification); Parallel prefix.

---

*The first author was partially supported by grant `Ku 996/3-1` of the DFG within the Schwerpunkt Deduktion at the University of Passau. The second author was partially supported by the DFG project RecuR2 and by the DAAD exchange programs ARC and PROCOPE.

# 1 Introduction

This paper addresses the problem of deriving correct parallel programs from functional specifications. It builds upon a rich body of research done in the framework of the Bird-Meertens formalism (BMF) [1, 2]. BMF comprises a collection of higher-order functions on arbitrary composite types (trees, arrays, etc.); expressions of the formalism are used for program specification. The program derivation process starts from an intuitively clear and correct specification and proceeds to the target program by applying semantically sound transformation rules of the formalism.

Parallelism is tackled in BMF via the notion of *homomorphism* which can be viewed as a specific form of the well-known divide-and-conquer (DC) paradigm. Classifications of the DC forms suitable for static parallelization are proposed in [3, 4]. The class of homomorphic functions exemplify data-parallelism and enjoy a common way of parallelization, and thus can be viewed as an algorithmic skeleton [5]. To use homomorphisms in the practical parallel programming, two tasks must be solved:

- *Extraction*: for a given function which is to be computed, find its representation as a homomorphism or adjust/customize it to a homomorphic form.

- *Implementation*: for different classes of homomorphisms, find an efficient way of implementing them on parallel architectures.

For both tasks, we aim at a systematic approach which would lead to a practically relevant parallel programming methodology. Recent results are presented in [3, 6, 7].

A systematic extraction method, proposed in [7], proceeds by generalizing two sequential representations of the function: on the `cons` and `snoc` lists. This so-called CS-method (CS for "Cons and Snoc") has been proved to be powerful enough for a broad class of *almost-homomorphisms* which include the classical problems like maximum segment sum and parsing the multi-bracket languages as examples.

This paper makes a further step in solving the extraction problem. Our contributions are, first, a precise formulation of the CS-approach to the extraction problem via term generalization and, second, a generalization procedure within the Bird-Meertens theory of lists, which could be used in the CS-method. We propose a new, mechanizable algorithm of generalization based on term rewriting, with the desirable properties of soundness, reliability and termination under reasonable assumptions.

The paper is structured as follows:

- Section 2 introduces the BMF notation and the notion of homomorphism.
- Section 3 contains a review of the problem and the known results.
- Section 4 introduces the notion of generalization, which plays the key rôle in our approach, and presents the related work.
- Section 5 formulates our method of homomorphism extraction, the CS-method, in the generalization framework.
- Section 6: we translate the theory of lists into the term rewriting framework.
- Section 7 derives a generalization calculus based on rewriting.
- Section 8 shows how to apply the generalization calculus to CS-generalization and offers a reliability result.
- Section 9 shows how to derive a terminating, deterministic procedure, i.e. an algorithm, for generalization.
- Section 10 summarizes the results and outlines future work.

The presentation is illustrated by a running example – the *scan* function, also known as the parallel prefix. For this practically relevant function, we first demonstrate the non-triviality of the extraction problem, then we show how the CS-method works for it and, finally, we apply the proposed generalization algorithm which successfully extracts the homomorphic form of the parallel prefix.

As we address problems and methods from traditionally separated areas, we prefer to review related work on the spot: in Section 3 for homomorphisms, in Section 4 for generalization and in Section 6 for term rewriting.

## 2 BMF and Homomorphisms

We restrict ourselves to non-empty lists and use a small fragment of BMF. The BMF expressions are built using the traditional backward functional composition denoted by $\circ$, and two second-order functions (for brevity, definitions are informal):

| | |
|---|---|
| $map\, f$ | *map* of unary function $f$, i.e. $map\, f\, [x_1, \cdots, x_n] \;=\; [f x_1, \cdots, f x_n]$; |
| $red\,(\odot)$ | *reduce* over a binary associative operation $\odot$, $red\,(\odot)\,[x_1, \cdots, x_n] \;=\; x_1 \odot x_2 \odot \cdots \odot x_n$. |

**Definition 2.1** *A list function $h$ is a homomorphism iff there exists a binary associative combine operator $\circledast$, such that for all lists $x$ and $y$:*

$$h\,(x +\!\!+\, y) = h\,(x) \;\circledast\; h\,(y) \tag{1}$$

*where $+\!\!+$ is list concatenation; hence, the value of $h$ on a list depends in a particular way (using $\circledast$) on the values of $h$ on the pieces of the list.*

The computations of $h(x)$ and $h(y)$ in (1) are independent and can be carried out in parallel, which expresses the well-known divide-and-conquer paradigm.

Examples of homomorphisms are simple functions, such as *length* yielding 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) [8].

An important property of homomorphisms is Bird's First Homomorphism Theorem [1]:

**Theorem 1 (Bird [1])** *Function $h$ is a homomorphism iff:*

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

*where $\circledast$ is from (1), and $f\,(a) \;=\; h\,([a])$.*

The theorem provides a standard parallelization pattern for all homomorphisms as a composition of two stages. The first stage in (2), *map*, is fully parallel; the parallelization of the second, reduction stage has been studied in [6]. Therefore, loosely speaking, a function is a homomorphism iff it is well parallelizable.

**Example 1 (Scan as a Homomorphism)** *Our illustrating example in the paper is the scan-function which, for associative $\odot$ and a list, computes "prefix sums". On a list of 4 elements e.g. it acts as follows:*

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

*Function scan has a surprisingly wide area of applications, including evaluation of polynomials, searching, etc. [8]; its parallelization has been extensively studied and meanwhile belongs to the folklore of parallel computing.*

*Function scan is a homomorphism with combine operator* $\circledast$*:*

$$\begin{aligned} scan\,(\odot)\,(x +\!\!+\, y) \;&=\; S_1 \circledast S_2 \;=\; S_1 +\!\!+\, (map\,((last\ S_1)\odot)\ S_2)\,, \\ \text{where } S_1 \;&=\; scan\,(\odot)\,x\,,\ S_2 \;=\; scan\,(\odot)\,y. \end{aligned} \tag{3}$$

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

# 3 Homomorphic vs. Sequential Forms

Whereas the desired parallel (homomorphic) representations use list concatenation, more traditional sequential functional programming is based on the constructors `cons` and `snoc`. We use the following notation:

- $\cdot\!:$ for `cons`, which attaches an element at the front of the list,
- $:\!\cdot$ for `snoc`, which attaches the element at the list's end.

Our goal is to use sequential representations of a given function to extract its parallel homomorphic representation.

**Definition 3.1** *List function h is called leftwards (lw) iff there exists a binary operation* $\oplus$ *such that* $h\,(a \cdot\!: y) = a \oplus h(y)$ *for all elements a and lists y. Dually, function h is rightwards (rw) iff, for some* $\otimes$, $h\,(y :\!\cdot\, a) = h(y) \otimes a$.

Note that $\oplus$ and $\otimes$ may be non-associative and include for instance conditional branching, so many functions are either *lw* or *rw* or both. For a given operation, we speak of $\oplus$-*lw* or $\otimes$-*rw* functions, respectively.

The following theorem combines the so-called second and third homomorphism theorems considered folk theorems in the BMF community. The former is simple to prove; the latter was proved by Meertens and presented systematically by Gibbons [9]; its importance for deriving parallel algorithms was first demonstrated in [10].

**Theorem 2** *A function on lists is a homomorphism iff it is both leftwards and rightwards.*

Unfortunately, as pointed out in [9, 10], the theorem does not provide a method to construct the homomorphic representation of a function from its leftwards and rightwards definitions. The extraction task can be thus formulated as finding the combine operation $\circledast$ from operations $\oplus$ and/or $\otimes$.

