Size Inference of Nested Lists in Functional Programs

Christoph A. Herrmann and Christian Lengauer

Fakultät für Mathematik und Informatik,
Universität Passau, Germany

{herrmann,lengauer}@fmi.uni-passau.de
http://www.fmi.uni-passau.de/~lengauer

Abstract. We propose a method for the static derivation of symbolic size information about the length of nested lists in functional programs. This work facilitates reasoning about the index space of a nested list structure which is more flexible than the hypercubic index space of an array and the polytopic index space of an affine loop nest.
Our interest in size inference stems from the desire to optimize target code which is generated in a program parallelization. Each nesting level of the list structure is represented by a size expression which is parametrized by its enclosing indices.

Keywords: size inference, Haskell, parallelization, skeleton, divide-and-conquer.

1 Introduction

A powerful way of expressing parallelism in functional programs is by the use of special skeletons, i.e., higher-order functions. A simple skeleton is `map`, which applies a function to all elements of a list. In order to obtain a compile-time (static) schedule and processor allocation, it is necessary to express these skeletons in terms of structural parameters, e.g., the problem size, number of processors, etc.

It is also necessary to provide an aggregated data type which allows the user to define data objects which are static in their structure but dynamic in their actual size. Such objects could be triangular, banded or block matrices, complete k-ary trees or graphs represented by adjacency lists sorted by the degree of the nodes. A large application example is the parallel simulation of static systems whose (dependent) component sizes are subject of the simulation input pattern. For an illustration of our inference method, we chose the smaller example of inverting a lower triangular matrix by divide-and-conquer.

We decided to use the list as the underlying structure because it is easily implementable and usable for simple applications but allows also for sophisticated index operations and graph representation. We denote the syntax of our functions in the lazy functional programming language Haskell [3, 14], but we adopt an eager semantics.

In Haskell, the following equation holds for finite lists:

```
map f xs = [ f (xs!!i) | i <- [0..n-1] ] where n = length xs
```

The left side of the equation is an elementwise application of function `f` to all elements of the list `xs`. The right side is expressed by a so-called *list comprehension*. It states that the list consists of elements with index `i` from `0` to `n-1`, and that the `i`th element of the list is the result of applying function `f` to `(xs!!i)`, which is the `i`th element of the input list. Provided that it is possible to deduce the value of `n` as a symbolic expression from the context instead of using the function `length` at run time, the parallel applications of `f` to the elements of `xs` can be assigned to a specific processor at compile time, allowing to organize the communication paths in advance. Formally, we express this case syntactically by a parallel loop, which does not necessarily imply an implementation on a SIMD architecture. For instance, on an MPI platform [11], `map` can be implemented by a parallel loop, where each processor applies a sequential `map` to its own, statically allocated part of the list.

Things get more complicated if the elements of `xs` are lists themselves, i.e., sublists, and the memory consumption and/or the computation time of the application of `f` may vary between different sublists. In such cases, it is necessary to keep length information—not only for each sublist but for every level of nesting. Then, large sublists can be distributed among many processors, whereas some small sublists may reside on a single processor.

Our interest in size inference has arisen in the implementation of a parallelizing compiler for a subset of Haskell with a focus on the divide-and-conquer ($\mathcal{DC}$) paradigm [12, 13], in which nested lists are the predominant data structure [4]. We call this sublanguage of Haskell $\mathcal{HDC}$ for $\mathcal{H}$igher-order $\mathcal{D}$ivide-and-$\mathcal{C}$onquer.

Size inference methods exist for arrays [16, 22]. Arrays have the property that the extent in each dimension is a constant. They can be represented by nested lists. E.g., a two-dimensional array can be represented by a list whose elements are lists of equal length. In general, the elements of a list do not have equal size. Therefore, a more flexible way of storing size information is necessary if the data is located in an index space with fuzzy bounds, as in most cases when nested list structures are used.

We identify the correct position of an element of a list, stored in a contiguous area of memory, as follows. If we want to access an element, we have to know its position and size. We obtain the position by calculating the sizes of all preceding elements. Because we permit nesting, we have to describe the extent of each list on each level in terms of the logical position (given by a sequence of indices) this list has in the entire structure. This is similar to describing a polytope by scanning it with a loop nest [20].

The following section gives an overview of our compiler. In the third section, we describe our model formally. The fourth section applies our size inference method to the example of inverting lower triangular matrices. The last section concludes.

2 The $\mathcal{HDC}$ compiler

The $\mathcal{HDC}$ compiler translates a subset of Haskell into an imperative language—at present, C with MPI calls. The main difference to Haskell is that $\mathcal{HDC}$ is strict in order to facilitate a compile-time parallelization. Two implementational differences to a typical Haskell compilation are that higher-order functions without a skeleton implementation are eliminated but list comprehensions are only simplified to single generator form. The reason is that higher-order functions complicate but list comprehensions simplify a static space-time mapping.

