Calculating List Homomorphisms For Parallel Bracket Matching ⋆†

Walter Dosch and Bernd Wiedemann

Institut für Softwaretechnik und Programmiersprachen
Medizinische Universität zu Lübeck

Seelandstr. 1a, D–23569 Lübeck, Germany
URL: http://www.isp.mu-luebeck.de

Abstract. This paper presents a case study for the systematic design of abstract data parallel programs. We formally derive different list homomorphic bracket matching algorithms from a common specification using equational and inductive reasoning. We explore how high-level design decisions influence the final parallel algorithm. We specialize the data parallel algorithm retrieving well-known sequential versions.

Keywords: bracket matching, list homomorphism, deductive design, parallel programming

1 Introduction

The design of efficient parallel programs is well understood on machine-oriented levels. Here different classes of parallel architectures lead to characteristic programming paradigms, in particular to programming models for message passing and shared memory. The resulting programs, however, are often burdened with architectural details; hence their code is difficult to understand, to modify and to reuse.

Opposite to the success on lower levels, the design of abstract parallel programs only makes gradual progress. The diversity of parallel architectures renders it difficult to discover adequate abstraction concepts [1, 2].

⋆ This research was partially supported by the Esprit Working group 8533 NADA on "New Hardware Design Methods".

† appeared already in: H. R. Arabnia (ed.): Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), Vol. IV. Las Vegas, July 13–16, 1998. Athens: CSREA Press 1998, 1710–1717

The skeletal approach to parallel programming [3, 4] combines generality with efficiency by encapsulating frequently occuring algorithmic patterns into a single higher order function. Then the design of a parallel algorithm is structured into phases separating correctness and efficiency concerns. First the specification is implemented by an abstract, that is architecture-independent [5] parallel program expressed with skeletons. Then the skeletons are compiled to a specific parallel architecture with predictable cost [6]. This approach is also elaborated in the SAT method to parallel program design [7].

The design of the abstract parallel program may follow the deductive approach [8–10]. Here a problem-oriented specification is transformed step by step into an implementation using sound transformation rules. For algorithms over the data structure of sequences, the theory of lists [11, 12] provides a collection of useful algebraic laws to calculate with functional programs [13]. The application to parallel programming originates from the homomorphism lemma: every function defined as a catamorphism by structural recursion on sequences allows a straightforward data parallel implementation.

This paper presents a case study for the systematic design of abstract parallel programs. We give a formal derivation of a list homomorphic algorithm for parallel bracket matching. We start with a constructive specification and derive various implementations using equational and inductive reasoning. When following different development lines, we explicate the effect of early high-level design decisions onto the final data parallel program. We also specialize the data parallel solution retrieving well-known sequential algorithms by calculation.

As our major contribution

- we present a *transformational derivation* of the parallel bracket matching algorithm on the functional level,
- we study the *formal redesign* of the data parallel algorithm by revising an early design decision,
- we *systematically specialize* the data parallel algorithm obtaining different sequential versions by narrowing the decomposition pattern.

The paper is organized as follows: Section 2 surveys the basic notions for data parallel programming on the functional level. Section 3 presents a constructive specification of the bracket matching problem. In Section 4 we derive the list homomorphic solution by fusion and tupling. In Section 5 we replay the derivation after revising an early design decision. Finally, in Section 6 we specialize the data parallel algorithm towards sequential versions by narrowing the decomposition pattern.

2 Sequences and Data Parallel Programming

In this section, we survey the basic notions about functions, sequences and data parallel programming.

2.1 Functions

The *composition* of two functions $f : \mathcal{C} \to \mathcal{D}$ and $g : \mathcal{A} \to \mathcal{B}$ with $\mathcal{B} \subseteq \mathcal{C}$ denotes the function $f \circ g : \mathcal{A} \to \mathcal{D}$ with $(f \circ g)(a) = f(g(a))$. The *tupling* of two functions $f : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{A} \to \mathcal{C}$ yields the function $f \diamond g : \mathcal{A} \to \mathcal{B} \times \mathcal{C}$ with $(f \diamond g)(a) = (f(a), g(a))$.