We start by imposing an additional restriction on the leftwards and rightwards functions.

**Definition 3.2** *Function $h$ is called left-homomorphic (lh) iff there exists $\circledast$ such that, for arbitrary list $x$ and element $a$, the following holds: $h\,(a \cdot\!\!: y) = h\,([a]) \circledast h\,(y)$. The definition of a right-homomorphic (rh) function is dual.*

Evidently, every *lh* (*rh*) function is also *lw* (*rw*, resp.), but not vice versa.

For a given function $f$, defined on elements, and operation $\oplus$, there is a unique $\oplus$-*lh* function $h$, such that $h\,([a]) = f\,(a)$ for arbitrary $a$; we denote this function by $lh\,(f, \oplus)$; similarly, notation $rh\,(f, \circledast)$ is introduced. Since operation $\circledast$ in (2) is necessarily associative on the range of $h$, a $\circledast$-homomorphic function is completely determined by its actions on singleton lists. So we may write $hom\,(f, \circledast)$ for the unique $\circledast$-homomorphic function $h$, such that $h\,([a]) = f\,(a)$, for all $a$.

**Theorem 3** *If $h = \mathrm{hom}\,(f, \circledast)$, then $h$ is both lh and rh, moreover $h = lh\,(f, \circledast) = rh\,(f, \circledast)$. Vice versa, if $h = lh\,(f, \circledast)$ or $h = rh\,(f, \circledast)$, where $\circledast$ is associative, then $h = \mathrm{hom}\,(f, \circledast)$.*

In [11], we prove a slightly stronger proposition.

Theorem 3 suggests a possible way to find a homomorphic representation: construct a `cons` definition of the function in the *lh* format (or, dually, find an *rh* representation on `snoc` lists) and prove that the combine operation is associative. Sometimes this simple method works, as the following example of function *length*, which yields the length of a list, demonstrates.

**Example 2 (Extraction for Length)** *Since $length\,([a]) = 1$, we have $f = one$, where $one\,(x) = 1$. The `cons`-definition of length is: $length\,(a \cdot\!\!: y) = 1 + length\,(y) = length\,([a]) + length\,(y)$, hence $length = lh\,(one, +)$. Since $+$ is associative, from Theorem 3 follows that $length = hom\,(one, +)$.*

Our next example demonstrates that the method with *lh*/*rh* representations does not always go that smoothly.

**Example 1 (Continued; extraction for Scan)** *Let us try the same simple method on the scan function introduced in Example 1. The sequential `cons` definition of scan is as follows:*

$$scan\,(\odot)\,(a \cdot\!\!: y) \;=\; a \;\cdot\!\!:\; (map\,(a\,\odot)\,(scan\,(\odot)\,y)) \tag{4}$$

*Representation (4) does not match the lh format because a is used where only scan [a] is allowed. Since scan [a] = [a], there are different possibilities to express a via scan [a], e.g., a = head (scan [a]) or a = last (scan [a]). Thus we immediately obtain four possible terms for ⊛; however, none of these terms defines an associative operation!*

*Let us try to use the right-homomorphy of the* `snoc` *definition:*

$$scan\,(\odot)\,(x :\cdot\, b) \;=\; (scan\,(\odot)\,x) \;:\cdot\; (last\,(scan\,(\odot)\,x) \odot b) \tag{5}$$

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

We believe to have shown that the problem to extract a homomorphism is nontrivial. For the *scan* function, it even led to errors in published parallel algorithms and required a formal correctness proof [12]. Now let us explain our plan to solve the problem.

# 4 Generalization

We intend to apply the framework of generalization to extract homomorphisms. First let us explain what generalization means.

A *substitution* is an assignment of variables to terms, which we will write as $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, or, more suggestively, in vector notation

$$\sigma = \left\{ \begin{matrix} x_1 \mapsto t_1 \\ \vdots \\ x_n \mapsto t_n \end{matrix} \right\}$$

For variables $z$ that are not mentioned in the enumeration, we assume $z \mapsto z$. For term $t$ and a substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, let $t.\sigma$ denote the result of simultaneously substituting in $t$ each variable $x_i$, $1 \leq i \leq n$, by the term $t_i$.

Given an equational theory, $E$, i.e. a set of formal equations $l = r$, the induced *conversion relation*, $\leftrightarrow_E^*$, is the smallest congruence relation that contains all pairs of $E$. The conversion relation is also called the *semantic equality* in view of its intended purpose.

Now that we come to reason about various equalities formally, it is better to distinguish their notation. We will write $l \leftrightarrow r$ for formal equations as opposed to syntactic equality $=$.

**Definition 4.1** *A term $t$ is called* more general *than a term $t'$ in the equational theory $E$ if there is a substitution $\sigma$ such that $t' \leftrightarrow_E^* t.\sigma$. Conversely, term $t'$ is called* more special *than $t$.*

Intuitively, a term $t$ is more general than every term that can be derived from $t$ by substitution applications and conversion steps.

**Definition 4.2** *A term $t_G$ is called a* generalizer *of terms $t_1$ and $t_2$ in the equational theory $E$ if there are substitutions $\sigma_1$ and $\sigma_2$ such that $t_G.\sigma_1 \leftrightarrow_E^* t_1$ and $t_G.\sigma_2 \leftrightarrow_E^* t_2$.*

In other words, a generalizer of $t_1$ and $t_2$ is a term that is more general than each $t_1$ and $t_2$. Obviously, there is always a (most general) generalizer: a variable, $x_0$, since $x_0.\sigma_1 = t_1$ and $x_0.\sigma_2 = t_2$ where $\sigma_1 =_{\text{def}} \{x_0 \mapsto t_1\}$ and $\sigma_2 =_{\text{def}} \{x_0 \mapsto t_2\}$. A term more general than a given generalizer is again a generalizer. So it is obvious that people prefer the *most special* generalizer, provided there is one.

In this respect, generalization is the dual to *unification* where, modulo conversion, a most general common special case is wanted. For this reason, generalization is sometimes also called "anti-unification"[13]. Plotkin has closely studied the special-case relation between terms [14]. In the case where $E$ is empty, the special-case relation forms a lattice, i.e. there are most general syntactic unifiers and most special syntactic generalizers. In contrast to unification, an intensively studied subject (see e.g. [15]), properties and methods for generalization for nonempty $E$ are hardly known.

A few generalization methods have been exercised for use in inductive theorem proving. As an intriguing example we mention the systematic construction of a term rewriting system of counting functions on the basis of an effective enumeration of elements [16, 17].

For our purpose to extract homomorphisms one can do simpler. There is no need to adapt the given equational theory, nor to introduce new functions. For our generalization method we start from Jouannaud [18] and Comon, Haberstrau and Jouannaud [19] who use a *resolvant* presentation of the equational theory to get a generalization algorithm. Our generalization calculus differs from theirs basically by the use of rewrite derivations instead of conversions.

Let us first identify what kind of generalization is required by our CS-method for the homomorphism extraction problem.

# 5 CS-Method

We present the CS-method of homomorphism extraction based on a generalization procedure and then illustrate its usage on the homomorphism extraction for the *scan* function which we started in Example 1.

Let us assume that function $h$ is a homomorphism with associative combine operation $\circledast$, i.e. (1) holds. Let $t_H$ denote a term over $u$ and $v$ that defines $\circledast$:

$$u \circledast v \leftrightarrow t_H \tag{6}$$

The following two terms, built from $t_H$ by substitutions: $t_L = t_H.\{u \mapsto h([a]), v \mapsto h(y)\}$ and $t_R = t_H.\{u \mapsto h(x), v \mapsto h([b])\}$, are obviously in the *lh* and *rh* format (see Definition 3.2), correspondingly.

