J. Functional Programming **9** (6): 649–673, November 1999. Printed in the United Kingdom
© 1999 Cambridge University Press

Parallelizing functional programs by generalization[*]

ALFONS GESER[1] and SERGEI GORLATCH[2]
[1] *Wilhelm-Schickard-Institut für Informatik, University, D-72076 Tübingen*
[2] *University of Passau, D–94030 Passau, Germany*

Abstract

List homomorphisms are functions that are parallelizable using the divide-and-conquer paradigm. We study the problem of finding homomorphic representations of functions in the Bird–Meertens constructive theory of lists, by means of term rewriting and theorem proving techniques. 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. The inductive claim is provable automatically. We illustrate the method and the procedure by the systematic parallelization of the *scan*-function (parallel prefix) and of the maximum segment sum problem.

Capsule Review

This paper presents a rewriting technique that automates the construction of concat-list homomorphisms from cons- and snoc-list honomorphism (Bird's third homomorphism theorem). Examples worked through by hand are given. Implementation of the technique described here would provide a powerful, but low complexity, way to derive complex programs. The result is particularly important because concat-list homomorphisms have a natural parallel implementation.

1 Introduction

This paper applies term rewriting and theorem proving techniques to formal reasoning about functional programs, specifically to program parallelization. It builds upon a rich body of research done in the framework of the Bird-Meertens formalism (BMF) (Bird, 1988; Skillicorn, 1994). Parallelism is tackled in BMF via the notion of *homomorphism* which captures a data-parallel form of the divide-and-conquer paradigm. A classification of divide-and-conquer forms suitable for static parallelization is proposed in Gorlatch and Bischof (1997).

[*] A preliminary version of this paper was presented at the *ALP'97 Conference*, Southampton, UK.

To use homomorphisms in the design of parallel programs, two tasks must be solved: (1) *Extraction*: for a given function find its representation as a homomorphism or adjust it to a homomorphic form; (2) *Implementation*: for different classes of homomorphisms, find an efficient way of implementing them on parallel machines. For both tasks, we aim at a systematic approach which would lead to a practically relevant parallel programming methodology (Gorlatch, 1996*a*; Gorlatch, 1996*b*; Gorlatch and Bischof, 1997). A systematic extraction method, proposed by Gorlatch (1996*b*), proceeds by generalizing two sequential representations of the function: on the `cons` and `snoc` lists. This so-called CS method (for 'Cons and Snoc') has proven to be powerful enough for a class of *almost-homomorphisms* which include famous 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, to 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.

The paper is structured as follows. Section 2 introduces the BMF notation and the notion of homomorphism. In section 3, we formulate the CS method of homomorphism extraction through term generalization. A formal embedding of the BMF theory of lists into the term rewriting framework is done in section 4. In section 5, we derive a generalization calculus for the theory of lists and formulate an algorithm of generalization. The presentation is illustrated by a running example – the *scan* function, also known as the parallel prefix (Blelloch, 1989). 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, in section 6 we apply the proposed generalization algorithm which successfully extracts the homomorphic form of scan. In section 7, we demonstrate the application of the CS method and the generalization algorithm on a more demanding case study – the famous MSS (maximum segment sum) problem. Finally, section 8 studies the termination property of the presented generalization algorithm.

2 BMF and homomorphisms

We restrict ourselves to non-empty lists, which can be constructed starting from singletons $[a]$ via list concatenation $+\!\!+$. The BMF expressions are built using functional composition denoted by $\circ$, and two second-order functions:

$map\, f$:	*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:

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

hence, the value of h on a list depends in a certain 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.

Theorem 1 (*Bird, 1988*)
Function h is a homomorphism iff there exists a binary associative *combine operator* $\circledast$ such that

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

where f is defined by $f\,(a) \;=\; h\,([a])$.

The theorem provides a standard parallelization pattern for all homomorphisms as a composition of two stages. A further refinement of (2) into coarse-grain SPMD parallel programs is studied elsewhere (Gorlatch, 1996*a*; Gorlatch and Bischof, 1997). Thus, 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 four 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. (Blelloch, 1989); 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 $((last\ S_1)\odot)$, which can be *map*ped.

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 2.2
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, so many functions are either *lw* or *rw* or both. The following theorem combines the so-called second and third homomorphism theorems considered folk theorems in the BMF community.

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

Unfortunately, as pointed out by Barnard *et al.* (1991) and Gibbons (1996), 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 $\otimes$.

Definition 2.3
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.*

Theorem 3
If function h is a homomorphism then h is both *lh* and *rh* with the same combine operator. If function h is *lh* or *rh*, and the combine operator is associative, then h is a homomorphism with this combine operator.

Gorlatch (1995) proves 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 is successful (e.g. for the function which computes the length of a list (Gorlatch, 1996*b*)), but already for *scan* it does not work:

Example 1 (*Continued; extraction for Scan*)
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. We have to guess which of possible substitutes for a: $scan\,[a]$, $head\,(scan\,(\odot)\,[a])$, $last\,(scan\,(\odot)\,[a])$, etc. should be used. We run into a similar problem for a `snoc` definition:

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

We believe we have shown that the problem of extracting a homomorphism is nontrivial. For the *scan* function, it even led to errors in published parallel algorithms and required a formal correctness proof (O'Donnell, 1994).

3 CS method: Extraction by generalization

In the previous section, we have seen some unsuccessful attempts to arrive at an associative combine operator, starting from either a `cons` or a `snoc` representation of a function. The idea of the proposed CS method (for 'Cons and Snoc') is to take both representations and *generalize* them.

Definition 3.1

A term t_G is called a *generalizer* of terms t_1 and t_2 in the equational theory E (E-generalizer) if there are substitutions σ_1, σ_2, such that $t_G.\sigma_1 \leftrightarrow_E^* t_1$ and $t_G.\sigma_2 \leftrightarrow_E^* t_2$.

Here, $\leftrightarrow_E^*$ denotes the semantic equality (conversion relation) in the equational theory E, and $t.\sigma$ denotes the result of applying substitution σ to term t. Generalization is the dual to unification and is sometimes also called 'anti-unification' (Heinz, 1994).

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

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

The homomorphism extraction problem can be formally specified as follows:

Given: A `cons` and a `snoc` definition of h:

$$\begin{array}{ccc} h([a]) \leftrightarrow t_B & \text{and} & h([b]) \leftrightarrow t'_B \\ h(a \cdot\!: y) \leftrightarrow t_C & & h(x :\!\cdot\, b) \leftrightarrow t_S \end{array}$$

Wanted: A definition $u \circledast v \leftrightarrow t_H$, such that $h(x +\!\!+\, y) \leftrightarrow h(x) \circledast h(y)$.