2.2 Sequences

For a set $\mathcal{A}$, the equation $\mathcal{A}^\star = \{\langle\rangle\} \cup \mathcal{A} \cup \mathcal{A}^\star \times \mathcal{A}^\star$ defines the set $\mathcal{A}^\star$ of all *finite sequences* over $\mathcal{A}$ assuming that $\times$ denotes the associative cartesian product. Sequences are constructed from the *empty sequence* $\langle\rangle$ by forming *singletons* $\langle . \rangle : \mathcal{A} \to \mathcal{A}^\star$ and by *concatenation* $\& : \mathcal{A}^\star \times \mathcal{A}^\star \to \mathcal{A}^\star$. Sequences will be denoted by capital letters, their items by small letters. The algebra $(\mathcal{A}^\star; \&, \langle\rangle)$ forms an inductively generated monoid. Similarly, the equation $\mathcal{A}^+ = \mathcal{A} \cup \mathcal{A}^+ \times \mathcal{A}^+$ defines the subset $\mathcal{A}^+$ of all nonempty sequences over $\mathcal{A}$.

2.3 Homomorphisms

A function $h : \mathcal{A}^\star \to \mathcal{B}$ is called a (*list*) *homomorphism*, if

$$h(\langle\rangle) = e \tag{1}$$
$$h(\langle a \rangle) = f(a) \tag{2}$$
$$h(A \,\&\, B) = h(A) \oplus h(B) \tag{3}$$

holds for some element $e \in \mathcal{B}$, some function $f : \mathcal{A} \to \mathcal{B}$ and some combining operation $\oplus : \mathcal{B} \times \mathcal{B} \to \mathcal{B}$. It follows that $\oplus$ is associative on the image $h(\mathcal{A}^\star)$ with unit e. We write $h = \mathit{hom}(e, f, \oplus)$ for the homomorphism uniquely determined by e, f and $\oplus$.

2.4 Data Parallel Algorithms

We introduce two list homomorphisms which are fundamental for data parallel programming.

The functional $\mathit{map} : [\mathcal{A} \to \mathcal{B}] \to [\mathcal{A}^\star \to \mathcal{B}^\star]$ with $\mathit{map}(f) = \mathit{hom}(\langle\rangle, \langle.\rangle \circ f, \&)$ applies a function $f : \mathcal{A} \to \mathcal{B}$ to all elements of a sequence. This skeleton allows simple parallel implementations, since there is no data dependency between the elements.

The functional $\mathit{red} : \mathcal{A} \times [\mathcal{A} \times \mathcal{A} \to \mathcal{A}] \to [\mathcal{A}^\star \to \mathcal{A}]$ with $\mathit{red}(e, \oplus) = \mathit{hom}(e, \mathit{id}_{\mathcal{A}}, \oplus)$ reduces a sequence to a singleton element by combining each two subsequent elements under an associative operation $\oplus$ with unit e. This suggests a data parallel implementation as a binary tree with logarithmic time complexity. The abstract computation tree can be flexibly embedded into the interconnection topology of a parallel machine exploiting the associativity of the combining operator.

The homomorphism lemma [11, 12]

$$\mathit{hom}(e, f, \oplus) = \mathit{red}(e, \oplus) \circ \mathit{map}(f) \tag{4}$$

lays the basis for data parallel algorithms. Any homomorphism can be computed in two parallel stages by first mapping f to all items of the sequence and then reducing the intermediate results with the combining operation $\oplus$. In this way homomorphisms provide a simple divide-and-conquer scheme for parallelization [14].

The high-level design of list homomorphic algorithms concentrates on finding suitable *algebraic structures* with *customizing functions* f and $\oplus$ solving a given problem.

3 Specification

In this section we specify the bracket matching problem as the contract for the subsequent development.

Let $\mathcal{A}$ be an alphabet and $\mathcal{K} = \{(,)\} \subseteq \mathcal{A}$ be the subset of brackets. We want to design a data parallel algorithm that tests a sequence over the alphabet $\mathcal{A}$ whether the brackets from $\mathcal{K}$ are correctly matched.

The problem has a well-known sequential solution where the sequence is inspected in linear time from left to right. A counter is initialized to zero and incremented or decremented as opening or closing brackets are encountered. The brackets are correctly matched iff the counter never falls below zero and meets zero at the right end.