Terms $t_L$ and $t_R$ are semantically equal variants to all `cons` and `snoc` representations, respectively, of function $h$. In the simple extraction method above, we try to pick either a `cons` term, $t_C$, or a `snoc` term, $t_S$, to transform them into the desired format using various equalities of the theory of lists and to guess from this format the combine operation. In Example 1, such attempts for the *scan* function with the right-hand sides of (4) and (5) for $t_C$ and $t_S$ have failed.

Our new attempt is to use the two sequential definitions of $h$, the `cons` definition $h(a \mathbin{\cdot\!:} y) \leftrightarrow t_C, h([a]) \leftrightarrow t_B$ and the `snoc` definition $h(x \mathbin{:\!\cdot} b) \leftrightarrow t_S, h([b]) \leftrightarrow t'_B$, together. Using their *lh* and *rh* properties, respectively, one obtains specifying equations of operation $\circledast$ in two special cases:

$$t_B \circledast h(y) \leftrightarrow t_C \tag{7}$$

and

$$h(x) \circledast t'_B \leftrightarrow t_S \ . \tag{8}$$

Now the basic idea is to generalize these given two equations to the equation (6) and thereafter extract $t_H$, the wanted piece of the definition of $h$ as a homomorphism.

Figure 1(a) illustrates the relations between the terms $t_H$, $t_C$, $t_S$, $t_L$, and $t_R$. Dotted arrows indicate application of substitutions; solid (black) arrows indicate conversion steps. By the subscripts (6), (7), (8) we indicate that the conversion steps are exactly the respective conversion rules above.

From Figure 1(a) one can read off that each substitution is applied to *two* terms — this is a simultaneous generalization problem. In order to work with the established notion of generalizer, we introduce a fresh binary

$$\begin{tikzcd}
t_B \circledast h(y) \arrow[r, leftrightarrow, "(7)"'] & t_C \arrow[r, leftrightarrow, "*", "E"'] & t_H.\sigma_1 \arrow[r, equal] & t_L \\
u \circledast v \arrow[u, dotted, "\sigma_1"'] \arrow[d, dotted, "\sigma_2"] \arrow[r, leftrightarrow, "(6)"'] & t_H \arrow[r, equal] & t_H \arrow[u, dotted, "\sigma_1"'] \arrow[d, dotted, "\sigma_2"] & \\
h(x) \circledast t'_B \arrow[r, leftrightarrow, "(8)"'] & t_S \arrow[r, leftrightarrow, "*", "E"'] & t_H.\sigma_2 \arrow[r, equal] & t_R
\end{tikzcd}$$

(a) as a simultaneous generalization problem

$$\begin{tikzcd}
t_B \circledast h(y) \rightharpoonup t_C \arrow[d, leftrightarrow, "*"', "E"] \\
h([a]) \circledast h(y) \rightharpoonup t_L \\
u \circledast v \rightharpoonup t_H \arrow[u, dotted, "\sigma_1"'] \arrow[d, dotted, "\sigma_2"] \\
h(x) \circledast h([b]) \rightharpoonup t_R \arrow[d, leftrightarrow, "*"', "E"] \\
h(x) \circledast t'_B \rightharpoonup t_S
\end{tikzcd}$$

(b) using rule term encoding

Figure 1: The relationships of the terms after a successful CS-Generalization; here: $\sigma_1 = \{u \mapsto t_B, v \mapsto h(y)\}$, $\sigma_2 = \{u \mapsto h(x), v \mapsto t'_B\}$.

function symbol, $\rightharpoonup$, and model pairs $(s,t)$ of terms as terms $s \rightharpoonup t$. Because we will use this construction for the pairs $(l,r)$ where $l \to r$ is a rewrite rule, we will call $l \rightharpoonup r$ a *rule term*. A rule term encodes a rewrite rule as a term. For example $u \circledast v \rightharpoonup t_H$ is a rule term that encodes the rule $u \circledast v \to t_H$.

With this encoding we get the view of Figure 1(b).

**Theorem 4** *Let $E$ be the theory of lists and let $t'_B = t_B.\{a \mapsto b\}$. If the two rule terms $t_B \circledast h(y) \rightharpoonup t_C$ and $h(x) \circledast t'_B \rightharpoonup t_S$ have an $E$-generalizer $u \circledast v \rightharpoonup t_H$ w.r.t. substitutions $\sigma_1 = \{u \mapsto t_B, v \mapsto h(y)\}$ and $\sigma_2 = \{u \mapsto h(x), v \mapsto t'_B\}$, and the operation $\circledast$ thus defined is associative, then:*

1. *The* `cons`, `snoc`, *and* $+\!\!+$ *definitions define the same function $h$.*

2. *Function $h$ is lh and rh, with $f(a) = t_B$.*

3. *Function $h$ is a homomorphism with $\circledast$ as combine operation,*

**Proof** *The first claim follows by Theorem 6(3) in Section 8. For the second claim, one shows $h(a \cdot\!\!: y) \leftrightarrow t_C \leftrightarrow t_B \circledast h(y) \leftrightarrow h([a]) \circledast h(y)$ and $h(x :\!\!\cdot\, b) \leftrightarrow t_S \leftrightarrow h(x) \circledast t'_B \leftrightarrow h(x) \circledast h([b])$. The third claim is equivalent to it by Theorem 3.*

Note that Theorem 4 does *not assume* that the `cons`- and the `snoc`-definitions define the same $h$; it rather establishes this property.

We call the generalization in Theorem 4 the *CS-generalization* (CS stands for "Cons + Snoc") in the theory of lists $E$; see Section 8 for a formal explanation. A successful CS-generalization yields the term $t_H$, and thereby the definition of $\circledast$. This allows one to formulate the claim that $\circledast$ is associative. If associativity can be proven inductively then Theorem 4 applies by which the $+\!\!\!+$ definition defines $h$ as a homomorphism representation equivalent to the given `cons`- and `snoc`-representations.

The envisaged CS-method of finding the combine operation proceeds in three steps as follows.

**CS-Method:**

1. The user is asked to provide two sequential definitions for a given function: a `cons` term $t_C$ and a `snoc` term $t_S$.

2. The CS-generalization, applied to the corresponding rule terms $t_B \circledast h(y) \rightharpoonup t_C$ and $h(x) \circledast t'_B \rightharpoonup t_S$, yields a rule term $u \circledast v \rightharpoonup t_H$, together with substitutions $\sigma_1 = \{u \mapsto t_B, v \mapsto h(y)\}$ and $\sigma_2 = \{u \mapsto h(x), v \mapsto t'_B\}$,

3. Associativity of the operation $\circledast$ defined by term $t_H$ is proven inductively.

Equipped with the CS-method, we again try to extract a homomorphism for the *scan* function.

**Example 1 (Continued)** *For function scan, the corresponding rule terms* $t_B \circledast h(y) \rightharpoonup t_C$ *and* $h(x) \circledast t'_B \rightharpoonup t_S$ *are:*

$$
\begin{aligned}
&[a] \ \circledast \ scan\,(\odot)\,y \rightharpoonup a \ \because \ (map\,(a\,\odot)\,(scan\,(\odot)\,y)) \\
&scan\,(\odot)\,x \ \circledast \ [b] \rightharpoonup (scan\,(\odot)\,x) \ \therefore \ (last\,(scan\,(\odot)\,x) \odot b)
\end{aligned}
$$

*Their CS-generalization yields a rule term*

$$u \circledast v \rightharpoonup u \ +\!\!\!+ \ map\,(last\,(u)\odot)\,v$$

*with the substitutions*

$$\sigma_1 = \{u \mapsto [a], v \mapsto scan(\odot)y\}, \tag{9}$$

$$\sigma_2 = \{u \mapsto scan(\odot)x, v \mapsto [b]\}\ . \tag{10}$$

*Moreover one can prove that the operation $\circledast$ thus defined*

$$u \circledast v \leftrightarrow u \mathbin{+\!\!+} map\,(last\,(u) \odot)\, v \tag{11}$$

*is associative. Therefore, according to Theorem 4, scan as defined by the* `cons` *definition (or, equivalently, as defined by the* `snoc` *definition), is a homomorphism: $scan\,(\odot) = hom\,([.], \circledast)$, with $\circledast$ defined by* (11)*.*

From the practical point of view, it would be advisable to present term $t_C$ in the *lw* format and term $t_S$ in the *rw* format, although the CS-method does not explicitly prescribe that. Such additional restriction would simplify the necessary generalization. Moreover, a difficulty in writing either an *rw* or an *lw* definition of a function is a good indicator that the function might not be a homomorphism.