Since h is a homomorphism, then according to Theorem 3, it is both *lh* and *rh* with combine operator $\circledast$, so that we can rewrite the definitions into the following:

$$t_B \circledast h(y) \leftrightarrow t_C \qquad \text{and} \qquad h(x) \circledast t'_B \leftrightarrow t_S \tag{7}$$

At the same time, 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 semantically equal variants of all `cons` and `snoc` definitions of h, respectively.

Figure 1(a) illustrates the relations between the introduced terms t_H, t_C, t_S, t_L and t_R. Dotted arrows indicate application of substitutions; solid arrows indicate conversion steps. Each substitution is applied to *two* terms – this is a simultaneous generalization problem. To work with the established notion of generalizer, we introduce a fresh binary function symbol, $\rightharpoonup$, and model each pair ($s \leftrightarrow t$) of terms as a single term $s \rightharpoonup t$, called a *rule term*. With this encoding we get the view of Figure 1(b).

The following theorem states that a generalization of the rule terms built of (7) yields term t_H from (6) – the wanted piece of the definition of h as a homomorphism.

Theorem 4

Let E be the theory of lists and let $t'_B = t_B.\{a \mapsto b\}$. If 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. $\sigma_1 = \{u \mapsto t_B, v \mapsto h(y)\}$ 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.

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

(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

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

Proof
The first claim follows by Theorem 5(3) in section 4. For the second claim, one shows $h(a \mathbin{\cdot\!:} y) \leftrightarrow t_C \leftrightarrow t_B \circledast h(y) \leftrightarrow h([a]) \circledast h(y)$ and $h(x \mathbin{:\!\cdot} b) \leftrightarrow t_S \leftrightarrow h(x) \circledast t'_B \leftrightarrow h(x) \circledast h([b])$. The third claim is equivalent to it by Theorem 3. $\square$

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

The generalization in Theorem 4 is called the *CS-generalization*: it is the key step of the following CS method of homomorphism extraction.

The 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. A successful CS-generalization, applied to the rule terms $t_B \circledast h(y) \rightharpoonup t_C$ and $h(x) \circledast t'_B \rightharpoonup t_S$, yields a rule term $u \circledast v \rightharpoonup t_H$.
3. If associativity of $\circledast$ defined by t_H can be proven inductively then, by Theorem 4, $\circledast$ is the desired combine operator.

Example 1 (Continued)
For *scan*, the rule terms follow from (4),(5):

$$[a] \circledast \mathit{scan}\,(\odot)\,y \rightharpoonup a \mathbin{\cdot\!:} (\mathit{map}\,(a\,\odot)\,(\mathit{scan}\,(\odot)\,y))$$
$$\mathit{scan}\,(\odot)\,x \circledast [b] \rightharpoonup (\mathit{scan}\,(\odot)\,x) \mathbin{:\!\cdot} (\mathit{last}\,(\mathit{scan}\,(\odot)\,x) \odot b)$$

As demonstrated further in section 6, their CS-generalization yields the desired combine operator of scan.

To apply the CS method practically, the generalization step of the method must be mechanized, i.e. an algorithm is required that yields the most special generalizer of two terms.

4 Expressing the CS method in term rewriting

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 $\sigma_2 =_{\text{def}} \{x_0 \mapsto t_2\}$. So it is obvious that people prefer the *most special* generalizer, provided there is one. In this respect, generalization is the dual to *unification* and is sometimes also called 'anti-unification' (Heinz, 1994). Plotkin has closely studied the special-case relation between terms (Plotkin, 1970). In the case where E is empty, there are most general syntactic unifiers and most special syntactic generalizers.

In contrast to unification, properties and methods for generalization for non-empty E have not received enough attention in the literature. A few generalization methods have been exercised for use in inductive theorem proving, i.e. the systematic construction of a term rewriting system of counting functions on the basis of an effective enumeration of elements (Lange, 1989; Smith, 1993). 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. We start from Comon *et al.* (1994) and Jouannaud (1990), where a *resolvant* presentation of the equational theory to get a generalization algorithm is used; our generalization calculus differs basically by the use of rewrite derivations instead of conversions.

We will adopt term rewriting methods for this step, so let us first embed a fragment of the Bird-Meertens theory of lists in term rewriting theory.

To this end, we henceforth restrict ourselves to first-order terms whence we suppress higher order parameters, and consider a fixed associative operation $\odot$. In our running example, we replace $\mathit{scan}\,(\odot)\,x$ by $\mathit{scn}(x)$, and $\mathit{map}\,(a\,\odot)\,(x)$ by $\mathit{mp}(a, x)$.

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'. Relation $\leftrightarrow_R^*$ is the equivalence closure of $\to_R$, i.e. the smallest binary relation on terms that contains $\to_R$ 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.

We start with the oriented associativity rules of $\odot$ and $+\!\!+$:

$$a \odot (b \odot c) \to (a \odot b) \odot c \qquad (A_{\odot})$$

$$x +\!\!+ (y +\!\!+ z) \to (x +\!\!+ y) +\!\!+ z \qquad (A_{+\!\!+})$$

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

$$a \cdot\!: y \to [a] +\!\!\!+\, y \qquad \text{and} \qquad x :\!\cdot\, b \to x +\!\!\!+ [b]$$

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 on lists are homomorphisms and that the extraction has already been 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 $+\!\!\!+$ definitions of h, respectively:

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

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

Example 1 (Continued)
Function *scn* calls functions *mp* and *last*. So, R_{base} becomes

$$last[a] \to a \tag{8}$$

$$last(x +\!\!\!+ y) \to last\; y \tag{9}$$

$$mp(a, [b]) \to [a \odot b] \tag{10}$$

$$mp(a, x +\!\!\!+ y) \to mp(a, x) +\!\!\!+ mp(a, y) \tag{11}$$

Next we get $t_B = [a]$, and accordingly, $t'_B = [b]$, next $t_C = [a] +\!\!\!+ mp(a, scn(y))$, and $t_S = scn(x) +\!\!\!+ [last(scn(x)) \odot b]$.

$$\begin{aligned} R_0 &= R \cup \left\{ \begin{array}{r} scn([a]) \to [a] \\ scn(x +\!\!\!+ y) \to scn(x) \circledast scn(y) \end{array} \right. \\ R_{cons} &= R_0 \cup \{[a] \circledast scn(y) \to [a] +\!\!\!+ mp(a, scn(y))\} \\ R_{snoc} &= R_0 \cup \{scn(x) \circledast [b] \to scn(x) +\!\!\!+ [last(scn(x)) \odot b]\} \end{aligned}$$

We are aiming at t_H such that R_{conc} defines the same function h as do R_{cons} and R_{snoc}.

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 σ 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 (e.g. see Bachmair, 1989).