The sequential algorithm inspects the bracket sum of all prefixes of the sequence. This motivates the following constructive, yet algorithmically inefficient specification where the predicate *correct* : $\mathcal{A}^\star \to \mathbb{B}$ is defined as a composition of simpler functions:

$$\begin{aligned} \mathit{correct} = ((\geq 0) \circ \mathit{min} \circ \mathit{Bra} \circ \mathit{pre}) \wedge \quad & (5) \\ ((= 0) \circ \mathit{bra}) \quad & (6) \end{aligned}$$

In expression (5), the function *pre* generates the finite set of all nonempty prefixes. Then the function *Bra* computes the bracket sum for each nonempty prefix. Finally the function *min* computes the minimum of the bracket sums which must be not less than zero. In expression (6), the function *bra* computes the bracket sum of the entire sequence which must be zero.

In the sequel we spell out the auxiliary functions *pre*, *bra*, *Bra* and *min* used in the specification.

3.1 Set of Nonempty Prefixes

The function $\mathit{pre} : \mathcal{A}^\star \to \mathcal{P}_{\mathit{fin}}(\mathcal{A}^+)$ computes the finite set of all *nonempty* prefixes of a sequence:

$$\mathit{pre}(\langle\rangle) = \emptyset \quad (7)$$

$$\mathit{pre}(\langle a \rangle) = \{\langle a \rangle\} \quad (8)$$

$$\mathit{pre}(A \,\&\, B) = \mathit{pre}(A) \cup \{A\} \,\&\, \mathit{pre}(B) \quad (9)$$

All base functions are extended elementwise to finite sets. By convention, they bind tighter than set operations.

3.2 Bracket Sum

The function $bra : \mathcal{A}^\star \to \mathbb{Z}$ computes the bracket sum of a sequence

$$bra(\langle\rangle) = 0 \tag{10}$$
$$bra(\langle a\rangle) = k(a) \tag{11}$$
$$bra(A \,\&\, B) = bra(A) + bra(B)\,, \tag{12}$$

where $k : \mathcal{A} \to \mathbb{Z}$ valuates the elements of $\mathcal{A}$:

$$k(a) = \begin{cases} +1 & \text{if } a = \texttt{(} \\ \;\;0 & \text{if } a \notin \mathcal{K} \\ -1 & \text{if } a = \texttt{)} \end{cases}$$

The bracket sum forms a list homomorphism with $bra = hom(0, k, +)$.

The function $Bra : \mathcal{P}_{\mathit{fin}}(\mathcal{A}^\star) \to \mathcal{P}_{\mathit{fin}}(\mathbb{Z})$ extends the bracket sum to finite sets of nonempty sequences:

$$Bra(\emptyset) = \emptyset \tag{13}$$
$$Bra(\{A\}) = \{bra(A)\} \tag{14}$$
$$Bra(M \cup N) = Bra(M) \cup Bra(N) \tag{15}$$

There are various algebraic laws combining set homomorphic and list homomorphic properties, for example ($A \in \mathcal{A}^\star$, $P \in \mathcal{P}_{\mathit{fin}}(\mathcal{A}^\star)$)

$$Bra(\{A\} \,\&\, P) = \{bra(A)\} + Bra(P)\,. \tag{16}$$

3.3 Minimum

The function $min : \mathcal{P}_{\mathit{fin}}(\mathbb{Z}) \to \mathbb{Z} \cup \{\infty\}$ determines the smallest number of a finite set of integers:

$$min(\emptyset) = \infty \tag{17}$$
$$min(\{a\}) = a \tag{18}$$
$$min(A \cup B) = min(A) \downarrow min(B) \tag{19}$$

Here $\downarrow : (\mathbb{Z} \cup \{\infty\})^2 \to \mathbb{Z} \cup \{\infty\}$ denotes the binary minimum operator on the set $\mathbb{Z}$ of integers extended by the neutral element ∞. The function min cooperates smoothly with addition, for example ($a \in \mathbb{Z}$, $B \in \mathcal{P}_{\mathit{fin}}(\mathbb{Z})$)

$$min(\{a\} + B) = a + min(B)\,. \tag{20}$$

4 Program Derivation

In this section we calculate a list homomorphic algorithm from the specification. First we fuse the functions of the composition into a single recursive function. Then we transform the function obtained by function tupling into a list homomorphism.