The compiler is based on the principle of compilation by transformation, which has already been used successfully in the Glasgow Haskell compiler [19], and consists of the following phases.

- **Scanner/Parser.** The source text is translated into a syntax tree represented as an algebraic data type in Haskell. The parser generator *happy* is used to generate a parse function from a grammar.
- **Pattern matching simplification.** Nested left-hand side patterns and patterns in arguments are eliminated.
- **Variable renaming.** All variables are made unique in order to simplify various substitutions in the subsequent phases.
- **Lambda lifting.** Lambda abstractions and `let` expressions are eliminated by introducing auxiliary functions [17].
- **List comprehension simplification.** Contrary to the complete elimination of list comprehensions like in [18], they are here only simplified to an index-based, single generator form. However, this transformation only works for finite lists. This form, which contains only a single generator from an integer sequence starting from 0, can be obtained by applying the rules in Fig. 1. ($E[x := y]$) denotes the substitution of every free occurrence of x in E by y.

$$
\begin{array}{ll}
(generatorLC) & [\ e \mid q_0, ..., q_n, x \leftarrow xs\ ] \\
\overset{\rightarrow}{=} & \texttt{concat}\ [\ [\ e \mid x \leftarrow xs\ ] \mid q_0, ..., q_n\ ] \\
 & \\
(guardLC) & [\ e \mid q_0, ..., q_n, p\ ] \quad \{p :: \mathrm{Bool}\} \\
\overset{\rightarrow}{=} & \texttt{concat}\ [\ \texttt{if}\ p\ \texttt{then}\ [e]\ \texttt{else}\ [] \mid q_0, ..., q_n\ ] \\
 & \\
(indexLC) & [\ e \mid x \leftarrow xs\ ] \quad \{xs\ \text{finite}, \text{fresh}\ i\} \\
\overset{\rightarrow}{=} & [\ (e[x := xs\,\texttt{!!}\,i]) \mid i \leftarrow [0..\texttt{length}\ \ xs - 1]\ ]
\end{array}
$$

Fig. 1. Simplification of list comprehensions

- **Typechecking.** The type checker is based on unification using the Martelli-Montanari rules [23]. A simple type class system is implemented by assigning type variables a set of possible types. The unification of two type variables then involves computing the intersection of both sets.
- **Monomorphization.** All type variables are eliminated and replaced by the types that are actually needed. This requires the duplication of each function for all possible types which occur in the context.
- **Elimination of functional arguments.** Functions in argument positions are replaced by codes which are interpreted by an apply function [2].
- **Dataflow graph optimizations.** Elimination of mutual recursion, inlining and common subexpression elimination.
- **Size inference.** The topic of this paper.
- **Space-time mapping.** If possible, computations are assigned a time step (schedule) and a processor (allocation). The function defining this assignment has structural parameters like the problem size, number of processors, etc.
- **Code generation.** Target code is in C using the parallel MPI library.

3 The size inference method

The size inference method we present requires some previous program simplification steps which are carried out by our compiler. Then, the size inference is applied to a program, which does not contain local function definitions, nested left-hand side patterns, polymorphism and overloading, higher-order functions and mutual recursion.

For the moment, we concentrate on the size inference for nested list structures, but we plan to include the use of tuples and algebraic data structures in combination with lists in the future.

In Subsect. 3.1, we discuss how size information of a function relates to its domain. Subsect. 3.2 describes the structure of the size information. The rules for size inference are given in Subsect. 3.3. Subsect. 3.4 discusses size inference of recursive functions.

3.1 Size information as a domain definition

Usually, functions are written for a domain which is more restrictive than the domain imposed by the type. This is due to limitations of a decidable type system. E.g., in the Hindley-Milner type system, it is impossible to restrict the type of a function which uses two lists as arguments to the case in which both lists have the same length. Such restrictions can be stated informally as documentation or formally by annotations which can be checked automatically.

We have chosen optional annotations. The compiler should perform the best inference possible but, if the result is not satisfactory, annotations can be provided.

From an implementational point of view, this means that the compiler must conduct a global analysis. Starting from the main function, the domains of the functions are inferred by the context of their use. The sizes of actual parameters and function results are derived by passing along the syntactic structure of the original function according to the rules presented in Subsect. 3.3.

3.2 The structure of size information

Following Hughes et al. [15], we perform first a Hindley-Milner type inference [9] which is not affected by the size information and, therefore, always terminates. Each expression in the program is annotated by a type and a size. The type annotations are performed during the type check. The size information is derived after the type check, since the according derivation rules require the presence of type information. The size derivation terminates because it consists of a single post-order traversal of the syntax tree.