Theorem 5 (*Reliability*)
Let the term rewriting systems R, R_{cons}, R_{snoc}, and R_{conc} be defined as above. If the two rule terms $t_B \circledast h(y) \rightharpoonup t_C$ and $h(x) \circledast t'_B \rightharpoonup t_S$ have an R-generalizer, $u \circledast v \rightharpoonup t_H$, w.r.t. the substitutions $\sigma_1 = \{u \mapsto t_B, v \mapsto h(y)\}$ and $\sigma_2 = \{u \mapsto h(x), v \mapsto t'_B\}$, then the following properties hold:

1. Relations $\leftrightarrow_{R_{cons}}^*$ and $\leftrightarrow_{R_{snoc}}^*$ are included in 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 $t_B \circledast h(y) \rightarrow t_C$ in R_{cons} not in R_{conc} can be replaced by a conversion $(u \circledast v).\sigma_1 \rightarrow_{R_{conc}} t_H.\sigma_1 \leftrightarrow_R^* t_C$ 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 $t \rightarrow_{R_{conc}}^* \leftrightarrow_R^* \leftarrow_{R_{conc}}^* t'$ for all ground terms t, t'. Specifically,

$$\begin{tikzcd}
h((t_1 \mathbin{+\!\!+} t_2) \mathbin{+\!\!+} t_3) \arrow[r, "R_{conc}"] \arrow[d, "R_{conc}"'] & h(t_1 \mathbin{+\!\!+} t_2) \circledast h(t_3) \arrow[r] & (h(t_1) \circledast h(t_2)) \circledast h(t_3) \arrow[d, leftrightarrow, "R"', "*"] \\
h(t_1 \mathbin{+\!\!+} (t_2 \mathbin{+\!\!+} t_3)) \arrow[r] & h(t_1) \circledast h(t_2 \mathbin{+\!\!+} t_3) \arrow[r] & h(t_1) \circledast (h(t_2) \circledast h(t_3))
\end{tikzcd}$$

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.

First, we assume associativity of $\circledast$ in the inductive theory of R_{conc}. Then it follows that the equation

$$t_H\,.\,\{u \mapsto t_H, v \mapsto w\} \;\leftrightarrow\; t_H\,.\,\{v \mapsto t_H\,.\,\{u \mapsto v, v \mapsto w\}\}\;, \tag{12}$$

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 (12) is also in the inductive theory of R.

Finally, we show that the only rule in $R_{conc} \setminus R_{cons}$ is also an element of 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). □

In other words, a generalizer of a certain shape and an inductive proof provide a solution to the homomorphism extraction problem.

Most special generalizers need not exist, as the following example shows.

Example 2
Given a term rewriting system R for the doubling function d on non-negative integers,

$$d(0) \rightarrow 0$$
$$d(s(x)) \rightarrow s(s(d(x)))$$

where s denotes the successor function, we may pose the generalization problem

$$\text{If } f(u'_1,\dots,u'_m) \xrightarrow[R]{\varepsilon\ *} u_i \text{ and } f(v'_1,\dots,v'_m) \xrightarrow[R]{\varepsilon\ *} v_i \text{ then} \qquad (13)$$

$$\frac{\sigma_1\|\{x_i \mapsto u_i\} \qquad \sigma_2\|\{x_i \mapsto v_i\} \qquad t_0}{\sigma_1\|\tau_1 \qquad \sigma_2\|\tau_2 \qquad t_0.\{x_i \mapsto f(x'_1,\dots,x'_m)\}}$$

where $\tau_1 = \{x'_1 \mapsto u'_1,\dots,x'_m \mapsto u'_m\}$ and $\tau_2 = \{x'_1 \mapsto v'_1,\dots,x'_m \mapsto v'_m\}$.

Fig. 2. 'Ancestor decomposition' rule.

$$\frac{\sigma_1\|\{x_i \mapsto u, x_j \mapsto u\} \qquad \sigma_2\|\{x_i \mapsto v, x_j \mapsto v\} \qquad t_0}{\sigma_1\|\{x_i \mapsto u\} \qquad \sigma_2\|\{x_i \mapsto v\} \qquad t_0.\{x_j \mapsto x_i\}}$$

Fig. 3. 'Agreement' rule.

for $t_1 = s(s(0))$ and $t_2 = s(s(s(s(0))))$ in R. We get two, incomparable, solutions: $t_0 = s(s(y))$ with $\sigma_1 = \{y \mapsto 0\}$ and $\sigma_2 = \{y \mapsto s(s(0))\}$, as opposed to $t_0 = d(z)$ with $\sigma_1 = \{z \mapsto s(0)\}$ and $\sigma_2 = \{z \mapsto s(s(0))\}$. For the first solution, no conversion is needed, for the second, we have the two conversions

$$t_0.\sigma_1 = d(s(0)) \to_R s(s(d(0))) \to_R s(s(0)) = t_1, \qquad \text{and}$$

$$t_0.\sigma_2 = d(s(s(0))) \to_R s(s(d(s(0)))) \to_R s(s(s(s(d(0)))))$$
$$\to_R s(s(s(s(0)))) = t_2$$

For the same reason the calculus usually is not confluent (Huet, 1980). The example also shows that it is reasonable to assume that the conversion is a *rewrite derivation* from $t_0.\sigma_i$ to t_i; we will exploit that in our algorithm.

5 A calculus for generalization

Following a good custom, we introduce our algorithm for generalization by means of a calculus. The objects of our generalization calculus are *configurations* consisting of three components: $(\sigma_1, \sigma_2, t_0)$, maintaining the invariant that t_0 is a generalizer of t_1 and t_2 via the substitutions σ_1 and σ_2, respectively. Starting from the initial configuration $(\{x_0 \mapsto t_1\}, \{x_0 \mapsto t_2\}, x_0)$, 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 assignment of σ_1 and σ_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.

We introduce two inference rules: *ancestor decomposition* and *agreement*, shown in Figures 2 and 3, respectively. The formula $t \xrightarrow[R]{\varepsilon\ *} t'$ means that t rewrites in zero or more steps to t' where every rewrite step takes place at the root position, ε, of t. For substitutions $\sigma = \{x_1 \mapsto s_1,\dots,x_m \mapsto s_m\}$, $\tau = \{y_1 \mapsto t_1,\dots,y_n \mapsto t_n\}$ for which $x_i \neq y_j$ for all i, j, their disjoint union is defined by $\sigma\|\tau = \{x_1 \mapsto s_1,\dots,x_m \mapsto s_m, y_1 \mapsto t_1,\dots,y_n \mapsto t_n\}$.