4.1 Fusion

In this subsection we concentrate on the function $\mathit{minbra} : \mathcal{A}^\star \to \mathbb{Z} \cup \{\infty\}$ that computes the minimal bracket sum of the finite set of all nonempty prefixes:

$$\mathit{minbra} = \mathit{min} \circ \mathit{Bra} \circ \mathit{pre} \tag{21}$$

It is the basis for task (5); we postpone task (6) to Section 4.3. The integration of the tests (≥ 0) and $(= 0)$ into the list homomorphism will be discussed in Section 4.4.

We aim at deriving a direct recursion for the function minbra. The induction bases for the empty sequence

$$\begin{aligned}
& \mathit{minbra}(\langle\rangle) && \{\text{unfold } \mathit{minbra} \text{ (21)}\}\\
= {} & \mathit{min}(\mathit{Bra}(\mathit{pre}(\langle\rangle))) && \{\text{unfold } \mathit{pre} \text{ (7)}\}\\
= {} & \mathit{min}(\mathit{Bra}(\emptyset)) && \{\text{unfold } \mathit{Bra} \text{ (13)}\}\\
= {} & \mathit{min}(\emptyset) && \{\text{unfold } \mathit{min} \text{ (17)}\}\\
= {} & \infty
\end{aligned}$$

and for singletons

$$\begin{aligned}
& \mathit{minbra}(\langle a\rangle) && \{\text{unfold } \mathit{minbra} \text{ (21)}\}\\
= {} & \mathit{min}(\mathit{Bra}(\mathit{pre}(\langle a\rangle))) && \{\text{unfold } \mathit{pre} \text{ (8)}\}\\
= {} & \mathit{min}(\mathit{Bra}(\{\langle a\rangle\})) && \{\text{unfold } \mathit{Bra} \text{ (14)}\}\\
= {} & \mathit{min}(\{\mathit{bra}(\langle a\rangle)\}) && \{\text{unfold } \mathit{bra} \text{ (11)}\}\\
= {} & \mathit{min}(\{k(a)\}) && \{\text{unfold } \mathit{min} \text{ (18)}\}\\
= {} & k(a)
\end{aligned}$$

are straightforward. On the right-hand side we mention for each deduction step the major transformations applied. The induction step for composite sequences exploits various distributive laws:

$$
\begin{aligned}
& minbra(A \,\&\, B) && \{\text{unfold } minbra \text{ (21)}\} \\
= {} & min(Bra(pre(A \,\&\, B))) && \{\text{unfold } pre \text{ (9)}\} \\
= {} & min(Bra(pre(A) \cup \{A\} \,\&\, pre(B))) && \{\text{unfold } Bra \text{ (15)}\} \\
= {} & min(Bra(pre(A)) \cup Bra(\{A\} \,\&\, pre(B))) && \{\text{by (16)}\} \\
= {} & min(Bra(pre(A)) \cup \{bra(A)\} + Bra(pre(B))) && \{\text{unfold } min \text{ (19)}\} \\
= {} & min(Bra(pre(A))) \downarrow min(\{bra(A)\} + Bra(pre(B))) && \{\text{by (20)}\} \\
= {} & min(Bra(pre(A))) \downarrow && \{\text{fold } minbra \text{ (21) twice}\} \\
& bra(A) + min(Bra(pre(B))) \\
= {} & minbra(A) \downarrow bra(A) + minbra(B)
\end{aligned}
$$

In summary, we obtain the algorithm

$$
minbra(\langle\rangle) = \infty \tag{22}
$$

$$
minbra(\langle a \rangle) = k(a) \tag{23}
$$

$$
minbra(A \,\&\, B) = minbra(A) \downarrow bra(A) + minbra(B) \tag{24}
$$

assuming that $+$ binds tighter than $\downarrow$. The function *minbra* does not form a list homomorphism wrt. the minimum operation.

4.2 Tupling

The central step towards a data parallel algorithm consists in finding a suitable structure for function tupling [15] . Equation (24) suggests to pair the function *minbra* with the homomorphism *bra* . This leads to the generalization $minbrah : \mathcal{A}^\star \to (\mathbb{Z} \cup \{\infty\}) \times \mathbb{Z}$ with

$$
minbrah = minbra \diamond bra\,. \tag{25}
$$

We retrieve the original task by selecting the first component.