There are three kinds of size annotations:

1. The empty annotation: in this case, the size of the data object which represents the expression depends on information not available before run time.
2. A compile-time value $\alpha(v)$: this describes a scalar value of type α (e.g., Bool, Int) which is known at compile time to have value v; v can be a symbolic expression in structural parameters.

3. Length information $\langle \alpha, [m_0, ..., m_{n-1}] \rangle$: the type of the expression is a nested list with nesting depth n and elements of type α which are not necessarily atomic. With respect to the pure type information ($=_{\text{type}}$), the following equations hold:

   (a) $\langle \alpha, [] \rangle \quad =_{\text{type}} \alpha$
   (b) $\langle \alpha, \_ : ns \rangle \quad =_{\text{type}} [\langle \alpha, ns \rangle]$
   (c) $\langle \alpha, ns\text{++}[\_] \rangle =_{\text{type}} \langle [\alpha], ns \rangle$

   Thus, m_i is a symbolic expression which describes the possibly different lengths of all lists at nesting level i in a uniform way. To obtain the length of a particular list at this level, the index variables $\uparrow 0 \ldots \uparrow (i-1)$, which possibly occur in the expression, have to be bound to the indices this list has in the entire structure. E.g., in the structure `[[x],[y,z]]`, `y` has the indices $\uparrow 0 = 1$ (it belongs to the second element #1 at the outermost level 0) and $\uparrow 1 = 0$ (it is the element #0 at level 1, i.e., in `[y,z]`). The idea was taken from the *de Bruijn* notation of lambda expressions [1].
   We allow a simple form of pattern matching, which extends the substitution operator ($[. := .]$) for the case of references ($((\uparrow i)[\uparrow j := f\ j]) = f\ i$), using a fresh variable i which is bound to the argument of $\uparrow$.

3.3 Derivation rules

The derivation rules for the sizes of nested lists are displayed in Fig. 2.

Rules (*nil*), (*cons*), (*null*), and the conditional rules constitute a basis for other size rules, e.g., (*head*) and (*tail*) can be derived from (*cons*), (*length*) can be derived from (*conditional3*), (*null*), and (*tail*) using induction, etc. Still, in order to make explicit use of the syntactic structure, which is especially important for a parallelization, we provide a number of rules, which we could derive from other rules, explicitly.

Let us provide a short explanation for each rule:

- (*nil*) The size of the empty list is 0. However, to apply, e.g., rule (*append*) to a list structure of nesting depth 2 and the empty list, rule (*nil*) has to be applied twice to yield the annotated type $\langle \langle \alpha, [0] \rangle, [0] \rangle$ which equals $\langle \alpha, [0, 0] \rangle$.
- (*cons*) If we access the first element ($\uparrow 0 = 0$) in `(x:xs)`, we must refer to the size information for `x` and rename all references to the next level of nesting. Otherwise ($\uparrow 0 > 0$), we have to take the information from `xs` but decrement every reference at the outermost level, because

(nil)

$$\frac{}{[\,] :: \langle \alpha, [0] \rangle}$$

$(cons)$

$$\frac{x :: \langle \alpha, ms \rangle \mid xs :: \langle \alpha, n : ns \rangle \mid \text{length } ms \;=\; \text{length } ns}{x\texttt{:}xs :: \left\langle \alpha,\; n{+}1 : \left\{ \begin{array}{r} \text{if } \uparrow 0 = 0 \text{ then } (ms[\uparrow i := \uparrow(i{+}1)]) \\ \text{else } (ns[\uparrow 0 := \uparrow 0 - 1]) \end{array} \right\} \right\rangle}$$

$(head)$

$$\frac{xs :: \langle \alpha, n : ns \rangle \mid n > 0}{\texttt{head}\ xs :: \langle \alpha, ((ns[\uparrow 0 := 0])[\uparrow i := \uparrow(i{-}1)]) \rangle}$$

$(tail)$

$$\frac{xs :: \langle \alpha, n : ns \rangle \mid n > 0}{\texttt{tail}\ xs :: \langle \alpha,\; n{-}1 : (ns[\uparrow 0 := \uparrow 0 + 1]) \rangle}$$

$(null)$

$$\frac{xs :: \langle \alpha,\; n : \_ \rangle}{\texttt{null}\ xs :: \text{Bool}(n = 0)}$$

$(length)$

$$\frac{xs :: \langle \alpha, n : \_ \rangle}{\texttt{length}\ xs :: \text{Int}(n)}$$