In the next section we provide the necessary background in term rewriting to speak about a generalization calculus.

# 6 The Switch to Rewriting

Next we describe our approach to generalization using term rewriting within the Bird-Meertens theory of lists. See Klop [20] for a good introductory text on term rewriting.

To have a simple technical apparatus, we henceforth restrict ourselves to first-order terms whence we suppress higher order parameters, and consider a fixed associative operation $\odot$. This renders it necessary to express a segment of the theory of lists by first-order functions and term rewriting rules. In our running example we replace $scan\,(\odot)\,x$ by $scn(x)$, and $map\,(a\,\odot)\,(x)$ by $mp(a, x)$.

Moreover from now on we use rewrite rules

$$f(t_1, \ldots, t_n) \to t_0$$

instead of conversion rules

$$f(t_1, \ldots, t_n) \leftrightarrow t_0$$

in order to account for oriented replacement, i.e. rewriting: A rewrite rule may be used to replace a term $t$ of the form $C[f(t_1, \ldots, t_n)]$ by $t'$ of the form $C[t_0]$, a fact expressed by $t \to t'$. By $C[t]$ we indicate that $t$ appears in the context $C$. If $R$ is a system of such rewrite rules, then one writes $t \to_R t'$ for the application of a rule from $R$ somewhere in $t$, yielding a term $t'$. The relation $\leftrightarrow_R^*$ is the equivalence closure of $\rightarrow_R$, i.e. the smallest binary relation that contains $\rightarrow_R$ and is reflexive, symmetric, and transitive.

To be able to reason in the rewriting framework, we have to fix a few term rewriting systems: The term rewriting system $R$ the conversion relation $\leftrightarrow_R$ of which will be the "semantic equality", and term rewriting systems $R_{cons}$, $R_{snoc}$, and $R_{conc}$ for the three versions of definitions of $h$.

Associativity of $\odot$ and of $+\!\!\!+$ are first to mention.

$$a \odot (b \odot c) \rightarrow (a \odot b) \odot c \tag{$A_\odot$}$$

$$x +\!\!\!+ (y +\!\!\!+ z) \rightarrow (x +\!\!\!+ y) +\!\!\!+ z \tag{$A_{+\!\!+}$}$$

The symbols $\cdot\!:$ and $:\!\cdot$ and their defining rules

$$a \cdot\!: y \rightarrow [a] +\!\!\!+ y$$

$$x :\!\cdot b \rightarrow x +\!\!\!+ [b]$$

will not appear explicitly in our method. Rather we replace throughout $t \cdot\!: t'$ and $t :\!\cdot t'$ by $[t] +\!\!\!+ t'$ and $t +\!\!\!+ [t']$, respectively.

To keep things simple, let us assume that $h$ is not mutually recursive, i.e. that the call relation does not contain cycles of length greater than one. Next we assume given a term rewriting system, $R_{base}$, to contain suitable rules for the functions that are called by $h$. We may by an inductive argument assume that all auxiliary functions are homomorphisms themselves and that the extraction is already done for them. In other words, each function $h'$ called by $h$ is given by means of $[.]$ and $+\!\!\!+$, and the right hand side of its definition does not use $x$ and $y$ unless in $h'(x)$ and $h'(y)$ respectively. (For functions in more than one parameter, we assume that parallelization is done for the last parameter, which is a list.)

Given a term rewriting system $R_{base}$ and terms $t_B$, $t_C$, $t_S$, $t_H$, we define a term rewriting system $R$ to describe the conversion relation, $\leftrightarrow_R^*$, that will serve as the "semantic equality", $\leftrightarrow_E^*$; and three term rewriting systems, $R_{cons}$, $R_{snoc}$, and $R_{conc}$, for reasoning about the `cons`, `snoc`, and *conc* definitions of $h$, respectively:

$$\begin{aligned}
R &= A_\odot \cup A_{+\!\!+} \cup R_{base} \\
R_0 &= R \cup \{h([a]) \rightarrow t_B,\ h(x +\!\!\!+ y) \rightarrow h(x) \circledast h(y)\} \\
R_{cons} &= R_0 \cup \{t_B \circledast h(y) \rightarrow t_C\} \\
R_{snoc} &= R_0 \cup \{h(x) \circledast t'_B \rightarrow t_S\} \\
R_{conc} &= R_0 \cup \{u \circledast v \rightarrow t_H\}
\end{aligned}$$

Here, $t'_B = t_B.\{a \mapsto b\}$ is just the term $t_B$ where $a$ is renamed by $b$.

**Example 1 (Continued)** *In our example, scn calls the functions mp and last. So,* $R_{base}$ *becomes*

$$\begin{aligned} last[a] &\to a && (12)\\ last(x \mathbin{+\!\!+} y) &\to last\, y && (13)\\ mp(a,[b]) &\to [a \odot b] && (14)\\ mp(a, x \mathbin{+\!\!+} y) &\to mp(a,x) \mathbin{+\!\!+} mp(a,y) && (15) \end{aligned}$$

*Next we get* $[a]$ *for* $t_B$*, so* $R_0$ *is* $R$ *together with the rules*

$$\begin{aligned} scn([a]) &\to [a] && (16)\\ scn(x \mathbin{+\!\!+} y) &\to scn(x) \circledast scn(y) && (17) \end{aligned}$$

*As* $t_C$ *is the term* $[a] \mathbin{+\!\!+} mp(a, scn(y))$*, we get for* $R_{cons}$ *the rules in* $R_0$ *plus the rule*

$$[a] \circledast scn(y) \to [a] \mathbin{+\!\!+} mp(a, scn(y)) \qquad (18)$$

*The term* $t'_B$ *is* $[b]$*, and the term* $t_S$ *is* $scn(x) \mathbin{+\!\!+} [last(scn(x)) \odot b]$*. So the rule in* $R_{snoc} \setminus R_0$ *reads*

$$scn(x) \circledast [b] \to scn(x) \mathbin{+\!\!+} [last(scn(x)) \odot b]\ . \qquad (19)$$

What we want is $t_H$ such that $R_{conc}$ defines the same function $h$ as do $R_{cons}$ and $R_{snoc}$. Designing a suitable CS-generalization procedure and investigating its properties is the topic of the next section.

# 7 A Calculus for Generalization

Following a good custom, we introduce our algorithm by means of a calculus. As the objects of the calculus we keep triples $(\sigma_1, \sigma_2, t_0)$, maintaining the invariant that $t_0$ is a generalizer of $t_1$ and $t_2$ via the substitutions $\sigma_1$ and $\sigma_2$, respectively. Starting from the triple $(\{x_0 \mapsto t_1\}, \{x_0 \mapsto t_2\}, x_0)$, where $x_0$ is the most general generalizer, we successively apply inference rules to specialize, until no more inference rule applies.

The basic idea is to repeat, as long as possible, the following step: Extract a common row of $\sigma_1$ and $\sigma_2$ and move it to $t_0$. Every such step, while preserving the generalization property of $t_0$, adds more speciality to it, until finally $t_0$ is maximally special.