For deriving a direct recursion, we start with the induction bases for the empty sequence

$$
\begin{aligned}
& minbrah(\langle\rangle) && \{\text{unfold } minbrah \text{ (25)}\} \\
= {} & (minbra(\langle\rangle), bra(\langle\rangle)) && \{\text{unfold } minbra \text{ (22) and } bra \text{ (10)}\} \\
= {} & (\infty, 0)
\end{aligned}
$$

and for singletons:

$$
\begin{aligned}
& minbrah(\langle a \rangle) && \{\text{unfold } minbrah \text{ (25)}\} \\
= {} & (minbra(\langle a \rangle), bra(\langle a \rangle)) && \{\text{unfold } minbra \text{ (23) and } bra \text{ (11)}\} \\
= {} & (k(a), k(a))
\end{aligned}
$$

In the induction step

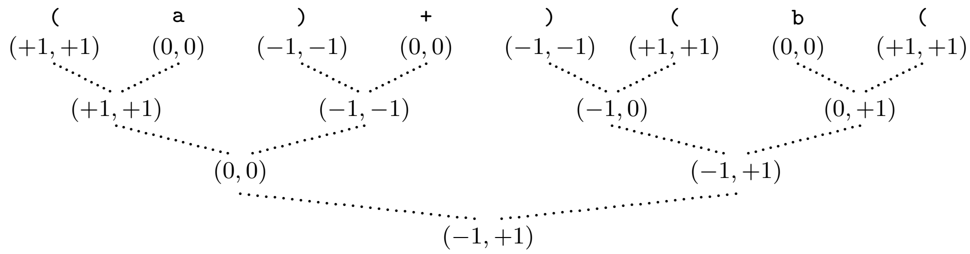

Fig. 1. List homomorphic bracket matching $\mathit{minbrah}(\langle\texttt{(a)+)(b(}\rangle)$

$$
\begin{aligned}
& \mathit{minbrah}(A \,\&\, B) && \{\text{unfold } \mathit{minbrah} \text{ (25)}\}\\
&= (\mathit{minbra}(A \,\&\, B), \mathit{bra}(A \,\&\, B)) && \{\text{unfold } \mathit{minbra} \text{ (24) and } \mathit{bra} \text{ (12)}\}\\
&= (\mathit{minbra}(A) \downarrow \mathit{bra}(A) + \mathit{minbra}(B), && \{\text{introduce } \oplus \text{ (26)}\}\\
&\quad\ \mathit{bra}(A) + \mathit{bra}(B))\\
&= (\mathit{minbra}(A), \mathit{bra}(A)) \oplus && \{\text{fold } \mathit{minbrah} \text{ (25) twice}\}\\
&\quad (\mathit{minbra}(B), \mathit{bra}(B))\\
&= \mathit{minbrah}(A) \oplus \mathit{minbrah}(B)
\end{aligned}
$$

we systematically abstract the combining operation $\oplus : ((\mathbb{Z} \cup \{\infty\}) \times \mathbb{Z})^2 \to ((\mathbb{Z} \cup \{\infty\}) \times \mathbb{Z})$ given by

$$(m, b) \oplus (m', b') = (m \downarrow b + m', b + b')\,. \tag{26}$$

So we have derived the data parallel algorithm

$$\mathit{minbrah}(\langle\rangle) = (\infty, 0) \tag{27}$$
$$\mathit{minbrah}(\langle a \rangle) = (k(a), k(a)) \tag{28}$$
$$\mathit{minbrah}(A \,\&\, B) = \mathit{minbrah}(A) \oplus \mathit{minbrah}(B)\,. \tag{29}$$

It forms a list homomorphism with

$$\mathit{minbrah} = \mathit{red}((\infty, 0), \oplus) \circ \mathit{map}(k \diamond k)\,; \tag{30}$$

the evaluation is shown in Fig. 1.

4.3 Reusing Results

The list homomorphism *minbrah* was developed to solve the first task (5). In the second component, it carries the bracket sum needed also for task (6). So we do not compute the bracket sum anew, rather we reuse the computed result for the second test. This gives the overall program

$$correct(A) = \lceil (m, b) = minbrah(A); \quad m \geq 0 \wedge b = 0 \rfloor \tag{31}$$

which — by construction — validates the specification.

4.4 Integrating Tests

The list homomorphic solution developed so far does not comprise the comparisons which are performed once and for all after both minimum and bracket sum have been computed.

Integrating the tests on zero into the data parallel algorithm leads to one additional component. Its Boolean value records whether the sequence is well bracketed. However, the subresults from the left and right subsequence are not used for computing the result of the composite sequence.

