UNIVERSITÄT PASSAU

Fakultät für Mathematik und Informatik

DISSERTATION

for acquisition of the degree Dr.rer.nat.

The Skeleton-Based Parallelization of Divide-and-Conquer Recursions

Christoph Armin Herrmann

Advisor:	Prof. Christian Lengauer, Ph.D.
Submitted:	15 June 2000
Defense:	12 July 2000
Revised:	14 March 2001

Abstract

The thesis of this dissertation is that an effective exploitation of the inherent parallelism in algorithms is possible at the high level of functional programming. Parallelism is a promising concept for achieving a high acceleration of computations, but most parallel programming languages and tools of today address mainly experts in parallel computing. This limits the use of massive parallelism by the average programmer.

Our approach to overcome this limit are skeletons, also called combinators or templates. They are polymorphic program schemata that have an efficient, possibly parallel, implementation. Skeletons are embedded into a functional source language and receive a special treatment in the compilation of the program to target code. We distinguish two views of a skeleton: the user's view and the implementer's view. A skeleton appears in the program as a single function call which is specialized by the user with problem-specific customizing functions. We focus on skeletons for the divide-and-conquer ($\mathcal{DC}$) paradigm, because $\mathcal{DC}$ algorithms provide a high potential for parallelism. In $\mathcal{DC}$, two of the customizing functions describe how the problem is divided into subproblems and how the partial solutions are combined. Starting from a skeleton that represents a general kind of $\mathcal{DC}$, we derive, successively, special cases which lead to an efficient parallel implementation. The functional definitions of the skeletons and the programs which use them are denoted in the language Haskell.

The implementer views a skeleton as an additional module of the compiler, which generates, for every skeleton application, an appropriate, optimized implementation in the target language. Thus, know-how of a problem domain, e.g., $\mathcal{DC}$, can be added to the compiler. The straight-forward way to parallelize $\mathcal{DC}$, by simply spawning independent tasks, can cause a serious load imbalance. Load balance can be established by an appropriate mapping of operations to time steps and processors. For the more specialized skeletons, this can be done completely at compile time. Otherwise, the space-time mapping can be expressed in dependence of run-time information. We transform the recursive definition of each of the $\mathcal{DC}$ skeletons presented to an index-based form in order to apply a space-time mapping to the set of operations. A transformation consists of a sequence of semantics-preserving rule applications. The power of recursion motivates the introduction of a computational model which is more sophisticated than the polytope model used for nested loops. The new model supports a potentially unbounded dimensionality of the index space.

A subset of the functional language Haskell, called $\mathcal{HDC}$, is used for experiments with example programs. Contrary to Haskell, $\mathcal{HDC}$ is strict. This is necessary to guarantee that the space-time mapping made at compile-time is respected by the execution. We present compilation techniques that transform polymorphic, higher-order functional programs into functional programs which can easily be compiled into C and linked together with the skeleton implementations.

Experimental results are presented for Karatsuba's polynomial multiplication, the n queens problem, the maximum independent set problem, the convex hull computation and sorting. It turns out that significant speedups can be achieved for realistic applications, but under special conditions on the algorithmic structure. Some of the skeletons enforce these conditions by construction. For the others, there is still a high potential for optimization left for the future.

Acknowledgements

Many people have supported and encouraged me while writing this thesis. I am indebted to my parents, friends and many members of the University of Passau.

Especially, I would like to thank my advisor, Prof. Christian Lengauer, Ph.D., for his confidence in my ideas from the beginning. He let me benefit from his knowledge in writing research papers. Thanks to his influence, this thesis aims to explain the concepts simply, in addition to the technical details. I am very grateful that he brought me together with many important people in the communities of functional and parallel programming.

Dr. John O'Donnell from the University of Glasgow taught me how to do powerful and sound reasoning in the functional language Haskell, which had a great impact on the quality and productivity of my proofs. His lectures at the University of Passau provided a lot of insight in the philosophy of functional programming which one cannot get only by reading the standard literature.

Four extremely motivated students deserve my special acknowledgement: Marian Musiol, Robert Günz, Christian Schaller and Jan Laitenberger. I owe them many thanks for their hard work, which made it possible to present working compiler technology in this thesis.

Peter Faber was always ready to help me in many difficult software installation problems. Prof. Dr. habil. Sergei Gorlatch deserves a special mention for reading several of my draft papers and pointing out a lot of connections to related work and potential sources for improvement. I also received valuable feedback from the other former and current members of our group: Nils Ellmenreich, Dr. Martin Griebl, Dr. Ulrike Lechner and Christoph Wedler.

Thanks to the Paderborn Center for Parallel Computing for access to their parallel machines and especially to Axel Keller and Andreas Krawinkel for their help.

This work was supported by the German National Research Foundation (DFG) with a four-year grant (project RecuR2).

Contents

Chapter 1

Introduction

1.1 Motivation

One of the main challenges of computer science is to increase ever more the execution speed of computations. Often, the speed determines whether the result of the computation remains of interest, e.g., for real-time applications, weather forecasting or large data base operations. Where systems are used to assist people, e.g., in design or optimization tasks, the speed has a significant influence on the productivity. In both situations, the expense spent for an increase in computation speed pays for a gain in productivity or time to market.

There are multiple possibilities to gain speed, e.g., by improving the performance of single processors or by connecting some or even many processors and letting them share a computational task. These possibilities differ in the amount of potential they provide. For most work in everyday life, especially for office applications, increasing speed requirements can be met with improvements of single processor architectures, exploiting parallelism at the instruction level, i.e., invisible to the programmer. This may be a reason for the widespread lack of attention to program parallelization.

As an example, let us consider a large-scale computation that we want to perform ten times faster than it would be possible on a high-quality personal computer of today. There are two obstacles for achieving this by the choice of a single faster processor: (1) one has to buy a processor which is extremely expensive or (2) one has to wait several years until a processor with the required speed is available.

Using multiple processors to gain speed, one does not have to wait and, as long as a dense communication network is not considered, the expense increases at most proportionally with the computational power.

If communication comes into play, a severe physical limitation enters the scene: the limit of the speed of light for signal transmission in combination with the restrictions of the three-dimensional space (Feldman and Shapiro, 1992). We assume that a fixed volume in space can only store a bounded number of processors. To achieve an arbitrary acceleration, the number of processors must be arbitrarily large. The consequence for a three-dimensional space is, that then the distance between any two processors cannot be bounded. Due to the restricted speed of light, the communication times between any two processors are then arbitrarily long, which can restrict the speedup.

As a consequence, modeling the communication times of data dependences by a fixed amount of real time, without imposing a constant bound on the number of processors, can lead to extremely unrealistic results. A similar consideration applies, if the amount of information units transmitted is not taken into account for the communication time. The complexity results presented in this thesis demonstrate the limitations dependent on different architectural parameters; see Sect. 4.8. However, compared to a single processor, the speed that can be achieved is superior by orders of magnitude, as the experimental results presented in this thesis demonstrate.

Programming languages popular today are mostly imperative. They are tailored for an efficient sequential execution, but cause severe difficulties for a compiler-driven parallelization. On the other hand, explicit parallel programming with additional language constructs complicates program design. The concept of *skeletons* (or *combinators*), i.e., predefined program schemata with an efficient parallel implementation, permits the user to specify the points where parallelism can be applied but leaves the actual organization of the parallel computation to the compiler. A similar concept named *template* exists in the world of imperative programming. We focus on skeletons for *divide-and-conquer* ($\mathcal{DC}$) algorithms, which contain a high potential of natural parallelism. Skeletons receive much of their expressive power from application specific, functional parameters. This motivates the use of a functional language. See Hudak (1989) for a survey of the development of functional languages and Hughes (1989) for a tutorial with example programs. We use the functional language *Haskell* (Haskell, 2000; Hudak and Fasel, 1992; Thompson, 1996; Bird, 1998; Peyton Jones and Hughes, 1999; Hudak, 2000).

In the application program, a skeleton appears just like an ordinary function call, but the implementation of the skeleton can control the parallel execution precisely, by laying out the individual computations in time and space (*space-time mapping*). Space-time mapping has received much attention in the parallelization of nests of for and while loops. The notion of a skeleton is not common in loop parallelization, and, until now, the skeleton approach has been viewed as an alternative approach. However, in this thesis, it is shown that the algorithmic idea behind a for or while loop can be captured by a *higher-order* function. Our way of nesting these higher-order functions is analogous to the construction of a nest of loops. Loops are not in the center of our interest here, they only serve as a useful construct to express iterating computational structures.

The expressive power of the divide-and-conquer skeletons investigated in this thesis is higher than that of nested for loops and requires a more sophisticated formalism. Space-time mapping is only applied to special forms of $\mathcal{DC}$ recursions, which promise a high potential for an efficient parallelization.

1.2 Conventions, terminology and notation

Throughout the thesis, we assume all data structures to be finite.

Definitions and the introduction of symbols appear in the text where they are used first. Basic notions of computer science like *termination*, *polynomial time*, *directed acyclic graph (DAG)* etc., are not defined except for those whose precise definition is crucial for large parts of the thesis, e.g., the notion of $\mathcal{DC}$. Also, common mathematical notation and symbols like $(\exists\ i \in \mathbb{N} : i^2 = i)$ are assumed to be known. Some parts of the thesis contain notation of the functional programming language Haskell, which we cannot explain in detail. The reader is referred here to the Haskell report (Peyton Jones and Hughes, 1999).

A transformation rule is written as a horizontal line which separates two expressions. The expression above the line is the object of the transformation, the expression below the result.

The *asymptotic notation* (Reischuk, 1990) refers, for a given function f, to the set of all functions with the same *asymptotic complexity* (Θ), upto the complexity of f (O), above the complexity of f (ω) or equal or above the complexity of f (Ω).

All indexing with natural numbers starts with 0, i.e., the term *first element* of a *list*, *vector*, etc. is equivalent to the term *element 0*. This applies also to children of a node in a tree, components in a tuple, dimensions in a vector space, etc.

We use sanserif font for top-level definitions in programs like map (predefined in *Haskell*), or dcA (our definition). In mathematical formulas, functions like the logarithm (log) appear, as usual, in Roman font. Variables are written in italics, except from type variables, which are named α, β and γ.

Haskell programs are written in *layout style*, in which indentation is used to form a block of syntactic elements. Sometimes, extra parentheses are used for clarity.

The central definition of a term is underlined but, as a matter of style, often not set apart in a dedicated definition section. Before the precise definition, a term can be introduced informally, just to indicate its presence or to explain its role. Expressions, which are not already in a special font, are emphasized by writing them in italics. The first occurrence of a term is not necessarily the only point where it is emphasized.

In the index, a bold number refers to an important occurrence of a term and an underlined number to the point of its definition. Terms in the index are not always literally identical with their occurrences in the text and not every occurrence in the text is referenced.

The end of environments for lemmas, examples, etc. is marked by $\square$.

1.3 Objectives

This thesis investigates the *parallelization* of $\mathcal{DC}$ *recursions*. The parallelization is based on particular program structures, so-called *skeletons*. Also, we want to demonstrate the *practical applicability* of the skeleton approach for the efficient parallelization of $\mathcal{DC}$ recursions. Let us specify the terms just introduced in detail.

1.3.1 Parallelization

In our understanding, parallelization is the method of analyzing a given algorithm with respect to independent computational parts and synthesizing control structures that exploit the independence of these parts by executing them at the same time on different processors. Formally, *independence* is defined in terms of a *partial order* on the parts of the computation. For two parts, say A and B, A is defined to be *smaller* than B if, and only if, B uses directly or indirectly the value produced by A. Obviously, if A is smaller than B, A must be computed before B to provide the value for B. If A and B are *incomparable* with respect to the partial order, they are called independent and could, in principle, be executed at the same time on different processors.

The main goals of a parallelization are *speedup*, *efficiency*, *time optimality* and *cost optimality*. Speedup and efficiency quantify the performance of a parallel program for an increasing number of processors for a fixed input size. Time optimality and cost optimality quantify the resource (time and space) consumption of a parallel program for an increasing input size.

Definition 1.1 (Speedup)
Let $\hat{T}$ be the execution time of the fastest sequential algorithm which solves a particular problem. Let $T(p)$ be the execution time of our parallel algorithm on p processors of the same type as before, for the same problem and on the same input. Then $\hat{T}/T(p)$ is called the (real) speedup on p processors (Akl, 1989). □

We use the so-called *relative* speedup instead of the real speedup in our considerations. A detailed discussion follows in Sect. 2.7.

Definition 1.2 (Efficiency)
If $S(p)$ denotes the speedup on p processors, then the term $S(p)/p$ is called the efficiency on p processors and reflects the average degree of utilization of a processor. □

The following measures are at first of theoretical interest, because they are based on the assumption that an unlimited number of processors is available. However, it is a promising strategy for parallel programming to start from an algorithm that fulfills the theoretical optimality criteria and then perform a mapping of the processors required by the algorithm (*virtual processors*) onto the processors provided by the real machine. There are several possibilities, e.g., (1) to control the granularity by introducing an artificial problem size that causes the algorithm to choose the number of processors available or (2) to emulate multiple virtual processors on a single real processor. The method of space-time mapping deals with such mappings in general, without problem-specific knowledge.

Definition 1.3 (Time Complexity)
The time complexity of a (parallel) program is given by $\Theta(time\ n)$, where $time\ n$ is the total number of discrete, equidistant time steps required for an input of size n for a particular parallel algorithm. It is assumed, that the algorithm chooses the number of processors only dependent on the value of n. □

Definition 1.4 (Space Complexity)
The space complexity of a parallel program is given by $\Theta(space\ n)$, where $space\ n$ is the total number of processors used for an input of size n for a particular parallel algorithm. □

In the analysis of sequential algorithms, the notion of *space complexity* stands for the number of memory cells used. We do not use this meaning here.

Definition 1.5 (Cost complexity)
If, for an input of size n, the time complexity is $\Theta(time\ n)$ and the space complexity is $\Theta(space\ n)$, then the cost complexity is $\Theta(time\ n \cdot space\ n)$. □

In the definition of the cost complexity, we assume that a particular set of processors is reserved exclusively for a single purpose, i.e., that idle times of processors cannot be used by other computations. Thus, the definition of the cost complexity does not necessarily reflect the total number of operations executed, because processors may be idle.

Definition 1.6 (Time optimality)
We say that an algorithm is time-optimal, if its time complexity is asymptotically minimal in the set of all algorithms that solve the given problem. □

Definition 1.7 (Cost optimality)
We say that an algorithm is cost-optimal, if its cost complexity equals asymptotically the cost complexity of the time-optimal sequential algorithm. □

The following properties of cost-optimality are worth noting:

1. Time-optimal sequential algorithms are always also cost-optimal.

2. Cost optimality requires that idle times and duplication of work are limited such that they do not influence the asymptotic behavior.

3. In a parallel execution, there is a tradeoff between time and cost optimality. While time optimality can be achieved only beyond a particular number of processors, cost optimality may require a limit on the number of processors. The goal is to achieve time and cost optimality in one implementation, which is not always possible because a time-optimal implementation can require a huge waste of processors.

4. In practice, the number of processors is limited by a constant. In this case, every parallel algorithm that is not cost-optimal is outperformed by an optimal sequential algorithm in time complexity.

1.3.2 Recursion

Recursion is a principle of definition in which the defining expression contains the object being defined. To avoid any possible confusion with object orientation here, especially when dealing with the notion of an *instance*, we restrict our discussion of recursion to functions, without loss of generality: Recursive functions can be employed to construct large objects from simpler objects of the same type, e.g., regular graphs with a particular property.

A recursively defined function or problem consists of a set of different *instances*, which can be demonstrated quite well with the example of the *Fibonacci function* (fib).

Example 1.1 (The fib function)
$\mathsf{fib} \in \mathbb{N} \rightarrow \mathbb{N}$
$\mathsf{fib}\ n = \underline{\mathtt{if}}\ n<2\ \ \underline{\mathtt{then}}\ n$
$\qquad\qquad\qquad\ \underline{\mathtt{else}}\ \mathsf{fib}\ (n-2) + \mathsf{fib}\ (n-1)$

Let us identify the instances of the recursion via the argument n. fib is defined only on the natural numbers. The basic cases of the recursion are for $n = 0$ and $n = 1$. Obviously, termination of the recursion and, thus, a sound definition of fib is guaranteed by choosing the identity on the input as a decreasing function into the natural numbers. □

fib shares with several other problems like the *Towers of Hanoi* the property that a problem instance is naturally identified via the input, which sometimes lead to misinterpretation in the following two ways:

- The first mistake that can be made is to assume that the theory of *recurrence equations* on integers will always be helpful in the *elimination of recursion*. The elimination of recursion in the examples of fib and the Towers of Hanoi benefits from multiple occurrences of identical subproblems. In general, this is not the case. E.g., although, for a fixed n, the subproblem of sorting n numbers may occur multiple times in a sorting algorithm, each of these subproblems has to be solved individually, since the input of the problem is not only n but contains also the values of the numbers to be sorted. Even for the n *queens* problem, a reuse of a subproblem solution is not possible, since additional, different constraints are imposed on the subproblems.

  In general, the elimination of recursion as a prerequisite for a parallelization must respect the call structure and the dependences of the algorithm. We are convinced that the skeleton approach is helpful here, since skeletons reflect particular call structures.

- The second erroneous conclusion is that a parallelization of fib must be inherently inefficient because the recomputation of identical subproblems leads to an *exponential complexity*. However, a recomputation is not necessarily a consequence of a parallelization. It only results from an application of $\mathcal{DC}$ here, which is not the appropriate paradigm for that particular problem, as discussed in Sect. 1.3.3.

In general, a *problem instance* is a restriction of the problem domain to a subset. Usually, elements of this subset have commonalities, e.g., the same size or the same computational effort. Sometimes, *structural parameters* are added explicitly to the definition of a problem to indicate how to construct the instances. E.g., the problem of multiplying two square matrices of the same size can be refined into instances by introducing a structural parameter n and defining instance n to be the multiplication of two $n \times n$ matrices.

A simple iterative computation of the Fibonacci numbers uses the numbering of the problem instances to construct the partial order of operations: at first, $\mathsf{fib}\,0$ and $\mathsf{fib}\,1$ are computed, then $\mathsf{fib}\,2 = \mathsf{fib}\,0 + \mathsf{fib}\,1$, $\mathsf{fib}\,3 = \mathsf{fib}\,1 + \mathsf{fib}\,2$ etc. This method works for several recursive functions, but must not be generalized. E.g., in the multiplication of matrices, this would erroneously introduce dependences between independent subproblems which are assigned to the same instance. Thus, we need to base the partial order on the call structure instead. We will later use the notions *call tree* and *call graph* to describe this partial order.

1.3.3 Divide-and-Conquer

The $\mathcal{DC}$ paradigm prescribes that a problem instance is recursively divided into independent parts of the same type (divide phase) until they are small enough to be solved directly (basic phase). The subproblem solutions are recursively combined to the solution of the original instance (combine phase).

Examples of $\mathcal{DC}$, categorized by application area, are:

1. **Combinatorial search**: n queens (Ex. 3.5); satisfiability/tautology test of Boolean formulas (Aho et al., 1974)

2. **Sorting**: quicksort (Ex. 3.3), mergesort (Ex. 3.6), bitonic merge (Ex. 3.8), bitonic sort (Ex. 3.9)

3. **Geometry**: convex hull (Sect. 6.5), component labeling (Leighton, 1992)

4. **Algebra**: multiplication of numbers/polynomials (Sect. 6.1)/matrices (Aho et al., 1974), the fast Fourier transform (FFT) (Aho et al., 1974), red/scan with an associative operation (Blelloch, 1993)

5. **Graph theory**: maximum independent set (Ex. 3.4), graph coloring (Aho et al., 1974)

We identify the independence of subproblems as *the* source of parallelism induced by $\mathcal{DC}$. Some sorting algorithms are well known examples of $\mathcal{DC}$, since the problem of sorting can be defined elegantly as reducing large problem instances to smaller instances. The use of the term $\mathcal{DC}$ in this thesis may differ from other common uses. We do not view the class called *sequential divide-and-conquer* by Mou (1990b), in which both subproblems have to be solved in a predefined sequence, as a subclass of $\mathcal{DC}$ because the independence constraint is violated. His example of sequential divide-and-conquer, the Gaussian elimination, is motivated by geometric intuition: the division of a matrix into blocks. For us, the geometric view is of minor importance. More important is to separate strictly between nested recursion, e.g., the Ackermann function (ack), and $\mathcal{DC}$, and we have to insist on the independence of the subproblems to exclude nested recursion. However, when the recursion depth is limited by the size of the input data, as in the matrix example Mou proprosed, sequential divide-and-conquer does at least not lead to an extreme complexity like the one of the Ackermann function. If sequential divide-and-conquer should really be required, it is worth to define a dedicated skeleton for it.

To clarify our view of $\mathcal{DC}$, we distinguish it from two related paradigms: *branch-and-bound* and *dynamic programming*. Both differ from $\mathcal{DC}$ in that they require dependences between the subproblems.

1. **Branch-and-Bound** (Finkel and Manber, 1987; McKeown et al., 1991; Glas et al., 1992; Quinn, 1994; Berman and Paul, 1996; Shinano et al., 1996). Papadimitriou and Steiglitz (1982) describe it as follows:

   > The branch-and-bound method is based on the idea of intelligently enumerating all the feasible points of a combinatorial optimization problem. The qualification *intelligently* is important here because, as should be clear by now, it is hopeless to look at all feasible solutions. Perhaps a more sophisticated way of describing the approach is to say that we try to construct a proof that a solution is optimal, based on successive partitioning of the solution space. The *branch* in branch-and-bound refers to this partitioning process, the *bound* refers to lower bounds that are used to construct a proof of optimality without exhaustive search.

   It is the way this bound is applied that makes branch-and-bound different from $\mathcal{DC}$. The bound is obtained from a feasible solution of one branch of the search tree and used to prune another branch of the search tree, if it can be estimated that, with the decisions already taken there, the solution found cannot be improved. Thus, the independence condition is violated by exchanging information between branches.

   An application area for branch-and-bound is the class of 0/1 linear optimization problems, a restricted form of integer linear optimization (Papadimitriou and Steiglitz, 1982). The search space is divided into two parts with respect to a particular decision variable. In one branch, this variable is assumed to carry the value 0, in the other the value 1. A special instance of this class is the *traveling salesman problem* (Papadimitriou and Steiglitz, 1982): given a directed graph with weighted edges, find a tour of minimal cost which visits each node exactly once. Here, edges can be represented by decision variables. A variable has the value 1, if the corresponding edge is contained in the tour, otherwise the value 0.