Figure 2 and Figure 3 show the two inference rules which we will use. The formula $t \xrightarrow[R]{\varepsilon}{}^{*}\, t'$ means that $t$ rewrites in zero or more steps to $t'$ where

$$\text{If } f(u'_1,\ldots,u'_m) \xrightarrow[R]{\varepsilon}{}^{*} u_i \text{ and } f(v'_1,\ldots,v'_m) \xrightarrow[R]{\varepsilon}{}^{*} v_i \text{ then} \tag{20}$$

$$\frac{\left\{\begin{matrix} x_1 \mapsto u_1 \\ \vdots \\ x_i \mapsto u_i \\ \vdots \\ x_n \mapsto u_n \end{matrix}\right\} \quad \left\{\begin{matrix} x_1 \mapsto v_1 \\ \vdots \\ x_i \mapsto v_i \\ \vdots \\ x_n \mapsto u_n \end{matrix}\right\} \quad t_0}{\left\{\begin{matrix} x_1 \mapsto u_1 \\ \vdots \\ x_{i-1} \mapsto u_{i-1} \\ x'_1 \mapsto u'_1 \\ \vdots \\ x'_m \mapsto u'_m \\ x_{i+1} \mapsto u_{i+1} \\ \vdots \\ x_n \mapsto u_n \end{matrix}\right\} \quad \left\{\begin{matrix} x_1 \mapsto v_1 \\ \vdots \\ x_{i-1} \mapsto v_{i-1} \\ x'_1 \mapsto v'_1 \\ \vdots \\ x'_m \mapsto v'_m \\ x_{i+1} \mapsto v_{i+1} \\ \vdots \\ x_n \mapsto v_n \end{matrix}\right\} \quad t_0.\{x_i \mapsto f(x'_1,\ldots,x'_m)\}}$$

Figure 2: "Ancestor Decomposition" rule

every rewrite step takes place at the root position, $\varepsilon$, of $t$. The "ancestor decomposition" rule in Figure 2 acts as follows. First a common variable, $x_i$, in the domain of the two substitutions is selected. If $u_i$ shares its root function symbol, $f$, with $v_i$ then the ancestor decomposition rule is applicable with void rewrite derivations, $f(u'_1,\ldots,u'_m) = u_i$ and $f(v'_1,\ldots,v'_m) = v_i$. The common function symbol is broken down, by splitting the mapping $x_i \mapsto f(u'_1,\ldots,u'_m)$ to $x_i \mapsto f(x'_1,\ldots,x'_m)$ and $x'_j \mapsto u'_j$ where $x'_j$ are fresh variables for each argument position, $j$, of $f$. Likewise $x_i \mapsto f(v'_1,\ldots,v'_m)$ is split to $x_i \mapsto f(x'_1,\ldots,x'_m)$ and $x'_j \mapsto v'_j$. Their common part, $x_i \mapsto f(x'_1,\ldots,x'_m)$, can now be transferred to $t_0$.

A term $t$ such that $t \xrightarrow[R]{\varepsilon}{}^{*} t'$ is called an *ancestor* of $t'$. To enable conversion by $R$ rules, we furthermore allow that $u_i$ or $v_i$ are replaced by some ancestor thereof. In other words, we allow non-void rewrite derivations in Condition (20).

The "agreement" rule in Figure 3 acts as follows. The mappings $x_i \mapsto u$ and $x_j \mapsto u$ with common right hand sides in the first substitution can be

$$\frac{\left\{\begin{array}{c} x_1 \mapsto u_1 \\ \vdots \\ x_i \mapsto u \\ \vdots \\ x_j \mapsto u \\ \vdots \\ x_n \mapsto u_n \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto v_1 \\ \vdots \\ x_i \mapsto v \\ \vdots \\ x_j \mapsto v \\ \vdots \\ x_n \mapsto v_n \end{array}\right\} \quad t_0}{\left\{\begin{array}{c} x_1 \mapsto u_1 \\ \vdots \\ x_i \mapsto u \\ \vdots \\ x_{j-1} \mapsto u_{j-1} \\ x_{j+1} \mapsto u_{j+1} \\ \vdots \\ x_n \mapsto u_n \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto v_1 \\ \vdots \\ x_i \mapsto v \\ \vdots \\ x_{j-1} \mapsto v_{j-1} \\ x_{j+1} \mapsto v_{j+1} \\ \vdots \\ x_n \mapsto v_n \end{array}\right\} \quad t_0.\{x_j \mapsto x_i\}}$$

Figure 3: "Agreement" rule

split into $x_j \mapsto x_i$ and $x_i \mapsto u$. If moreover $x_i \mapsto v$ and $x_j \mapsto v$ in the second substitution, then likewise $x_j \mapsto x_i$ and $x_i \mapsto v$ can be split there. The common part, $x_j \mapsto x_i$, of the two can be transferred to $t_0$.

Occasionally we will justify an application of the ancestor decomposition rule by a remark of the form "AncDec for $f$; $t \underset{R}{\overset{\varepsilon}{\longrightarrow}}{}^{*} t'$", or just "AncDec for $f$" if the rewrite derivation $t \underset{R}{\overset{\varepsilon}{\longrightarrow}}{}^{*} t'$ is void, i.e. $t = t'$. To justify an application of the agreement rule we will put a remark like "Agr for $x_i$, $x_j$".

It is fairly easy to prove that our calculus is sound in the following sense:

**Theorem 5 (Soundness)** *For every inference step $(\sigma_1, \sigma_2, t_0) \vdash (\sigma'_1, \sigma'_2, t'_0)$, we have $t_0.\sigma_1 \leftarrow_R^* t'_0.\sigma'_1$ and $t_0.\sigma_2 \leftarrow_R^* t'_0.\sigma'_2$.*

**Corollary 1** *If $t_0$ is a generalizer of $t_1$ and $t_2$ w.r.t. the term rewriting system $R$, and $(\sigma_1, \sigma_2, t_0) \vdash (\sigma'_1, \sigma'_2, t'_0)$ is an inference step, then $t'_0$ is a more special generalizer of $t_1$ and $t_2$ w.r.t. $R$.*

We conjecture that under reasonable restrictions there is a completeness result as well, but for a good reason we do not pursue this subject further here. Our aim is practical application. So we will turn the calculus into a terminating, deterministic procedure (i.e. an algorithm), for which it is essential to strengthen Condition (20) of the agreement rule. For this restricted form completeness does not hold, anyway.

**Example 1 (Continued)** *Now let us study our scn example to see how our calculus operates. Here we have*

$$\begin{aligned} t_1 &= ([a] \circledast scn(y) \rightharpoonup [a] +\!\!+\, mp(a, scn(y))) \\ t_2 &= (scn(x) \circledast [b] \rightharpoonup scn(x) +\!\!+\, [last(scn(x)) \odot b]) \end{aligned}$$

*A derivation is listed in Figure 4.*

We have seen examples of generalization that are, properly, CS-generalizations as needed for our CS-method. Next we are going to show how these fit into the CS-method.

# 8 CS-Generalization

In the previous section we established that the calculus is sound: every derivation yields a generalizer. Not every generalizer obtained this way, however, needs to be suitable for our purpose, CS-generalization. In this aspect it is natural to coin the following notion of success:

**Definition 8.1** *A derivation in the generalization calculus is called* successful *if it ends in a triple of the form*

$$\{u \mapsto t_B, v \mapsto h(y)\} \qquad \{u \mapsto h(x), v \mapsto t'_B\} \qquad u \circledast v \rightharpoonup t_H$$

*modulo bijective renaming of the variables $u$ and $v$.*

Please note that there may be more than one successful derivation, or no successful derivation at all.