$(conditional1)$

$$\frac{x :: \text{Bool} \mid y :: s \mid z :: t \mid s = t}{\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z :: s}$$

$(conditional2)$

$$\frac{x :: \text{Bool}(r) \mid ys :: \langle \alpha, ms \rangle \mid zs :: \langle \alpha, ns \rangle \mid \text{length } ms \;=\; \text{length } ns}{\texttt{if}\ x\ \texttt{then}\ ys\ \texttt{else}\ zs :: \langle \alpha,\ \text{if } r \text{ then } ms \text{ else } ns \rangle}$$

$(conditional3)$

$$\frac{x :: \text{Bool}(r) \mid y :: \alpha(s) \mid z :: \alpha(t)}{\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else}\ z :: \alpha(\text{if } r \text{ then } s \text{ else } t)}$$

$(append)$

$$\frac{xs :: \langle \alpha,\; n : ns \rangle \mid ys :: \langle \alpha,\; m : ms \rangle \mid \text{length } ns \;=\; \text{length } ms}{xs\ \texttt{++}\ ys :: \langle \alpha,\; n{+}m :\ \text{if } \uparrow 0 < n \text{ then } ns \text{ else } (ms[\uparrow 0 := \uparrow 0 - n]) \rangle}$$

$(sequence)$

$$\frac{x :: \text{Int}(m) \mid y :: \text{Int}(n) \mid m \leq n}{\texttt{[}x\,\texttt{..}\,y\texttt{]} :: \langle \text{Int}, [n{-}m{+}1] \rangle}$$

$(take, drop)$

$$\frac{xs :: \langle \alpha, n : ns \rangle \mid y :: \text{Int}(m) \mid 0 \leq m \leq n}{\begin{array}{l} \texttt{take}\ y\ xs :: \langle \alpha, m : ns \rangle \\ \texttt{drop}\ y\ xs :: \langle \alpha,\; n{-}m : (ns[\uparrow 0 := \uparrow 0 + m]) \rangle \end{array}}$$

$(indexing)$

$$\frac{xs :: \langle \alpha, n : ns \rangle \mid y :: \text{Int}(m) \mid 0 \leq m < n}{xs\ \texttt{!!}\ y :: \langle \alpha, ((ns[\uparrow 0 := m])[\uparrow i := \uparrow(i{-}1)]) \rangle}$$

$(concat)$

$$\frac{xss :: \langle \alpha, [n, m] \rangle \mid \text{fresh } i}{\texttt{concat}\ xss :: \left\langle \alpha, \left[\sum_{i=0}^{n-1} (m[\uparrow 0 := i]) \right] \right\rangle}$$

$(singleGen)$

$$\frac{f :: \text{Int}(i) \rightarrow \langle \alpha, ns \rangle \mid m :: \text{Int}(n{-}1)}{[\ f\ i \mid i \leftarrow [0..m]\ ] :: \langle \alpha, n : ((ns[\uparrow j := \uparrow(j{+}1)])[i := \uparrow 0]) \rangle}$$

Fig. 2. Derivation rules

the new reference $\uparrow 0$ refers to the list `x:xs` which has an additional element at the first position at the outermost level.

- (*head*) The size information for the first element is obtained by instantiating $\uparrow 0$ with 0 and renaming all other references because the nesting level has decreased.
- (*tail*) The length of the list is decremented and, to obtain the size information for an element, one has to refer to the original list with outermost index, which is incremented because it had an additional element at its head.
- (*null*) Derives a compile-time truth value stating whether the list is empty.
- (*length*) Derives the length of the list as a compile-time integer value.
- (*conditional1*) If, in a conditional, both alternatives share the same size, this rule establishes the size of the result without a need to know the value of the predicate at compile time.
- (*conditional2*) If the predicate is a compile-time value and both alternatives are lists with size information, the result is a list with size information, which is constructed by using the compile-time value of the predicate in the size expression itself.
- (*conditional3*) If the predicate and both alternatives are compile-time values, the result is a compile-time value.
- (*append*) The concatenation of two lists is a list whose size is the sum of the sizes of both arguments, and the sizes of the sublists are defined by an alternation, which checks whether the element originated from the first or the second argument.
- (*sequence*) The length of the list containing the integers from m to n is $n-m+1$. This rule facilitates the exploitation of range information common in matrix and vector algorithms.
- (*take*) The size of the list is the first argument of `take` if it is a compile time value.
- (*drop*) The size of the result is the difference between the original size and the number of elements dropped.
- (*indexing*) This rule is similar to (*head*), but it refers to an arbitrary index instead of index 0.
- (*concat*) The size of the result list is the sum of the sizes of the sublists. The sum quantifier can be eliminated if the sizes of the sublists are piecewise polynomials, e.g.
- (*singleGen*) List comprehensions are simplified after lambda lifting to the form used in this rule. The elements of the resulting list are expressed by an auxiliary function f which makes the variable i bound by the generator accessible from outside. The size of the result is the

size n of the generator list. The sizes of the sublists are obtained by renaming the references due to increase of nesting level and substituting the variable i by its scope-independent counterpart $\uparrow 0$.