The ancestor decomposition rule in Figure 2 reconstructs a common top symbol of a pair of terms, potentially after a few reverse rewrite steps at the top. First, a common variable, x_i, in the domain of the two substitutions is selected. If the corresponding right-hand sides, u_i and v_i, have a common root function symbol, f, then the rule splits $x_i \mapsto f(u'_1, \ldots, u'_m)$ into $x_i \mapsto f(x'_1, \ldots, x'_m)$ and $x'_j \mapsto u'_j$, where x'_j is a fresh variable 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 assignment, $x_i \mapsto f(x'_1, \ldots, x'_m)$, is 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 (13).

The 'agreement' rule in Figure 3 joins variables that map to the same term. Mappings $x_i \mapsto u$ and $x_j \mapsto u$ with common right-hand sides in the first substitution can be split into $x_j \mapsto x_i$ and $x_i \mapsto u$. Likewise if, there are $x_i \mapsto v$ and $x_j \mapsto v$ in the second substitution then they are split into $x_j \mapsto x_i$ and $x_i \mapsto v$. The common assignment, $x_j \mapsto x_i$, of the two is transferred to t_0.

Occasionally we will justify an application of the ancestor decomposition rule by a remark of the form '`AncDec` for x_i; f; $t \xrightarrow[R]{\varepsilon}{}^{*} t'$', or just '`AncDec` for x_i; f' if the rewrite derivation $t \xrightarrow[R]{\varepsilon}{}^{*} t'$ 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 6 (*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 assumptions there is a completeness result as well. However, for practical applicability we will have to strengthen Condition (13) of the ancestor decomposition rule such that completeness is lost anyway. So we leave the completeness question open.

6 Generalization for scan

To turn our generalization calculus into an algorithm, one has to decide on a rule application strategy, and prove its termination. We adopt the strategy to prefer the 'agreement' rule, and then 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 rewrites that justifies Condition (13) at index i. We are going to study the termination property of this algorithm in section 8, but first let us see how it works.

For an arbitrary function h, the generalization process starts from the configuration with the rule terms given by $t_B \circledast h(y) \rightharpoonup t_C$ and $h(x) \circledast t'_B \rightharpoonup t_S$. The first two steps in the generalization are always the same, independently of the particular function h:

$$\{x_0 \mapsto (t_B \circledast h(y) \rightharpoonup t_C)\} \qquad \{x_0 \mapsto (h(x) \circledast t'_B \rightharpoonup t_S)\} \qquad x_0$$

$$\vdash \quad (\textsf{AncDec for } x_0;\ \rightharpoonup)$$

$$\left\{\begin{array}{c} x'_1 \mapsto t_B \circledast h(y) \\ x'_2 \mapsto t_C \end{array}\right\} \qquad \left\{\begin{array}{c} x'_1 \mapsto h(x) \circledast t'_B \\ x'_2 \mapsto t_S \end{array}\right\} \qquad x'_1 \rightharpoonup x'_2$$

$$\vdash \quad (\textsf{AncDec for } x'_1;\ \circledast)$$

$$\left\{\begin{array}{c} x'_3 \mapsto t_B \\ x'_4 \mapsto h(y) \\ x'_2 \mapsto t_C \end{array}\right\} \qquad \left\{\begin{array}{c} x'_3 \mapsto h(x) \\ x'_4 \mapsto t'_B \\ x'_2 \mapsto t_S \end{array}\right\} \qquad x'_3 \circledast x'_4 \rightharpoonup x_2$$

Here, x'_1, x'_2, x'_3, x'_4 are fresh variables introduced by the two applications of the ancestor decomposition rule. Since the two inference steps shown above have to be done for every function, we omit them in the sequel, and start generalization for an arbitrary function h from the following configuration, which is obtained from the latter by simple variable renaming:

$$\left\{\begin{array}{c} x_1 \mapsto t_B \\ x_2 \mapsto h(y) \\ x_3 \mapsto t_C \end{array}\right\} \qquad \left\{\begin{array}{c} x_1 \mapsto h(x) \\ x_2 \mapsto t'_B \\ x_3 \mapsto t_S \end{array}\right\} \qquad x_1 \circledast x_2 \rightharpoonup x_3 \tag{14}$$

Example 1 (*Continued*)

Now let us study our scan example to see how our calculus operates. Here we have, according to (4)–(5):

$$\begin{aligned} t_C &= ([a] \circledast scn(y) \rightharpoonup [a] \mathbin{+\!\!+} mp(a, scn(y))) \\ t_S &= (scn(x) \circledast [b] \rightharpoonup scn(x) \mathbin{+\!\!+} [last(scn(x)) \odot b]) \\ t_B &= [a] \\ t'_B &= [b] \end{aligned}$$

Generalization proceeds as follows:

$$\left\{\begin{array}{c} x_1 \mapsto [a] \\ x_2 \mapsto scn(y) \\ x_3 \mapsto [a] +\!\!+ \\ \quad mp(a, scn(y))) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto scn(x) \\ x_2 \mapsto [b] \\ x_3 \mapsto scn(x) +\!\!+ \\ \quad [last(scn(x)) \odot b] \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_3$$

$\vdash$ (AncDec for x_3; $+\!\!+$)

$$\left\{\begin{array}{c} x_1 \mapsto [a] \\ x_2 \mapsto scn(y) \\ x_4 \mapsto [a] \\ x_5 \mapsto mp(a, scn(y))) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto scn(x) \\ x_2 \mapsto [b] \\ x_4 \mapsto scn(x) \\ x_5 \mapsto [last(scn(x)) \odot b] \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_4 +\!\!+ x_5$$

$\vdash$ (Agr for x_1, x_4)

$$\left\{\begin{array}{c} x_1 \mapsto [a] \\ x_2 \mapsto scn(y) \\ x_5 \mapsto mp(a, scn(y))) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto scn(x) \\ x_2 \mapsto [b] \\ x_5 \mapsto [last(scn(x)) \odot b] \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_1 +\!\!+ x_5$$

$\vdash$ (AncDec for x_5; mp; $mp(last(scn(x)), [b]) \xrightarrow[(10)]{\varepsilon\quad *} [last(scn(x)) \odot b]$)

$$\left\{\begin{array}{c} x_1 \mapsto [a] \\ x_2 \mapsto scn(y) \\ x_6 \mapsto a \\ x_7 \mapsto scn(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto scn(x) \\ x_2 \mapsto [b] \\ x_6 \mapsto last(scn(x)) \\ x_7 \mapsto [b] \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_1 +\!\!+ mp(x_6, x_7)$$

$\vdash$ (Agr for x_2, x_7)

$$\left\{\begin{array}{c} x_1 \mapsto [a] \\ x_2 \mapsto scn(y) \\ x_6 \mapsto a \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto scn(x) \\ x_2 \mapsto [b] \\ x_6 \mapsto last(scn(x)) \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_1 +\!\!+ mp(x_6, x_2)$$