Therefore the tests are better *not* integrated in the list homomorphism, but carried out once and for all at the end.

5 Redesign

Given a program derivation for one particular algorithm, we can explore further development lines by following alternative design decisions. This results in a family tree of related algorithms. A redesign should not just textually modify the final code; rather it should "replay" the derivation under modified assumptions.

Let us revise the early design decision where we restricted ourselves to the set of *nonempty* prefixes. Alternatively we now consider the set of *all* prefixes — including the empty prefix.

5.1 Modified Specification

The modified function $epre : \mathcal{A}^\star \to \mathcal{P}^+_{\mathit{fin}}(\mathcal{A}^\star)$ computes the finite set of all prefixes:

$$epre(\langle\rangle) = \{\langle\rangle\} \tag{32}$$

$$epre(\langle a\rangle) = \{\langle\rangle, \langle a\rangle\} \tag{33}$$

$$epre(A \,\&\, B) = epre(A) \cup \{A\} \,\&\, epre(B) \tag{34}$$

The base cases (32)–(33) differ from (7)–(8), whereas the inductive case (34) agrees with (9). The functions *bra*, *Bra* and *min* remain unchanged. This results in the modified specification $ecorrect : \mathcal{A}^\star \to \mathbb{B}$ given by

$$ecorrect = ((\geq 0) \circ min \circ Bra \circ epre) \wedge ((= 0) \circ bra)\,.$$

5.2 Redevelopment

We start with the modified function $eminbra : \mathcal{A}^\star \to \mathbb{Z}$ defined by

$$eminbra = min \circ Bra \circ epre\,, \tag{35}$$

compare (21). Now we can replay the derivations of Sections 4.1 and 4.2, and focus on the effects of the modification. At the end, we obtain the algorithm $eminbrah : \mathcal{A}^\star \to \mathbb{Z} \times \mathbb{Z}$ with

$$eminbrah(\langle\rangle) = (0, 0) \tag{36}$$

$$eminbrah(\langle a\rangle) = (0 \downarrow k(a), k(a)) \tag{37}$$

$$eminbrah(A \,\&\, B) = eminbrah(A) \oslash eminbrah(B)\,, \tag{38}$$

compare (27)–(29). Its combining operation

$$(m, b) \oslash (m', b') = (m \downarrow b + m', b + b') \tag{39}$$

agrees with (26) on the simplified domain $\mathbb{Z}\times\mathbb{Z}$. The modifications (36)–(37) reflect the changes (32)–(33) in the decomposition function *epre*. So the list homomorphism

$$eminbrah = red((0, 0), \oslash) \circ map(l \diamond k)\,, \tag{40}$$

compare (30), uses a different valuation function $l : \mathcal{A} \to \mathbb{Z}$ on the minimum component:

$$l(a) = \begin{cases} 0 & \text{if } a \in \mathcal{A} \cup \{\,(\,\} \\ -1 & \text{if } a = \,) \end{cases}$$

5.3 Related Work

The list homomorphic algorithm (40) for parallel bracket matching was first published by M. Cole [16]. This paper contains excellent motivation, but "*the justification of the algorithms' correctness remains intuitive rather than being verifiable by a sequence of checkable transformations*" [16], p. 202. We systematically derived the data parallel algorithm, and additionally provided a new version (30).

6 Sequential Versions

In this section, we systematically specialize the list homomorphic solution obtaining two sequential versions. The left-to-right sequential algorithm traverses *cons*-lists $\mathcal{A}^\star = \{\langle\rangle\} \cup \mathcal{A}\times\mathcal{A}^\star$, whereas the right-to-left sequential algorithm works on *snoc*-lists $\mathcal{A}^\star = \{\langle\rangle\} \cup \mathcal{A}^\star\times\mathcal{A}$.

6.1 Left-to-Right Sequential Algorithm

For deriving a left-to-right sequential version, we first generalize the constant in the data parallel algorithm (31). In the generalization $\mathit{lrcorr} : \mathbb{Z} \times \mathcal{A}^\star \to \mathbb{B}$ with

$$\begin{aligned} \mathit{lrcorr}(c, A) = \lceil &(m, b) = \mathit{minbrah}(A);\\ &c + m \geq 0 \wedge c + b = 0 \rfloor \end{aligned} \tag{41}$$

the counter c accumulates the bracket sum of the left context. With the specialization $\mathit{lrcorr}(0, A) = \mathit{correct}(A)$ we retrieve the original task.