Note that these rules do not cover all possibilities:

- If run-time values influence the length of a list, a default rule has to be used which does not derive size information. This may require a dynamic space-time mapping for the respective part of the program.
- Some special situations, in which size information can also be derived, are not treated here. According rules can be added as needed.

3.4 Size inference of recursive functions

The application of our method to a recursive function leads to a recursive size function. Recursive size functions have the disadvantage that the simplification of expressions containing them is restricted. This property propagates to all size functions which depend on them. Size expressions are used as parameters for the space-time mapping, i.e., they have to be evaluated during the execution.

The fact that the size function is recursive does not necessarily incur a lot of effort at run time. Consider the case that the original function and the corresponding size function both have the same recursion depth. Let the original function have two recursive calls but the size function only one. Then, the number of recursive calls of the size function is only the logarithm of the number of recursive calls of the original function. This case occurs, e.g., if in a $\mathcal{DC}$ algorithm the size does not depend on run-time values and the original function generates two subproblems of the same size. Then, the result of both subproblems will have the same size which means that it suffices to compute the size only once.

If the size function is recursive in only a single parameter and linear, it can be solved by a standard mathematical tool, e.g., with the *rsolve* function of Maple [6]. We do not know of a mathematical tool which is able to solve recurrence equations in multiple variables. This case arises in the size inference of a function with two arguments, which calls itself recursively modifying both arguments.

Take the following example of function `double` [22]:

```
double xs = if null xs then []
               else head xs : head xs : double (tail xs)
```

The abstracted size information is:

$\texttt{double} :: \langle \alpha, [n] \rangle \rightarrow \langle \alpha, [s(n)] \rangle$
$\quad\quad$ where $s(n) \;=\;$ if $n = 0$ then 0 else $2 + s(n-1)$

The solution of the recurrence equation in s delivers:

$\texttt{double} :: \langle \alpha, [n] \rangle \rightarrow \langle \alpha, [2n] \rangle$

In general, annotations cannot be avoided if recursive size functions must be solved. This does not mean that the programmer has to supply them. We prefer that the programmer should replace all recursion that is to be parallelized by the composition of predefined skeletons (combinators). The person defining a new skeleton must annotate it with a corresponding, non-recursive size function.

Take the following, better definition of function `double` using a list comprehension:

```
double xs = [ xs!!i | i<-[0..length xs -1], _<-[0,0] ]
```

This leads to the same result without the need to solve recurrence equations. Furthermore, a later parallelization does not have to establish that the only dependences from the ith element of the input list are to the elements with indices $2i$ and $2i+1$ of the output.

4 Example: Inversion of lower triangular matrices

Our example problem is the inversion of lower triangular matrices, which plays an important role in the inversion of full matrices, LU- and QR-decomposition [8, Subsect. 5.3.2].

4.1 A $\mathcal{DC}$ algorithm for the inversion of lower triangular matrices

A matrix of size 1×1 is inverted by inverting its element. A lower triangular matrix of size $n \times n$ with $n > 1$ is inverted as shown in Fig. 3; the algorithm must be applied recursively to two triangular matrices `a` and `c` of smaller size [8, Subsect. 5.3.2]. Therefore, we can express the algorithm in terms of a $\mathcal{DC}$ skeleton.

The inversion of lower triangular matrices can be defined by the following $\mathcal{HDC}$ (and Haskell) function `minv`. A triangular matrix is represented by a list of its rows; the ith row (counting from 0) is a list of $i+1$ column elements. Therefore, the type of a lower triangular matrix annotated with size information is $\langle \alpha, [n, \uparrow 0+1] \rangle$, where n is the number of rows.

