Parallelizing Functional Programs by Generalization

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. email: gorlatch@fmi.uni-passau.de

Abstract. List homomorphisms are functions that are parallelizable 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.

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) [3, 19]. Parallelism is tackled in BMF via the notion of *homomorphism* which captures a data-parallel form of the divide-and-conquer (DC) paradigm. A classification of the DC forms suitable for static parallelization is proposed in [13].

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/customize 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 [11, 12, 13]. A systematic extraction method, proposed in [12], proceeds by generalizing two sequential representations of the function: on the cons and snoc lists.

* Partially supported by grant Ku 996/3-1 of the DFG within the Schwerpunkt Deduktion at the University of Tübingen.

** Partially supported by the DFG project RecuR2 and by the DAAD exchange programs ARC and PROCOPE.

This so-called CS-method (CS 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. Section 3 introduces the notion of generalization and formulates the CS method of homomorphism extraction, in the generalization framework. In Section 4, we derive a generalization calculus for the theory of lists, based on term rewriting. Section 5 shows how to apply the generalization calculus to CS-generalization and derives a terminating, deterministic procedure, i.e. an algorithm, for generalization. Section 6 summarizes the results and outlines future work.

The presentation is illustrated by a running example – the *scan* function, also known as the parallel prefix [4]. 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 scan.

The full version of the paper, with proofs of all claims, is [7].

2 BMF and Homomorphisms

We restrict ourselves to non-empty lists, which are 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. $\mathit{map}\,f\,[x_1,\cdots,x_n] = [f x_1,\cdots,f x_n]$;
red $(\odot)$ *reduce* over a binary associative operation $\odot$,
$\mathit{red}(\odot)\,[x_1,\cdots,x_n] = x_1 \odot x_2 \odot \cdots \odot x_n$.

Definition 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:

$$h\,(x +\!\!+ y) = h\,(x) \circledast h\,(y) \qquad (1)$$

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.

Theorem 2 Bird [3]. *Function h is a homomorphism iff there exists a binary associative combine operator* $\circledast$ *such that*

$$h = \mathit{red}(\circledast) \circ (\mathit{map}\,f) \qquad (2)$$

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. The first stage in (2), *map*, is fully parallel; the parallelization of the second, reduction stage has been studied in [11]. 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. [4]; 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 3. 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 4. *A function on lists is a homomorphism iff it is both leftwards and rightwards.*

Unfortunately, as pointed out in [2, 9], 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 5. 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 $\circledast$-*lh* function h, such that $h\,([a]) = f\,(a)$ for arbitrary a; we denote this function by $lh\,(f, \circledast)$; similarly, notation $rh\,(f, \circledast)$ is introduced. We also write $hom\,(f, \circledast)$ for the unique $\circledast$-homomorphic function h, such that $h\,([a]) = f\,(a)$, for all a.

Theorem 6. *If* $h = \hom\,(f, \circledast)$, *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* $h = \hom\,(f, \circledast)$.

In [10], we prove a slightly stronger proposition.

Theorem 6 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 (e.g., for the function which yields the length of a list [12]), but already for *scan* it does not.

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 $scan\,(\odot)\,[a]$, e.g., $a = head\,(scan\,(\odot)\,[a])$ or $a = last\,(scan\,(\odot)\,[a])$. Unfortunately, each of possible terms for $\circledast$ obtained from (4), for instance: $head\,(u) \circledast (map\,(last\,(u) \odot)\,v)$, defines a non-associative operation! We run into a similar problem for a cons definition [12].

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 [17].

3 CS Method: Extraction by Generalization

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 $\circledast$:

$$u \circledast v \leftrightarrow t_H \tag{5}$$

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 to all **cons** and **snoc** representations, respectively, of function h:

$$t_B \circledast h(y) \leftrightarrow t_C \tag{6}$$
$$h(x) \circledast t'_B \leftrightarrow t_S \ . \tag{7}$$

The homomorphism extraction problem is formally specified as follows:

> **Given:** A **cons** definition $h(a \cdot\!: y) \leftrightarrow t_C, h([a]) \leftrightarrow t_B$ and a **snoc** definition $h(x :\!\cdot\, b) \leftrightarrow t_S, h([b]) \leftrightarrow t'_B$ for the same function h on lists.
> **Wanted:** A definition $u \circledast v \leftrightarrow t_H$, such that $h(x +\!\!+ y) \leftrightarrow h(x) \circledast h(y)$.

We intend to apply the framework of generalization for this problem.

Definition 7. A term t_G is called a *generalizer* of terms t_1 and t_2 in the equational theory E if there are substitutions σ_1 and 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.

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"[14]. Plotkin has closely studied the special-case relation between terms [18]. 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 are hardly known. 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 [16, 20]. 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 [5, 15], 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.