**Definition 8.2** *For a successful derivation,*

$$\begin{aligned} & (\{x_0 \mapsto (t_B \circledast h(y) \rightharpoonup t_C)\}, \\ & \ \{x_0 \mapsto (h(x) \circledast t'_B \rightharpoonup t_S)\}, x_0) \\ \vdash \ & (\{u \mapsto t_B, v \mapsto h(y)\}, \\ & \ \{u \mapsto h(x), v \mapsto t'_B\}, u \circledast v \rightharpoonup t_H) \end{aligned}$$

*we call $t_H$ its* computed solution *of the problem instance $t_B, t_C, t_S$.*

$$\left\{x_0 \mapsto t_1\right\} \qquad \left\{x_0 \mapsto t_2\right\} \qquad x_0$$

$$\vdash \quad (\textsf{AncDec for } \rightharpoonup)$$

$$\left\{\begin{array}{c} x_1 \mapsto [a] \circledast scn(y) \\ x_2 \mapsto [a] \mathbin{+\!\!+} \\ \qquad mp(a, scn(y))) \end{array}\right\} \qquad \left\{\begin{array}{c} x_1 \mapsto scn(x) \circledast [b] \\ x_2 \mapsto scn(x) \mathbin{+\!\!+} \\ \qquad [last(scn(x)) \odot b] \end{array}\right\} \qquad x_1 \rightharpoonup x_2$$

$$\vdash \quad (\textsf{AncDec for } \circledast)$$

$$\left\{\begin{array}{c} x_3 \mapsto [a] \\ x_4 \mapsto scn(y) \\ x_2 \mapsto [a] \mathbin{+\!\!+} \\ \qquad mp(a, scn(y))) \end{array}\right\} \qquad \left\{\begin{array}{c} x_3 \mapsto scn(x) \\ x_4 \mapsto [b] \\ x_2 \mapsto scn(x) \mathbin{+\!\!+} \\ \qquad [last(scn(x)) \odot b] \end{array}\right\} \qquad x_3 \circledast x_4 \rightharpoonup x_2$$

$$\vdash \quad \textsf{AncDec for } \mathbin{+\!\!+})$$

$$\left\{\begin{array}{c} x_3 \mapsto [a] \\ x_4 \mapsto scn(y) \\ x_5 \mapsto [a] \\ x_6 \mapsto mp(a, scn(y))) \end{array}\right\} \qquad \left\{\begin{array}{c} x_3 \mapsto scn(x) \\ x_4 \mapsto [b] \\ x_5 \mapsto scn(x) \\ x_6 \mapsto [last(scn(x)) \odot b] \end{array}\right\} \qquad x_3 \circledast x_4 \rightharpoonup x_5 \mathbin{+\!\!+} x_6$$

$$\vdash \quad \textsf{Agr for } x_3,\ x_5)$$

$$\left\{\begin{array}{c} x_3 \mapsto [a] \\ x_4 \mapsto scn(y) \\ x_6 \mapsto mp(a, scn(y))) \end{array}\right\} \qquad \left\{\begin{array}{c} x_3 \mapsto scn(x) \\ x_4 \mapsto [b] \\ x_6 \mapsto [last(scn(x)) \odot b] \end{array}\right\} \qquad x_3 \circledast x_4 \rightharpoonup x_3 \mathbin{+\!\!+} x_6$$

$$\vdash \quad (\textsf{AncDec for } mp;\ mp(last(scn(x)), [b]) \xrightarrow[(14)]{\varepsilon} [last(scn(x)) \odot b])$$

$$\left\{\begin{array}{c} x_3 \mapsto [a] \\ x_4 \mapsto scn(y) \\ x_7 \mapsto a \\ x_8 \mapsto scn(y) \end{array}\right\} \qquad \left\{\begin{array}{c} x_3 \mapsto scn(x) \\ x_4 \mapsto [b] \\ x_7 \mapsto last(scn(x)) \\ x_8 \mapsto [b] \end{array}\right\} \qquad x_3 \circledast x_4 \rightharpoonup x_3 \mathbin{+\!\!+} mp(x_7, x_8)$$

$$\vdash \quad (\textsf{Agr for } x_4,\ x_8)$$

$$\left\{\begin{array}{c} x_3 \mapsto [a] \\ x_4 \mapsto scn(y) \\ x_7 \mapsto a \end{array}\right\} \qquad \left\{\begin{array}{c} x_3 \mapsto scn(x) \\ x_4 \mapsto [b] \\ x_7 \mapsto last(scn(x)) \end{array}\right\} \qquad x_3 \circledast x_4 \rightharpoonup x_3 \mathbin{+\!\!+} mp(x_7, x_4)$$

$$\vdash \quad (\textsf{AncDec for } mp;\ last([a]) \xrightarrow[(12)]{\varepsilon} a)$$

$$\left\{\begin{array}{c} x_3 \mapsto [a] \\ x_4 \mapsto scn(y) \\ x_9 \mapsto [a] \end{array}\right\} \qquad \left\{\begin{array}{c} x_3 \mapsto scn(x) \\ x_4 \mapsto [b] \\ x_9 \mapsto scn(x) \end{array}\right\} \qquad x_3 \circledast x_4 \rightharpoonup x_3 \mathbin{+\!\!+} mp(last(x_9), x_4)$$

$$\vdash \quad (\textsf{Agr for } x_3,\ x_9)$$

$$\left\{\begin{array}{c} x_3 \mapsto [a] \\ x_4 \mapsto scn(y) \end{array}\right\} \qquad \left\{\begin{array}{c} x_3 \mapsto scn(x) \\ x_4 \mapsto [b] \end{array}\right\} \qquad x_3 \circledast x_4 \rightharpoonup x_3 \mathbin{+\!\!+} mp(last(x_3), x_4)$$

Figure 4: A successful derivation for *scn*

A ground term is a term that does not contain any (free) variable. The inductive theory of a term rewriting system, $R$, is the set of all formal equations $s \leftrightarrow t$ between terms such that $s.\sigma \leftrightarrow_R^* t.\sigma$ holds for all substitutions $\sigma$ such that $s.\sigma$ and $t.\sigma$ are ground. A term rewriting system $R'$ is called a *conservative extension* of $R$ if every equation $s \leftrightarrow t$ in the inductive theory of $R'$, where $s$ and $t$ are terms over the signature of $R$, is already in the inductive theory of $R$. See, e.g., [21] for an introduction to inductive theorem proving based on rewriting.

**Theorem 6 (Reliability)** *For every computed solution, $t_H$, of the problem instance $t_B, t_C, t_S$, and term rewriting systems defined as in Section 6, the following properties hold:*

1. *The conversion relations $\leftrightarrow_{R_{cons}}^*$ and $\leftrightarrow_{R_{snoc}}^*$ are included in the conversion relation $\leftrightarrow_{R_{conc}}^*$.*

2. *If $\circledast$ is associative in the inductive theory of $R_{conc}$ then $R_{conc}$ is a conservative extension of $R$.*

3. *If $\circledast$ is associative in the inductive theory of $R_{conc}$ then the inductive theories of $R_{cons}$, $R_{snoc}$, and $R_{conc}$ coincide.*

**Proof** *By the success of derivation we have a situation as depicted in Figure 1(b). The only rule $t_B \circledast h(y) \to t_C$ in $R_{cons}$ not in $R_{conc}$ can be replaced by a conversion $(u \circledast v).\sigma_1 \to_{R_{conc}} t_H.\sigma_1 \leftrightarrow_R^* t_C$ in $R_{conc}$. This way every conversion in $R_{cons}$ can be translated into one in $R_{conc}$. By symmetry the same holds for conversions in $R_{snoc}$.*