$\vdash$ (AncDec for x_6; $last$; $last([a]) \xrightarrow[(8)]{\varepsilon\quad *} a$)

$$\left\{\begin{array}{c} x_1 \mapsto [a] \\ x_2 \mapsto scn(y) \\ x_8 \mapsto [a] \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto scn(x) \\ x_2 \mapsto [b] \\ x_8 \mapsto scn(x) \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_1 +\!\!+ mp(last(x_8), x_2)$$

$\vdash$ (Agr for x_1, x_8)

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

It remains to prove the associativity of the obtained operator $\circledast$:

$$u \circledast v \leftrightarrow u \ +\!\!+ \ map\,(last\,(u)\,\odot)\; v \tag{15}$$

The inductive proof of associativity may be carried out by a mechanized inductive theorem prover. In our *scn* example, we have used the semi-automatic inductive prover TIP (Fraus and Hußmann, 1992; Geser, 1995) to produce a proof of associativity of $\circledast$ based on the lemmas

$$last(mp(a, x)) \to a \odot last\, x \qquad \text{and} \qquad mp(a, mp(b, x)) \to mp(a \odot b, x)$$

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

Therefore, *scan* is a homomorphism in the sense of (2), with $\circledast$ defined by (15) and $f = [.]$, where function $[.]$ creates singleton list of an element.

7 Case study: The MSS problem

Many practical non-homomorphic functions are so-called *almost-homomorphisms* (name coined by M. Cole): they are convertible to a composition of a homomorphism and some adjusting function.

Actually, every function h can be tupled together with the identity function, resulting in the function $g = \langle h, id \rangle$. Obviously, such g is a homomorphism: $g\,(x +\!\!\!+\, y) = g\,x \circledast g\,y$, where $(u, x) \circledast (v, y) = \big(\, h\,(x +\!\!\!+\, y), x +\!\!\!+\, y \,\big)$. The original function is computed from g by projection, $h = \pi_1 \circ g$, where π_1 yields the first 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 operator $\circledast$ reveals the snag: it does not make use of the computed values, u and v, and computes function h from scratch!

Fortunately, there are also examples where a conversion to a 'true' tuple homomorphism exists. Cole reports several elegant case studies (Cole, 1994), where the main remaining difficulty is to guess which *auxiliary functions* must be included in a tuple and to find the combine operator. Usually, this requires a lot of ingenuity from the developer, hence a more systematic approach is desired. Cole says:

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

We will demonstrate that the CS method allows us to systematically construct almost-homomorphisms, known from the literature.

We consider solution of the *maximum segment sum* (*mss*) problem – a *programming pearl* (Bentley, 1984), studied by many authors (Bird, 1988; Cole, 1994; Skillicorn, 1994; Smith, 1987). Given a list of integers, function *mss* finds the contiguous, non-empty list segment whose members have the largest sum among all such segments and returns this sum. For example, in the notation of Cole (1994):

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

where the result is contributed by the segment $[2, -1, 6]$.

Let us apply the CS method to the MSS problem. First, we have to express function *mss* over `cons` lists. For some element a and list y, it may well be the case that $mss\,(a \mathbin{\cdot\!:} y) = a \uparrow mss\,(y)$, where $\uparrow$ returns the larger of its two arguments. But we must not overlook the possibility that the true segment of interest includes both a and some initial segment of y. To cater for that, we have to introduce auxiliary function *mis* which yields the sum of the *maximum initial segment*:

$$mss\,(a \mathbin{\cdot\!:} y) \;=\; a \uparrow (a + mis\,(y)) \uparrow mss\,(y) \tag{16}$$

The next step of the CS method, `snoc` definition, requires the introduction of auxiliary function *mcs*, yielding the sum of the *maximum concluding segment*. The obtained `cons` definition of *mss* is as follows:

$$mss\,(x :\cdot\, b) \;=\; mss\,(x) \uparrow (mcs\,(x) + b) \uparrow b \tag{17}$$

The value of *mss* on a singleton list is the element itself:

$$mss\,[a] \;=\; a \tag{18}$$

To get a closed definition of *mss*, we need to define the auxiliary functions, *mis* and *mcs*, on both `cons` and `snoc` lists. Trying to find these definitions, we see that the concluding segment of $a \cdot\!: y$ may be the whole list, so we need its sum, which no (combination) of the functions from the triple can yield. Therefore, we have to introduce one more auxiliary function, *ts* (for *total sum*).

As the result, we arrive at the quadruple of functions: $\langle mss, mis, mcs, ts\rangle$. The functional program for *mss* consists now of definitions of four functions of the quadruple:

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

Since functions in the quadruple are depending on each other, they should be generalized together. To keep the presentation clear enough, we demonstrate the generalization process in parts; and then comment on how these parts work together in the generalization process.

Let us start with the simple case of function *ts* which does not depend upon any other function:

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto ts\,(y) \\ x_3 \mapsto a + ts\,(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto ts\,(x) \\ x_2 \mapsto b \\ x_3 \mapsto ts\,(x) + b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_3$$

$\vdash$ (AncDec for x_3; $+$)

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto ts\,(y) \\ x_4 \mapsto a \\ x_5 \mapsto ts\,(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto ts\,(x) \\ x_2 \mapsto b \\ x_4 \mapsto ts\,(x) \\ x_5 \mapsto b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_4 + x_5$$

$\vdash$ (Agr for x_1, x_4)

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto ts\,(y) \\ x_5 \mapsto ts\,(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto ts\,(x) \\ x_2 \mapsto b \\ x_5 \mapsto b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_1 + x_5$$

$\vdash$ (Agr for x_2, x_5)

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto ts\,(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto ts\,(x) \\ x_2 \mapsto b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_1 + x_2$$

Thus, *ts* is a homomorphism with $+$ as the combine operator.

Let us now consider the most complicated function in the quadruple, *mss*. Like *mis* and *mcs*, it depends on other functions. We start to generalize *mss* as usual, and then show how this dependency will be taken care of.

The generalization algorithm proceeds for *mss* as shown in Figure 4.

We have arrived at a point where every applicable inference rule misses the target, but the desired final form is still not reached. The reason is that two other functions *mis* and *mcs*, are used in the definition of *mss*. To cater for that, we modify the generalization algorithm in a natural way: in addition to variables x_1, x_2 of (14) for the function under generalization h, we introduce a pair of variables x_1^g, x_2^g for each function g, such that g is used in the definition of h. In case of *mss*, we enrich the configuration obtained in Figure 4 by variables for *mis* and *mcs*, and proceed with generalization as shown in Figure 5.