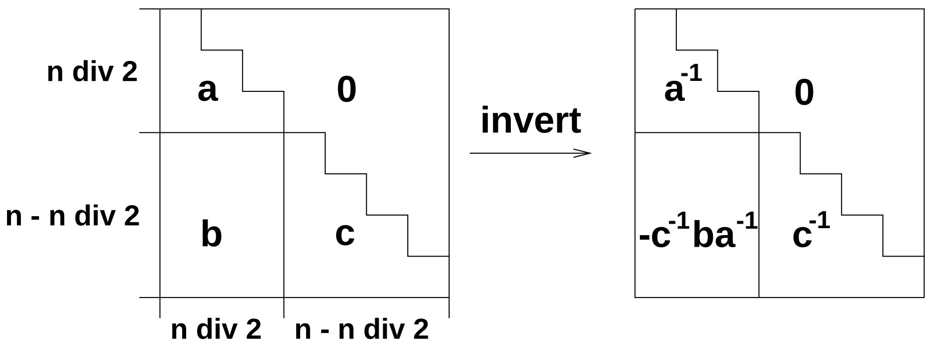

Fig. 3. Inversion of lower triangular matrices (n=7)

```
minv :: [[Double]] -> [[Double]]
minv xs =
 let b = map (map (\x -> 1.0 / x))
     d x = let m  = length x `div` 2
               a  = take m x
               bc = drop m x
               c  = map (drop m) bc
           in [a,c]
     c x [ai,ci] = let m   = length x `div` 2
                       bc  = drop m x
                       b   = map (take m) bc
                       bai = mmrd b ai
                       nci = map (map (\x -> (-x))) ci
                       ll  = mmld nci bai
                       low = zipWith (++) ll ci
                   in ai ++ low
 in dc_fixed_degree_count_levels 2 b d c
                                 (ilog2 (length xs)) xs
```

`minv` is an instance of our skeleton `dc_fixed_degree_count_levels`. This is a slight modification of our skeleton `dc3` in [13], i.e., a $\mathcal{DC}$ skeleton which imposes a fixed degree of problem division and a fixed number of levels of recursion. It is called with the following actual parameters: the division degree 2, the basic function `b`, the divide function `d`, the combine function `c`, the number of levels of recursion, and the input data `xs`. The number of levels is computed using function `ilog2`, which delivers for an argument n the smallest natural number whose power of 2 is at least n. The basic function inverts matrices of size $0{\times}0$ or $1{\times}1$. As can be seen in

Fig. 3, for the lower left part of the result, we need a function to multiply a full matrix with a lower triangular matrix from the left and from the right side. The functions are named `mmld` and `mmrd`, respectively, and they are defined in terms of list comprehensions.

Let us discuss the parts of function `minv` in more detail:

- **basic function** `b`:
  Inverts a matrix of size 0×0 or 1×1.
- **divide function** `d`:
  `a`/`bc` contains the left/right part of the outer list which represents the upper/lower half of the matrix. `a` is already a lower triangular matrix. `c` is obtained from `bc` by dropping the first `m` column elements in each row. The subproblems consist of the upper left part (`a`) and the lower right part (`c`) of the original matrix.
- **combine function** `c`:
  The arguments of the combine function are the original matrix in `x` and the inverses `ai` and `ci` of the lower triangular matrices `a` and `c`, respectively. `bc` is the same as in the divide function. `b` is the lower left part of the original matrix, obtained from `bc` by taking the first `m` column elements in each row. `bai` is the result of multiplying the full matrix `b` with the lower triangular matrix `ai`. `nci` is the negative of `ci`. `ll` is the lower left part of the result, which is the product of the lower triangular matrix `nci` and the full matrix `bai`. `low` is the lower half of the result matrix, obtained by concatenating `ll` and `ci` row by row. The result matrix is the concatenation of the upper part `ai` with the lower part `low`.

4.2 Deriving the size information

Because the code at this compilation phase is voluminous and not easy to read, we base our presentation on the original source code. Also, we restrict the presentation to the two most important parts, i.e., the divide function `d` and the combine function `c`.

We assume that the programmer has provided the following annotation for `minv`: $\langle Double, [p, \uparrow 0+1]\rangle \rightarrow \langle Double, [p, \uparrow 0+1]\rangle$.

divide function `d`**:** Using the annotation $\langle Double, [p, \uparrow 0+1]\rangle$ for the argument of `minv`, which in the $\mathcal{DC}$ schema becomes the argument of `d`, we can derive successively the sizes of the other expressions in `d` as shown in Fig. 4, using a fresh name n for p. When we cite rules in bold font, we refer to the function in the source text which is transformed before size inference.