We aim at deriving a direct recursion for the function lrcorr. The induction basis is simple:

$$\begin{aligned}
& \mathit{lrcorr}(c, \langle\rangle) && \{\text{unfold } \mathit{lrcorr}\ (41)\}\\
={}& \lceil (m, b) = \mathit{minbrah}(\langle\rangle); && \{\text{unfold } \mathit{minbrah}\ (27)\}\\
& c + m \geq 0 \wedge c + b = 0 \rfloor &&\\
={}& \lceil (m, b) = (\infty, 0);\ c + m \geq 0 \wedge c + b = 0 \rfloor && \{\text{unfold const. } m, b\}\\
={}& c + \infty \geq 0 \wedge c + 0 = 0 && \{\text{simplify}\}\\
={}& (c = 0) &&
\end{aligned}$$

In the induction step we specialize the general pattern $A \mathbin{\&} B$ of a composite sequence to the pattern $\langle a\rangle \mathbin{\&} A$ of *cons*-lists. This enforces a sequential evaluation from left to right.

With the abbreviation $(m, b) = \mathit{minbrah}(A)$ we first calculate

$$\begin{aligned}
& \mathit{minbrah}(\langle a\rangle \mathbin{\&} A) && \{\text{unfold } \mathit{minbrah}\ (29)\}\\
={}& \mathit{minbrah}(\langle a\rangle) \oplus \mathit{minbrah}(A) && \{\text{unfold } \mathit{minbrah}\ (28)\}\\
={}& (k(a), k(a)) \oplus \mathit{minbrah}(A) && \{\text{introduce } (m, b)\}\\
={}& (k(a), k(a)) \oplus (m, b) && \{\text{unfold } \oplus\ (26)\}\\
={}& (k(a) \downarrow k(a) + m, k(a) + b) &&
\end{aligned}$$

and then continue

$$
\begin{aligned}
& lrcorr(c, \langle a \rangle \,\&\, A) && \{\text{unfold } lrcorr \text{ (41)}\} \\
={}& c + (k(a) \downarrow k(a) + m) \geq 0 \,\wedge && \{+ \text{ distributes over } \downarrow\} \\
& c + (k(a) + b) = 0 && \\
={}& (c + k(a) \downarrow c + k(a) + m) \geq 0 \,\wedge && \{\geq \text{ distributes over } \downarrow\} \\
& c + k(a) + b = 0 && \\
={}& c + k(a) \geq 0 \wedge (c + k(a) + m) \geq 0 \,\wedge && \{\text{fold } lrcorr \text{ (41)}\} \\
& c + k(a) + b = 0 && \\
={}& c + k(a) \geq 0 \wedge lrcorr(c + k(a), A)\,. &&
\end{aligned}
$$

In the resulting algorithm

$$
\begin{aligned}
lrcorr(c, \langle\rangle) &= (c = 0) \\
lrcorr(c, \langle a \rangle \,\&\, A) &= c + k(a) \geq 0 \wedge lrcorr(c + k(a), A)
\end{aligned}
$$

the conjunction can be sequentialized:

$$
\begin{aligned}
lrcorr(c, \langle\rangle) &= (c = 0) \\
lrcorr(c, \langle a \rangle \,\&\, A) &= \begin{cases} F & \text{if } c + k(a) < 0 \\ lrcorr(c + k(a), A) & \text{if } c + k(a) \geq 0 \end{cases}
\end{aligned}
$$

This enforces early termination upon a bracket underflow.

Altogether the left-to-right sequential version results from the data parallel algorithm by specializing the decomposition pattern to *cons*-lists and by imposing early termination.

6.2 Right-To-Left Sequential Algorithm

The right-to-left sequential version can be derived analogously. We obtain the algorithm $rlcorr : \mathcal{A}^\star \times \mathbb{Z} \to \mathbb{B}$ with

$$
\begin{aligned}
rlcorr(\langle\rangle, c) &= (c = 0) \\
rlcorr(A \,\&\, \langle a \rangle, c) &= \begin{cases} F & \text{if } k(a) + c > 0 \\ lrcorr(A, k(a) + c) & \text{if } k(a) + c \leq 0\,. \end{cases}
\end{aligned}
$$

In summary, we have derived left-to-right and right-to-left sequential versions by specializing the data parallel algorithm. In the opposite direction, the CS–method attempts to fuse the homomorphic solution from sequential algorithms on *cons*- and *snoc*-lists [7].