The generalization is successful, and the right-hand side of the third component of the obtained configuration yields the combine operator of the function *mss* from the quadruple:

$$mss\,(x +\!\!\!+\, y) \;=\; mss\,(x) \uparrow (mcs\,(x) + mis\,(y)) \uparrow mss\,(y) \tag{19}$$

The generalization for the remaining two auxiliary functions, *mis* and *mcs*, proceeds analogously, with an additional variable introduced for function *ts* which both of them depend upon. From the presented generalization for *ts* and *mss*, it should be clear that all four functions of the quadruple can be generalized together, with two variables x_1 and x_2 introduced for each of them.

The resulting combine operator for the quadruple is of the form:

$$\begin{array}{l} (mss\,(x), mis\,(x), mcs\,(x), ts\,(x)) \;\circledast\; (mss\,(y), mis\,(y), mcs\,(y), ts\,(y)) \;= \\ \quad \big(\, mss\,(x) \uparrow (mcs\,(x) + mis\,(y)) \uparrow mss\,(y)\,,\; mis\,(x) \uparrow (ts\,(x) + mis\,(y))\,, \\ \quad\; mcs\,(y) \uparrow (mcs\,(x) + ts\,(y))\,,\; ts\,(x) + ts\,(y)\,\big) \end{array} \tag{20}$$

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto mss\,(y) \\ x_3 \mapsto a \uparrow (a + mis\,(y)) \uparrow \\ mss\,(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto mss\,(x) \\ x_2 \mapsto b \\ x_3 \mapsto mss\,(x) \uparrow \\ (mcs\,(x) + b) \uparrow b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_3$$

$\vdash$ (AncDec for x_3; $\uparrow$)

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto mss\,(y) \\ x_4 \mapsto a \uparrow (a + mis\,(y)) \\ x_5 \mapsto mss(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto mss\,(x) \\ x_2 \mapsto b \\ x_4 \mapsto mss\,(x) \uparrow (mcs\,(x) + b) \\ x_5 \mapsto b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_4 \uparrow x_5$$

$\vdash$ (Agr for x_2, x_5)

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto mss\,(y) \\ x_4 \mapsto a \uparrow (a + mis\,(y)) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto mss\,(x) \\ x_2 \mapsto b \\ x_4 \mapsto mss\,(x) \uparrow (mcs\,(x) + b) \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_4 \uparrow x_2$$

$\vdash$ (AncDec for x_4; $\uparrow$)

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto mss\,(y) \\ x_6 \mapsto a \\ x_7 \mapsto a + mis\,(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto mss\,(x) \\ x_2 \mapsto b \\ x_6 \mapsto mss\,(x) \\ x_7 \mapsto mcs\,(x) + b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_6 \uparrow x_7 \uparrow x_2$$

$\vdash$ (Agr for x_1, x_6)

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto mss\,(y) \\ x_7 \mapsto a + mis\,(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto mss\,(x) \\ x_2 \mapsto b \\ x_7 \mapsto mcs\,(x) + b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_1 \uparrow x_7 \uparrow x_2$$

$\vdash$ (AncDec for x_7; $+$)

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto mss\,(y) \\ x_8 \mapsto a \\ x_9 \mapsto mis\,(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto mss\,(x) \\ x_2 \mapsto b \\ x_8 \mapsto mcs\,(x) \\ x_9 \mapsto b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_1 \uparrow (x_8 + x_9) \uparrow x_2$$

Fig. 4. Generalization for *mss*: first part.

Since $\circledast$ is associative, our quadruple is the homomorphism with $\circledast$ defined by (20) and f providing the result of the quadruple singleton:

$$f\ a \;=\; \langle mss, mis, mcs, ts \rangle\ [a] \;=\; (a\,,\, a\,,\, a\,,\, a) \tag{21}$$

The target function *mss* is therefore computable as follows:

$$mss \;=\; \pi_1 \circ red\,(\circledast) \circ (map\ f) \tag{22}$$

where $\circledast$ and f are defined by (20),(21), and π_1 yields the first component of a quadruple.

Let us estimate the parallel time complexity of the derived homomorphic algorithm (22) for the MSS problem. Since both function f and operator $\circledast$ require a constant number of communicated elements and executed operators, the total time on n processors is $O(\log n)$. The number of processors can be reduced to $n/\log n$ by simulating lower levels of the tree sequentially, based on Brent's theorem (Quinn, 1994). Therefore, the direct tree-like algorithm is both time and cost optimal.

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto mss\,(y) \\ x_8 \mapsto a \\ x_9 \mapsto mis\,(y) \\ x_1^{mis} \mapsto a \\ x_2^{mis} \mapsto mis\,(y) \\ x_1^{mcs} \mapsto a \\ x_2^{mcs} \mapsto mcs\,(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto mss\,(x) \\ x_2 \mapsto b \\ x_8 \mapsto mcs\,(x) \\ x_9 \mapsto b \\ x_1^{mis} \mapsto mis\,(x) \\ x_2^{mis} \mapsto b \\ x_1^{mcs} \mapsto mcs\,(x) \\ x_2^{mcs} \mapsto b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_1 \uparrow (x_8 + x_9) \uparrow x_2$$

$\vdash$ (Agr for x_1^{mcs}, x_8)

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto mss\,(y) \\ x_9 \mapsto mis\,(y) \\ x_1^{mis} \mapsto a \\ x_2^{mis} \mapsto mis\,(y) \\ x_1^{mcs} \mapsto a \\ x_2^{mcs} \mapsto mcs\,(y) \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto mss\,(x) \\ x_2 \mapsto b \\ x_9 \mapsto b \\ x_1^{mis} \mapsto mis\,(x) \\ x_2^{mis} \mapsto b \\ x_1^{mcs} \mapsto mcs\,(x) \\ x_2^{mcs} \mapsto b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_1 \uparrow (x_1^{mcs} + x_9) \uparrow x_2$$

$\vdash$ (Agr for x_2^{mis}, x_9)

$$\left\{\begin{array}{c} x_1 \mapsto a \\ x_2 \mapsto mss\,(y) \\ x_1^{mis} \mapsto a \\ x_2^{mis} \mapsto mis\,(y) \\ x_1^{mcs} \mapsto a \\ x_2^{mcs} \mapsto mcs\,y \end{array}\right\} \quad \left\{\begin{array}{c} x_1 \mapsto mss\,(x) \\ x_2 \mapsto b \\ x_1^{mis} \mapsto mis\,(x) \\ x_2^{mis} \mapsto b \\ x_1^{mcs} \mapsto mcs\,(x) \\ x_2^{mcs} \mapsto b \end{array}\right\} \quad x_1 \circledast x_2 \rightharpoonup x_1 \uparrow (x_1^{mcs} + x_2^{mis}) \uparrow x_2$$

Fig. 5. Generalization for *mss*: second part.

8 The termination property

Of course one would like the generalization procedure to terminate, i.e. be an algorithm. The procedure, however, does not always terminate.