*For the second claim, let $\circledast$ be associative in the inductive theory of $R$. Then $R_{conc}$ is a conservative extension of $R$ as we show next. We first note that $R_{conc}$ is ground confluent modulo $R$, i.e. $t \leftrightarrow_{R_{conc}}^* t'$ implies $t \to_{R_{conc}}^* \leftrightarrow_R^* \leftarrow_{R_{conc}}^* t'$ for all ground terms $t$, $t'$. Specifically,*

$$\begin{array}{ccccc}
h((t_1 +\!\!+ t_2) +\!\!+ t_3) & \xrightarrow{R_{conc}} & h(t_1 +\!\!+ t_2) \circledast h(t_3) & \longrightarrow & (h(t_1) \circledast h(t_2)) \circledast h(t_3) \\
{\scriptstyle R_{conc}}\big\downarrow & & & & {\scriptstyle R}\big\updownarrow{\scriptstyle *} \\
h(t_1 +\!\!+ (t_2 +\!\!+ t_3)) & \longrightarrow & h(t_1) \circledast h(t_2 +\!\!+ t_3) & \longrightarrow & h(t_1) \circledast (h(t_2) \circledast h(t_3))
\end{array}$$

*and the proper rules in $R_{conc}$ have a $h$ or $\circledast$ symbol at the left hand side. It is a folk theorem in term rewriting that a ground confluent term rewriting system where every "new" rewrite rule contains a "new" symbol at the left hand side, is a conservative extension of the "old" system. This extends to ground confluence modulo the "old" system in a straightforward way.*

*The inductive theory of $R_{cons}$ is included in the inductive theory of $R_{conc}$ since its conversion is. We have to prove the converse.*

*We first derive that from associativity of $\circledast$ in the inductive theory of $R_{conc}$ it follows that the equation*

$$t_H\{u \mapsto t_H, v \mapsto w\} \;\leftrightarrow\; t_H\{v \mapsto t_H\{u \mapsto v, v \mapsto w\}\}\ , \tag{21}$$

*too, is in the inductive theory of $R_{conc}$: For arbitrary ground terms $t_1$, $t_2$, and $t_3$, one gets*

$$\begin{array}{rl} & t_H.\{u \mapsto t_H.\{u \mapsto t_1, v \mapsto t_2\}, v \mapsto t_3\} \\ \leftarrow_{R_{conc}} & (t_1 \circledast t_2) \circledast t_3 \\ \leftrightarrow^*_{R_{conc}} & t_1 \circledast (t_2 \circledast t_3) \\ \rightarrow_{R_{conc}} & t_H.\{u \mapsto t_1, v \mapsto t_H.\{u \mapsto t_2, v \mapsto t_3\}\} \end{array}$$

*As $R_{conc}$ is a conservative extension of $R$, we conclude that* (21) *is also in the inductive theory of $R$.*

*Finally we show that the only rule in $R_{conc}$ not in $R_{cons}$ is in the inductive theory of $R_{cons}$:*

$$h(t +\!\!+\, t') \leftrightarrow^*_{R_{cons}} t_H.\{u \mapsto h(t), v \mapsto h(t')\}$$

*holds for all ground terms $t$, $t'$.*

*We define $R'$ to be $R_{cons}$ but the associativity of $+\!\!+$ (i.e. Rule $(A_{+\!\!+})$) oriented in the reverse direction. It is easy to show that $R'$ is terminating and that every ground term has a $R'$-normal form of the shape $[a] +\!\!+\, t''$. We get that $t +\!\!+\, t' \rightarrow^*_{R'} [a] +\!\!+\, t''$ for suitable $a$ and $t''$. Hence there is the conversion*

$$\begin{array}{rl} & h(t +\!\!+\, t') \\ \leftrightarrow^*_{R_{cons}} & h([a] +\!\!+\, t'') \\ \rightarrow_{R_{cons}} & t_C.\{y \mapsto t''\} \\ \leftarrow^*_{R} & t_H.\{u \mapsto t_B, v \mapsto h(y)\}.\{y \mapsto t''\} \\ = & t_H.\{u \mapsto t_B, v \mapsto h(t'')\} \\ \leftarrow^*_{R_{cons}} & t_H.\{u \mapsto h([a]), v \mapsto h(t'')\} \\ \leftrightarrow^*_{R_{cons}} & t_H.\{u \mapsto h(t), v \mapsto h(t')\} \end{array}$$

*We have thus shown that $h(u +\!\!+\, v) \leftrightarrow t_H$ is in the inductive theory of $R_{cons}$, and so that $R_{conc}$ is included in the inductive theory of $R_{cons}$. This finishes the proof of Claim* (3).

As we have demonstrated, the generalization calculus may yield the computed solution. An inductive theorem prover may finally be used to prove associativity of $\circledast$.

**Example 1 (Continued)** *In our scn example, we have used the semi-automatic inductive prover* TIP *[22, 23] to produce a proof of associativity of $\circledast$ based on the lemmas*

$$\begin{aligned} last(mp(a,x)) &\to a \odot last\, x \\ mp(a, mp(b,x)) &\to mp(a \odot b, x) \end{aligned}$$

*and, in turn, a proof of each lemma. The three proofs are obtained by rewriting induction, without any user interaction.*

# 9 An Algorithm

To turn our calculus into an algorithm, one has to stick to a strategy, and prove its termination. We adopt the strategy to prefer the "agreement" rule, and to choose the smallest index $i$ for which an inference rule applies. If only the "ancestor decomposition" rule applies, we branch for every pair of justifying rewrites. Certainly, the complexity may become exponential this way, but experience shows that branching is harmless: Rarely is there more than one branch.

In the general case, we have to be aware of non-terminating derivations.

**Example 3** *Let $s$ be the successor and $p$ the predecessor on integers, together with the rewrite rules*

$$p(s(x)) \to x \qquad s(p(x)) \to x \ .$$

*Then we get the infinite derivation*

$$\begin{array}{ll} & (\{x \mapsto s(0)\}, \{x \mapsto p(p(0))\}, x) \\ \vdash & (\{x' \mapsto s(s(0))\}, \{x' \mapsto p(0)\}, p(x')) \\ \vdash & (\{x'' \mapsto s(0)\}, \{x'' \mapsto p(p(0))\}, s(p(x''))) \\ \vdash & \cdots \end{array}$$

The standard way to get finiteness of derivations, and so termination of the procedure, is to take care that derivations $(\sigma_1^1, \sigma_2^1, t_0^1) \vdash (\sigma_1^2, \sigma_2^2, t_0^2) \vdash \cdots$ satisfy $(\sigma_1^i, \sigma_2^i, t_0^i) \succ (\sigma_1^{i+1}, \sigma_2^{i+1}, t_0^{i+1})$ for an appropriate well-founded order $\succ$ on triples.

In term rewriting finiteness of rewrite derivations is ensured by the requirement $l > r$ for every rule $l \to r$ in $R$, where $>$ is a suitable *termination order*: a well-founded order on terms that is closed under contexts and substitution applications. The knowledge $l > r$ however is useless for our purposes since we need to apply rules in their *reverse direction*. It is not realistic to require $r > l$ instead; this property is usually violated. For instance $l = last[a]$, $r = a$ does not satisfy $l > r$ for any termination order since no term can be greater than a superterm.