2. **Dynamic Programming** (Aho et al., 1974; Papadimitriou and Steiglitz, 1982; Sedgewick, 1988; Rajopadhye, 1989). Aho et al. (1974) motivate the use of dynamic programming instead of $\mathcal{DC}$ for some cases the following way:

   > However, if the obvious division of a problem of size n results in n problems of size $n-1$, then a recursive algorithm is likely to have exponential growth. In this case a tabular technique called *dynamic programming* often results in a more efficient algorithm.
   >
   > In essence, dynamic programming calculates the solution to all subproblems. The computation proceeds from the small subproblems to the larger subproblems, storing the answers in a table. The advantage of the method lies in the fact that once a subproblem is solved, the answer is stored and never recomputed.

   The tabulation technique used in dynamic programming requires that a table entry produced in one branch of the call tree can be consumed in a different branch. Thus, the independence condition of $\mathcal{DC}$ is violated.

   A very simple example of tabulation is an improved version of function fib presented above in which, in a call of fib n, the first recursive call fib $(n-2)$ delivers not only the value of fib $(n-2)$ but also the value of fib $(n-3)$, such that the value of fib $(n-1)$ can be computed without further recursion. This technique has been generalized by Chin and Khoo (1993).

   A powerful application of dynamic programming is the Cocke-Younger-Kasami (CYK) algorithm (Atallah, 1999) for deciding if a string is an element of a formal language which is given by a context-free grammar. Here, common parts of overlapping substrings are analyzed only once.

1.3.4 Skeletons

A skeleton is a syntactic schema which can be used to define the structure of an object, e.g., an algorithm. In the context of a functional language, a skeleton is represented by a *higher-order* and, possibly, *polymorphic* function.

Definition 1.8 (Polymorphic function)
A function is called (parametrically) polymorphic if, and only if, its type contains at least one universally quantified type variable. In ad-hoc polymorphism, often called overloading, the type variables are restricted to a type class, e.g., the class of numbers for which addition is defined. □

It is not the purpose of a skeleton to perform combining operations on application-specific data. These operations are done by so-called *customizing* functions passed to the skeleton. Contrary to a parametrically polymorphic function, an overloaded function depends, at some point, on a dedicated implementation for each particular type.

Definition 1.9 (Higher-order function)
A function of type $\alpha \to \beta$ is called a higher-order function, if, and only if, the definition of α or β requires directly or via type definitions the use of an arrow ($\to$). □

No type instantiation of a higher-order function can prevent that at least one part of its argument or result can be used to carry a function.

The identity function id is first-order polymorphic, but not higher-order unless instantiated appropriately (Hammond, 1998). Since in the presence of polymorphism the distinction between *second order*, *third order*, etc., is of no use to us, we just use the notion *higher order*.

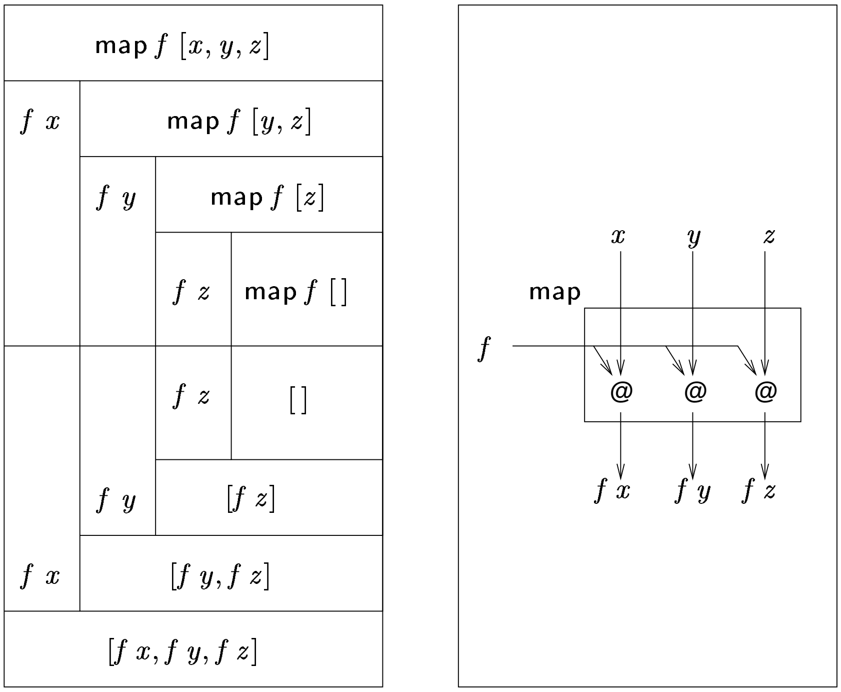
A skeleton can be viewed as a representative of a class of objects, i.e., those which can be defined in terms of the skeleton. If used as building blocks of programs, skeletons reflect particular programming paradigms like *divide-and-conquer*, *branch-and-bound*, *dynamic programming*, etc.

If a skeleton reflects a programming paradigm P, an object obtained after instantiation of the skeleton is a P *algorithm.* An algorithm is not entirely described by the structure imposed by P. The customizing functions add the problem-specific parts by instantiating the functional arguments of the skeleton. Take the following example of the map skeleton, which constitutes, in combination with conditional and recursion, the $\mathcal{DC}$ schema. The principle of map is to form a set of independent computations by applying a function f to every element of a list independently. The functional language Haskell (Peyton Jones and Hughes, 1999) is, due to its higher-order capabilities, well suited for defining skeleton-based programs. The definition of map in Haskell is:

Definition 1.10 (The map function)
map $\in (\alpha{\to}\beta) \to [\alpha] \to [\beta]$
map f [] = []
map f $(x{:}xs)$ = $f\ x$: map $f\ xs$
□

map takes a function f of type $\alpha \to \beta$ and delivers a function, which maps a list with elements of type α to a list with elements of type β. (The type arrow ($\to$) associates to the right, the function application operator (space between names) to the left.) Element i of the output list is the result of applying f to element i of the input list. map itself does not combine data elements, it only organizes the computation. Thus, map can be defined polymorphically, which is reflected by the type variables α and β. map is defined inductively by pattern matching on the list constructors [] and (:). If the input list is empty, the output list is also empty. If the list consists of an element x added in front of a list xs, the output is an application of f to x added in front of the result of applying map with f to xs.

The usual application of map in a user program will specify a fixed function f. For this reason, we call f a customizing function of the map skeleton. Users might prefer skeletons in the definition of customizing functions and also pass



(a) recursive unfolding (b) parallel execution

Figure 1.1: Views of the map skeleton

them around in the program in order to implement different strategies. Sacrificing type safety, a straight-forward solution for restricted combinations of skeletons could be implemented directly in a *low-level* language like C, but we prefer the presence of higher-order functions for methodical reasons:

1. to detect structural and logical inconsistencies in the use of a skeleton by a type error at compile-time,

2. to be aware of the parallelization problems that might arise, e.g., if data structures of unknown size are used (Sect. 4.4) and

3. to exploit properties that functions provide, e.g., as a *compact description* of large lists, instead of representing the elements individually (Sect. 5.2.3.2).

Let us take a closer look at the dependences in map to identify the possibilities for a parallelization. map is defined recursively on the list constructor (:). If this constructor is used also for the internal representation, a computational structure as depicted in Fig. 1.1(a) would be the result, i.e., elements could be accessed only in sequence. Each horizontal line corresponds to a separation of two time steps. The black-box property of skeletons helps us to abstract from the artificial dependences introduced by the list constructor (:) and to consider only the dependences between the input and the output data elements of the skeleton. In Fig. 1.1(b), adapted from O'Donnell (1997), an implementation of map is shown which applies f simultaneously to all elements. The application is depicted by the symbol @.

In part (a), the computation is laid out in time (due to the dependences), in part (b), it is laid out in (processor) space. To enable the layout in space, a direct, i.e., constant time, access to the elements of the list must be provided. The straight-forward implementation is by an array. Then, the constructors [] and (:) do not reflect the structure of the representation. To emphasize the change of implementational properties one could use the name *finite sequence* instead of *list*. Since the functional properties do not differ from the ones of lists, we simply keep the name *list*. If both implementations are required in the same program, a type annotation could be used for distinction.

The `map` example demonstrates that it is useful to eliminate artificial dependences in a computational structure. The same holds for $\mathcal{DC}$. Some kinds of $\mathcal{DC}$ provide a regularity that makes it possible to replace the structure imposed by recursion by another structure with fewer dependences. Therefore, we classify $\mathcal{DC}$ with respect to restrictions of the computational structure and define each class by a specialized skeleton. Programs expressed in terms of skeletons inherit partially their performance properties.

1.3.5 Practical applicability

It is a commonly held belief that a practical approach to parallelism must address non-academic problems, be easy to use, and yield speedups which justify the employment of a parallel system.

1. **Relevance for non-academic problems.** The applications of interest can be complex and involve a framework of heterogeneous tools. We see our contribution to this framework in the support of computationally intensive, non-interactive parts which are specified in the high-level functional style.

   Usually, computations do not consist of a single application of $\mathcal{DC}$. Thus, there is the need to provide a programming language which permits the user to write entire programs, including input and output. The availability of a compiler for this language establishes confidence that the methods presented are sufficiently developed such that their application does not require human interaction.

2. **Ease of use.** This is achieved by hiding the implementation of the skeleton from the application programmer.

   Let us assume the following scenario. The user encounters a particular computationally intensive problem, e.g., to answer a new, complicated question about a large system like a data base, a processor design, a set of equations, etc. The user wants to solve the problem in one day by taking an algorithm from the literature and not spend a week to develop a fine-tuned parallel algorithm which uses explicit communications.

   We assume the basics of functional programming but not, that the user is an expert in functional or parallel programming. Thus, the use of skeletons should not require extra work other than the appropriate structuring of the program with respect to the use of particular paradigms. This approach fits well with a *top-down* program design. If the user is an expert in parallel programming and discovers a potential for improvement, he/she also should be able to modify existing skeleton implementations or even to define new skeletons.

3. **Significant speedups.** Most people take speedup to be the only reason for a parallelization. Systems providing skeletons should implement methods that exploit parallelism where it accelerates the computation (by dividing work into large chunks) and ignore parallelism where it slows down (if large communication costs arise without compensation by an appropriate amount of work delegated). It is known that the use of $\mathcal{DC}$ can lead to significant speedups if the problem is large enough for the given number of processors.

   However, the best method will not be able to establish a parallelization which leads to an optimal speedup for every program because some communications or computations may depend on undecidable conditions. Therefore, experts in parallel computing who are not involved in the implementation of the skeletons should be empowered to use their experience to control *schedule* and *allocation* (the mappings of operations to time and space) at a high level of abstraction. They should not be bothered with considerations of the machine-specific behavior or pitfalls of low-level parallel programming like *races* or *deadlocks*.

We achieve these goals by embedding skeletons into a purely functional programming language, $\mathcal{HDC}$ (for Higher-order Divide-and-Conquer). $\mathcal{HDC}$ is syntactically a subset of Haskell. The semantics differs from Haskell only in that $\mathcal{HDC}$ is almost strict while Haskell is mostly non-strict. Note that $\mathcal{HDC}$ is not a skeleton coordination language, which would require thinking in two semantics: the one of the coordination language and the one of the functional programming language. There is no visible difference between skeletons and other functions in $\mathcal{HDC}$. Any predefined or user-defined function can be implemented as a skeleton, if this is useful, without changing its type.

1.4 Overview

In Chapter 2, we discuss related work on functional programming, parallelization and Haskell. Then, we enter into the three main parts of this thesis:

- **Divide-and-Conquer skeletons:** In Chapter 3, we propose our classification of $\mathcal{DC}$ by skeletons. We start from the general skeleton for $\mathcal{DC}$ and derive further skeletons by successively imposing restrictions, first on the structure of the recursion, then on the structure of the data. For each of the skeletons encountered, we present small example applications.

- **Space-time mapping:** The classification of the skeletons provides only a sketch of the possibilities for parallel implementations. The implementation ideas of the specialized skeletons appear to be simple for special cases, i.e., if the structural parameters are fixed. Otherwise, the amount of detail makes it necessary to derive the implementation for the general case of each skeleton formally. The method of *equational reasoning* makes us to trust the correctness of the implementation. Chapter 4 develops a space-time mapping of some of these skeletons and the structure of the corresponding parallel target code depending on this mapping. We consider this to be of real help for implementers.

- **Compilation techniques:** Knowledge of the parallel implementation of the skeletons alone is not sufficient for their comfortable use. It is not our aim to provide just parallel library functions in the style of MPI. This would still require the user to deal with memory organization, marshaling, type specialization, encoding of functional closures, etc. Probably, it would then be easier for the user to write the entire program in C, than to embed Haskell parts into handwritten C+MPI parts and consider appropriate encodings of data and functions. Instead, we propose in Chapter 5 a combination of advanced compilation techniques, which allow us to integrate skeletons in a functional program as cleanly as any other function.

Experimental results are presented in Chapter 6, in order to give an impression of the performance of the generated code. The thesis concludes with a reflection of the main observations and results in Chapter 7. Long proofs and derivations can be found in the appendix.

Chapter 2

Relation to Previous Work

A lot of work has already been done, which deals with statically mapped skeletons on the one side and with the implementation of a higher-order functional language on the other. The language *Divacon* (Mou, 1990b) provides a special support for restricted forms of $\mathcal{DC}$. The functional languages *Caliban* (Kelly, 1989) and *Eden* (Breitinger et al., 1995) use separate network descriptions, i.e., *skeleton coordination languages*, in which the functional program is embedded. In our approach, the skeletons are integrated into the functional language like any other function and their mappings are dependent on program and machine parameters.

Implementations of functional languages usually transform complicated language constructs into compositions of simpler ones, thereby giving up the structural information contained in the complicated constructs, e.g., skeletons. The $\mathcal{HDC}$ compiler uses a tailored implementation for each skeleton and language constructs like list comprehensions are compiled into compositions of skeletons.

Sect. 2.1 deals with the transformation of recursion. The use of the functional style for transformations is discussed in Sect. 2.2. Sect. 2.3 describes transformations of $\mathcal{DC}$, directed towards parallelization. The skeleton approach is addressed in Sect. 2.4. The transformations in this thesis are performed in the purely functional language Haskell (Peyton Jones and Hughes, 1999). Sect. 2.5 gives an overview of Haskell. We use a pseudo-formal notation to express mathematical objects, where abstraction from detail is crucial for clarity, and Haskell as a formally sound language, where precision is necessary. Haskell is also used as a programming language for the examples and the compiler prototype. In Sect. 2.6, some important approaches of parallelizing functional programs are presented. Sect. 2.7 discusses two different definitions of speedup and motivates our choice of the relative speedup.

2.1 Transformation of recursion

Work on transformation of recursion into iteration dates back to the Seventies. The main issue was to avoid the recursion stack by changing the evaluation order or exploiting semantic properties. Parallelism was not an issue in those days and, therefore, a breadth-first traversal of a call tree as the final result of a transformation, as is proposed in this thesis, was unacceptable due to its huge memory consumption. Darlington and Burstall (1976) submitted an article presenting a system which automatically improves imperative programs. Their approach can be viewed as skeleton-based because it aims at detecting predefined recursion schemata in programs for which an efficient implementation had been developed manually before. The results of the transformations are programs based on while loops. The transformations of Partsch and Pepper (1976) led to code based on for loops and data access by indexing. Burstall and Darlington (1977) concentrated on the functional setting and on a formalism based on rules like *unfolding* (the replacement of a function name by its definition), *folding* (the inverse of unfolding), *instantiation*, *associativity* and *eureka* (tupling values with auxiliary information). Promising approaches have been developed in recent years (Chin and Khoo, 1993), but it is not obvious how to apply the *eureka* rule in general. A survey of transformation rules in the style of Burstall and Darlington (1977) can also be found in (Partsch, 1990). These rules are a subset of the rules we use in the equational reasoning proofs. Backus (1978) presented a functional programming style which is much more appropriate for equational reasoning than the imperative style, although not as comfortable as Haskell.

The common idea of transformation of recursion was to fit the recursion into a *linear schema*, i.e., a definition in which the function being defined occurs only once on the right-hand side. In this case, the call structure changes from a tree (partial order) to a chain (total order), and this chain can be traversed by iteration.

At this point, the ordering of the call instances in the chain plays a role for parallelization. If the elements are ordered according to a depth-first traversal of the original call tree, then additional artificial dependences obstruct a parallel execution. This is not the case if the elements are partitioned into levels according to a breadth-first traversal. The absence of dependences in the original partial order is the source of parallelism. Therefore, we have to take care to select transformations which do not add dependences to this order.

Referring to the so-called *linear expansion theorem*, Backus (1981), Harrison (1988) and Harrison and Khoshnevisan (1992) proposed the employment of a *predicate transformer* and of *tupling* to enable the unfolding of recursion. Even $\mathcal{DC}$ recursions can be transformed this way (de Guzmán et al., 1993) broadening the concept of *linearity*. The difficulty is that linearity was not defined by a purely syntactic schema but by a definition containing existential quantifiers, which was obviously intended to cover as many functions as possible. Our approach in the definitions of skeletons is syntactic, except for simple constraints which can be verified by a compiler. This has the following advantages:

1. Linearity is a *decidable* property.

2. The computational structures and, for some skeletons, even the time and space consumption are known at compile time.

3. The approach is constructive.

To see the danger of a semantic definition of linearity, take the example of the *continuation-passing style* (Appel, 1992) in the higher-order functional setting. With continuation passing, linearity can easily be achieved, but the price that has to be paid for it is the possible occurrence of a huge memory consumption for functional closures. A functional closure is a reference to a function together with an environment for the function's free variables. The huge space is the result of successively adding the remaining computations of a function to the closure carried along.

A similar example is the derivation of a pipeline for $\mathcal{DC}$ (de Guzmán et al., 1993), in which control over the balance of work and space between different pipeline stages is lost. We discuss work balancing issues at several points in this thesis.

The disadvantage of our syntactic approach is that programs have to be adapted to a particular skeleton. In the case that a skeleton is used as a definition of a particular paradigm, the difficulty is to decide whether a given algorithm reflects this paradigm. Intuitively, adaptations which only incur a small overhead in execution time and space, should be permitted.

In the text we will use expressions like *we can handle* (a particular algorithmic schema) also if simple transformations have to be applied additionally in order to make it fit into one of our skeletons.

2.2 Functional styles for proofs and derivations

Functional programming is based on the λ-*calculus* (Barendregt, 1984), a formalism which describes computation in terms of successive substitution. The relation between computations and the λ-calculus was investigated in 1936 by Church, Turing and Kleene. The result was that the notion of *Turing computability* is equivalent to λ-*definability*. The influence of the λ-calculus on programming languages started with the development of LISP by McCarthy (1960).

In his Turing Award Lecture, Backus (1978) advocated the functional programming style and the language *FP* based on it, beginning with the following criticism of the conventional, imperative languages:

> Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor – the von Neumann computer, their close coupling of semantics to state

> transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

Later, Backus (1981) discussed equational reasoning in the functional style. However, this style, which was quite progressive those days, lacks the comfort a modern functional language like Haskell can offer:

- **Absence of types.** The domain and codomain of functions are not clear. Equality is not symmetric: e.g., if the identity function were rewritten to $(-1) \circ (+1)$, a derived implementation would work on numbers but fail otherwise. Absence of type means loss of quality concerning documentation, because a type gives abstract information about the function at hand.

- **Absence of variable names.** Backus proposed a style without names for variables and without λ-abstractions. In our opinion, names, if carefully chosen, are useful for human comprehension – especially, in case of recursion and in cases in which arguments of functions are used in multiple positions. Furthermore, every occurrence of a name provides the potential of a rule application by substitution, e.g., when unfolding a function.

- **Absence of the higher-order polymorphic concept in general.** (Backus, 1981) pointed out:

  > We shall suggest that (a) the FP style leads to "structured" functional programs, whereas the lambda style leads to unstructured ones, and that (b) the FP style encourages reasoning at the "function level" whereas the lambda style leads to reasoning at the "object level".

  In our opinion, a distinction between function level and object level is an unacceptable complication for programming and equational reasoning. This is confirmed by the extension *FFP* of FP which establishes user-defined functionals by the introduction of a *representation function* ρ (Backus, 1978).

  We prefer to treat functions like other objects, where they are not applied to an argument. A functional program can gain the necessary amount of structure if explicit recursion is replaced by skeletons, preferably by those which reflect the strategies used in the algorithm.

One of the successors of FP, the *Bird-Meertens formalism (BMF)* (Bird, 1988; Skillicorn, 1990/1992; Pepper, 1993; Gorlatch and Lengauer, 1995; Skillicorn and Cai, 1995) provides a set of higher-order functions on lists together with a set of *algebraic identities*. The list is taken as a potentially parallel data structure, and the skeletons map and red (Sect. 5.2.1) are used as a basis for the implementation of a restricted form of $\mathcal{DC}$.

The majority of the researchers in the BMF community prefers transformations based on the combinators of data structures instead of indices. In contrast, for classical loop parallelization, indices are absolutely essential. In this thesis, we require both: combinators in the early phases and indices in the late phases. In our opinion, indices simplify a mapping of nested structures, matrices or regularly structured trees, to the linearly organized memory of real machines.

The functional language Haskell offers the concepts of types and variable names and powerful ways of indexing. Due to its higher-order polymorphism, a distinction between first-order functions and *functionals* (higher-order functions) becomes meaningless. Haskell has a large community of users and many tools support program development in the language. This facilitates the exchange of examples, acceleration of the program development and comparison of compilation results.

Reasoning in the FP style used to be of the following form:

> Let f be a solution of ... . Then, f satisfies the hypothesis of the recursion theorem with ... . Therefore ...

This style may lead to seemingly short proofs, but it is hard to understand, hard to check automatically and error-prone due to the informal semantics of the natural language. The BMF style, like it is used by Bird (1988), provides some improvements to FP: each step of the proof focuses on the application of a particular algebraic law. Bird (1989) also proposed a style of reasoning in which the expressions are alike to a functional programming style which makes the proofs more comfortable to read, in our opinion.

O'Donnell (1994a/b) based *equational reasoning* on expressions in the language Haskell. Thus, the syntax and semantics of expressions is precise enough such that they can be parsed and evaluated automatically with a Haskell interpreter or compiler. Lemmas are applied explicitly by using syntactic equality and substitution. Proofs are constructive; proofs by contradiction are not permitted. Conditions like "if the list xs is not empty" in "$f\ xs\ =\ \mathsf{tail}\ xs$" can be expressed syntactically using pattern matching, by "$f\ (\_{:}xr)\ =\ xr$". Instead of the imprecise use of the three dots like in $[x_{0,0}, ..., x_{0,n}, ..., x_{m,0}, ..., x_{m,n}]$, we prefer expressions like the list comprehension $[\ x\,!!\,i\,!!\,j \mid i \leftarrow [0\,..\,m],\ j \leftarrow [0\,..\,n]\ ]$ which has a formally defined semantics in Haskell.