value	size	rule(s)
`x`	$\langle Double, [n, \uparrow 0 + 1] \rangle$	—
`length x`	$\text{Int}(n)$	*length*
`m`	$\text{Int}(n \text{ div } 2)$	arithmetic
`a`	$\langle Double, [n \text{ div } 2, \uparrow 0 + 1] \rangle$	*take*
`bc`	$\langle Double, [n - n \text{ div } 2, \uparrow 0 + 1 + \text{n div } 2] \rangle$	*drop*
`c`	$\langle Double, [n - n \text{ div } 2, \uparrow 0 + 1] \rangle$	***map***, *drop*
`[]`	$\langle Double, [0, 0, 0] \rangle$	*nil*
`[c]`	$\left\langle Double, 1 : \begin{cases} \uparrow 0 = 0 : & [n - n \text{ div } 2, \uparrow 1 + 1] \\ \text{otherwise} : \bot & \end{cases} \right\rangle$	*cons*
`[a,c]`	$\left\langle Double, 2 : \begin{cases} \uparrow 0 = 0 : & [n \text{ div } 2, \uparrow 1 + 1] \\ \uparrow 0 = 1 : & [n - n \text{ div } 2, \uparrow 1 + 1] \\ \text{otherwise} : \bot & \end{cases} \right\rangle$	*cons*

Fig. 4. Deriving the size information for `d`

Deriving the size of the result of the recursive call: The size information for the result of a recursive function can be verified by structural induction. In the case of $\mathcal{DC}$, one has to verify the size behaviour of the basic function as induction base.

The function in the basic case is `map (map (\x -> 1.0/x))`. Two applications of rule ***map*** show that the basic function transforms the sizes in the way the annotation for `minv` prescribes:

$$\langle Double, [p, \uparrow 0 + 1] \rangle \to \langle Double, [p, \uparrow 0 + 1] \rangle.$$

Next, we have to check the induction step. We have to apply rule ***map*** here because, in the $\mathcal{DC}$ scheme, the algorithm is being mapped recursively. It turns out that the list of subproblem solutions has the same shape as the list of subproblems. We have two subproblems. For the first, we have to unify the formal parameter $[p', \uparrow 0 + 1]$ of the annotation with the actual size $[n \text{ div } 2, \uparrow 0 + 1]$ of the first subproblem, for the second, $[p'', \uparrow 0 + 1]$ with $[n - n \text{ div } 2, \uparrow 0 + 1]$. We must use copies of the annotation in which we replace p by p' resp. p'', because the annotation is used in two different contexts. Applying the size function given by the annotation, which is the identity, delivers $[p', \uparrow 0 + 1]$ resp. $[p'', \uparrow 0 + 1]$, and backsubstitution $[n \text{ div } 2, \uparrow 0 + 1]$ resp. $[n - n \text{ div } 2, \uparrow 0 + 1]$. Now, because this matching is done inside a map, according to the rule ***map***, $\uparrow 0$ of an element corresponds to $\uparrow 1$ in the list of the two subproblem sizes.

combine function `c`: We use the induction hypothesis to obtain the sizes of `ai` and `ci` and then derive successively the other sizes in Fig. 5.

value	size	rule(s)
`x`	$\langle \mathit{Double}, [n, \uparrow 0 + 1] \rangle$	—
`[ai,ci]`	$\left\langle \mathit{Double}, 2 : \begin{cases} \uparrow 0 = 0 : & [n \text{ div } 2, \uparrow 1 + 1] \\ \uparrow 0 = 1 : & [n - n \text{ div } 2, \uparrow 1 + 1] \\ \text{otherwise} : \bot & \end{cases} \right\rangle$	hypothesis
`ai`	$\langle \mathit{Double}, [n \text{ div } 2, \uparrow 0 + 1] \rangle$	*head*
`[ci]`	$\left\langle \mathit{Double}, 1 : \begin{cases} \uparrow 0 = 0 : & [n - n \text{ div } 2, \uparrow 1 + 1] \\ \text{otherwise} : \bot & \end{cases} \right\rangle$	*tail*
`ci`	$\langle \mathit{Double}, [n - n \text{ div } 2, \uparrow 0 + 1] \rangle$	*head*
`length x`	$\text{Int}(n)$	*length*
`m`	$\text{Int}(n \text{ div } 2)$	arithmetic
`bc`	$\langle \mathit{Double}, [n - n \text{ div } 2, \uparrow 0 + 1 + \text{n div } 2] \rangle$	*drop*
`b`	$\langle \mathit{Double}, [n - n \text{ div } 2, \text{n div } 2] \rangle$	***map***, *take*
`bai`	$\langle \mathit{Double}, [n - n \text{ div } 2, \text{n div } 2] \rangle$	***mmrd***
`nci`	$\langle \mathit{Double}, [n - n \text{ div } 2, \uparrow 0 + 1] \rangle$	***map*** twice
`ll`	$\langle \mathit{Double}, [n - n \text{ div } 2, \text{n div } 2] \rangle$	***mmld***
`low`	$\langle \mathit{Double}, [n - n \text{ div } 2, \text{n div } 2 + \uparrow 0 + 1] \rangle$	***zipWith***, *append*
`ai ++ low`	$\langle \mathit{Double}, [n, \uparrow 0 + 1] \rangle$	*append*

Fig. 5. Deriving the size information for `c`

Completing the structural induction: We have obtained that the result of the combine function has the size specified in the annotation for function `minv`. Therefore, the induction proof is completed and the annotation is valid.