To get on the safe side, we replace Condition (13) of the ancestor decomposition rule by:

$$f(u_1',\dots,u_m') \xrightarrow[R]{\varepsilon\ \ X} u_i, \qquad f(v_1',\dots,v_m') \xrightarrow[R]{\varepsilon\ \ Y} v_i, \qquad X + Y \leqslant 1 \tag{23}$$

Here $t \xrightarrow[R]{\varepsilon\ X} t'$ means that t rewrites in exactly X steps to t' where each step takes place at the root position, ε, of t. We call the resulting calculus the *restricted generalization calculus*.

In the restricted generalization calculus, as opposed to the unrestricted, the ancestor decomposition rule is finitely branching, and its applicable instances are computable, provided that R contains no *erasing* rules, i.e. rules $l \rightarrow r$ where l contains a variable not in r. For instance, Rule (9) is erasing: it has x on its left, but not on its right-hand side.

Condition (23) is not as hard in practice as it may seem. Observe, for instance, that the derivations for both scan and MSS are within the restricted generalization calculus. The restricted generalization calculus is not complete for the simple fact that many function definitions need recursion and so an unbounded number of steps in the ancestor decomposition rule.

Even in the restricted generalization calculus there are 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 $r > l$ 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 off the left hand side. Stripping off the top symbol can be considered a decrease, provided that the reverse rule application does not harm it.

Definition 8.1
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 $1 \leqslant j \leqslant n$.

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.

Proof
We show that for every inference step $(\sigma_1, \sigma_2, t_0) \vdash (\sigma_1', \sigma_2', t_0')$ in the calculus, we have $(\sigma_1, \sigma_2, t_0) \succ (\sigma_1', \sigma_2', t_0')$ where $\succ$ is a well-founded order on triples defined below.

Let $>$ be a well-founded order on pairs of terms, closed under substitution application, that extends the component-wise subterm order. Let $>_{mult}$ denote the extension of $>$ to finite multisets of pairs of terms. Finally, let $\succ$ be defined as follows:

$(\{x_1 \mapsto u_1, \ldots, x_m \mapsto u_m\}, \{x_1 \mapsto v_1, \ldots, x_m \mapsto v_m\}, t_0) \succ$
$(\{x'_1 \mapsto u'_1, \ldots, x'_n \mapsto u'_n\}, \{x'_1 \mapsto v'_1, \ldots, x'_n \mapsto v'_n\}, t'_0)$, if
$\{\{(u_1, v_1), \ldots, (u_m, v_m)\}\} >_{mult} \{\{(u'_1, v'_1), \ldots, (u'_n, v'_n)\}\}$

This leaves to prove that each inference step strictly decreases the triple w.r.t. $\succ$. For the 'agreement' rule, we get that a pair (u, v) is deleted from the multiset, an obvious decrease by definition of multiset extension.

For the application of the 'ancestor decomposition' rule, we note that a pair (u_i, v_i) is removed, and the pairs $(u'_1, v'_1), \ldots, (u'_m, v'_m)$ are added. Let R be reversely guarded by $>$. We claim that $(u_i, v_i) > (u'_j, v'_j)$ holds for all $1 \leqslant j \leqslant m$.

In the case where no step is made, we have $u_i = f(u'_1, \ldots, u'_m)$ and $v_i = f(v'_1, \ldots, v'_m)$ whence u'_j and v'_j are subterms of u_i and v_i, respectively. So $(u_i, v_i) > (u'_j, v'_j)$ by the component-wise subterm property of $>$.

If $f(u'_1, \ldots, u'_m) \xrightarrow[R]{\varepsilon} u_i$ and $v_i = f(v'_1, \ldots, v'_m)$ then reverse guardedness $(u_i, f(x_1, \ldots, x_m)) > (u'_j, x_j)$ entails

$$(u_i, v_i) = (u_i, f(v'_1, \ldots, v'_m)) > (u'_j, v'_j)$$

by closure under application of the substitution $\{x_1 \mapsto v'_1, \ldots, x_m \mapsto v'_m\}$.

The remaining case is symmetric. We conclude that in each case the resulting multiset is smaller w.r.t. $\succ$, so we are done. □

A term rewriting system that contains erasing rules is not reversely guarded. The term rewriting system R for our *scn* function is therefore *not* reversely guarded. As we argued for Condition (23), it is sensible to exclude erasing rules for computability reasons.

Example 1 (*Continued*)
The term rewriting system $R \setminus \{(9)\}$ 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 (8):

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

So every derivation in the restricted generalization calculus that refuses to use Rule (9) is finite: We obtain an algorithm that computes a finite set of generalizers.

By the branching at each application of the ancestor decomposition rule the complexity of the generalization algorithm may become exponential. But experience shows that branching is harmless: Rarely is there more than one branch.

Conclusion and discussion

Extraction of a homomorphism from a `cons` and a `snoc` definition of a function on lists is a systematic, powerful technique for designing parallel programs. Its advantage over the direct construction of a parallel solution is that, unlike the latter, it does not rely heavily on the user's intuition.

We have successfully applied two techniques from the area of term rewriting, *generalization* and *rewriting induction*, to attack the nontrivial, intriguing extraction problem. If the rules for `cons` and `snoc` are encoded in terms then each generalizer of a certain form encodes the definition of the homomorphism. This, provided that an associativity result can be proven inductively, e.g. by an automated inductive theorem prover. We proved reliability of this method to extract a homomorphism.

We have introduced a simple, sound calculus for generalization of terms. 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.

From a term rewriting point of view it is interesting to ask for properties such as termination or confluence of the term rewriting systems. For an introductory text to term rewriting see Klop (1992), for example. A term rewriting system terminates if all its rewrite derivations are finite. A term rewriting system is confluent if all pairs of branching rewrite derivations can be continued so as to join again. All term rewriting systems R_{cons}, R_{snoc}, R_{conc} in the *scan* example terminate, as can easily be proven by recursive path order. None of these term rewriting systems, however, is confluent. In other words, rewriting may compute different results, depending on the chosen derivation strategy. The system R_{conc} extended by associativity of $\circledast$ and the two inductively proven lemmas is confluent, by the Knuth/Bendix critical pair theorem. So the development of a homomorphic presentation yields a terminating, confluent term rewriting system of a peculiar form.

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 is to implement the algorithm.