2.3 Derivations of parallel $\mathcal{DC}$ programs

Mou and Hudak (1988) presented an algebraic model for $\mathcal{DC}$ based on a generalization of the *homomorphism*, which they call *pseudomorphism*. The Ph.D. thesis of Mou (1990b) contains fundamental results about the structure and taxonomy of a subclass of $\mathcal{DC}$. The concept of the homomorphism was later also used by Gorlatch (1996c/1997b) for proofs and derivations. We build partly on the geometric aspects of the work of Goodman and Mou (1991), Mou et al.

(1992) and Mou and Wang (1993), where we show how to lay out the computations in time and space. Mou used multi-dimensional arrays to model structures of the problem (matrices) as well as structures of target architectures (multi-dimensional grids). His n-dimensional domain decompositions are binary divisions in each of the n dimensions and lead to 2^n subdomains. Because we prefer a uniform interface for our skeletons, we use plain lists also for multi-dimensional data aggregates and blockwise division of the list into 2^n parts. We call the number of parts the *degree of data division*. This degree influences the time and space complexity.

Carpentieri and Mou (1991), Gibbons and Ziani (1991), Lo et al. (1996), Musiol (1996) and Gorlatch (1998) worked on the mapping of a particular subclass of $\mathcal{DC}$ to grids (meshes). This subclass can mainly be characterized as performing a recursive decomposition on balanced partitions while restricting to a small number of regular communication patterns like *broadcast* or *correspondent communication*. In a correspondent communication, the element i of one block is combined with the element i of another block.

This schema is similar to the schema induced by a skeleton that we present in this thesis (dcF). Early work of implementing the dcF example *bitonic merge* on grids was done by Nassimi and Sahni (1979).

In this thesis, we use transformational techniques for programs described by Feather (1987). Our techniques of *tupling* and *decomposition* of recursive functions into compositions of combinators are, in a remotely related way, also employed by Hu et al. (1999). They concentrate on special forms which exploit the associativity of operators. Geser and Gorlatch (1999) also apply algebraic methods based on the associativity of operators. Their method, called *generalization*, derives the tupling information (eureka rule) for a parallel form of so-called *almost-homomorphisms* from the tupling information of two different sequential forms, a leftward and a rightward representation of the function.

Misra (1994) invented a formalism based on the so-called *powerlist*. A powerlist is a list with a length of a power of two. This data structure is used for a balanced construction of large lists. Two operations combine two powerlists of the same size: *zip*, which concatenates them, and *tie*, which shuffles them. The strength of the powerlist lies in the concise formulation of some proofs and derivations (Kornerup, 1995/1997a). However, one should be aware of the limitations of the powerlist model: (1) problems always have to be divided into two parts, (2) the division of work is coupled with the division of data, and (3) the data dependences are restricted to a static schema. Powerlist algorithms correspond to parallel algorithms on the *hypercube* network topology (Def. 4.4), which perform communication in parallel along a single dimension at a time. We speak of ascending or descending algorithms if, during the computation, the number of this dimension is counted up resp. down.

The powerlist was also taken by Achatz and Schulte (1995/1996), who applied it to the $\mathcal{DC}$ skeleton of Geerling (1994). Kornerup (1997b) extended the powerlist to a structure called *parlist* in order to deal with lists of arbitrary size. Unfortunately, the elegance of the proofs for powerlists is lost in these extensions. In (Herrmann and Lengauer, 1997a), we presented a generalization

active phase	only combine phase	only divide phase
Mou's - postmorphism - premorphism	 left/right odd/even	 odd/even left/right
hypercube	ascending	descending
powerlist	zip	tie
BMF	special map/red	—

Table 2.1: One-sided dcF schemata

of the powerlist –the *powerstructure*– to deal also with bases other than 2; see Def. 3.6. Kornerup (1998) called this structure a *plist*.

Huang et al. (1992), Kumar et al. (1995) and Pauca et al. (1998) used operator strings in tensor algebra to transform block-recursive algorithms, like Strassen's matrix multiplication (Strassen, 1969; Aho et al., 1974), into a loop-based form. Although the level of abstraction provided by tensor algebra is similar to the powerlist approach, more flexibility can be gained from indexing operations if needed. The loop programs obtained for Strassen's matrix multiplication are quite similar to the programs developed in this thesis by equational transformations in Haskell. Loop programs provide a convenient way to describe *data-parallel* algorithms (Hillis and Steele, 1986).

Like Kornerup and Huang et al., also Kumar and Skillicorn (1995) and Amor (1997) took operator strings for describing communication. These operator strings can be viewed as compositions of abstract combinators.

In fact, some of the schemata introduced by Mou (1990b), Misra (1994), Huang et al. (1992) and Amor (1997) are again related to one of our skeletons (dcF). Tab. 2.1 shows the relation between $\mathcal{DC}$ schemata in the literature, which have in common that the computations between list elements are either done exclusively in the divide phase or exclusively in the combine phase of dcF. The other phase serves only for a scattering or gathering of data. A computation restricted to the divide phase can be converted to one restricted to the combine phase, and vice versa, except from reduction (red), which is by definition restricted to perform the operations in the combine phase.

The row for the hypercube demonstrates that, in dcF, a single divide phase is isomorphic to a single combine phase, because the ascending schema differs from the descending schema only in the enumeration of the dimensions. Likewise, zip and tie are isomorphic modulo a bit-reversal permutation of the powerlist. dcF covers block-recursive algorithms with correspondent communication like bitonic sort (Batcher, 1968), FFT (Cooley and Tukey, 1965), scan (Blelloch, 1989/1993), polynomial multiplication (Karatsuba and Ofman, 1962) and matrix multiplication (Strassen, 1969). FFT, polynomial and matrix multiplication are also described by Aho et al. (1974) and Gathen and Gerhard (1999). Contrary to the other schemata in this thesis, dcF introduces fine-grained parallelism, which has to be chosen with care and respect to the parallel machine in order to avoid a slowdown. For special examples like scan or polynomial evaluation, Mou et al. (1992) applied a technique called *compress-and-conquer* to cope with this situation. Typical for these examples is their low cost complexity of $\Theta(n)$ instead of $\Theta(n \cdot \log n)$ (e.g., of mergesort or FFT). The higher cost complexity (of $\Theta(n \cdot \log n)$) is always enforced when dcF is applied with a binary problem division, because dcF performs $\Theta(\log n)$ steps, each of which doing $\Theta(n)$ operations. This is of disadvantage for scan because the optimal cost for scan is $\Theta(n)$.

The concept of a *data field* unifies many different forms of data aggregation (Lisper, 1989a/b; Hammarlund and Lisper, 1993; Lisper and Collard, 1994; Lisper, 1996). Data fields are formally represented as functions which map from an index set to a set of data. Simple data fields are one-dimensional arrays, whose index set is a range of the natural numbers. n-dimensional arrays are appropriately modeled by an n-dimensional index set. Trees can be indexed by a set of labels for node locations, as we do in Sect. 3.3. In Sect. 4.4.1, we will see that the implementation of data fields represented by functions is quite simple in a higher-order language. The efficiency of the access to elements of the data field depends on the way the data field is constructed. Violard proposed a layout algebra of data fields (Violard and Perrin, 1992; Violard, 1994; Violard et al., 1997) which can be used for different space-time mappings. *Change of basis* rules describe redistributions of data.

Similar to the data field approach, with an emphasis on the geometrical structure, are *covers* (Pepper et al., 1993) and *shapes* (Jay et al., 1996; Jay and Sekanina, 1997).

2.4 Skeleton approaches to $\mathcal{DC}$

The use of skeletons differs from the approaches above in the point that the combinators it provides do not capture simple parallel operations, but entire algorithms. Thus, it is a top-down approach which is guided by instantiation, as opposed to a bottom-up approach which is guided by composition.

Bentley (1980) presented a schema for multi-dimensional geometrical application of $\mathcal{DC}$. A d-dimensional problem with n points is divided into two d-dimensional problems with $n/2$ points and a $(d-1)$-dimensional problem of at most n points.

Peters (1981) proposed a so-called *tree machine* as an abstract machine model for the execution of $\mathcal{DC}$ recursions. The network topology of the machine reflects the call structure of $\mathcal{DC}$. Each node of the machine corresponds to an instance of a subproblem. The computation proceeds down the tree during the division, then up the tree during the combination of the subproblems. If programmed, one only has to distinguish between the program at the root, at the leaves, and at the inner nodes of the tree. Such considerations are often made when programming in the *SPMD* (single program, multiple data) style.

Another virtual topology, which was designed to fit a particular $\mathcal{DC}$ structure, is the $\mathcal{N}$*-graph* presented by Gorlatch (1997a). It is an extension of a complete binary tree by a single node and some additional edges, which enables the use of all processors for computing the basic cases and requires only direct communications. Cole (1989) presented choices for the layout of a so-called balanced fixed-degree $\mathcal{DC}$ skeleton (which we call dcC) in a two-dimensional grid.

Boiten et al. (1993) and Geerling (1994) transformed a particular $\mathcal{DC}$ skeleton (dcC with a restriction to two subproblems) into a hypercube implementation. The independence of subproblems is exploited to achieve a tail-recursive (linear) form, using a rule called *partial inversion*. We cannot use the linearization rule of Geerling (1994) for processor arrays, because it applies only to cases in which the input data is not used in the combine function.

A skeleton coordination language named P^{3}L and an interactive system for the composition of imperative skeletons and especially, for the construction of pipelines, was developed by a group at Pisa (Pelagatti, 1993; Bacci et al., 1995; Ciarpaglini et al., 1997; Pelagatti, 1997/1998; Gorlatch and Pelagatti, 1999). The treatment of nested skeletons, e.g., the definition of $\mathcal{DC}$ by recursion and map, appears to be difficult due to the absence of higher-order functions. The approach of Darlington et al. (1995), who use a coordination model for structuring programs with skeletons, points in a similar direction.

Skeletons more complicated than $\mathcal{DC}$ have been investigated. McKeown et al. (1991) proposed an, albeit imperative, skeleton for parallel branch-and-bound. Further work on imperative skeletons was done by Botorog and Kuchen (1995/1996).

A model called *abstract parallel machines (APM)* (O'Donnell and Rünger, 1997; Ellmenreich et al., 1999; Winstanley, 1999) is used for deriving parallel programs from functional specifications. Functional specifications can even be used to describe complex digital circuits (O'Donnell, 1995). The APM derivation forms a DAG, in which the operations of each node are refined by the node's children. Haskell was chosen as a specification language at all levels of abstraction. Derivations are based on equational reasoning and are, thus, formally sound. Winstanley (1999) employed *vertical* as well as *horizontal* transformations. Horizontal transformations perform optimizations at the same level of abstraction. In vertical transformations, implementational detail is introduced, e.g., the exploitation of independence, distribution, explicit communications, and processor view. At the very end a language translation to C+MPI is made.

The derivations presented in this thesis differ in method from the APM approach. The aim of the APM approach is to investigate the effect of possible design decisions at different levels of abstraction, i.e., the alternatives are chosen using human expertise. This is necessary because the APM approach is not restricted to a particular kind of $\mathcal{DC}$, but tries to exploit the power of program transformations, like those of Gorlatch (1996a/b/1997b).

We impose restrictions only in the derivation of a new skeleton. The transformations of a skeleton to an implementation are based on equational reasoning which does not impose further restrictions.

A similarity to the APM approach lies in a decrease of the level of abstraction in the derivation of an SPMD implementation. The specification of a skeleton is based on combinators which may carry an algorithmic principle. The transformations in this thesis are directed towards data parallelism and, thus, are based on iterators and indexed structures. To achieve an implementation in the SPMD style, the parallel computation must be expressed from the viewpoint of a processor, i.e., as an alternating sequence of local computations and communications (in the synchronous case).

The direction towards data parallelism gives a global view on the operations without imposing too much structure. Thus, we keep a high potential of freedom for mapping computations to time and space.

2.5 The functional language *Haskell*

Like the functional language *ML* (Paulson, 1996), Haskell provides higher-order functions and a *static polymorphic type system* (Damas and Milner, 1982). Contrary to ML, function application in Haskell is *non-strict*. Haskell provides a set of user-extendable type classes, e.g., the type class `Num` for types whose elements can be added, subtracted and multiplied by overloading the operators $+$, $-$ and $*$. In Haskell, lists can be defined in a *comprehension style*, similar to comprehensions in set theory and the language Setl (Schwartz et al., 1986). This style has strong connections to loops in imperative languages; we use it for the target programs we derive.

Haskell has been developed at several universities, especially, at *Glasgow*, *Yale* and *Chalmers*. There exist several interpreters and compilers for Haskell, as well as tools for program development.

The four main properties of Haskell stated below have been taken from the official report (Peyton Jones and Hughes, 1999). We make use of the first three, but neglect the fourth (non-strictness) due to our interest in parallelization.

1. **Higher-orderness.** Haskell encourages the use of *higher-order functions*. In Haskell, functions are *first-class citizens*, i.e., it is possible to deal with them in the same way as with elements of other data types. Non-functional languages are usually not higher-order. In the language C, function pointers can be stored in data structures, but then the type information about the functions is lost. Also, C neither provides nested function definitions to build up *environment*s nor permits the user to apply functions partially to create new functions. In Pascal, due to its ability to nest functions with local scope, one can at least pass a function which is instantiated with run-time values, but one cannot store this instantiation in a data structure. However, one can encode higher-order functions explicitly by data structures in a first-order language, but this is not the level of abstraction which we recommend for programming.

   Haskell provides a lot of *syntactic sugar*, i.e., comfortable language constructs that can easily be compiled into compositions of simpler constructs. Functions are usually defined using tuples and other patterns as formal arguments; this comfortable representation is translated internally into abstractions, applications and pattern-matching `case`-expressions.

2. **Purity.** A purely functional language is characterized by the property of referential transparency (Leibniz' Law), which states that two semantically equal expressions in the same context carry the same value (Hankin, 1994).

   With this property, programs can be transformed into more efficient, semantically equivalent ones by successively substituting expressions by equals. The Glasgow Haskell compiler (GHC) is based on the principle of compilation by transformation, which relies on referential transparency, as Peyton Jones et al. (1993) state:

   > A consistent theme runs through all our decisions, namely that *most of the compilation process is expressed as correctness-preserving transformations of a purely-functional program.*

   The technique of *equational reasoning* based on a denotational semantics can be used for proving program properties much more elegantly than it is possible by using an *axiomatic* or *operational semantics.*

3. **Static type system.** Every Haskell expression can be given a type at compile time. Thus, the semantics of Haskell functions can be understood without considering the complicated evaluation mechanism of Haskell. A function whose argument and result do not contain functions and are not polymorphic, can be translated to C by replacing the Haskell expressions by the corresponding C expressions.

   The Haskell type system is decidable and very close to the Hindley-Milner type system (Damas and Milner, 1982) used for the language ML, which has had a major influence on the development of static typing techniques. Like in ML, functions can be polymorphic and operators can be overloaded. Additionally, Haskell allows the user to add type class definitions and instance declarations.

4. **Non-strictness.** In Haskell, the application of functions and constructors is, in general, not strict. This makes it possible to work with intermediate infinite data structures which can simplify the program, e.g., if systems with feedback loops are expressed by Haskell functions. Non-strictness is implemented by *lazy graph reduction*: arguments of functions are evaluated only when their value is actually needed by a strict operation and, if so, this value is kept for further reference. Graph reduction is a way of performing the calculation by successive restructurings of directed graphs, which represent values during program execution. It is especially useful for the call-by-need parameter passing mechanism which induces laziness, i.e., a combination of the *leftmost-outermost* β-reduction strategy (Barendregt, 1984; Hankin, 1994) and the sharing of common subexpressions.

A Haskell function definition consists of an (optional) type definition and a defining equation, as presented in the Fibonacci example in Sect. 1.3.2. We write Haskell programs in a more beautiful style than the plain ASCII style:

keywords are underlined, top-level functions are in sanserif font, variables are in italics, the length function is represented by #, etc.

2.6 Parallelization of functional programs

Functional programs contain much inherent parallelism.

Evaluation in the functional style is formally defined in terms of β-reduction, i.e., substitution. Parallelism results from the fact that variables representing functions can appear in an expression more than once. This means that, if such a variable is substituted by a λ-abstraction or a predefined operator, multiple further reductions are triggered, which can be executed in parallel. The map function presented in Sect. 1.3.4 is such an example, since in its body the customizing function appears twice.

The *Church-Rosser property* (Barendregt, 1984) guarantees that the order of evaluation does not affect the result of the computation in case of termination (Turner, 1995). Parallel graph reduction can perform multiple independent computations in parallel (Rabhi and Manson, 1991).

Another option is the concept of *futures* (Halstead, 1985). If a function is called, the evaluation of the argument can be spawned as a parallel thread. If the argument is required for a strict operation, the completion of its evaluation must be awaited. Time is saved in either case if organizational overhead is not considered. If the evaluation of the argument does not terminate and the argument is not needed, the computation can proceed with a waste of computational resources.

Glasgow parallel Haskell (GpH), the extension of the Glasgow Haskell compiler (GHC) for parallelism (Trinder et al., 1998), requires only a single additional combinator, called par. par takes two arguments and evaluates them in parallel. To enforce evaluation, the expression containing the par has to be used as first argument of the Haskell seq operator.

Sisal (Streams and Iterations in a Single Assignment Language) (Feo et al., 1990; Skedzielewski, 1991) is a first-order functional language which aims to mimic imperative arrays efficiently. For some algorithms, it can compete successfully with Fortran. Cann (1992) gives an example in which Sisal outperforms Fortran significantly:

> The Sisal version of WEATHER, which is the most complex program in the suite, yields the best parallel performance improvement over Fortran (a speedup of 5.7). The reason for this difference was the lack of interprocedural analysis and a means to manage separate compilation.

$\mathcal{DC}$ received special attention in the ZAPP (zero assignment parallel processor) approach (McBurney and Sleep, 1987). A spawned thread resides on the processor that created it until it is *stolen* by another processor which does not have work to do. Hammond (1991) developed a parallel implementation of Standard ML by translation to an intermediate target language called Dactl. Dactl is based on a *graph rewriting* model, i.e., its run-time environment is akin to the usual environment for functional languages. It differs from the imperative run-time system of the $\mathcal{HDC}$ compiler, which is oriented towards a composition of skeletons with a static space-time mapping.

The approaches described so far led to a dynamic parallel execution with the advantage that load balancing can be performed automatically. The disadvantage is that schedule and allocation cannot be controlled by the programmer, which can result in loss of performance because of the overhead of task management. The programmer can help by making a parallel execution depend on the amount of work that may be required, e.g., in GpH by applying `par`. However, the global heap management can introduce dependences between different tasks, which cannot be controlled by the programmer.

Hudak (1986) followed a more static approach by adding schedule and allocation annotations to a functional language. In $\mathcal{HDC}$, one can add control parameters to skeletons which also let the user take influence on the space-time mapping. Kelly (1989) uses many small skeletons as pieces of a computational network. Laziness becomes an issue, due to feedback loops in the network. The purpose of the feedback loop is to describe pipelining of unbounded streams. If we picture the time component of a parallel execution directed from top to bottom, we distinguish between vertical parallelism, i.e., pipelining, where the processors are also directed vertically, and horizontal parallelism, where the processors are directed horizontally. Both kinds can be mixed and the polytope model (Lengauer, 1993; Feautrier, 1996) unifies them in a single theory. In this thesis, we concentrate on the exploitation of horizontal parallelism. If the space-time mapping should impose a pipelined solution for a computation domain we came up with, this is a different issue.

Another network approach is combined with the development of the language *Eden* (Breitinger et al., 1995/1997; Galán et al., 1996) which is built on top of *Concurrent Haskell* (Peyton Jones et al., 1996), a concurrent extension of Haskell. Here, an ensemble of skeletons, which cooperate via communication channels, can be created. Mou (1990a) developed a language called *Divacon* that supports the special kinds of $\mathcal{DC}$ schemata he proposed (Mou, 1990b). In Divacon, it is possible to construct computations in an irregular way, but the examples presented contain the application of elementwise schemata to a high extent. A prototype implementation of Divacon was made for the *Connection Machine* (Hillis, 1985).

Parallelism in the language *Nesl* (Blelloch et al., 1994; Blelloch, 1995) is based on the *sequence*, i.e., a list with a direct element access, as the *parallel data structure*. The implementation exploits the *parallel scan* for *indexing* (Blelloch, 1989/1990/1993), a technique which we adopt for parallelization of one of our dynamic skeletons. Nesl concentrates on data parallelism and lacks task parallelism, i.e., it excludes many $\mathcal{DC}$ algorithms from a parallelization. Task parallelism was introduced in the successor of Nesl: *Proteus* (Goldberg et al., 1994).

Busvine (1993) used a restricted class of higher-order functions for a system that extracts parallel structures. Bratvold (1993/1994), from the same group, developed a compiler for a restriction of ML, that used parallel skeletons. This compiler is not able to deal with higher-order functions except in very simple cases: those that can be implemented by *expansion*. In this thesis, we present a method that can handle all higher-order functions, based on the simplification schema developed by Bell et al. (1997), which is complete for programs stated in the Hindley-Milner type system.

2.7 Speedup

Amdahl stated a law which estimates the achievable *speedup*. Quinn (1994) describes it as follows:

> **Amdahl's law** (Amdahl, 1967). Let f be the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup S achievable by a parallel computer with p processors performing the computation is
>
> $$S \leq \frac{1}{f + (1-f)/p}$$

In the Seventies, Amdahl's law led to pessimism about the potential of parallel computing. The mistake was to neglect that the number of processors is often increased only when the need to solve larger problems arises. Gustafson (1988) pointed out that the sequential parts in a computation require a fixed amount of time rather than a constant fraction, i.e., the parallelism scales up with the problem size. Parallelism in $\mathcal{DC}$ algorithms usually does not scale up linearly with the problem size, but nearly, because the sequential amount of work grows slowly with the problem size. Therefore, we are optimistic that an arbitrary amount of speedup can be achieved if the problem size can be chosen dependent on the number of processors, and if communication costs need not be considered.

Closely related to Amdahl's law is *Brent's theorem* which is used by Quinn (1994) to calculate the number of processors that establishes both an asymptotically time-optimal and cost-optimal parallel implementation.