The generalization applied to (6) and (7), should yield t_H of (5) – 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 substitutions, solid arrows indicate conversion steps. 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 function symbol, $\rightharpoonup$, and model pairs (s, t) of terms as terms $s \rightharpoonup t$, called *rule terms*. With this encoding we get the view of Figure 1(b).

Theorem 8 [7]. *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 $\sigma_2 = \{u \mapsto h(x), v \mapsto t'_B\}$,*

$$\begin{tikzcd}
t_B \circledast h(y) \arrow[r, leftrightarrow, "(6)"'] & t_C \arrow[r, leftrightarrow, "*", "E"'] & t_H.\sigma_1 \arrow[r, equal] & t_L \\
u \circledast v \arrow[u, dotted, "\sigma_1"'] \arrow[d, dotted, "\sigma_2"'] \arrow[r, leftrightarrow, "(5)"'] & t_H \arrow[r, equal] & t_H \arrow[u, dotted, "\sigma_1"'] \arrow[d, dotted, "\sigma_2"'] & \\
h(x) \circledast t'_B \arrow[r, leftrightarrow, "(7)"'] & t_S \arrow[r, leftrightarrow, "*", "E"'] & t_H.\sigma_2 \arrow[r, equal] & t_R
\end{tikzcd}$$

(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 CS-Generalization; here: $\sigma_1 = \{u \mapsto t_B, v \mapsto h(y)\}$, $\sigma_2 = \{u \mapsto h(x), v \mapsto t'_B\}$.

and the operation $\circledast$ thus defined is associative, then: (1) The cons, snoc, *and* $\mathbin{+\!\!+}$ *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.*

For the proof we only remark that (1) follows immediately from Theorem 9. The generalization in Theorem 8 is called the *CS-generalization* in the theory of lists E. We can now summarize our considerations as the following method of finding the combine operation which proceeds in two steps.

CS-Method:

1. 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 $u \circledast v \rightharpoonup t_H$.
2. If associativity of $\circledast$ defined by t_H can be proven inductively then, by Theorem 8, $\circledast$ is the desired combine operator.

Example 1 Continued. For *scan*, the corresponding rule terms are:

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

Their CS-generalization yields $u \circledast v \rightharpoonup u \mathbin{+\!\!+} \mathit{map}\,(\mathit{last}\,(u) \odot)\, v$. One can prove that the operation $\circledast$ thus defined

$$u \circledast v \leftrightarrow u \mathbin{+\!\!+} \mathit{map}\,(\mathit{last}\,(u) \odot)\, v \tag{8}$$

is associative, so *scan* is a homomorphism with $f(a) = [a]$ and $\circledast$ defined by (8).

In practice, it is advisable to present t_C in the *lw* format and t_S in the *rw* format, which 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.

4 Expressing the CS Method in Term Rewriting

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

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 fragment 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 $f(t_1,\ldots,t_n) \rightarrow t_0$ 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 $t \rightarrow t'$. By $C[t]$ we indicate that t appears in the context C. If R is a system of such rewrite rules, then one writes $t \rightarrow_R t'$ for the application of a rule from R somewhere in t, yielding a term t'. Relation $\leftrightarrow_R^*$ is the equivalence closure of $\rightarrow_R$, i.e. the smallest binary relation on terms 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.

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

$$a \odot (b \odot c) \rightarrow (a \odot b) \odot c \qquad (A_\odot)$$

$$x +\!\!+ (y +\!\!+ z) \rightarrow (x +\!\!+ y) +\!\!+ z \qquad (A_+)$$

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

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

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 $+\!\!+$ 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. In our example, *scn* calls the functions *mp* and *last*. So, R_{base} becomes

$$last[a] \to a \tag{9}$$
$$last(x +\!\!+ y) \to last\, y \tag{10}$$
$$mp(a, [b]) \to [a \odot b] \tag{11}$$
$$mp(a, x +\!\!+ y) \to mp(a, x) +\!\!+ mp(a, y) \tag{12}$$

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

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

What we want is t_H such that R_{conc} defines the same function h as do R_{cons} and R_{snoc}. This is speaking about an inductively valid equation.

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. See, e.g., [1] for an introduction to inductive theorem proving based on rewriting.

Theorem 9 Reliability [7]. *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 a 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. *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.*

In other words, a generalizer of a certain shape and an inductive proof provide a solution to the homomorphism extraction problem. The inductive proof of associativity of $\circledast$ may be carried out by a mechanized inductive theorem prover.

Example 1 Continued. In our *scn* example, we have used the semi-automatic inductive prover TIP [6, 8] 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.

This leaves to solve the generalization problem. 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 σ_1 and σ_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 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.

Figure 2 and 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 every rewrite step takes place at the root position, ε, of t. For substitutions $\sigma = \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$, $\tau = \{y_1 \mapsto t_1, \ldots, y_n \mapsto t_n\}$ for which $x_i \neq y_j$ for all i, j, their disjoint union is defined by $\sigma \| \tau = \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m, y_1 \mapsto t_1, \ldots, y_n \mapsto t_n\}$.