Space precludes describing the related work in detail, so we just mention the most closely related approaches in both term rewriting and parallelizing functional programs. Little research has been done on the generalization or anti-unification of terms. We reported such work in section 4. The only reference available to us for combining theorem proving with the Bird–Meertens calculus is (Martin and Nipkow, 1990): the authors used the Larch Prover to automatically verify Bird's development of the sequential maximum segment sum algorithm. Their approach differs from ours in one essential point. The development of the parallel algorithm is still completely up to the programmer. In the area of parallelization, data parallel languages such as NESL (Blelloch, 1993) can also be viewed as implicitly parallelizing functions over lists. Our method can extend the power of compilers for such languages in revealing the available parallelism automatically. Another approach is to delegate parallelism management tasks to the runtime system, but to allow the programmer the opportunity to give advice on a few critical aspects; relevant work includes evaluation strategies (Trinder *et al*., 1998) and para-functional programming (Hudak, 1991). The skeleton approach (Cole, 1988) takes the view that implementing good dynamic behavior on a parallel machine is hard, and encapsulates the commonly encountered patterns of behavior using parallel higher-order functions like *scan*, *red*, *map* considered in this paper. Parallel implementations of skeletons on particular parallel architectures are prepackaged in the corresponding programming systems such as SCL (Darlington *et al*., 1995) and P3L (Bacci *et al*., 1995). In contrast, our approach does not require the user to express the problem using some fixed collection of skeletons. The restriction of our approach is that it is based on a fixed strategy of parallelism expressed by the homomorphy property. A promising approach to integrate different arts of parallelism are abstract parallel machines (O'Donnell and Rünger, 1997).

We have illustrated our method on two examples. The *scan* example, while being very well-known in the functional community, was chosen to demonstrate the details of generalization. Though parallel algorithms for *scan* are a common knowledge, an automatic parallelization from only sequential representations is an original feature of our approach. Our second example, the *mss* case study, is more elaborated. We have demonstrated that the CS method is applicable to almost-homomorphisms if extended by 'tupling' all auxiliary functions which arise in the process of building closed `cons` and `snoc` definitions. For the *mss* problem, the result of systematically applying the method coincides with the result obtained by Cole (1994) and Smith (1987), but unlike them we have not used our intuition about parallelism in the derivation process.

Acknowledgements

We are grateful to Murray Cole, Jeremy Gibbons, Christian Lengauer and Lambert Meertens for discussing different parts of the manuscript. The anonymous referees were helpful in improving the presentation.

The second author was partially supported by a travel grant from the EU INTAS Programme.

References

Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S. and Vanneschi, M. (1995) P^{3}L: A structured high level programming language and its structured support. *Concurrency: Practice & Exper*., **7**(3), 225–255.

Bachmair, L. (1989) *Canonical Equational Proofs: Research Notes in Theoretical Computer Science*, John Wiley & Sons.

Barnard, D., Schmeiser, J. and Skillicorn, D. (1991) Deriving associative operators for language recognition. *Bulletin of EATCS*, **43**, 131–139.

Bentley, J. (1984) Programming pearls. *Commun. ACM*, **27**, 865–871.

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

Blelloch, G. (1989) Scans as primitive parallel operations. *IEEE Trans. Computers*, **38**(11), 1526–1538.

Blelloch, G. (1993) NESL: a nested data-parallel language (Version 2.6). *Technical Report CMU-CS-93-129*, School of Computer Science, Carnegie Mellon University.

Cole, M. (1988) 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.

M. Cole. Parallel programming with list homomorphisms. *Parallel Process. Lett.*, **5**(2), 191–204.

Comon, H., Haberstrau, M. and Jouannaud, J.-P. (1994) Syntacticness, cycle-syntacticness, and shallow theories. *Infor. & Computation*, **111**, 154–191.

Darlington, J., Guo, Y., To, H. W. and Jing, Y. (1995) Skeletons for structured parallel composition. *Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Fraus, U. and Hußmann, H. (1992) 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*. Clarendon Press.

Geser, A. (1995) Mechanized inductive proof of properties of a simple code optimizer. In: P. D. Mosses, M. Nielsen and M. I. Schwartzbach, editors, *Proc. 6th Theory and Practice in Software Development (TAPSOFT): Lecture Notes in Computer Science 915*, pp. 605–619. Springer-Verlag.

Gibbons, J. (1996) The third homomorphism theorem. *J. Functional Programming*, **6**(4), 657–665.

Gorlatch, S. (1995) Constructing list homomorphisms. *Technical Report MIP-9512,* Universität Passau, Germany.

Gorlatch, S. (1996*a*) 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*, pp. 401–408. Springer-Verlag.

Gorlatch, S. (1996*b*) 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*, pp. 274–288. Springer-Verlag.

Gorlatch, S. and Bischof, H. (1997) 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*, pp. 80–94.

Heinz, B. (1994) Lemma discovery by anti-unification of regular sorts. *Technical Report 94-21*, TU Berlin, Germany.

Hudak, P. (1991) *Para-functional Programming in Haskell*, pp. 159–196. ACM Press.

Huet, G. (1980) Confluent reductions: abstract properties and applications to term rewriting systems. *J. ACM*, **27**(4), 797–821.

Jouannaud, J.-P. (1990) Syntactic theories. In: B. Rovan, editor, *Mathematical Foundations of Computer Science: Lecture Notes in Computer Science 452*, pp. 15–25. Banská Bystrica. Springer-Verlag.

Klop, J. W. (1992) Term rewriting systems. In: S. Abramsky, D. M. Gabbay and T. Maibaum, editors, *Handbook of Logic in Computer Science*, Vol. I, Chap. 6, pp. 1–116. Clarendon Press.

Lange, S. (1989) Towards a set of inference rules for solving divergence in Knuth-Bendix completion. In: K. P. Jantke, editor, *Proc. Analogical and Inductive Inference: Lecture Notes in Computer Science 397*, pp. 305–316. Springer-Verlag.

Martin, U. and Nipkow, T. (1990) Automating Squiggol. In: M. Broy and C. Jones, editors, *Proceedings of IFIP Working Group 2.2*, pp. 233–246. North Holland.

O'Donnell, J. (1994) A correctness proof of parallel scan. *Parallel Process. Lett.*, **4**(3), 329–338.

O'Donnell, J. and Rünger, G. (1997) A methodology for deriving parallel programs with a family of abstract parallel machines. In: C. Lengauer, M. Griebl and S. Gorlatch, editors, *Parallel Processing. Euro-Par'97: Lecture Notes in Computer Science 1300*, pp. 661–668. Springer-Verlag.

Plotkin, G. D. (1970) Lattice-theoretic properties of subsumption. *Technical Report Memo MIP-R-77*, University of Edinburgh.

Quinn, M. J. (1994) *Parallel Computing*. McGraw-Hill.

Skillicorn, D. (1994) *Foundations of Parallel Programming*. Cambridge University Press.

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

Thomas, M. and Watson, P. (1993) Solving divergence in Knuth-Bendix completion by enriching signatures. *Theoret. Comput. Sci.*, **112**(1), 145–185.

Trinder, P., Hammond, K., Loidl, H.-W. and Jones, S. P. (1998) Algorithm + strategy = parallelsim. *J. Functional Programming*, **8**(1), 23–60.