To determine the *real speedup*, the execution time of the fastest sequential algorithm known is compared with that of the parallel algorithm. Thus, the real speedup does not only regard the organizational overhead in the parallel execution and the delay caused by communication, but also the efficiency loss due to structured computations, which are often easier to implement in parallel, because they contain fewer dependences.

The real speedup is an objective metric for the customer. However, the fastest algorithm may not yet be known and, thus, the real speedup may be object to future changes.

Xavier and Iyengar (1998) presented different notions of speedup. The kind of speedup used in this thesis is called *relative speedup*. The relative speedup is parameterized in the problem size and the number of processors p and is simply the fraction of the execution time on one processor over the time on p processors, leaving the algorithm constant.

We prefer a clear distinction between the overhead caused by the choice of the algorithm and the overhead of parallelization. For the first topic, we present an absolute comparison of the sequential execution times and, for the second, we present the relative speedup.

One could argue that the use of skeletons may cause unnecessary overhead in a sequential execution. For the experiments presented in Chapter 6, *compiled graph reduction* (using GHC) leads to an execution time of the same order of magnitude as the compilation by the $\mathcal{HDC}$ compiler using skeletons.

Chapter 3

Divide-and-Conquer Skeletons

In this chapter, the terrain of $\mathcal{DC}$ is studied and charted, using the skeleton approach as a formal basis.

At first sight, the characterization of $\mathcal{DC}$ as recursively dividing a problem into independent subproblems seems to be sufficiently precise. However, when one compares statements made by different communities, one discovers that opinions of what the essence of $\mathcal{DC}$ actually is differ quite a lot. Everybody will probably agree that quicksort is a $\mathcal{DC}$ algorithm, but there are many definitions of $\mathcal{DC}$ that exclude this algorithm. Some do not allow the split of a problem into two unequal parts, others do not permit non-trivial operations in a problem division, etc. With our approach of starting from the most general case and permitting arbitrary specializations, we manage to be both as general as necessary and as specific as possible.

To make the classification precise, we had to choose a particular formalism. In Sect. 3.1, we motivate our major choices. General recursion is used as a starting point of our hierarchical classification in Sect. 3.2. An abstract computational model for $\mathcal{DC}$, the *call tree model*, is presented in Sect. 3.3. Sect. 3.4 then describes the derivation of the different classes of $\mathcal{DC}$, represented by skeletons, from the general case. The case of nested skeletons is discussed in Sect. 3.5.

3.1 Motivation

3.1.1 The use of skeletons

As pointed out above, the class of $\mathcal{DC}$ algorithms is quite heterogeneous. An aspect that causes part of this diversity is the discrepancy between *static* and *dynamic* data dependences. A static dependence is known at compile time, while a dynamic dependence is determined by run-time values. Static dependences have the advantage that time-consuming organizational computations at run time can be avoided, e.g., the redistribution of work in case of a load imbalance. The overhead caused by organizational work increases with the number of processors involved.

One of our goals is to exploit the potential that static dependences provide by imposing them on $\mathcal{DC}$. We employ this by *specialization*. Intuitively, a specialization S' of a skeleton S inherits part of its power from S, while imposing particular restrictions.

Definition 3.1 (Specialization)
A function S' is a specialization of S, if, and only if, the following conditions are satisfied:

1. The body of S' contains a single application of S, possibly curried to take several arguments.

2. S' is strict in the result of the application of S.

3. The application of S is *outermost*, i.e., the scope of the application equals the scope of the body of S'.

4. The cost complexity caused by function applications in the body of S', aside from the application of S itself, but including the computation of the arguments of S, must be smaller than the one of the application of S.

□

Conditions 1 and 2 establish that S is really required. The purpose of Condition 1 (*single* application) and of Conditions 3 and 4 is that the power of S is used once only.

Note that a program for which Condition 1 holds for each branch of a conditional can be easily transformed to fit Condition 1. Likewise, Condition 3 can be established by a rewriting, if (a) the application of S appears inside a `let` or `where` construct, (b) the local definitions only serve for defining the arguments of S in a more readable form, and (c) Condition 3 holds aside from the local definitions.

The conditions do not exclude the instantiation of a skeleton with itself (Sect. 3.5). S or a specialization of it can be passed as a partial application to S. In this case, the object passed is a customizing function and its call is controlled by the called skeleton itself.

In transiting from a skeleton S to its specialization S', the customizing functions of S can be decomposed into a structural part, which is added to the specification of S', and a problem-specific part, which remains for the customizing functions of S'. Due to this decomposition, the signature of S' can differ from the one of S. Structural parameters can be added to S', which specify explicitly what had been encoded implicitly before. E.g., a matrix representation by a list of lists can be simplified to a plain list representation, if the extents of the matrix are given by structural parameters.

The (parallel) implementation of a skeleton organizes the predefined structural parts by an efficient predefined schema of computations and communications.

As an example, take two algorithms for the problem of sorting that have a different run-time behavior. In the algorithm *mergesort* (Ex. 3.6), the dependences in the combine function depend on run-time data. For the algorithm *bitonic sort* (Ex. 3.9), this is not the case. To make the dependences static, the number of communications is increased such that the value-based functions min and max in the example are applicable instead of control-oriented conditionals, as used in mergesort.

The dependence structure of a program has a strong influence on the possibilities of the implementation. If the dependences are static, the communication schema can be organized more efficiently as when they are dynamic. Take the example of sorting networks (Knuth, 1998). The static dependences make it easy to implement the bitonic sort (Jájá, 1992) for a fixed-size input (of length of a power of 2) by a network of combining elements computing the min and max function on their two inputs. A particular implementation could be by a nested butterfly network (Leighton, 1992), in which the switches act such that one output line always carries the minimum of both input elements and the other always the maximum. A network of combining elements can also be used for the *fast Fourier transform* (*FFT*), see (Leighton, 1992).

3.1.2 Skeletons as representatives of classes of algorithms

One of our goals is to employ skeletons in the definition of classes of algorithms. Not every class of algorithms may be described appropriately by a skeleton. Still, for the classes for which it is possible, a skeleton should be taken as a representative. If different skeletons are appropriate for describing the same class, one may be selected arbitrarily.

Let us motivate this particular role of skeletons. It turns out that the same skeleton (dcF) can be used for bitonic merge (the combine phase of bitonic sort) and FFT. Aside from their structure, bitonic merge and FFT only use simple comparison resp. arithmetic operations. It seems reasonable to expect that other algorithms which match dcF, might also be implementable by a network of combining elements. The classification of algorithms provides us with knowledge about the relationships between problems of different application areas. The employment of a skeleton as a formal criterion of membership in a class guarantees that the class specific properties of one algorithm carry over to the other, i.e., the complexity calculations, the code of the implementation, etc., can be reused after minor modifications.

Every class of $\mathcal{DC}$ we investigate is represented by a skeleton. The distinguished property of the skeleton is that it defines exactly the commonalities of the algorithms in its class, while exposing the differences via the instantiation of its arguments with algorithm-specific customizing functions.

Often, it turns out to be useful to implement algorithms which are not asymptotically cost-optimal. E.g., the use of an $\Theta(n^2)$ algorithm for polynomial multiplication on a systolic array is well motivated by its regularity and nearest neighbor communications. Another example is the bitonic merge algorithm, which does not have the optimal cost complexity for merging. Every implementation of the bitonic merge would be inefficient with respect to the cost complexity of the best sequential algorithm, which is linear. However, the bitonic merge is relevant for a fast execution, due to its restricted dependence structure, which provides a simple implementation in hardware with fixed connections. In order to describe the quality of an implementation of such a non-optimal algorithm, we compare the time/space consumption of the parallel program with respect to the algorithm we want to implement and not to the one which is best to solve the problem.

The choice of a skeleton does not necessarily enforce a particular implementation, but it encourages particular implementations. Also, it has consequences for efficiency that are due to the operations performed by the customizing functions.

The applicability of a skeleton with respect to the pure functional semantics does not imply that the skeleton is appropriate for this algorithm in the sense that it results in an efficient implementation. We describe two directions of aberration and state conditions to avoid pathological cases:

1. The skeleton is <u>too general</u>, i.e., it does not reflect the essential principles of the algorithm, although it might cover some of its properties. This occurs, e.g., if $\mathcal{DC}$, which is inherently recursive, is described by a task-farming skeleton, which is not recursive. In this case, the asymptotic cost-complexity of the remaining tasks is still the same as the one of the original problem instance. Thus, task farming only serves for a single distribution of work, but does not solve the problem of recursive $\mathcal{DC}$.

   We will present some skeletons for $\mathcal{DC}$, of which one is more general than the other. This does not mean, that the former is *too* general, since the principles of the algorithm are still reflected, but it can mean that its use can be extremely inefficient as soon as parallelism comes into play.

   The condition we impose to exclude the case of too high generality is that every single application of every customizing function supplied to the skeleton must have a lower asymptotic cost complexity than the entire skeleton application.

2. The skeleton is <u>too restricted</u>, e.g., it excludes important possibilities for data exchange and induces the need to compensate for this by emulation. An example is the implementation of scan by dcF (Sect. 3.1.3) upto the basic cases. The complexity of scan is in $\Theta(n)$, for input size n, whereas dcF with a division degree of 2 induces a complexity of $\Theta(n \cdot \log n)$.

   We want to support the parallel implementation of particular algorithms, be they cost-optimal or not. Thus, we impose the condition that the solution with the particular skeleton is asymptotically cost-optimal for the algorithm *chosen*.

   A consequence of this condition is that a $\mathcal{DC}$ skeleton cannot be appropriate for the algorithm denoted by program fib (Ex. 1.1) because, due to the exponential overhead we discussed in Sect. 1.3.3, it does not have the optimal cost complexity. The exponential overhead is caused by unnecessary recomputations due to the lack of tabulation, which could be established by using dynamic programming.

Unfortunately, there seems to be no simple way to construct a skeleton that represents exactly the algorithms belonging to the particular class one might have in mind. Let us consider the case that we want to define a $\mathcal{DC}$ skeleton with the property that all its instances terminate, but without limiting the depth of the recursion. This is impossible because termination is undecidable.

Of course, there are several other possibilities to restrict a class of algorithms, e.g., constraints on the time and space consumption, but these are not considered in this thesis.

3.1.3 Our classification

The territory of $\mathcal{DC}$ is divided into classes that overlap in part. One distinguished class is the one covering the entire territory. We call this class dcA.

Each class is represented by a skeleton, denoted in Haskell. We use the same name for both the class and the skeleton it represents.

We say that an algorithm belongs to the class of algorithms specified by a skeleton S, if the algorithm is given by a function that is a specialization of S, and if S is neither too general nor too restricted for the algorithm. The membership of an algorithm in a class is decidable if the cost complexity of the relevant parts of the algorithm can be calculated.

dcA itself can be derived from the class of recursive functions, which we call rec. A class B can be derived from a class A by defining the skeleton for B as a specialization of the skeleton for A.

dcA constitutes a basis from which all other $\mathcal{DC}$ skeletons in this thesis are derived by specialization. It specifies the idea of the $\mathcal{DC}$ paradigm in generality. We took the specification of dcA from the $\mathcal{DC}$ skeletons proposed by Horowitz and Zorat (1983), Kelly (1989) and Rabhi (1995). We model the relationships between different classes by a graph, in which the nodes are skeletons and the directed edges are derivations.

This way of classifying $\mathcal{DC}$ permits us to add classes as needed, as far as they can be defined purely functionally and in terms of dcA, directly or via other $\mathcal{DC}$ skeletons. To keep the classification useful for describing the territory of $\mathcal{DC}$, we adopt the following informal discipline: a skeleton is only added to the classification if it represents a class with at least two algorithms from different application areas that do not match another skeleton with minor adaptations.

In this thesis, we concentrate on dcA plus five other skeletons dcB, dcC, dcD, dcE and dcF. The derivation graph is depicted in Fig. 3.1. The edges are labeled with restrictions imposed on the skeleton which are chosen with respect to the opportunities that arise for a compile-time parallelization. This will be discussed in Sect. 3.4.

On the way down in Fig. 3.1, the class of algorithms is being restricted while the potential for compile-time parallelization increases. The approach is open for additional specializations with respect to other perspectives or refinements. Thus, the set of possible specializations will likely form a DAG rather than just a chain. However, a sequence of several derivations is of special interest to us, because it reflects the amount of refinement steps that the higher-order concept provides, as long as polymorphic types exist. The six skeletons we present are sufficient to present our method of classification; they are not meant to be complete in any particular perspective.

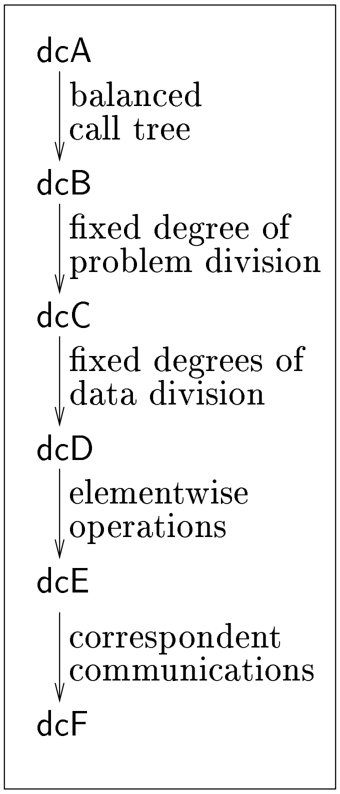


Figure 3.1: Derivations

The purpose of a specialization is to achieve a form that can be implemented more efficiently. The instantiation involved in a specialization is formally expressed by a substitution which, at first glance, complicates the appearance of the form. Algebraic identities can simplify parts of the substituted expressions. Often, the signature of the skeleton is affected by this simplification, e.g., if arguments become obsolete or if structural parameters are added. Structural parameters carry information about the shape and extent of data and control structures and are used to

1. free the run-time execution from a time-consuming analysis by exploiting information extracted by the compiler,

2. permit a flat data representation, e.g., matrices represented by lists in row-major order,

3. provide useful information in advance, e.g., to allocate buffers,

4. emphasize the directions in which the domain of the skeleton is intended to vary (complexity results are stated in terms of these structural parameters) and

5. simplify derivations of implementations by equational reasoning.

3.1.4 Generality of the classification

We see a potential danger of a classification in that the criteria for separation of the classes are chosen from a particular perspective and, thus, the classification might not suit the taste of the user. The problem is that *the* right perspective does not exist. Which classification should we choose to open our approach to the needs of many people?

We choose not to let our skeletons play the role of separators. Especially, for a given skeleton, we do not present a skeleton that describes its complement.

This means that algorithms that belong to the complement are, in our classification, covered by a more general skeleton. Also, two skeletons can have a non-empty intersection, even if one is not a specialization of the other. This is tolerable, since its purpose is to point out possibilities of implementation but not to propose a partitioning of the class of all algorithms.

Several people have analyzed special forms of $\mathcal{DC}$. Of major importance is the classification by Mou (1990b) and the corresponding language Divacon (Mou, 1990a). The structures Mou discussed describe the terrain around our $\mathcal{DC}$ skeleton dcF. In his classification, the structures are divided by orthogonal criteria, e.g.:

1. the division of a list, say $[x_0, ..., x_{n-1}]$:

   (a) *left/right*: splits into $[x_0, ..., x_{\lfloor n/2 \rfloor -1}]$ and $[x_{\lfloor n/2 \rfloor}, ..., x_{n-1}]$.
   (b) *odd/even*: splits into $[x_0, x_2, ..., x_{2\lfloor (n-1)/2 \rfloor}]$ and $[x_1, x_3, ..., x_{2\lfloor n/2 \rfloor -1}]$
   (c) *head/tail*: splits into x_0 and $[x_1, ..., x_{n-1}]$

2. the communication between $[x_0, ..., x_{n-1}]$ and $[y_0, ..., y_{n-1}]$:

   (a) *correspondent*: between x_i and y_i $(0 \le i < n)$
   (b) *mirror-image*: between x_i and y_{n-i-1} $(0 \le i < n)$
   (c) *broadcast*: from x_k for a particular k $(0 \le k < n)$ to all y_i $(0 \le i < n)$

3. the active phases of $\mathcal{DC}$:

   (a) *morphisms*: only the basic phase (equals the map function)
   (b) *premorphisms*: only the phases divide and basic
   (c) *postmorphisms*: only the phases basic and combine
   (d) *pseudomorphisms*: all phases

This view is compatible with our classification. The balanced divisions left/right and odd/even, as well as the different kinds of communication (Mou, 1990b), are covered by our skeleton dcD. Skeleton dcF supports an efficient treatment of elementwise operations, thereby restricting $\mathcal{DC}$ with a division degree of 2 to left/right division and correspondent communication. Odd/even division or mirror-image communication can be established by introduction of skeletons related to dcF or by our skeleton dcE. dcE enforces elementwise operations but does not restrict the communication pattern.

We do not intend to place our classification in contrast to others, but we prefer to use a formalism that is as comfortable as possible for our purposes. Therefore, we recommend to consider the relation to other classifications at an abstract level.

Even if a skeleton has been described very precisely with respect to the class of algorithms it should represent, a large variety of design decisions remains: the representation of the domains of input and output data, the type of the customizing functions, the body expression of the skeleton, the choice of the functional language, etc.

However, for a formal treatment, we need a particular representation. This representation must be chosen carefully to offer the flexibility we need, yet be as simple as possible. Our choice is arbitrary: skeletons which are assigned to the same class will have the same expressive power, regardless of the chosen representation details. Therefore, the skeletons we present can be treated as representatives.

Our classification of $\mathcal{DC}$ focuses on particular conditions that are motivated from the perspective of compile-time parallelization. This choice does not exclude additional aspects from being used in the derivation.

3.1.5 The role of $\mathcal{DC}$ aside from dc$i$

We remind the reader, that dcA covers all kinds of $\mathcal{DC}$, which we justified by the commonly used informal idea of the $\mathcal{DC}$ paradigm. However, due to the ubiquity of $\mathcal{DC}$, it can sometimes be useful to implement the paradigm directly instead basing it on a dc$i$ skeleton.

- From a formal point of view, the map skeleton is a special kind of $\mathcal{DC}$. On the other hand, map occurs as a function in the definition of dcA, which would make its derivation from dcA unsound. Thus, map is defined independently.

- From the implementation point of view, the skeletons red (Sect. 5.2.1.2) and scan (Sect. 5.2.1.3) could be based on one of our $\mathcal{DC}$ skeletons, but dedicated skeleton implementations for them are more efficient because these functions are very simple.

Often, $\mathcal{DC}$ is present implicitly in the program by independent recursive calls. Then, a compiler can introduce map or even dcA by a source-to-source transformation to enable a parallelization. Beware: without an automatic cost estimation, this can lead to inefficient target code.

3.1.6 The need for a higher-order polymorphic functional language

We make strong demands on the programming comfort with skeletons. If the application of a skeleton deep inside the nesting structure of the program turns out to be useful, its application at that point should be possible without the need to manipulate the surrounding parts of the program. Also, customizing functions of skeletons may contain skeletons. Often, it is not clear at compile time whether a particular application of a functional argument incurs the application of a skeleton or not. Polymorphism frees the programmer from consideration of the later use of a function.

Due to our requirements, we consider skeletons to be first-class citizens of the programming language in which they are used. This distinguishes our approach from programming with a parallel library like MPI and also from skeleton coordination languages which are placed on top of an existing programming language

(Darlington et al., 1995; Bacci et al., 1995). The first-class citizenship means that skeletons are functions of our programming language and, because they take other, customizing functions as arguments, they have to be higher-order.

3.1.7 Combination of parallel programming styles

We do not restrict ourselves to a single style of parallel programming. On the one hand, we do not want to exclude algorithms. On the other hand, we want to use the style which is best suited for each individual algorithm. Thus, we feel the need to impose severe restrictions only for some parts of a program, but not in general. Skeletons can be helpful here because they can be used as environments which receive an individual treatment by the compiler. We discuss three criteria for characterizing style.

1. **Explicit parallelism vs. implicit parallelism**. Explicit parallelism is parallelism specified by language constructs. Implicit parallelism is obtained by exploiting program properties. Programming with parallel skeletons is often viewed as explicit parallelism. However, the way we use skeletons does not enforce parallelism, but only points out a potential for parallelization. The parallelization depends on the particular run-time environment, the space-time mapping, available resources, etc. Due to resource restrictions, only a small fraction of the skeleton applications in our programs will actually be executed in parallel. The advanced user may wish to help the compiler in selecting parallelism where it is accelerating instead of where it is slowing down. This is possible by adding a customizing function to a skeleton that calculates, e.g., the number of processors that the skeleton implementation should allocate. Thus, our approach has aspects of both explicit and implicit parallelism.

2. **Static vs. dynamic parallelization**. Parallelization is connected with space-time mapping, i.e., the assignment of the operations of a computation to processors and instances in time. The parallelization is static if this mapping is known at compile time, i.e., does not depend on run-time values. Structural parameters are an exception here because they are just designed to abstract from run-time information in the compile-time parallelization. Their values are often of a very simple type, e.g. integers and thus, can easily be used by a mapping function. Parallelization, which does not use a default or a generated mapping function for schedule and allocation, but follows a strategy based on a run-time analysis of problem sizes, is called dynamic.

   We prefer static parallelization over dynamic parallelization, because administrative and communication overhead, which results from the need to redistribute data, is avoided. Dynamic parallelization must be applied where run-time values determine the existence of dependences or where a compile-time dependence analysis fails. If a function is given as a skeleton, it is assumed that its dependence structure is mostly known to the compiler, such that a dependence analysis is not necessary or at least simplified substantially. Therefore, we view the instantiation of a predefined parallel skeleton implementation by the compiler as a form of static parallelization.

It is not necessary to parallelize the program either completely statically or completely dynamically. A skeleton with a static dependence structure can be called in an environment that parallelizes dynamically. Also, customizing functions of a statically parallelized skeleton can be parallelized dynamically.

3. **Data vs. task parallelism**. Data parallelism is the parallel, individual application of a single function to some elements of a distributed regular data structure. In contrast, task parallelism is the parallel application of possibly different functions to individual data structures. Data parallelism can be implemented with a simpler machine architecture than task parallelism and is appropriate for a high number of small independent subcomputations.

   The problem of data parallelism is that it reflects the capabilities of particular machines and, thus, enforces that the function to be applied must be very simple: e.g., floating point operations may be permitted, but not multiplications of integers of arbitrary size. This has the consequence that the parallel implementation of matrix multiplication based on our skeleton `dcF` is data-parallel, if the entries of the matrices are floating point numbers, but task-parallel if they are unbounded integers. Unbounded integers cannot be multiplied in a time step with a fixed duration. Thus, one cannot always tell whether a polymorphic skeleton can be implemented data-parallel without looking at the particular type instantiations. Therefore, we prefer to make the implementation dependent on the types.