7 Conclusion

The systematic design of parallel algorithms suffers from a lack of *adequate* abstraction concepts. The abstraction mechanism must hide irrelevant implementation details from the programmer. At the same time, the abstraction must not conflict with performance by allowing efficient implementations on a variety of parallel machines. One possible solution is to provide a hierarchy of abstractions ranging from problem oriented to architecture specific ones, to relate them by transformations, and to endow them with cost functions. Then the design could gradually introduce the functionality principles of a target architecture into a derivation guided by cost considerations.

The skeletal approach to parallel programming offers higher order functions as control abstractions and sequences as data abstractions. The algebraic reasoning with (higher order) functional programs is elegant and works well for recursive data types over simple data structures. Efficient parallel algorithms on dynamic data structures, however, present a great challenge to programming methodology.

References

1. D.B. Skillicorn, D. Talia: *Programming Languages For Parallel Processing.* Los Alamitos: IEEE Computer Society Press 1995
2. D. Talia: *Parallel Computation Still Not Ready For the Mainstream.* Communications of the ACM *40*:7, 98–99 (1997)
3. M.I. Cole: *Algorithmic Skeletons: Structured Management of Parallel Computation.* Cambridge: M.I.T. Press 1989
4. J. Darlington, A.J. Field, P. G. Harrison, P. H.J. Kelly, D.W.N. Sharp, Q. Wu: *Parallel Programming Using Skeleton Functions.* In: A. Bode, M. Reeve, G. Wolf (eds.): Parallel Architectures and Languages Europe. Lecture Notes in Computer Science *694*. Berlin: Springer 1993, 146–160
5. D.B. Skillicorn: *Architecture-Independent Parallel Computation.* IEEE Computer *23*:12, 38–50 (1990)
6. D.B. Skillicorn, W. Cai: *A Cost Calculus for Parallel Functional Programming.* Journal of Parallel and Distributed Computing *28*:1, 65–83 (1995)
7. S. Gorlatch: *Stages and Transformations in Parallel Programming.* In: M. Kara, J.R. Davy, D. Goodeve, J. Nash (eds.): Abstract Machine Models. IOS Press 1996, 147–161
8. W. Dosch: *Design Decisions For the Parallel Evaluation of Polynomials.* In: H.R. Arabnia (ed.): Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Volume II. CSREA Press 1997, 636–644

9. A.M. Geerling: *Program Transformations and Skeletons: Formal Derivation of Parallel Programs.* In: N. Mirenov (ed.): Proceedings of the Aizu International Symposium of Parallel Algorithms and Architecture Synthesis. Los Alamitos: IEEE Computer Society Press 1995, 250–256
10. P.G. Harrison: *Towards the Synthesis of Static Parallel Algorithms.* In: B. Möller (ed.): Constructing Programs From Specifications. Amsterdam: Elsevier 1991, 49–69
11. R.S. Bird: *An Introduction to the Theory of Lists.* In: M. Broy (ed.): Logic of Programming and Calculi of Discrete Design. NATO ASI Series. Series F: Computer and Systems Sciences *36*. Berlin: Springer 1987, 5–42
12. R.S. Bird: *Lectures on Constructive Functional Programming.* In: M. Broy (ed.): Constructive Methods in Computing Science. NATO ASI Series. Series F: Computer and Systems Sciences *55*. Berlin: Springer 1989, 151–216
13. R.S. Bird, O. de Moor: *Algebra of Programming.* London: Prentice Hall 1997
14. S. Gorlatch, C. Lengauer: *Parallelization of Divide-and-Conquer in the Bird-Meertens-Formalism.* Formal Aspects of Computing *7*:6, 663–682 (1995)
15. Z. Hu, H. Iwasaki, M. Takeichi: *Construction of List Homomorphisms by Tupling and Fusion.* In: W. Penczek, A. Szalas (eds.): Mathematical Foundations of Computer Science. Lecture Notes in Computer Science *1113*. Berlin: Springer 1996, 407–418
16. M.I. Cole: *Parallel Programming with List Homomorphisms.* Parallel Processing Letters *5*:2, 191–203 (1995)