5 Conclusions

Size inference is an important technique in the static parallelization of data-parallel programs [7, 16]. With the restriction to affine size expressions, much work has been done on the polytope model [20, 10] and some work on generalized array structures such as data fields [21].

We are not aware of any related work on reasoning about non-affine size expressions of nested list structures, which occur, e.g., in the static parallelization of functional programs. Here, we have presented a method for deriving the sizes of nested lists. If complete size information is not available at compile time one can pursue a semi-dynamic approach by keeping auxiliary index information in tables initialized at run time. A related technique uses scans and segment descriptors [5].

Performance measurements of the compiled code are not available yet because the phases of space-time mapping and code generation are still under development. Preliminary experiments with divide-and-conquer skeletons in C/MPI on a 1024-processor network indicate that the speed-up will scale well with increasing problem size.

The main point which makes us expect good performance is that we obtain very simple target code via the direct correspondences, e.g., between Haskell list comprehensions and C loop nests. The intermediate code will contain a large amount of auxiliary functions, but most of them are eliminated by inlining, using common subexpression elimination to avoid code explosion.

Acknowledgements

The German Research Foundation (DFG) is supporting this work under project RecuR2. We thank the Paderborn Center for Parallel Computing for machine access. We are grateful to John O'Donnell for his suggestion to replace explicit recursion by the use of combinators, as far as possible, and Peter Kleinschmidt for remarks on recurrence equations. We are indepted to Sergei Gorlatch and some participants of the IFL'98 workshop, especially Thorsten Zörner, for comments on an earlier version of this paper.

References

1. Hendrik P. Barendregt. *The Lambda Calculus.* Studies in Logic 103. North-Holland, 1984.
2. Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. *ACM SIGPLAN Notices*, 32(8):25–37, 1997. *Proc. ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'97).*
3. Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Series in Computer Science. Prentice Hall Europe, second edition, 1998.
4. Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.
5. Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddharta Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel and Distributed Computing*, 21(1):4–14, April 1994.
6. Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V Language Reference Manual.* Springer-Verlag, 1991.
7. Siddharta Chatterjee, Guy E. Blelloch, and Allan L.Fisher. Size and access inference for data-parallel programs. Technical Report CMU-CS-91-118, Department of Computer Science, Carnegie-Mellon University, 1991.
8. Michel Cosnard and Denis Trystram. *Parallel Algorithms and Architectures.* Int. Thomson Computer Press, 1995.

9. Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. on Principles of Programming Languages (POPL'82)*, pages 207–212. ACM Press, 1982.
10. Paul Feautrier. Automatic parallelization in the polytope model. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 79–103. Springer-Verlag, 1996.
11. Message Passing Interface Forum. MPI: A message passing interface. In *Supercomputing*, pages 878–883. IEEE Computer Society Press, 1993.
12. Christoph A. Herrmann and Christian Lengauer. On the space-time mapping of a class of divide-and-conquer recursions. *Parallel Processing Letters*, 6(4):525–537, 1996.
13. Christoph A. Herrmann and Christian Lengauer. Transformation of divide & conquer to nested parallel loops. In Hugh Glaser, Pieter Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementation, Logics, and Programs (PLILP'97)*, Lecture Notes in Computer Science 1292, pages 95–109. Springer-Verlag, 1997.
14. Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5):T1–T53, May 1992.
15. John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proc. 23rd Ann. ACM Symp. on Principles of Programming Languages (POPL'96)*, pages 410–423. ACM Press, 1996.
16. Barry Jay and Milan Sekanina. Shape checking of array programs. *Australian Computer Science Comm.*, 19:113–121, 1997. *Proc. Australasian Theory Seminar 1997*.
17. Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA'85)*, Lecture Notes in Computer Science 201. Springer-Verlag, 1985.
18. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall Int., 1987.
19. Simon L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In Hanne Riis Nielson, editor, *Programming Languages and Systems (ESOP'96)*, Lecture Notes in Computer Science 1058, pages 18–44. Springer-Verlag, 1996.
20. Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.
21. Björn Lisper. Data parallelism and functional programming. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 220–251. Springer-Verlag, 1996.
22. Björn Lisper and Jean-François Collard. Extent analysis of data fields. In Baudouin Le Charlier, editor, *Static Analysis (SAS'94)*, Lecture Notes in Computer Science 864, pages 208–222. Springer-Verlag, 1994.
23. Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. on Programming Languages and Systems*, 4(2):258–282, April 1982.