   This insight encourages program optimizations that lead to a large target program in favor of a faster execution: type instantiations of skeletons and inlining of simple customizing functions like floating point multiplication known at compile time. Without this inlining, a data-parallel execution might not be possible if a call of a customizing function is not a simple operation for the particular processor type.

   Let us summarize that a program is not necessarily exclusively data-parallel or task-parallel and that an increase of the power of the processors of a data-parallel machine can make more problems amenable for a data-parallel execution.

3.1.8 The choice of the language Haskell

We use the syntax of the language Haskell (Peyton Jones and Hughes, 1999) to represent skeletons. The corresponding denotational semantics is also taken from Haskell, aside from the strictness we added in order to guarantee that the space-time mappings chosen are not compromised. The reason for choosing

Haskell was that equational reasoning in Haskell is sound, due to referential transparency, and that Haskell is very popular and well supported. Thus, one need not learn a special-purpose programming language to understand the semantics of our skeletons, and we can build on the experience and thought that went into the design of Haskell.

However, the definition of our skeletons is in standard Haskell, and we keep strictness as a side condition. Since the construction of infinite data structures leads to nontermination, we restrict the calculus of our transformations to finite data structures, especially finite lists. This simplifies the reasoning in the non-strict language Haskell, as the following example demonstrates.

Example 3.1 (Transformation rule for map)
Let us revisit the view of a parallel map in Fig. 1.1. Our aim is to express it by the Haskell list comprehension [*elem* | *indexspec*]. Here, *elem* defines the value of a particular element of the list and *indexspec* constructs the index set. We use the symbol # for the function length which computes the length of a list. The length of the result is the same as the length of the original list xs. We define an ordered index set ranging from 0 to $\#xs - 1$ and i as the index variable belonging to this set. Element i of the result is $(f\ (xs\,!!\,i))$.

$$\frac{\mathsf{map}\ f\ xs}{[\, f\ (xs\,!!\,i) \mid i \leftarrow [0\,..\,\#xs-1]\,]}$$

If finite data structures are not assumed, an application of this rule could transform a terminating program into a non-terminating one. If, e.g., only the head element of $(\mathsf{map}\, f\ xs)$ is required, the recursive definition of map unfolds once only and ignores the tail of xs, which is crucial in the case that xs is infinite. Contrary, the evaluation of the list comprehension does not terminate if xs is infinite because function # requires a complete traversal of xs. □

The transformation in Ex. 3.1 is directed towards an index-based form, which is a prerequisite for a space-time mapping in the computation domains we present in Chapter 4. It makes data dependences visible at a more concrete level. Assume for the moment that map does not have a parallel skeleton implementation, but list comprehensions like in the result of the rule above have. After analysis of the outcome of the rule application, a compiler knows that the result of the application of $(\mathsf{map}\, f\ xs)$ is a list, that it has the same length as xs, and that element i of the result requires firstly f and secondly element i of xs. Function f, if not even inlined in the case that it is constant, will likely be provided by a *broadcast communication* (Carpentieri and Mou, 1991) from a single processor to every processor involved. If we assume xs and the result of the application of map to be distributed in an identical shape, a *correspondent communication* (Carpentieri and Mou, 1991) can be used to provide element i of xs to the processor that computes element i of the result. Thus, the compiler is able to employ communications of high regularity like in HPF (Coelho and Germain, 1996), parameterized by the length of xs.

Such transformation rules are also useful for the author of a new parallel implementation of a skeleton, especially for one with a complicated dependence structure. As we will see in Chapter 4, equational reasoning in Haskell, applying rules like the one above, will prove useful in achieving a skeleton representation in an almost index-based form, that can easily be converted into an imperative program with explicit communications.

These conversions require that arguments of functions have a corresponding implementation on the imperative side. This is guaranteed, if functions in argument positions, so called functional arguments, are encoded by data structures that can easily be expressed in the imperative style. Thus, an execution schema other than graph reduction appears to be appropriate, and this motivates the development of dedicated compiler phases, especially the code generation, for our language $\mathcal{HDC}$ despite the fact that it is a subset of Haskell.

3.2 rec: general recursion

According to Sect. 1.3.2, a recursive function is a function that contains at least one application of itself in its definition.

We define a skeleton for recursion by case distinction of the *basic case* and the *recursive case*. The basic case does not contain calls to the function, while the recursive case does. This distinction helps for the comprehension of the semantics of terminating recursive functions. They also have a simple structurally inductive definition: the basic case forms the basic case of the induction, while the recursive case forms the inductive case.

Definition 3.2 (rec: Recursion as a higher-order function)
rec $\in (\alpha\to\mathbb{B})\to(\alpha\to\beta)\to((\alpha\to\beta)\to(\alpha\to\beta))\to(\alpha\to\beta)$
rec $p\ b\ e$ = r
  where $r\ x$ = if $p\ x$ then $b\ x$
                            else $e\ r\ x$
□

Skeleton rec is polymorphic in the types α and β. $\mathbb{B}$ is the set of Boolean values. rec takes three customizing functions: p, b and e. The predicate p determines the non-recursive case, in which the basic function b is applied. The customizing function e describes the recursive case in terms of the instantiated skeleton r and the input data x. Even if the customizing functions are fixed, function r can still be polymorphic. This is the case if variables remain in the type of the customizing functions.

Example 3.2 (Ackermann function)
As an example, we present the Ackermann function ack in terms of rec. This function was given by Rice (1965) and Péter (1981) as an example of a recursive function that is not primitively recursive.

```
ack ∈ (ℕ×ℕ) → ℕ
ack = rec p b e
  where p (n, _)    = n=0
        b (_, m)    = m+1
        e r (n, m)  = if m=0 then r (n−1, 1)
                             else r (n−1, r (n, m−1))
```

□

Mutual recursion can also be handled by skeleton rec, since its transformation into direct recursion is a minor adaptation: a new function has to be constructed that unites all functions involved in the mutual recursion, combines their types of domains and codomains in an algebraic data type and selects the appropriate semantics via a case distinction. This is discussed in detail in Sect. 5.3.8.

A customizing function can refer to the skeleton itself, e.g., if a customizing function of $\mathcal{DC}$ is itself of the $\mathcal{DC}$ type. However, this reference is not recognized as a recursive call because it does not belong to the defining expression of the skeleton. The complexity analysis of a skeleton is independent of its use; it is parameterized in the skeleton's free variables.

Skeleton rec specifies recursion in its general form. This permits nested recursion, i.e., that the input of a recursive call depends on the result of a previous recursive call. Obviously, the number of recursive calls can increase dramatically with the size of the input, because the first recursive call can compute a large number that is used as input for the second call, like in the Ackermann function.

Such a dramatic development cannot happen with $\mathcal{DC}$, because of our requirement that the customizing functions have to be of lower asymptotic complexity than the skeleton itself, which prohibits the instantiation of a $\mathcal{DC}$ function with itself. Still, $\mathcal{DC}$ is not as moderate as it may appear at first sight, e.g., if the quicksort algorithm is taken as a typical representative for $\mathcal{DC}$. The depth of a terminating $\mathcal{DC}$ recursion can still be exponential, i.e., of size k^n, where k ($k>1$) is the size of the input alphabet and n the length of the input: consider the alphabet $\{0, ..., k-1\}$ and a decreasing function whose value, represented in the radix k number system, equals the input string. Since the number of recursive calls can grow exponentially with the depth of the recursion, the largest possible number of basic cases of $\mathcal{DC}$ is at least doubly exponential, i.e., $\Omega(p^{k^n})$, for a natural number p greater than 1. Hofmann (1999) uses linear types to control growth in nested iteration.

3.3 Call trees and call graphs

The aim of this section is to give the reader some intuition of the structure of $\mathcal{DC}$, using a few example graphs. These examples are special cases of a huge set of possibilities. The structure of the whole set cannot even be described formally if the subgraphs of the customizing functions are expanded. Thus, the examples should not be taken as representatives of the general context. The general representatives of computational structures are the skeletons we present.

A call tree is a tree which reflects the call structure of a program or a part of it. Every node of the call tree represents an activation of a function. The root represents the initial activation of the main function of the program or a particular function under investigation. The children of a node N represent the activations of the functions called by N. In a sequential execution, the order of the calls corresponds to the preorder in the tree. For a parallel execution, in general, we must distinguish between calls that have to be executed in sequence, due to data dependences, and calls that can be executed in parallel. For $\mathcal{DC}$, no dependences exist between recursive calls made by the same activation. Therefore, all children of a node can be executed in parallel.

In the call tree of $\mathcal{DC}$, the nodes represent problem instances. We divide the call tree into levels; see Fig. 3.2. The node that represents the entire problem is located at level 0, the topmost level. In the figure, the instance of the entire problem is divided into three subproblem instances.

A node located at level i is connected by an edge with every node that represents one of its subproblem instances. All these nodes are located at level $i+1$.

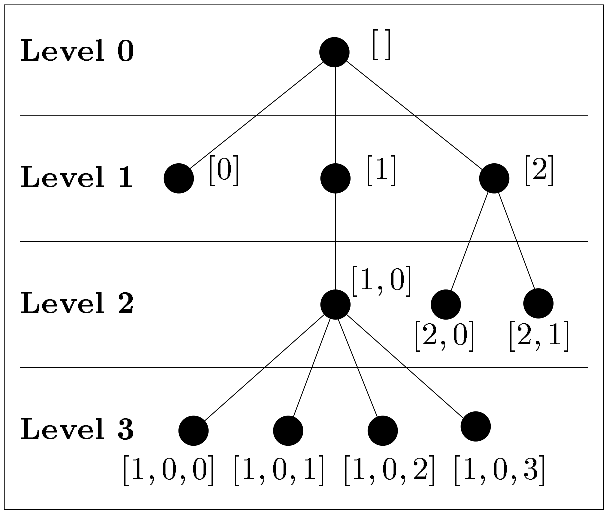


Figure 3.2: Example $\mathcal{DC}$ call tree

Note that different branches originating at the same node are independent of each other. We permit that the number of subproblems generated is 0, i.e., leaves of the call tree do not necessarily compute the basic function but can instead compute the divide function, producing an empty list of subproblems, and then the combine function on an empty list of results. This is motivated by search problems, for which constraints leave an empty set of possibilities, although the formal termination criterion is not fulfilled. As an example, we present the n queens problem in Sect. 3.4.2. Expanding the termination criterion with these constraints would complicate the program and its parallelization unnecessarily.

The nodes of the call tree are uniquely labeled with lists of natural numbers. These labels depend only on the structure of the tree. A node at level i is labeled with a list of length i. The root of the call tree is labeled with the empty list. Child j of a node N is labeled by appending the number j to the label of N; see Fig. 3.2.

The call tree is helpful for an abstract view, but not sufficient for a more detailed analysis, since it does not distinguish between the problem division and the combination of problem solutions. A mapping of the nodes to time without a distinction between these phases would violate the data dependences. Another motivation for a more general model is that the environment in which

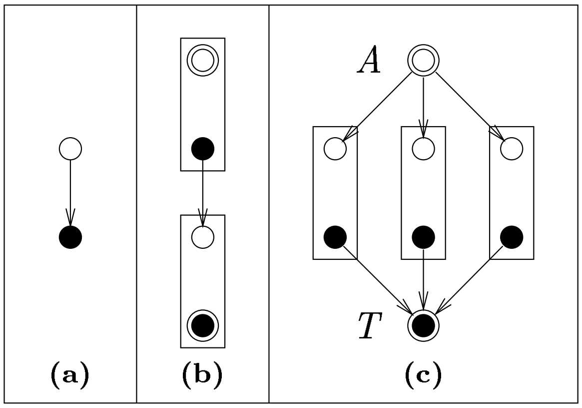


Figure 3.3: Call graph construction

$\mathcal{DC}$ is used possibly contains call structures which cannot be described by a tree because dependences between calls exist.

We propose a call graph, which is a DAG, whose node set is partitioned into *activation nodes* that represent function invocations and *termination nodes* that represent terminations of called functions.

A call graph can be constructed in a bottom-up fashion, as depicted in Fig. 3.3, where the activation nodes are drawn unfilled and the termination nodes are filled.

(a) An activation node, a termination node and an edge from the activation node to the termination node form an atomic call graph. This graph models an activation of a function which does not contain function calls of interest.

(b) n (here: $n=2$) call graphs can be combined sequentially by adding, for every i $(0 \leq i \leq n-2)$, an edge from the termination node of graph i to the activation node of graph $i+1$. The activation node of the first and the termination node of the last graph (both highlighted by an extra circle) become the activation resp. termination node of the sequence. This construction models a sequence of calls.

(c) n (here: $n = 3$) call graphs can be composed in parallel by adding an activation node A and a termination node T for the parallel composition. A/T are connected by an edge with the activation/termination node of each of the composed graphs.

For a presentation of the call structure, we make the following simplifications in order to abstract from uninteresting detail:

1. Contraction of a call graph into an atomic call graph.
2. Elimination of an atomic call graph which appears in a sequential composition directly before or after another call graph. The work assigned to the eliminated call graph is transferred to the activation resp. termination node of the other graph.

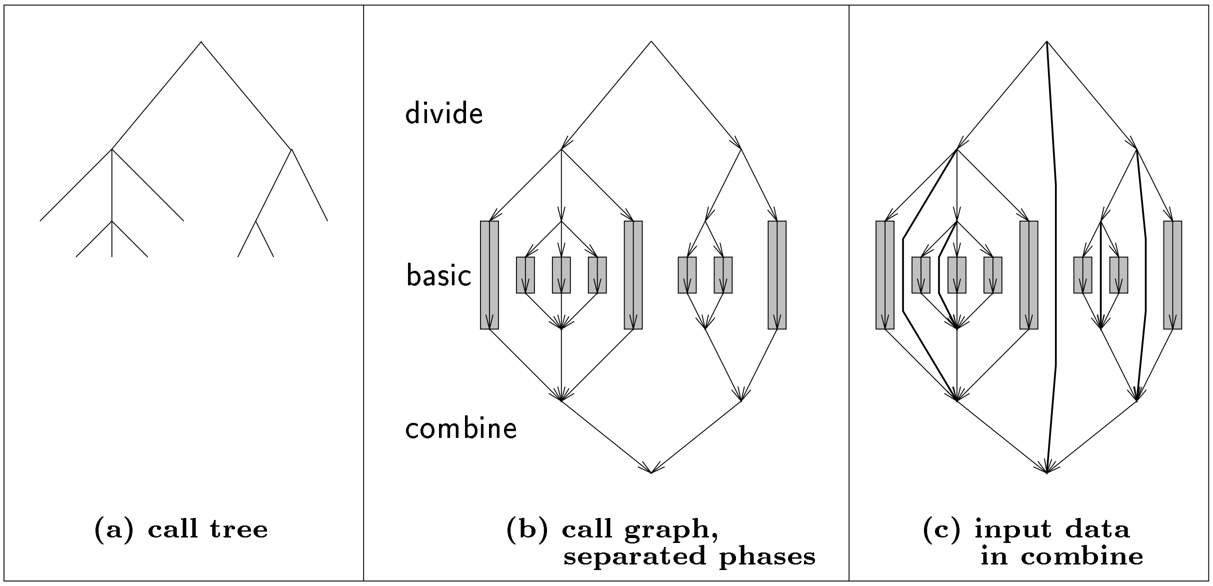
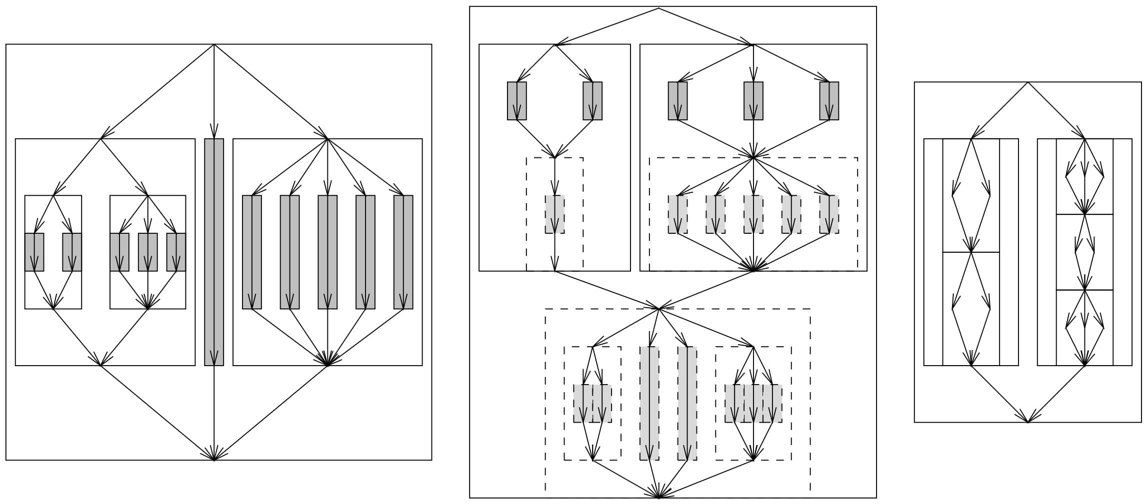


Figure 3.4: Call tree and corresponding call graph

3. In a sequential composition: merge of the termination node of part i with the activation node of part $i+1$. Here, the distinction between activation nodes and termination nodes is lost and, thus, we refrain from drawing the nodes.

The relationship between the call graph and the call tree of $\mathcal{DC}$ can be seen in Fig. 3.4. The call tree (a) shows only the division of the problem, while the call graph (b) exposes the phases of $\mathcal{DC}$. The parts of the basic phase are shaded in grey. The divide phase, above the basic phase, has the same structure as the call tree. The graph of the combine phase, below the basic phase, has the structure of the divide phase, but mirrored horizontally and with reversed edges. In the example depicted, we divide always into at least two subproblems: the divide function can be assigned to the nodes with more than one outgoing edge, the basic function to the nodes with exactly one ingoing edge and the combine function to the nodes with more than one ingoing edge. The call graph forms a complete partial order in which the initial node is the minimum and the termination node is the maximum. In principle, direct data dependences have to be assumed between all nodes that are connected by an edge in the transitive closure of the call graph. For $\mathcal{DC}$, in particular, we can restrict these dependences. Fig. 3.4 (c) shows the entire set of direct dependences in $\mathcal{DC}$: the call graph is enriched with edges between the node at which the input data of an instance activation is available and its use in the combine function of that instance.

The call graph helps us to understand the structure of $\mathcal{DC}$: first to offset it against other kinds of recursion and then to combine the parallelism in the $\mathcal{DC}$ skeleton with the parallelism in its customizing functions. Fig. 3.5 depicts call graphs related to $\mathcal{DC}$. Graph (a) represents single $\mathcal{DC}$. The problem is divided into independent subproblems, which are solved recursively, and then the solutions are combined. The frames indicate the levels inherited from the



(a) single $\mathcal{DC}$ **(b) nested** $\mathcal{DC}$ **(c) not** $\mathcal{DC}$

Figure 3.5: Dependences in the call structure

call tree. The grey boxes show the application of the basic cases.

Graph (b) can be viewed as $\mathcal{DC}$, in which the combine phase is itself of the $\mathcal{DC}$ type. The recursive calls of the combine function of the outer skeleton are depicted with dashed boxes. The basic phase of the outer skeleton is shaded in dark grey, the basic phases of the inner skeleton applications are shaded in light grey.

Graph (c) shows a call graph which does not reflect the $\mathcal{DC}$ structure at more than the outermost level of recursion because, at the second level, recursive calls occur in sequence. One can detect this from splits after merges at the same call instance. In contrast, the splits after merges in Graph (b) are not due to a recursive call of the outer skeleton, but due to a call of the combine function (dashed boxes), which is recursive itself.

We are only interested in call graphs of nested $\mathcal{DC}$ if additional restrictions apply, e.g., if the number of subproblems is fixed. Later, we demonstrate the usefulness of a special treatment of nested regular skeletons with the example of the bitonic sort (Sect. 3.5). A nesting of irregular $\mathcal{DC}$ does not enable compile-time optimizations, but is treated like any unspecific call structure, like in Graph (c).

3.4 The $\mathcal{DC}$ hierarchy

We have developed our hierarchy of $\mathcal{DC}$ skeletons with respect to issues of static parallelization. We propose a derivation sequence, starting with our most general form of $\mathcal{DC}$, dcA, and continuing with dcB, etc., until we reach dcF. Other specializations are also possible, but they do not lead to the parallelization issues we want to investigate here. Schemata similar to dcF have been used quite often in related work, see Sect. 2.3.

restriction	with input	without input
independent subproblems	dcA (dc0) - qsort - max_independent_set	(dc1) - tautology check in the propositional calculus
bounded depth of call tree	dcB —	(dc2) - queens - radix sort
fixed division degree	dcC - Karatsuba integer multiplication	(dc3) - naïve integer multiplication - convex hull - mergesort - component labeling
multiple block recursion	dcD - triangular matrix inversion	(dc3io) - bitonic sort
elementwise operations	dcE - matrix-vector multiplication	—
correspondent communications	—	dcF (dc4io) - FFT - bitonic merge - polynomial multiplication - matrix multiplication

Table 3.1: Overview of the skeletons

Tab. 3.1 provides an overview of the skeletons and the restrictions imposed on them. The column labeled *without input* indicates that the input data cannot be used in the combine function of $\mathcal{DC}$. An algorithm that does not require the input data can also use the skeleton that provides it, but not vice versa.

A table of the types of the skeletons dcA–dcF is given in App. E. The names in parentheses refer to previous publications (Herrmann and Lengauer, 1997a/b/1999). For simplicity, the availability of the input data in the combine function was dropped very early in our previous derivations. We found that the result of those derivations is based on a regular representation of the levels of the call tree. In this thesis, we extend this representation to the case of irregularity and store the history of the input data, in order to provide the input data level by level to the combine function. Thus, we achieve a loop program already for dcA. Sect. 4.6 gives an overview of the parts of the correctness proof.

The implementations of the specialized $\mathcal{DC}$ skeletons are derived from the implementation of dcA by imposing the restrictions with respect to dcA taken from the specification. Following this route, all our $\mathcal{DC}$ implementations provide the input data to the combine function, except for dcF, for which the input data exceeds the size used in the correspondent communications.