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 $f(v'_1, \ldots, v'_m) =$

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

v_i. The common function symbol is broken down, by splitting the mapping $x_i \mapsto f(u'_1,\dots,u'_m)$ to $x_i \mapsto f(x'_1,\dots,x'_m)$ and $x'_j \mapsto u'_j$ where x'_j are fresh variables for each argument position, j, of f. Likewise $x_i \mapsto f(v'_1,\dots,v'_m)$ is split to $x_i \mapsto f(x'_1,\dots,x'_m)$ and $x'_j \mapsto v'_j$. Their common part, $x_i \mapsto f(x'_1,\dots,x'_m)$, 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 (13).

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 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; $t \xrightarrow[R]{\varepsilon}{}^{*} t'$", or just "AncDec for 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 10 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 11. *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.

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

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

A derivation is listed in Figure 4.

5 A Generalization Algorithm

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 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.

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

$$f(u'_1, \ldots, u'_m) \xrightarrow[R]{\varepsilon}{}^{X} u_i, \quad f(v'_1, \ldots, v'_m) \xrightarrow[R]{\varepsilon}{}^{Y} v_i, \quad X + Y \leq 1 \tag{14}$$

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 $l \to r$ where l contains a variable not in r. For instance, Rule (10) is erasing: it has x on its left, but not on its right hand side.

Condition (14) is not as hard in practice as it may seem. Observe, for instance, that the derivations in Example 2 and in Figure 4 each 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 2. 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

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

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

$\vdash$ (AncDec for $\rightharpoonup$)

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

$\vdash$ (AncDec for $\circledast$)

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

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

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

$\vdash$ Agr for x_3, x_5)

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

$\vdash$ (AncDec for mp; $mp(last(scn(x)), [b]) \xrightarrow[(11)]{\varepsilon} [last(scn(x)) \odot b]$)

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

$\vdash$ (Agr for x_4, x_8)

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

$\vdash$ (AncDec for mp; $last([a]) \xrightarrow[(9)]{\varepsilon} a$)

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

$\vdash$ (Agr for x_3, x_9)

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

Fig. 4. A successful derivation for *scn*

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 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 12. 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 13 Termination [7]. *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 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 (14), it is sensible to exclude erasing rules for computability reasons.

Example 1 Continued. The term rewriting system $R \setminus \{(10)\}$ 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 (9):

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

So every derivation in the restricted generalization calculus that refuses to use Rule (10) is finite: We obtain an algorithm that computes a finite set of generalizers.

6 Conclusion

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, intuition-based construction of a parallel solution is even more clearly visible for more complex problems than scan, e.g., so-called almost homomorphisms [12], which are not considered here because of lack of space.

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 prove reliability of this method to extract a homomorphism.

We have introduced a very 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.

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.

Acknowledgements. Thanks to Chris Lengauer and to the four anonymous referees for helpful remarks.

References

1. Leo Bachmair. *Canonical Equational Proofs.* Research Notes in Theoretical Computer Science. Wiley and Sons, 1989.
2. D. Barnard, J. Schmeiser, and D. Skillicorn. Deriving associative operators for language recognition. *Bulletin of EATCS*, 43:131–139, 1991.
3. 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.
4. G. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

5. Hubert Comon, Marianne Haberstrau, and Jean-Pierre Jouannaud. Syntacticness, cycle-syntacticness, and shallow theories. *Inform. and Computation*, 111:154–191, 1994.
6. 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.
7. A. Geser and S. Gorlatch. Parallelizing functional programs by term rewriting. Technical Report MIP-9709, Universität Passau, April 1997. Available from http://brahms.fmi.uni-passau.de/cl/papers/GesGor97a.html.
8. 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.
9. J. Gibbons. The third homomorphism theorem. *J. Fun. Programming*, 6(4):657–665, 1996.
10. S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Universität Passau, August 1995.
11. 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.
12. 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.
13. 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.
14. B. Heinz. Lemma discovery by anti-unification of regular sorts. Technical Report 94-21, TU Berlin, May 1994.
15. Jean-Pierre Jouannaud. Syntactic theories. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, pages 15–25, Banská Bystrica, 1990. LNCS 452.
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. J. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, 1994.
18. Gordon D. Plotkin. Lattice-theoretic properties of subsumption. Technical Report Memo MIP-R-77, Univ. Edinburgh, UK, 1970.
19. D. Skillicorn. *Foundations of Parallel Programming*. Cambridge Univ. Press, 1994.
20. Muffy Thomas and Phil Watson. Solving divergence in Knuth-Bendix completion by enriching signatures. *Theoretical Computer Science*, 112(1):145–185, 1993.