Now we observe that not only do we apply the rule in the reverse direction, but also strip the top symbol of the left hand side. The stripping of the top symbol can be considered a decrease, provided that the reverse rule application does not harm it. To be on the safe side, we replace Condition (20) of the agreement rule by

$$\begin{cases} f(u'_1, \ldots, u'_m) \overset{\varepsilon}{\underset{R}{\to}}{}^{X} \; u_i, \\ f(v'_1, \ldots, v'_m) \overset{\varepsilon}{\underset{R}{\to}}{}^{Y} \; v_i \\ \text{where} \quad X + Y \leq 1 \;\; . \end{cases} \tag{22}$$

Here $t \overset{\varepsilon}{\underset{R}{\to}}{}^{X} t'$ means that $t$ rewrites in exactly $X$ steps to $t'$ where each step takes place at the root position, $\varepsilon$, of $t$. We call the resulting calculus the *restricted generalization calculus*. Condition (22) is not as hard in practice as it may seem. Observe, for instance, that the derivations in Example 3 and in Figure 4 each are within the restricted generalization calculus.

**Definition 9.1** *Let $>$ be an order on pairs of terms. A term rewriting system $R$ is called* reversely guarded by $>$ *if for every rewrite rule $f(l_1, \ldots, l_m) \to r$ in $R$, both $(r, f(x_1, \ldots, x_m)) > (l_j, x_j)$ and $(f(x_1, \ldots, x_m), r) > (x_j, l_j)$ hold for all $1 \leq j \leq n$.*

**Theorem 7 (Termination)** *Let $>$ be a well-founded order on pairs of terms, closed under substitution application, that extends the component-wise subterm order. If $R$ is reversely guarded by $>$ then the restricted generalization calculus admits no infinite derivations.*

A rigorous proof can be found in the technical report [24]. A term rewriting system that contains rules $l \to r$ where $l$ contains a variable not in $r$ is not reversely guarded. For instance, Rule (13) has $x$ on its left, but not on its right hand side. The term rewriting system $R$ for our *scn* function is therefore *not* reversely guarded.

It makes sense to exclude such rules on principle since they need guessing the right term to substitute for the new variable and so cannot be used effectively anyway. Note that we do not use Rule (13) in the generalization for *scan*.

**Example 1 (Continued)** *The term rewriting system* $R \setminus \{(13)\}$ *is reversely guarded. To prove this, assign each function symbol,* $f$*, its weight, a non-negative integer,* $\#f$*. The weight* $\#t$ *of a term* $t$ *is defined to be the sum of all weights of function symbols in* $t$*. Now a relation* $>$ *on pairs of terms is defined as follows.*

> $(t_1, t_2) > (t'_1, t'_2)$, *if*
> *every variable occurs in* $(t_1, t_2)$ *at least as often as in* $(t'_1, t'_2)$*, and*
> $\#t_1 + \#t_2 > \#t'_1 + \#t'_2$

*It is straightforward to show that* $>$ *is a well-founded order, closed under substitution application, and that* $>$ *contains the component-wise subterm relation.*

*To show reverse guardedness, we show* $\#r + 2\#f > \#l$ *for every rule* $l \to r$ *where the root symbol on the left is* $f$*. The weight assignment* $\#\,last = 2$ *and* $\#f = 1$ *for all* $f$ *else obviously does the job. Here is the reasoning for Rule* (12)*:*

$$\#r + 2\#f = 0 + 2 \cdot 2 > 2 + 1 = \#l$$

*So every derivation in the restricted generalization calculus that refuses to use Rule* (13) *is finite: We have got an algorithm that computes a finite set of generalizers among which there may be a computed solution.*

# 10 Conclusion

*Extraction of a homomorphism* from a `cons` and a `snoc` definition of a function on lists is a powerful technique for obtaining parallel programs. We have successfully applied a technique from the area of term rewriting, *generalization*, to attack this nontrivial, intriguing problem.

We have introduced a very simple calculus for generalization of terms that contains two inference rules: *ancestor decomposition* and *agreement*. The former rule reconstructs a common top symbol of a pair of terms, potentially after a few reverse rewrite steps at the top. The latter joins variables that map to the same term.

If the rules for `cons` and `snoc` are encoded in terms then generalization may yield a term that encodes the definition of the homomorphism. This, provided that an associativity result can be proven inductively. The calculus is shown sound as a calculus for generalization, and reliable for use within a method to extract a homomorphism. The associativity proof can conveniently be done by an automated inductive theorem prover.

We impose a strategy and a sensible restriction on the calculus, under which we obtain a terminating, deterministic procedure. There is an exponential upper bound in worst-case time complexity, but the algorithm appears to perform almost linearly in practice. We cannot guarantee that a computed solution will be found in any case, but why not try?

Our work is, to the best of our knowledge, the very first successful mechanization of the homomorphism extraction problem.

We can imagine improvements of the calculus, such as rewriting modulo associativity or a more powerful ancestor decomposition rule which is able to "jump" over a number of non-top rewrite steps, together with an appropriately weakened application condition. Of course, the next step to do is an implementation of the algorithm.

# References


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

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

[3] S. Gorlatch and H. Bischof. Formal derivation of divide-and-conquer programs: A case study in the multidimensional FFT's. In D. Mery, editor, *Formal Methods for Parallel Programming: Theory and Applications. Workshop at IPPS'97*, pages 80–94, 1997.

[4] C. A. Herrmann and C. Lengauer. Parallelization of divide-and-conquer by translation to nested loops. Technical Report MIP-9705, Fakultät für Mathematik und Informatik, Universität Passau, March 1997.

[5] M. Cole. Algorithmic skeletons: A structured approach to the management of parallel computation. Ph.D. Thesis. Technical Report CST-56-88, Department of Computer Science, University of Edinburgh, 1988.

[6] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[7] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In H. Kuchen and D. Swierstra, editors, *Programming languages: Implementation, Logics and Programs*, Lecture Notes in Computer Science 1140, pages 274–288. Springer-Verlag, 1996.

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

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

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

[11] S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Universität Passau, August 1995.

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

[13] B. Heinz. Lemma discovery by anti-unification of regular sorts. Technical Report 94-21, TU Berlin, May 1994.

[14] Gordon D. Plotkin. Lattice-theoretic properties of subsumption. Technical Report Memo MIP-R-77, Univ. Edinburgh, UK, 1970.

[15] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 257–321. MIT Press, 1990.

[16] Steffen Lange. Towards a set of inference rules for solving divergence in Knuth-Bendix completion. In Klaus P. Jantke, editor, *Proc. Analogical and Inductive Inference*, pages 305–316. LNCS 397, 1989.

[17] Muffy Thomas and Phil Watson. Solving divergence in Knuth-Bendix completion by enriching signatures. *Theoretical Computer Science*, 112(1):145–185, 1993.

[18] Jean-Pierre Jouannaud. Syntactic theories. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, pages 15–25, Banská Bystrica, 1990. LNCS 452.

[19] Hubert Comon, Marianne Haberstrau, and Jean-Pierre Jouannaud. Syntacticness, cycle-syntacticness, and shallow theories. *Inform. and Computation*, 111:154–191, 1994.

[20] Jan Willem Klop. Term rewriting systems. In Samson Abramsky, Dov M. Gabbay, and Tom Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, chapter 6, pages 1–116. Clarendon Press, Oxford, UK, 1992.

[21] Leo Bachmair. *Canonical Equational Proofs.* Research Notes in Theoretical Computer Science. Wiley and Sons, 1989.

[22] Ulrich Fraus and Heinrich Hußmann. Term induction proofs by a generalization of narrowing. In C. Rattray and R. G. Clark, editors, *The Unified Computation Laboratory — Unifying Frameworks, Theories and Tools*, Oxford, UK, 1992. Clarendon Press.

[23] Alfons Geser. Mechanized inductive proof of properties of a simple code optimizer. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *Proc. 6th Theory and Practice in Software Development (TAPSOFT)*, LNCS 915, pages 605–619. Springer, 1995.

[24] Alfons Geser and Sergei Gorlatch. Parallelizing functinal programs by term rewriting. Technical Report MIP-97??, Universität Passau, Germany, 1997.