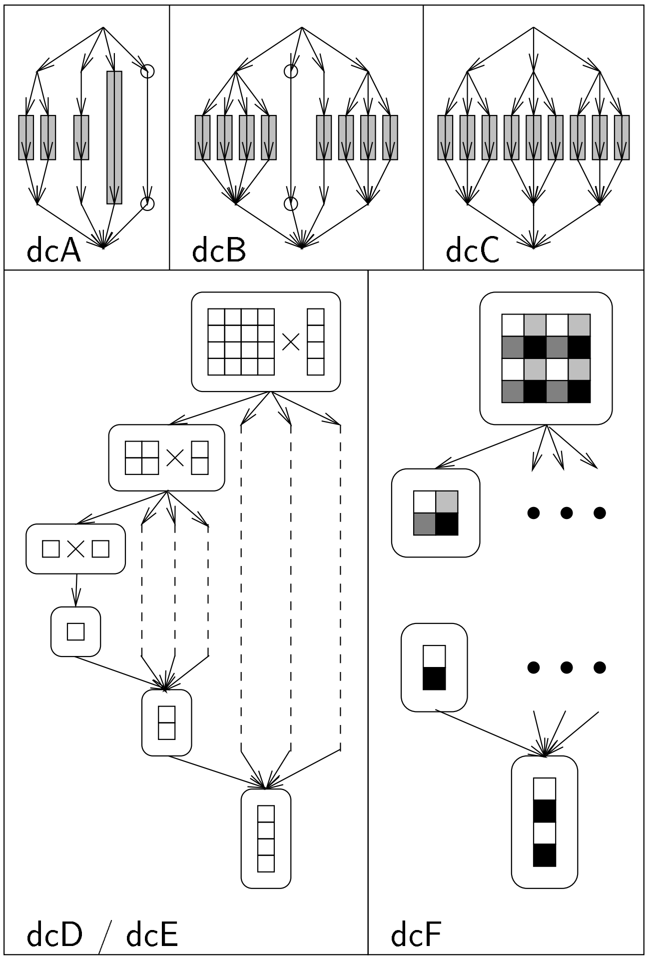


Figure 3.6: Structure of the skeletons

See Fig. 3.6 for an overview of the changes in the call graphs of the $\mathcal{DC}$ skeletons due to the derivation. In the lower row of the figure (dcD/dcE and dcF), a view of the inside of the nodes of the call graph displays example data structures a node has to deal with. In the upper row (dcA, dcB and dcC), no constraints are put on the data structures. The boxes where the basic function is applied are shaded in grey. Circles indicate a division into zero subproblems. The direct data dependences that point from the input data to the combine function are not shown.

dcA is our most general form of $\mathcal{DC}$. It is a direct translation of the informal specification to Haskell, with a rigorous interpretation of the condition that, in $\mathcal{DC}$, the subproblems are independent of each other. Unfortunately, dcA cannot enforce that the subproblems are really smaller, although the $\mathcal{DC}$ specification says so. Because termination is undecidable, a solution would be that the user delivers a termination proof to be checked by the compiler. This will not be acceptable for almost all users. Therefore, we have to live with the fact that dcA describes slightly more than $\mathcal{DC}$, i.e., $\mathcal{DC}$ plus $\perp$ (nontermination). This nontermination can be caused by an infinite chain of recursive calls of dcA or by a nonterminating call of a customizing function.

Fig. 3.6 (dcA) shows an example, in which not all leaves of the call tree reside at the same level of recursion. This is typical for dcA, in general. In the figure, the problem is divided into four subproblems. The first divides into two parts. The second divides into one part. For the third, the predicate is fulfilled, i.e., the divide function is not called. For the fourth, the divide function produces an empty list of subproblems. Thus, the arrow between the two circles represents no computation, but only the passing of information that an empty list occurred.

The undecidability of the termination of dcA means that the number of remaining levels of recursion at a particular point in the call tree cannot be determined and, thus, that it cannot be guaranteed that two children of a node in the call tree have the same remaining depth of recursion. Consequently, dcA cannot enforce that all leaves of the call tree reside at the same level.

Skeleton dcB enforces this by controlling the remaining levels of recursion of each node. In particular, the remaining numbers of levels of two siblings in the call tree must be the same. In Fig. 3.6 (dcB), this is demonstrated by restricting the grey boxes to a particular level. The number of levels of recursion is 2. Like for dcA, the arrow between the circles depicts a passing of input data without computation of the basic function. Control over the levels of recursion is achieved by determining the recursion depth before each call of dcB. Thus, we can give a *schedule* here, which is not meant to be optimized for a particular final implementation, but only used to order computations in a regular fashion for formal treatment in Chapter 4. A final schedule, e.g., one which respects resource restrictions, is *not* affected by our choice here. We choose the free schedule or ASAP (as soon as possible) schedule in the divide phase and the ALAP (as late as possible) schedule in the combine phase. By *ASAP*, we consider each *operation* to be scheduled as early as possible, but at least one step after the computation of every potentially required operand is scheduled. The term *ALAP* is analogous. We use ALAP in the combine phase to schedule all nodes, which are located at the same level of the call tree, at the same time, like ASAP does in the divide phase.

During division, the levels of the call tree are enumerated from the root to the leaves. Subsequently, enumeration continues while traversing the levels from the leaves to the root. This may require an amount of memory which grows exponentially with the depth of recursion, because the data of all basic cases has to be kept in memory at the same time. Often, the memory required will grow more than linearly with the size of the input. As long as the memory is distributed among a sufficiently large set of processors, this is not a problem. But, for a sequential execution, where the memory is concentrated on a single processor, it is unacceptable. The same problem occurs already if a level of the call tree is entirely or partly mapped to time because then several nodes have to be allocated to one processor. In this case, a schedule based on a depth-first traversal of the call tree is preferable, because the memory required grows, usually, only linearly with the input size.

For dcB, we still cannot give a static *allocation*. Static allocation requires knowledge of the existence of a particular branch in the tree. Here, two kinds of tree properties are involved: (1) the existence of a level and (2) the degree of a node. The first is also critical for the existence of a static schedule. Therefore, we impose the control over the recursion depth first (in dcB) and examine the degree of nodes afterwards (in dcC).

To get a static allocation, we add the restriction that the degree of nodes, i.e., the number of subinstances of every problem instance is a constant k. The value of k is given as a structural parameter to skeleton dcC. This restriction enables simple allocations and keeps the organizational overhead in a parallel execution low. Fig. 3.6 (dcC), in which $k=3$, illustrates the simplification enforced by this restriction. It is easy to see that, e.g., the number of basic cases is k^n, if n is the number of levels of recursion.

Skeletons dcD, dcE and dcF pay attention to the amount and structure of data, which becomes important when the data has to be distributed among the processors, for the following reasons:

1. **Memory requirements.** The memory of a single processor might be too small to carry all of the data.

2. **I/O performance.** A single disk is considered too slow to read the input or store the output. Every processor could have its own local disk.

3. **Link bandwidth.** The subproblems created by the root processor are still too large to be communicated fast enough to the other processors.

4. **Long divide or combine times.** Often the divide and combine functions take quite a large fraction of computation time. If these operations are working regularly on the data and the data is large, data parallelism can be exploited, e.g., by an application of the owner-computes rule. This rules states that every processor computes the values that have to reside on it according to the space-time mapping.

Some algorithms, like mergesort, which will serve as an example for dcC, allow for balanced partitions. However, the distribution of data at the higher levels of the call tree is of no advantage, due to the dynamic dependence structure. The best case of merging two sorted sequences in the way mergesort does requires time linear in the length of the shorter sequence. Also, the space consumption has to be considered, e.g., by choosing a processor with a sufficient amount of memory or by pipelining. For sorting a huge amount of data, the bitonic sort algorithm is more convenient. It performs the merging in parallel, by dcF, and is nested inside the dcD skeleton which performs the sorting. We discuss this in Sect. 3.5.

dcD introduces a balanced partitioning of the data. In Fig. 3.6 the input consists of two items: a matrix that is divided into four parts, and a vector, that is divided into two parts. We only show the leftmost of the 16 paths. How the division of a matrix or a vector is performed depends on the customizing functions. With dcD, it is possible that elements of one data structure are used for another data structure in a subproblem. However, this is not required in matrix-vector multiplication. dcD controls the amount of data that is produced during division or combination, but it does not restrict the data dependences. This permits operations like broadcast, which makes a single value, e.g., a pivot, accessible at all positions.

dcE restricts dcD to elementwise operations in the customizing functions. These functions do not construct data aggregates, but receive a set of indices which describe the position of a single element in the result. The lists of structural parameters that could be computed at run time in skeleton dcD not only influence the performance of dcE but are absolutely essential to determine the ranges of values the customizing functions have to be applied to. The structural difference between dcD and dcE is too small to be exposed at the level of Fig. 3.6.

dcE establishes, in an application of a customizing function, that all elements of the result data aggregate are computed in parallel. This leads to a huge degree of parallelism. A potential drawback is that sharing of common subexpressions among the computations of different elements is not possible.

dcF additionally restricts the data dependences to what Carpentieri and Mou (1991) call *correspondent communication*, i.e., it permits an operation only to use sources that have the same index in their particular data structure. This simplifies the communication enormously, by enabling a regular communication schema for the entire set of result elements. The restriction to one common index for all sources limits the number of arguments for each customizing function to the number of blocks.

If the algorithm is defined on more than one input block (e.g., two in matrix multiplication), they all must have the same degree of division. This holds analogously for the elements of the result. Therefore, it is not useful to keep tuples for input and output. Instead, the user has to zip the arguments and unzip the elements of the result.

Thus, dcF has a single input and output data aggregate. Fig. 3.6 shows a division phase, in which a matrix is recursively divided, and a combination phase, in which a vector is recursively composed. Only data depicted with the same shade can be combined for division and the result of the operation has the same shade in the subproblem. In the divide phase, the matrix is divided into four parts of equal shape and size: the upper left, the upper right, the lower left and the lower right. After division, each subproblem contains a matrix of the size of such a part. In the figure, four operations can be performed in parallel for each of the subproblems. In general, there are 4^n parallel corresponding operations for matrices, where n counts the remaining levels of recursion.

3.4.1 dcA: general $\mathcal{DC}$

We define the general form of divide-and-conquer ($\mathcal{DC}$), dcA, as follows:

Definition 3.3 (dcA)
dcA $\in (\alpha\to\mathbb{B})\to(\alpha\to\beta)\to(\alpha\to[\alpha])\to(\alpha\to[\beta]\to\beta)\to\alpha\to\beta$
dcA *istrivial basic divide combine* = r
  where $r\ x$ = if *istrivial* x then *basic* x
  else *combine* x (map r (*divide* x))

□

Our skeleton enforces neither termination nor the existence of at least two recursive calls. Because a violation of these two conditions of $\mathcal{DC}$ does not make the implementation fail, we keep them as informal side conditions.

The independence of the subproblems is accomplished by storing them in a list and processing them independently with the map skeleton.

Lemma 3.1 (dcA_by_rec)
dcA can be expressed by rec.

Proof.

```
dcA istrivial basic divide combine x
    = [unfolding dcA]
let r x = if istrivial x then basic x else combine x (map r (divide x))
in r x
    = [introducing e]
let  e r x = combine x (map r (divide x))
     r x = if istrivial x then basic x else e r x
in r x
    = [folding rec]
let e r x = combine x (map r (divide x))
in rec istrivial basic e x
```

□

As pointed out in Sect. 1.3.3, dcA is a restriction of rec, in which all recursive calls are independent of each other. Such forms are called map-recursive (Suciu and Tannen, 1994). This is the only formal restriction we impose on $\mathcal{DC}$. Therefore, dcA can be taken as our most general form of $\mathcal{DC}$.

Because of the dynamic nature of dcA, it is impossible to give complexity estimates of time or space without knowledge of the customizing functions. Also, space-time mapping must be deferred to run time.

A functional counterpart of the *quicksort* algorithm (Hoare, 1961; Knuth, 1998; Aho et al., 1974) can be expressed in terms of dcA.

Example 3.3 (qsort: a functional version of *quicksort*)
The quicksort algorithm takes as input a list and delivers as output a permutation of this list that is sorted according to a particular ordering. The algorithm computes a so-called *pivot* value from a set of sample values of the list. Then, all elements, which are smaller/greater than the pivot, are filtered into a list A/B. The algorithm is applied recursively to A and B. Then the sorted list A is concatenated with the elements that equal the pivot and the sorted list B. An important difference between qsort and the original quicksort algorithm is that the original works *in place*, i.e., requires no additional memory. This is achieved by reusing the space of the input as space for A and B.

Quicksort is known to have an average complexity of $\Theta(n \cdot \log n)$ for input size n and a worst-case complexity of $\Theta(n^2)$.

Variants of qsort can be found as application examples for other functional languages like Nesl (Blelloch et al., 1994; Blelloch, 1995/1996) or Proteus (Goldberg et al., 1994).

```
qsort ∈ [ℤ]→[ℤ]
qsort = dcA istrivial basic divide combine
  where pivot xs = let l=#xs
                   in ((xs!!0)+(xs!!(l`div`2))+(xs!!(l−1))) `div` 3
        istrivial = (≤1) ∘ #
        basic = id
        divide xs = [filter (< pivot xs) xs, filter (> pivot xs) xs]
        combine xs [le, gr] = le ++ filter (= (pivot xs)) xs ++ gr
```

□

The problem of sorting is in the complexity class $\mathcal{NC}$ (Akl, 1989; Reischuk, 1990; Leighton, 1992; Quinn, 1994) which contains problems that can be solved in parallel with a polylogarithmic number of steps and with a polynomial number of processors. Algorithms in $\mathcal{NC}$ are considered to run fast in parallel. Note that this does not imply efficiency, which becomes an issue if the sequential algorithm is already very fast.

There are also difficulties with the efficient parallelization of mergesort, which has a worst-case complexity of $\Theta(n \cdot \log n)$ for problem size n. Richard Cole (1988) has worked on a reduction of the complexity of parallel mergesort. The delicacy is that the sequential versions of both quicksort and mergesort are already very fast. If the I/O is centralized (transmitted via a single processor), this incurs a parallel time of at least $n \cdot c$ for some constant factor c. The overhead factor introduced by the parallelization must be smaller than the logarithm of the input size, i.e., $n \cdot c < n \cdot \log n$. This is difficult since already the communication of a value requires a multiple of the time for a comparison, i.e., c is large. Much effort must be spent if quicksort is to be parallelized efficiently (Brown and Xiong, 1993): also the separation of the input list has to be parallelized. It is then convenient to use a special machine for parallel quicksort (Heidelberger et al., 1990).

We have to be aware of the fact that, for algorithms with a sequential complexity of not more than $\Theta(n \cdot (\log n)^k)$ for a natural k and input size n, an efficient parallelization is very difficult to achieve, and a general $\mathcal{DC}$ implementation is not appropriate.

Fortunately, the sequentially hard problems require acceleration more urgently than, e.g., the problems of complexity $\Theta(n \cdot \log n)$. Although they remain hard, parallelization can reduce computation time more significantly than for simple problems. $\mathcal{DC}$ is appropriate for exhaustive search problems. Some $\mathcal{NP}$-hard problems (Garey and Johnson, 1979), which occur in the scheduling of everyday situations, assignment problems, integer linear optimization, etc., might be appropriate for $\mathcal{DC}$. Others are better solved with a branch-and-bound strategy because its average expected solution time is likely to be smaller than that of $\mathcal{DC}$.

Let us present an example of an $\mathcal{NP}$-complete problem from graph theory: the *maximum independent set* problem (Wilf, 1986).

Example 3.4 (Maximum independent set)
Given a graph, the problem is to find a subset of the nodes of maximal cardinality, which does not contain adjacent elements. An application could be an assignment problem, in which the edges of the graph represent conflicts. The solution algorithm finds a maximal set of nodes that can be assigned to the same entity without encountering a conflict.

The input of function max_independent_set is a pair of a set of nodes and a set of edges. The node set is represented by a list of natural numbers as identifiers. The edge set is represented by an adjacency list. The element i of this list contains the target points of the edges whose source point is i. The output is the list of identifiers of the nodes in a maximum independent set.

```
max_independent_set ∈ ([ℕ]×[[ℕ]]) → [ℕ]
max_independent_set = dcA istrivial basic divide combine
  where
   istrivial (_, edges)  = all ((=0) ∘ #) edges
   basic (nodes, _)      = nodes
   divide g = let  sel = selectnode g
                   h = removenode g sel
              in [ h, foldl removenode h (neighbors g sel) ]
   combine g [n, m] = if  #n > #m
                      then n
                      else selectnode g : m
   selectnode (nodes, edges)
    = head [ u | u←nodes, #(edges!!u) = maximum (map # edges) ]
   removenode (nodes, edges) u
    = ( filter (≠u) nodes,
        [ if i=u then [] else filter (≠u) (edges!!i) | i←[0..#edges −1] ])
   neighbors (_, edges) v  = edges!!v
```

The algorithm works as follows. The basic case of $\mathcal{DC}$ is that the remaining graph does not contain edges. In this case, all nodes form the maximum independent set. Otherwise, a node of maximum degree is selected and the problem is divided into two subproblems: one that contains this node and one that does not. The result is formed by selecting a solution with the maximum number of nodes. The exhaustive search is accompanied by the strategy of node selection, together with the simplification of removing the neighborhood if the node is selected. □

The example demonstrates that it is useful to access the input data in the combine function. Otherwise, the node selected would have to be stored by passing it down and up the call tree, in order to perform the combine operation. However, this tupling with the input data may lead to communication and space overhead (Herrmann and Lengauer, 1997a).

The $\mathcal{DC}$ skeletons presented by Horowitz and Zorat (1983), Kelly (1989), Suciu and Tannen (1994) and Rabhi (1995) provide the input data to the combine function, while the skeletons presented by Axford (1992), Cole (1989) and Darlington et al. (1993) do not.

3.4.2 dcB: $\mathcal{DC}$ with a static schedule

As stated above, dcB enables a compile-time schedule for $\mathcal{DC}$ by enumerating the levels of the call tree first downwards and then upwards. The remaining levels of recursion are enumerated using the parameter n instead of the predicate *istrivial* of dcA. If the algorithm does not state the number of levels naturally, an upper bound on the depth of recursion has to be chosen for n. Also, the divide and combine functions have to be made <u>overrun-tolerant</u> (Williams, 1982): even if, according to the predicate, the basic case has to be applied, further division does no failure. This can be guaranteed by a formal transformation. The value of n must be delivered by the environment (the user or an outer skeleton) because, in general, the value of n is not computable at compile time.

The choice of n is a degree of freedom for the advanced programmer to control the degree of parallelism explicitly. If the number of processors permits us to lay out m levels of recursion in space, in the case that $m < n$, then $n-m$ levels of recursion must be executed sequentially on each processor, preferably, by a depth-first traversal of the involved part of the call tree. If a better sequential algorithm exists, the advanced programmer can choose this algorithm as a substitute for the basic case and select a smaller value for n. A motivation for selecting a smaller value for n could be that more parallelism does not pay off.

Since the value of n depends on the level of recursion and, thus, changes more often than the customizing functions, it appears late in the list of skeleton parameters.

Because it turns out to be useful in the later derivations, we pass the current value of n as first parameter to the divide and combine function. The advanced user can manipulate the parameter n to control parallelism. In this case, the basic function has to be prepared to deal with non-trivial cases and the divide and combine function must be modified to invert the manipulation of n.

Definition 3.4 (<u>dcB</u>)
dcB $\in (\alpha\to\beta)\to(\mathbb{N}\to\alpha\to[\alpha])\to(\mathbb{N}\to\alpha\to[\beta]\to\beta)\to\mathbb{N}\to\alpha\to\beta$
dcB *basic divide combine* = r
  <u>where</u> $r\ n\ x$ = <u>if</u> $n{=}0$ <u>then</u> *basic* x
                <u>else</u> *combine* $n\ x$ (map $(r\ (n-1))$ $(divide\ n\ x)$)
□

Lemma 3.2 (dcB_by_dcA)
dcB can be derived from dcA.

Proof.

dcB *basic divide combine n x*

$= \boxed{\text{unfolding dcB}}$

```
let r n x = if n=0  then basic x
                    else combine n x (map (r (n-1)) (divide n x))
in r n x
```

$= \boxed{\text{uncurrying } r}$

```
let r (n,x) = if n=0  then basic x
                      else combine n x (map  (λy → r (n-1,y))
                                             (divide n x))
in r (n,x)
```

$= \boxed{\text{property } \circ}$

```
let r (n,x) = if n=0  then basic x
                      else combine n x (map  (r ∘ (λy → (n-1,y)))
                                             (divide n x))
in r (n,x)
```

$= \boxed{\text{map/}\circ}$

```
let r (n,x) = if n=0  then basic x
                      else combine n x (map r (map  (λy → (n-1,y))
                                                    (divide n x)))
in r (n,x)
```

$= \boxed{\text{introduction of auxiliary functions}}$

```
let  tr (n,_)      = n=0
     ba (_,x)      = basic x
     di (n,x)      = map (λy → (n-1,y)) (divide n x)
     co (n,x) ys   = combine n x ys
     r z = if tr z  then ba z
                    else co z (map r (di z))
in r (n,x)
```

$= \boxed{\text{folding dcA}}$

```
let  tr (n,_)       = n=0
     ba (_,x)       = basic x
     di (n,x)       = map (λy → (n-1,y)) (divide n x)
     co (n,x) ys    = combine n x ys
in dcA tr ba di co (n,x)
```

□

This lemma is used in the derivation of the implementation of dcB (itB) in Sect. B.2. An example of the use of dcB is the n queens problem introduced in 1850 by C.F. Gauss (*Damenproblem* in (Claus and Schwill, 1993)).

Example 3.5 (n queens problem)
The problem is to find a placement for n chess queens on an $n \times n$ board such that no two queens can attack each other, i.e., are placed in the same row, column or diagonal.

Quoting Abramson and Yung (1989):

> Over the past few decades, this problem has become important to computer scientists by serving as the standard example of a globally constraint problem which is solvable using backtracking search methods.

We consider the variant of this problem to compute, for a given n, the number of solutions of the n queens problem. As far as we know, these numbers have been computed upto $n = 23$ (Sloane, 2000).

For this thesis, we have chosen a naïve algorithm which is similar to the one from the examples suite of the Haskell interpreter Hugs. It is not our aim to be competitive with this algorithm.

```
queens ∈ ℕ → ℕ
queens n = dcB basic divide combine n ([],[0..n−1])
  where basic _ = 1
        divide _ (placed, remain) =
          let diagonal_attack i
                = or [ (#placed − j) = abs (i − placed !! j)
                     | j←[0..#placed −1] ]
          in [ (placed++[i], filter (≠i) remain)
             | i←remain, not (diagonal_attack i) ]
        combine _ _ = sum
```

□

The algorithm starts with an empty board and adds recursively, row by row, a column position at which a queen can be placed. The number of subproblems for a placement already made is determined by the number of possible placements for the next row.

The input data for dcB is a pair of the placements already made and the column positions which have not been allocated yet. The divide function only has to check for an attack of two queens on the same diagonal, since the rows are distinguished by the position in the list of placed queens and the columns are distinguished by the list of free column positions.

3.4.3 dcC: $\mathcal{DC}$ with a static schedule and allocation

Skeleton dcB permits us to compute the free schedule at compile time, but the allocation remains dynamic because the number of children of a particular node in the call tree is unknown. Thus, if given a label, the existence of the node which carries that label is dubious. Even if the existence of this node is assumed, the node's position in its level of the call tree cannot be given. However, this is required for a static allocation with our ASAP/ALAP schedule, since all nodes at a fixed level of the call graph are mapped to the same time step.

In contrast, skeleton dcC enforces that the number of children of every non-leaf node is fixed, say k. Then, level r of the call tree contains k^r nodes and we can define a function that maps each node, given by its label, to a distinguished virtual processor in the range $[0\,..\,k^r-1]$. Chapter 4 is about the details of such mappings.

Skeleton dcC takes the constant k as an additional parameter:

Definition 3.5 (dcC)

dcC $\in \mathbb{N}\to(\alpha\to\beta)\to(\mathbb{N}\to\alpha\to[\alpha])\to(\mathbb{N}\to\alpha\to[\beta]\to\beta)\to\mathbb{N}\to\alpha\to\beta$

```
dcC k basic divide combine n x =
   dcB basic  (λ r xs →  let  subs = divide r xs
                         in   [subs!!i | i←[0..k−1]])
              combine n x
```

□

Lemma 3.3 (dcC_by_dcB)

dcC can be derived from dcB.

Proof. Follows directly from the use of dcB in the definition of dcC. □

The condition that the number of children of every non-leaf node is k must be established by a divide function which always produces a list of length k. This is a side condition which is not enforced by the skeleton itself. If this condition is not satisfied, the skeleton implementation is undefined and its execution will very likely incur a run-time error. The parameter k is used to fix part of the communication and computation structure of the target code in advance, i.e., at a time at which the validity of the condition cannot be checked because the result of the divide function has not yet been computed. The compiler can try to check this condition by *size inference* and may produce warnings in cases of correct uses of dcC that cannot be verified.

To liberate the derivation of the implementation of dcC from side conditions, we introduce this condition in its definition, by enumerating the subproblems via a list comprehension.

Quicksort does not match dcC because its worst-case depth of recursion is linear, which implies that its cost is exponential using dcC. Thus, it cannot be implemented cost-optimally as required in Sect. 3.1.2.

As an example for the use of dcC, we present the mergesort algorithm (Aho et al., 1974) whose depth of recursion is always logarithmic. In contrast to quicksort, mergesort guarantees a worst-case complexity of $\Theta(n \cdot \log n)$ for input size n.

Example 3.6 (Mergesort)

The call tree of an activation of mergesort is always complete for inputs whose size is a power of 2, i.e., it matches dcB. Also, the divide function of mergesort always produces two subproblems, i.e., mergesort matches dcC cost-optimal with

$k = 2$. In our example, the functions *divide* and *combine* are overrun-tolerant and, thus, the program also works for inputs whose size is not a power of 2.

Mergesort divides a list into its left and right part, sorts these parts independently of each other and then merges the sorted parts in linear time in the combine phase by applying function *merge*. (Linear time refers to code produced by GHC; using the $\mathcal{HDC}$ compiler with the plain list representation requires the use of a special skeleton for a linear-time merge.) Function ilog2 computes the floor of the logarithm to base 2. Function *merge* does not use the input data in the combine function. The bitonic sort algorithm, which is presented in Sect. 3.5, parallelizes also the merge.

```
mergesort ∈ Ord α ⇒ [α] → [α]
mergesort as = dcC 2 basic divide combine n as
 where basic                = id
       divide _ xs          = [left xs, right xs]
       combine _ _ [ys, zs] = merge ys zs
       n                    = if #as = 0 then 0 else ilog2 (2*#as−1)
       merge [] ys          = ys
       merge xs []          = xs
       merge (x:xs) (y:ys)  = if x<y  then x : merge xs (y:ys)
                                      else y : merge (x:xs) ys
```

With this program, a compiler can check easily that the side condition is satisfied with $k = 2$, since the result of the divide function is given by a list pattern with two entries. □

3.4.4 dcD: block-recursive $\mathcal{DC}$

dcD restricts the class of algorithms for which a static space-time mapping at the abstraction level of the call tree can be given (represented by dcC) to those with a balanced partitioning of blockwise distributed data. Thus, we adopt the notion of *block recursion* from Huang et al. (1992) for these algorithms. Other kinds of distribution are, e.g., *cyclic* or *block-cyclic* (MPI Forum, 1995). In our view, the choice of the distribution is an implementational issue which does not affect the nature of the algorithm but which is connected with the allocation function.

Simply put, block recursion decomposes the input data aggregates in the division, performs the algorithm recursively, and composes the output data aggregates in the combine phase of $\mathcal{DC}$. For simplicity, we restrict the set of data aggregates involved in the division to *powerstructures*. The name was adapted from the name *powerlist* invented by Misra (1994).

Definition 3.6 (Powerstructure)
A powerstructure (Herrmann and Lengauer, 1997a) with a degree of data division d over a type α is a list with elements of type α and of length d^m, where $m \in \mathbb{N}$. A powerstructure of size d^{m+1} is composed of d powerstructures of size d^m. □

In specifications, we use the list type of Haskell for powerstructures. It appears convenient to choose an array implementation of lists for powerstructures in $\mathcal{HDC}$. If $d=2$, we can think of a powerlist of size 2^m, if $d=4$, we can imagine a square matrix of size $2^m \times 2^m$. Skeleton dcD is applied in Ex. 3.9.

In general, the input and output of algorithms are tuples of data structures. Each input argument and each output component can be given a separate division degree of data. As one can see in Fig. 3.6, which depicts the structure of matrix-vector multiplication for dcD, the input data is decomposed into four parts for the matrix and two parts for the vector, i.e., the division degree of the matrix is 4 and the degree of the vector is 2. The output data is a vector which is recursively composed of two parts.

Therefore, dcD receives the following structural information concerning division degrees:

k: the degree of the problem division inherited from dcC
ids/ods: lists with the division degrees of the input/output data structures

Definition 3.7 (dcD)

```
type TDiv α     = ℕ→[[α]]→[[[α]]]
type TCom α β = ℕ→[[α]]→[[[β]]]→[[β]]

dcD ∈ ℕ→[ℕ]→[ℕ]→([α]→[β])→TDiv α→TCom α β→ℕ→[[α]]→[[β]]
dcD k ids ods basic divide combine n
  = dcC k ba di co n where
    ba = (map (:[]) ∘ basic ∘ map head)
    di level inputs
      = let subs = divide level inputs
        in [ [ [ subs !! subprob !! component !! item
               | item←[0..(ids!!component)^(level−1)−1] ]
             | component←[0..#ids −1] ]
           | subprob←[0..k−1] ]
    co level inputs partsols
      = let sol = combine level inputs partsols
        in [ [ sol !! component !! part !! item
             | part←[0..ods!!component−1],
               item←[0..(ods!!component)^(level−1)−1] ]
           | component←[0..#ods −1] ]
```

□

Lemma 3.4 (dcD_by_dcC)
dcD can be derived from dcC.
Proof. Follows directly from the use of dcC in the definition of dcD.

□

Some side conditions have to be satisfied which specify the size of particular list structures. Like the degree of problem division in dcC, these conditions are made visible in the definition by enumerating list elements in comprehensions.

The divide function creates a triply nested list in which the levels have the following meaning:

– outer level: the list of subproblems
– middle level: the list of components (different input arguments)
– inner level: the list that carries the aggregated data items of the component

The input data is a list of components and each component is a list of items. To access a particular block inside a component, the base address of the block, i.e., the product of the block size and the block index has to be added to the item index. This flat organization is chosen for faster access.

The combine function receives the input data for the particular problem instance and a triply nested list of subproblem solutions (*partsols*). The levels of the nesting represent:

– outer level: the subproblems
– middle level: the components inside the result of a subproblem
– inner level: the aggregated data items of the component

The nesting levels of the result of the combine function are:

– outer level: the list of components (different elements of result tuple)
– middle level: the list of blocks (*part*)
– inner level: the list of aggregated data items inside a block

For simplicity, all components are of the same type. The input data and the output data is a list of components. Each component is a list of elements. Components can also be scalars. In this case, the degree of data division is 1 and the component list always has length 1.

Let the degree of the problem division be k and the division degree of the input data be d. At level n, the number of parallel subproblems is k^n, each of component size d^{m-n}, where m is the total number of levels. Thus, the amount of all data for a particular component at level n is $k^n \cdot d^{m-n}$. Consider the simple case of only one component with $d=k$ (e.g., in bitonic merge with $k=2$). In this case, $k^n \cdot k^{m-n} = k^m$. Here, the total amount of data per level remains constant.

In the problem division, the processor space allocated for distributed data is successively converted into processor space for independent problems. In general, the situation is more complicated due to different division degrees. Also, the stack used for the input data has to be taken into account. Chapter 4 develops these aspects in detail.

3.4.5 dcE: block-recursive $\mathcal{DC}$ with elementwise operations

Often, the data items of the result of the divide or combine function can be computed efficiently in isolation, i.e., without reusing common subexpressions. Matrix-vector multiplication is an example.

dcE differs from dcD in that the divide and combine function do not return multiply nested lists, but deliver the result for only one item which is determined by a particular multi-dimensional index passed to the customizing function.

Definition 3.8 (dcE)

```
type TDiv_elem α    = ℕ→ℕ→ℕ→ℕ→[[α]]→α
type TCom_elem α β = ℕ→ℕ→ℕ→ℕ→[[α]]→[[[β]]]→β

dcE ∈ ℕ→[ℕ]→[ℕ]→
        (ℕ→[α]→β)→TDiv_elem α→TCom_elem α β→ℕ→[[α]]→[[β]]
dcE k ids ods basic divide combine n
   = dcD k ids ods ba di co n where
      ba x = [ basic i x | i←[0..#ods −1] ]
      di level inputs
        = [ [ [ divide level subprob component item inputs
                | item←[0..(ids!!component)^(level−1)−1] ]
              | component←[0..#ids −1] ]
            | subprob←[0..k−1] ]
      co level inputs partsols
        = [ [ [ combine level component part item inputs partsols
                | item←[0..(ods!!component)^(level−1)−1] ]
              | part←[0..ods!!component−1] ]
            | component←[0..#ods −1] ]
```

□

Lemma 3.5 (dcE_by_dcD)
dcE can be derived from dcD.
Proof. Follows directly from the use of dcD in the definition of dcE. □

To choose the appropriate skeleton for a particular algorithm, dcD and dcE can be distinguished as follows:

1. dcD is especially suited for divide and combine functions that exploit sharing of common subexpressions. As an example, we present the bitonic sort (Sect. 3.5), in which the divide function belongs itself to a $\mathcal{DC}$ schema (dcF).

2. dcE is appropriate if elements of the results of the customizing functions depend only on a few elements of the input blocks, such that all elements in the result can be computed efficiently in parallel. We present matrix-vector multiplication with elementwise operations (mvm_elem) in Ex. 3.7.

Another example for dcE is the multiplication of large integers due to Karatsuba (Karatsuba and Ofman, 1962; Aho et al., 1974; Gathen and Gerhard, 1999). The essence of this algorithm responsible for its complexity is the polynomial multiplication, which can be expressed by skeleton dcF. An extension to the multiplication of numbers requires a reduction of the coefficient values in the result by carry propagation. This can be done during or after the polynomial multiplication. For an immediate propagation, knowledge of the input data is necessary which requires skeleton dcE instead of dcF. A propagation at the end has the disadvantage of a linear traversal or an additional $\mathcal{DC}$ application and requires larger memory blocks to carry intermediate values.

Example 3.7 (Matrix-vector multiplication)
In this example, a matrix of size $2^n \times 2^n$, represented by a list of row vectors, is multiplied with a vector of size 2^n.

```
mvm_elem ∈ [[ℤ]] → [ℤ] → [ℤ]
mvm_elem xs ys = dcE 4 [4,2] [2] basic divide combine n [concat xs, ys] !! 0
 where n = ilog2 (#ys)
       basic _ [x,y] = x*y
       divide l subprob component i [mat,vec]
         = let le2 = 2^(l-1)
               row = i`div`le2
               col = i`mod`le2
           in  if component=0
                 then mat!!(( row +(subprob`div`2)*le2)*2*le2
                            + col +(subprob`mod`2)*le2)
                 else vec!!(i+(subprob`mod`2)*le2)
       combine l component part i _ [[ul],[ur],[ll],[lr]]
         = if part=0 then ul!!i + ur!!i else ll!!i + lr!!i
```

The left part of the matrix is multiplied with the upper part of the vector, which yields two subproblems covering the upper and lower part of the left part of the matrix. The same applies to the right part of the matrix and the lower part of the vector.

The first parameter of dcE, the degree of the problem division, is 4. The second parameter is the list of degrees of the input data division. This list contains the degrees 4 (for matrix partitioning) and 2 (for vector partitioning). The third parameter is the list of degrees of the output data composition. This list contains 2 (for the result vector). The basic function takes the only element of a 1×1 matrix and the only element of a vector of length 1 and computes the only entry of the result vector of length 1. The divide function distinguishes between two components (0 for matrix and 1 for vector). The combine function receives the part of the result (0/1 for the upper/lower part of result vector). n is the depth of recursion, obtained by the logarithm of base 2 of the size of the vector. The input matrix, represented as a list of lists, is concatenated in row-major order. Indexing the result with (!!0) selects its only component. □

A common way to perform a parallel matrix-vector multiplication is to divide the matrix by rows, ending up with dot products of vectors of the initial size. This is motivated by machine support for vector operations. A problem occurs for very large problem sizes, if the vector is duplicated or not spatially distributed. If the elements of the vector are, e.g., huge numbers, memory consumption is critical in the case of duplication. Our dcE schema guarantees that all data aggregates are shrinking in the division. An imbalance between different elements of the same type, e.g., unbounded integers, may remain and has to be taken into account by the allocation.

3.4.6 dcF: block-recursive $\mathcal{DC}$ with correspondent communication

We have a strong interest in providing a $\mathcal{DC}$ skeleton that exploits the simplicity of a restriction to correspondent communication, because many algorithms share this property, e.g., the bitonic merge, FFT, Karatsuba's polynomial multiplication and Strassen's matrix multiplication.

Because correspondent communication is not possible between blocks of different sizes, we use unique degrees of data division in the divide and combine phases, respectively. For the same reason, we drop the use of the input in the combine function. With these restrictions, we obtain skeleton dcF from dcE.

Definition 3.9 (dcF)

$\mathsf{dcF} \in \mathbb{N}\to\mathbb{N}\to\mathbb{N}\to(\alpha\to\beta)\to(\mathbb{N}\to[\alpha]\to\alpha)\to(\mathbb{N}\to[\beta]\to\beta)\to\mathbb{N}\to[\alpha]\to[\beta]$

```
dcF k indeg outdeg basic divide combine n x =
  dcE k [indeg] [outdeg] ba di co n [x] !! 0
  where ba _ x = basic (x!!0)
        di lev subprob _ item x = divide subprob
                                    [ x!!0!!(item+i*indeg^(lev−1))
                                    | i←[0..indeg−1] ]
        co _ _ part item _ sol = combine part
                                    [ sol!!j!!0!!item
                                    | j←[0..k−1] ]
```

□

Lemma 3.6 (dcF_by_dcE)
dcF can be derived from dcE.
Proof. Follows directly from the use of dcE in the definition of dcF.

□

The parameters of dcF have the following meaning:

- k: The degree of problem division, i.e., the number of subproblems which are generated for every problem not trivially solved.
- *indeg*: The degree of division of the input data; it specifies the number of blocks the input data is divided into.

problem	k	*indeg*	*outdeg*
FFT, bitonic merge	2	2	2
polynomial multiplication	4 (3)	2	2
matrix multiplication	8 (7)	4	4

Table 3.2: Example $\mathcal{DC}$ division degrees

- *outdeg*: The degree of composition of the output data; it specifies the number of blocks the output data is composed of.
- *basic*: The function to be applied in the trivial case.
- *divide*: A function which takes a number *subprob* and a list of corresponding elements of blocks and delivers the corresponding element of the subproblem *subprob*.
- *combine*: A function which takes a number *part* and a list of corresponding elements of the subproblem solutions and generates the corresponding element of the output partition *part*.
- n: The number of levels of recursion.
- x: The input data, a powerstructure of size $indeg^n$, which is subject to a division into blocks. Analogously, the output data is a powerstructure of size $outdeg^n$.

indeg and *outdeg* depend much on the data representation, e.g., for vectors they have the value 2 (left and right part), for matrices the value 4 (upper left part, upper right part, lower left part and lower right part); see Tab. 3.2.

The parenthesized values in Tab. 3.2 are for the optimized version of the respective algorithm, i.e., for Karatsuba's polynomial multiplication and Strassen's matrix multiplication (Aho et al., 1974; Gathen and Gerhard, 1999).

The example chosen here is the bitonic merge algorithm.

Example 3.8 (Bitonic merge)
The algorithm (Batcher, 1968; Huang and Lengauer, 1986; Leighton, 1992) sorts any *bitonic sequence* whose length is a power of 2.

A bitonic sequence is a list with elements of an ordered type and the following property: if the list is separated at a particular position, the first part is non-strictly increasing and the second part is non-strictly decreasing, e.g., $[1,3,5,7,8,6,4,2]$. The bitonic merge also sorts any cyclicly shifted bitonic sequence, e.g., $[3,4,1,2]$, but fails to sort arbitrary sequences, e.g., $[1,3,2,4]$.

$\mathsf{bitonicMerge} \in [\mathbb{Z}] \to [\mathbb{Z}]$
$\mathsf{bitonicMerge}\ xs = \mathsf{dcF}\ 2\ 2\ 2\ basic\ divide\ combine\ n\ xs\ \underline{\texttt{where}}$
$\quad n = \mathsf{ilog2}\ (\#xs)$
$\quad basic\ x = x$
$\quad divide\ subprob\ [x,y] = \underline{\texttt{if}}\ subprob{=}0\ \ \underline{\texttt{then}}\ \mathsf{min}\ x\ y$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \underline{\texttt{else}}\ \mathsf{max}\ x\ y$
$\quad combine\ part\ [x,y] = \underline{\texttt{if}}\ part{=}0\ \underline{\texttt{then}}\ x\ \underline{\texttt{else}}\ y$

□

The restriction of dcF to the correspondent communication schema often requires a permutation of the input and output data to match the representation used in the rest of the program. E.g., it is natural to represent matrices by rows or columns, but the dcF skeleton requires a composition of the powerstructure by the blocks of the matrix for Strassen's algorithm. In contrast, dcE provides the possibility to perform the permutation step by step during the divide and combine phase.

3.5 Nested skeletons

The use of higher-order functions turns out to be especially helpful for *nested* skeletons, a third form of composition aside from the *sequential* and *parallel* composition. A simple form of nesting is to express only the basic function of $\mathcal{DC}$ in terms of some other skeleton. This could, in principle, be realized by sequential composition, if $\mathcal{DC}$ were split into a *divide skeleton* and a *combine skeleton*, as Cole (1997) advocates. More interesting is the case that not the basic function but the divide or combine function is expressed in terms of another skeleton. In this case, the recursive schema is refined in each step, i.e., the structure of the inner skeleton depends on the current level of recursion of the outer skeleton. An algorithm in which the combine phase is itself of the $\mathcal{DC}$ type, is the bitonic sort.

Example 3.9 (Bitonic sort)
Quoting Batcher (1968):

> A sorter for arbitrary sequences can be constructed from odd-even merges or bitonic sorters using the well-known sorting-by-merging schema.

Note that, like in the quotation, the term *bitonic sort* was sometimes used for bitonic *merge*. We follow the common use in which it refers to the sorting of *arbitrary* sequences.

The bitonic sort differs from the mergesort in that the combine phase (the merge) is done in parallel, i.e., is a bitonic merge instead of a sequential merge. To apply the bitonic merge step, the second sorted sequence has to be appended in reverse to the first sorted sequence, yielding a bitonic sequence. In the mergesort algorithm, the dynamic data dependences keep us already from using dcA, the most general form of $\mathcal{DC}$, *efficiently* for the merge. In bitonic sort, the application of dcF becomes possible because, in the bitonic merge, the operations are elementwise on corresponding elements of the left and the right part. However, the static dependences are not for free: the bitonic merge emulates an abstract network of switching elements which must be able to establish all possible permutations, thus, inducing an additional factor of $\Theta(\log n)$ in complexity for a sequence of size n.

```
bitonicSort ∈ [ℤ] → [ℤ]
bitonicSort xs = dcD 2 [2] [2] basic divide combine n [xs] !! 0 where
  n = ilog2 (#xs)
  basic x = x
  divide _ [x] = [[left x], [right x]]
  combine lev _ [[x],[y]]
   = let sub = dcF 2 2 2 b d c lev (x ++ reverse y)
     in  [[left sub, right sub]]
         where b x = x
               d s [x,y] = if s=0 then min x y
                                  else max x y
               c s l = l!!s
```

For a comprehensive view of the bitonic sort, consider Fig. 3.7. Only the interesting parts of the call structure of $\mathcal{DC}$ are displayed. Of the outer dcD skeleton, the combine phase is shown. The divide phase only creates singleton problem instances. Of the inner dcF skeleton (the bitonic merge), the focus is on the divide phase. Here, the combine phase is trivial.

Let us have a look at the free schedule. All boxes that are located at the same vertical position are assigned to the same time step. Only the steps that perform non-trivial computations are counted here. The schedule is two-dimensional and given on the right margin. The first component corresponds to the steps of the outer skeleton, the second to the steps of the inner skeleton. This two-dimensional schedule can be mapped to one-dimensional real time according to the *lexicographic ordering*. □

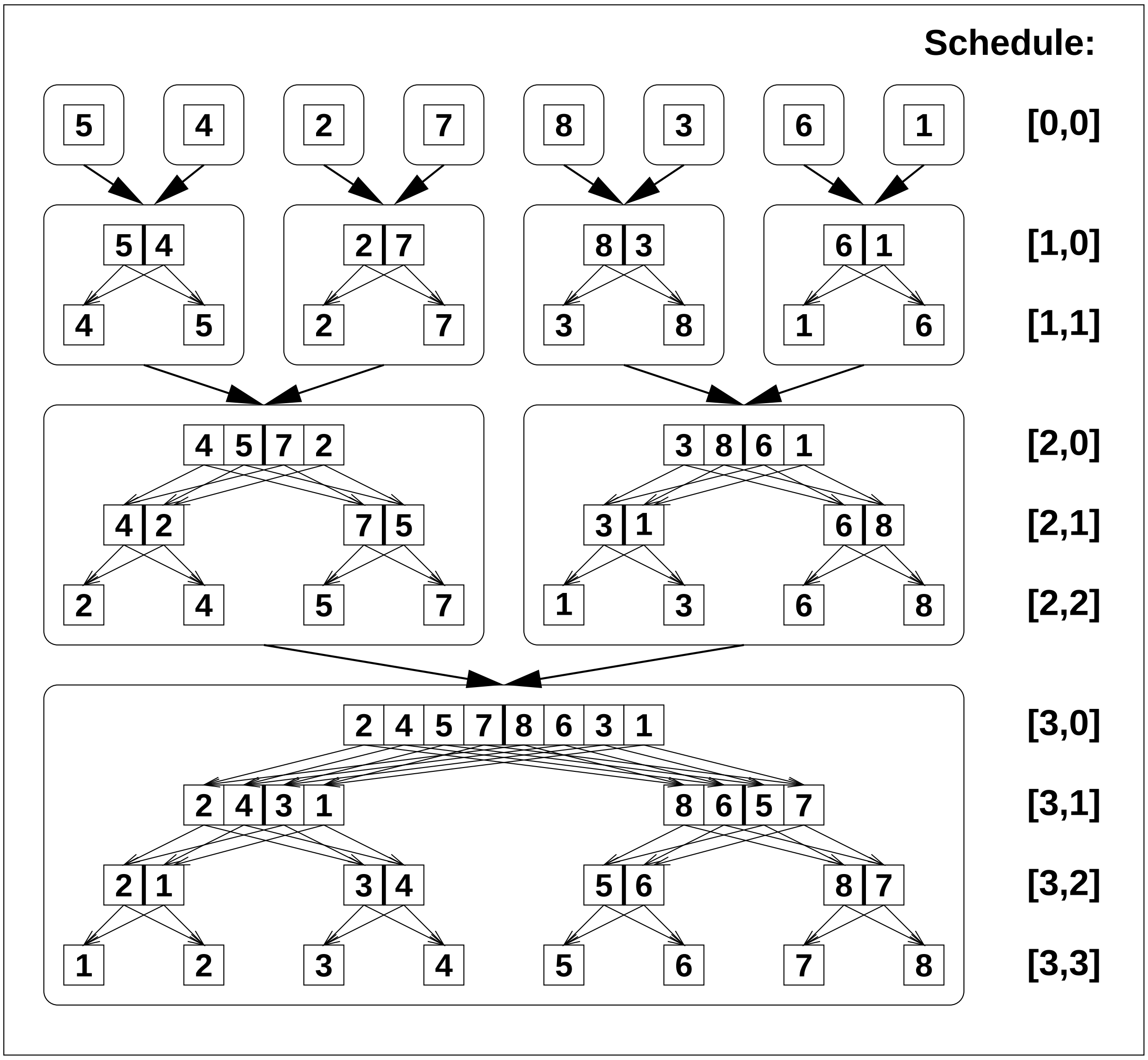


Figure 3.7: Combine phase of bitonic sort

Chapter 4

Space-Time Mapping

In the previous chapter, we have become acquainted with a series of $\mathcal{DC}$ skeletons. Among them, dcA, dcB and dcC appear to have obvious parallel implementations based on a recursive spawning of tasks. However, such implementations can lead to highly unbalanced load and, thus, to bad speedup results. The other $\mathcal{DC}$ skeletons, dcD, dcE and dcF, consider data aggregates. We have an intuition about their parallelization but, due to the amount of detail, the implementation is not obvious at all and deserves a formal derivation.

Even if an efficient implementation for each skeleton has been found, this does not imply that an arbitrary combination of skeletons is efficient. An automatic composition of small skeletons to larger skeletons is critical with respect to performance. In this chapter, we present a method which provides a formal basis for the coordination of parts of the computation. Its use can support the manual design of skeleton implementations with respect to safety and efficiency of the parallel execution. Under certain restrictions, it can even enable an automatic optimization.

Space-time mapping is a method by which one assigns each operation in a computation a position in *time* and a position in *space*, i.e., a processor. We distinguish a *static* (compile-time) from a *dynamic* (run-time) space-time mapping. The *polytope model* (Lamport, 1974; Lengauer, 1993; Feautrier, 1996) has been used successfully in the static space-time mapping of nested loop programs. In this thesis, the problem domain is not loops but $\mathcal{DC}$ recursions. It turns out that some ideas of the polytope model can be adapted for $\mathcal{DC}$, but the process of parallelization is different.

The use of skeletons makes it possible to anticipate a part of the space-time mapping at the time at which a skeleton is designed. This part is developed by the skeleton implementer who can use appropriate tools and can spend quite a large amount of time in achieving a good solution. The other part, which cannot be anticipated because it depends on a particular use of the skeleton, is done by the compiler and expected to deliver a feasible and good solution within a short time. This means that a compiler can optimize the mapping in the sense that it calculates some of its parameters, but not that it finds optimal structures for computation and communication with respect to a machine topology described by a set of constraints. Thus, it remains the task of the skeleton implementer to design different solutions for different architectures, e.g., shared memory systems, grids of processors, clusters of shared memory systems, etc.

The introduction of this chapter presents a few basic notions of the polytope model and an example of the parallelization of nested loops. Sect. 4.2 motivates our departure from the polytope model. The new model is described in Sect. 4.3. The technique of size inference presented in Sect. 4.4 is independent of this model, but it is here motivated by the employment for the model, i.e., for determining the sizes of data structures that have to be regarded in a space-time mapping. Sect. 4.5 presents example computation domains for several skeletons. Sect. 4.6 deals with the elimination of recursion. This leads to a significant simplification, since a computation domain of an unbounded dimensionality can be transformed to a computation domain of a fixed dimensionality. Sect. 4.7 builds on this transformation to derive the iterative forms of our $\mathcal{DC}$ skeletons dcA to dcF, which can be converted easily into C programs, as described at the beginning of Chapter 5.

4.1 Introduction

Space-time mapping has had a long tradition in computer science and dates back to the methods of Karp et al. (1967) for the scheduling of uniform recurrence equations and to the wavefront model of Lamport (1974). Further work was done by Quinton (1987) and Lisper (1989b) on synthesizing synchronous systems by static scheduling in space-time. Rajopadhye (1989) proposed a solution for affine recurrence equations by pipelining. Huang and Mou (1992) discussed the reduction of dependences for *parallel expansion*, a generalization of *recursive doubling*. Recursive doubling (Akl, 1989; Berman and Paul, 1996) is a schema which reduces the problem of computing $2n$ values to the problem of computing n values. It is an instance of our skeleton dcF. An application example for recursive doubling is scan (Sect. 5.2.1.3) with an *associative* combining operator.

In our view, space-time mapping is a formal method for the parallelization of programs based on a *computation domain*. The approach unifies many prior transformation techniques that operate on loop programs. Lamport (1974), Lengauer (1993) and Feautrier (1996) showed how loop programs can be transformed using a computation domain without working on the program as a syntactic object. An extension of the polytope model to the so-called *polyhedron model* was made by Griebl and Lengauer (1994) and Collard (1995) to handle while loops. Redon and Feautrier (1993) and Barreteau and Feautrier (1995) applied the model to scans and reductions, i.e., simple instantiations of dcF.

Before we can say more about what a computation domain is, we have to define a few basic notions. The notions of a *unit of time* and a *virtual processor* are only used to describe single entities of the computation. They are, at this point, not viewed to form a time dimension or a space dimension.

We use the informal notion of a level of resolution, which refers to a particular degree of power of the operators and to a particular size of data aggregates. E.g., activations of the same $\mathcal{DC}$ function at different levels of the call tree are assigned to different levels of resolution. This corresponds to the fact that the dimensionality of the computation domain increases with the problem size.

Definition 4.1 (Unit of time)
The unit of time is a basic measure for the current level of resolution. □

Definition 4.2 (Virtual processor)
The virtual processor is an abstract computational unit which is able to perform every function at the current level of resolution in one unit of time, by a particular instruction. □

Definition 4.3 (Operation)
An operation is an instance of an instruction of a virtual processor at a distinguished position in the computation. □

The instantiation of polymorphic and overloaded functions can lead to different levels of resolution for the same function symbol, e.g., multiplication. It appears sensible to assign one unit of time to the multiplication of two word-size integers. For the multiplication of two arbitrary integers, this is not appropriate since one unit of time corresponds to a fixed amount of real time. For arbitrary integers, the number of time units required can be calculated from the algorithm which performs this multiplication, in terms of the sizes of the operands. Refining the resolution increases the potential of static compile-time optimization, e.g., load balancing.

Each operation is due to an *application* in a functional program or a *statement* in an imperative program. Whether an application or a statement causes no, one or many operations, depends on the number of its activations. Thus, to refer to a particular operation formally, we have to keep an *identifier* for the application or statement, as well as an *index* for the particular activation. Indexing is difficult if branching conditions or functional arguments depend on run-time values. The transformations of skeletons from the recursive to the iterative forms reduce branching conditions and simplify indexing.

The *computation domain* of a program is an enumerable index space, which depends on structural parameters, together with a description of operations to be performed. The precise definition of the notion of a computation domain is presented later. For each input, for which the program is defined, there exists a one-to-one mapping between the elements (points) of the index space and the operations executed. The values of the structural parameters are obtained from information about the sizes of the input data structures.

The index set of a single for loop with stride 1 is the series of integer points from the lower to the upper bound. The index set of a nest of k for loops can be described by a bounded intersection of half-spaces in $\mathbb{Z}^k$, which is a polytope and has given the *polytope model* its name. This model is not appropriate for the $\mathcal{DC}$ structures we deal with in this thesis. We explain this in Sect. 4.2. Although

```
procedure A(n ∈ ℕ);                 procedure B(i ∈ ℕ);
begin                               begin
  var i ∈ ℕ;                          var j ∈ ℕ;
  for i := 1 to n do                  for j := 1 to i do
    B(i)                                c[i,j] := c[i-1,j]+c[i,j-1]
end;                                end;
```

Figure 4.1: Loops separated by procedure bounds

the space-time mapping principle of the polytope model is used, the concrete methods applied are different. The idea of an index space as a subset of $\mathbb{Z}^k$ with time and space dimensions has already been used beyond the scope of the polytope model, i.e., for dynamic structures (Fox, 1989), but we are especially interested in those structures which are known at compile time.

We believe that an entire program is too large and irregular to be subject of a global optimization and concentrate the optimization on those parts of the program that can be explained in terms of computation domains with a considerable amount of regularity. On the other hand, the larger the part assigned to a computation domain is, the higher is the potential for extracting parallelism.

An example, for which quite a large computation domain is sensible, is the bitonic sort algorithm (Sect. 3.5). This algorithm can be expressed by skeleton dcD, which is instantiated with skeleton dcF. The dependences are static such that, for a particular problem size, the algorithm can be described by a sorting network, as depicted in Fig. 3.7. The computation domain of the bitonic sort can be described by the nesting of the domains of dcD and dcF.

We show with a simple example, denoted in pseudo-Pascal, how convenient the nesting of computation domains can be for exploiting parallelism.

Example 4.1 (Nested loop programs)
Consider a procedure A which contains a loop. Every iteration calls procedure B consisting of a loop which performs an operation on a global array c (see Fig. 4.1).

Neither procedure A nor procedure B can be parallelized because there is a *loop-carried* dependence in i as well as in j. A dependence is called carried by loop i, if i is the iteration variable of the outermost loop in which the source and the target of the dependence are assigned to different iterations.

We believe that a global optimization of a large program is not feasible, but we choose for a local optimization in combination with inlining. Let us assume, for example, an inlining of B into A, say A', shown on the left side of Fig. 4.2. The dependences carried by i and j remain but, in the two-dimensional space we have obtained, there are more possibilities for a parallelization than in two separated one-dimensional spaces. On the right side of Fig. 4.2, one can see the polytope (the convex hull of filled circles) for the value $n = 4$. Circles indicate points of the domain. Unfilled circles indicate input points, i.e., points at which *input operation*s are located. An input operation is an operation which does not

```
procedure A'(n ∈ ℕ);
begin
   var i,j ∈ ℕ;
   for i := 1 to n do
      for j := 1 to i do
         c[i,j] :=
             c[i−1,j]+c[i,j−1]
end;
```

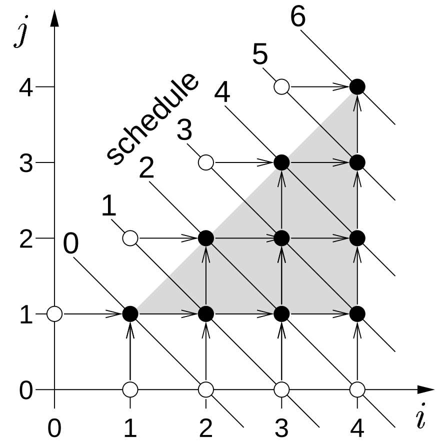


Figure 4.2: Inlined procedure, polytope with $time[i,j] = i+j-1$

perform computational work, but acts as a dummy that carries an element of the input. Here, input operations are assigned to elements of c which are accessed read-only. Data dependences are depicted by arrows. A schedule which depends only on i or only on j violates the dependence constraint in the orthogonal direction. However, in the figure, all points for which the sum $i+j$ is equal, i.e., which lie on the same diagonal can be computed in parallel. Obviously, the dependences are respected. In this case, the number of time steps is $2n-1$, for $n(n+1)/2$ non-input operations. This means that the average degree of parallelism, i.e., the number of operations divided by the number of time steps is roughly $n/4$.

The example reflects our anticipated situation that a nesting of computation domains can be transformed into a single domain, using knowledge about the involved skeletons (here: for loops). □

4.2 Modifications of the polytope model

Before we motivate our changes to the polytope model, we discuss the relation between the polytope model and skeletons. From the view point of structure, a loop can be treated as a small imperative template: it is instantiated with the name of the loop variable, its bounds, its stride and the code of its body. In our functional view, bounds and stride are structural parameters, while the body is represented by a customizing function. Griebl and Lengauer (1996) presented a hierarchy of different loop templates, ranging from restricted for loops to fully dynamic while loops. In this chapter, we present functional counterparts of for and while loops. In loop parallelization, the automatic parallelization process comes into play when multiple loops are combined, preferably by nesting. The same holds for the combination of skeletons in a functional setting. However, the focus of this thesis is on $\mathcal{DC}$ and, thus, on the efficient implementation of *single* $\mathcal{DC}$ skeletons. The properties of a single skeleton, e.g., possibilities of layout in time and space can be investigated by the developer of the skeleton, presented in a document and exploited in the implementation.

Our idea of space-time mapping stems from the polytope model, but there are several reasons why the polytope model itself is not appropriate for $\mathcal{DC}$, that are discussed in the following subsections. We identify necessary changes by referring to properties of $\mathcal{DC}$ that we aim to establish. These properties should be comprehensible without prior knowledge of the new model, which is presented in technical detail in Sect. 4.3.

4.2.1 Objectives and challenges

Skeletons are often developed with a particular schema of parallel computation in mind. Thus, one could argue that they are not challenging objects for a space-time mapping. This may be true as long as the focus of the parallelization is on the detection of data dependences and on finding a free schedule. These two major challenges of the polytope model are caused by the presence of side effects in imperative programs.

For functional skeletons, the situation is different. Due to the lack of side effects, the set of potential dependences does not exceed the set of real dependences significantly, as it does in the imperative setting. (A potential dependence that is not real can be caused by a conditional with unknown condition in which only one branch uses a particular value.) This motivates us to base the definition of the *free schedule* in Sect. 3.4 on *potential* dependences. With this definition, the free schedule can be determined easily.

Challenges are the integration of skeletons into a programming language and the optimization of skeleton combinations under resource restrictions. If resource restrictions are ignored, no useful time measure can be established since the steps enumerated by the free schedule can vary wildly in the amount of real time they require. Consider, e.g., a $\mathcal{DC}$ call tree in which many basic cases have to be computed by a single processor in a particular step, but only a single problem division in the first step. Thus, it may be profitable to stay away from the free schedule and try to match the structure of the skeleton with the particular architecture as best as possible, which requires the employment of a cost model.

Often the goal is to optimize the entire program and not a single skeleton application. Cost models which do not pay special attention to the combination of skeletons might fail, because an optimized implementation of one skeleton can be detrimental to the optimization of the combination. *Symbolic* cost information provides the power to choose the implementation parameters for the skeletons such that a global optimization is achieved. We present a way to combine computation domains manually. The most powerful example is the computation domain of itF which consists of combinations of loop skeletons.

Be aware that the success of static parallelization depends strongly on the ability to find a good solution, which often is a difficult task. Any dynamic (run-time) parallelization that allows for a quick assignment of jobs to idle processors requires a considerably large, but often not unacceptable overhead.

4.2.2 Constraints of the problem domain

$\mathcal{DC}$ algorithms can enumerate a number of points of the index space which is exponential in the problem size. On the other hand, a polytope in $\mathbb{Z}^n$ contains not more than k^n points, where k is the maximum of the extents in all of the n dimensions, i.e., it is polynomial since n is fixed. In the polytope model, k is supposed to be bounded by an affine linear expression in the problem size and n is the nesting depth of the loop nest. There are at least two ways of overcoming the polynomial bound of the number of points in the index space of a computation domain for $\mathcal{DC}$: by dropping the restriction of (1) the fixed number of dimensions or (2) the affine extent.

A look at the *hypercube* (Leighton, 1992) proves to be helpful:

Definition 4.4 (Hypercube)
An n-dimensional hypercube is a graph with 2^n nodes, which are indexed by n-tuples with component values 0 or 1. Two nodes are connected by an edge if, and only if, their indexing tuples differ in exactly one position. □

The hypercube has served as an abstract network topology for the description of algorithms so often that one could call it a template (Foster, 1995). The edges of the graph model the direct connections for communication. The fact that one often makes the dimensionality n of the hypercube dependent on the problem size demonstrates the hypercube's role as a model, instead of a real target topology that has to be of a fixed dimensionality. Also, hypercube algorithms are often mapped to other topologies or the hypercube is provided only virtually by a parallel programming environment.

The hypercube model is often connected with a special kind of $\mathcal{DC}$, namely, the class covered by our skeleton dcF, in which all three division degrees (problem division, input data division and output data composition) have the value 2 (Akl, 1989; Leighton, 1992; Quinn, 1994; Mayr and Werchner, 1996).

A generalization of the hypercube is the k-ary n-cube (Dally, 1990), which extends the hypercube in the following ways: (1) the component values can be taken from the set $\{0, ..., k-1\}$ and (2) two nodes are connected by an edge if, and only if, they differ in exactly one component and the difference in this component is 1 (modulo k).

Additional modifications are necessary in order to deal with different forms of $\mathcal{DC}$, as in this thesis. E.g., a subgraph of a 3-ary n-cube, enriched with a dimension for time, is appropriate to model the index space of Karatsuba's polynomial multiplication.

We consider two representations for our model:

1. The index space is the union of all vector spaces with a finite number of dimensions. Every point in this space is identified by a list of coordinates. The length of the list equals the dimensionality required for the recursion depth of the particular problem instance, and entry i of the list contains the coordinate in dimension i. The coordinate values are elements of a small finite set.

2. The dimensionality of the computation domain is constant for all problem instances but the extent of at least one dimension depends exponentially on the recursion depth. Consider a particular, regular case: the nodes of the k-ary n-cube can be labeled uniquely with integers, by interpreting the list of coordinates as a sequence of digits of a number in radix k. Thus, a one-dimensional space with an exponential extent of k^n points is sufficient. Note that n is not fixed, like in the polytope model, but depends (e.g., logarithmically) on the size of the input.

Since both representations are related to each other, we prefer the first as long as we need to refer to particular coordinates, and the second when we deal with linearly enumerated entities, e.g., processor numbers.

For both representations, linear mappings of index space points to time and processor space, like in the polytope model, are not sufficient.

4.2.3 Speedup goals

Without regarding a restricted number of processors and communication times, many $\mathcal{DC}$ algorithms with a polynomial cost complexity could be computed in parallel in polylogarithmic time. Even if communication times are taken into account, at least a sublinear execution time can be achieved for some problems; see Sect. 4.8.

In contrast, the lower bound of the parallel time obtained by space-time mapping in the polytope model is at least linear, except for trivial cases for which it is constant. Thus, our approach can provide a higher speedup than the polytope model, from a theoretical point of view, i.e., with an unbounded number of processors.

4.2.4 Flexibility to choose efficient algorithms

The polytope model is mainly tailored for loop programs with affine data dependences. The *naïve* algorithms for polynomial or matrix multiplication fit neatly into this schema. For their more efficient counterparts, Karatsuba's polynomial multiplication and Strassen's matrix multiplication, this is not the case.

A loop program for a $\mathcal{DC}$ algorithm must somehow emulate the non-linearly recursive structure. Since the number of nested loops and, thus, the number of dimensions of the index space has to be fixed, non-linear dependences, which require integer remainder, have to be taken into account. Held and Kienhuis (1995) presented a method to replace integer remainder in index expressions by auxiliary dimensions. Other complications are that the shape of the $\mathcal{DC}$ call tree is not convex and that the loop bounds are not affine. We do not believe that even the advanced optimization techniques of the polytope model can outweigh the overhead caused by the adaptation of $\mathcal{DC}$ algorithms to fit into the model. Therefore, we propose a new model which is tailored to the requirements of $\mathcal{DC}$.

4.3 The new model

The purpose of the new model is to make index-based space-time mapping techniques applicable for skeleton programs. We pay for the departure from the polytope model, especially the fixed dimensionality and the affine dependences, with a high complexity of the model and the loss of the applicability of many techniques.

Functional skeletons lack the geometrical view useful for a space-time mapping. In the polytope model, geometry is provided by the predefined construction of a computation domain derived from the loop program. An arbitrary skeleton has far too large a variety of structure to give it a computation domain that is useful in all contexts.

We presented a preliminary solution for a special case of dcF based on geometric considerations (Herrmann and Lengauer, 1996). It is not our intention to introduce our model by taking a special case of it here: for one, the non-trivial concepts cannot be understood by looking at a special case, for another, the need for a model is dubious if only a special case is considered.

Computation domains constructed from particular geometrical perspectives have the drawback that they cannot be composed as simply as their corresponding skeletons in a functional language. The construction of a computation domain for a particular skeleton, and also the composition of computation domains, requires a manual design, guided by the following considerations:

1. What are the structural parameters, what are the customizing functions, and what is the input data? Structural parameters determine the shape of the index space.

2. Which functional arguments are expected to carry a large environment? We discuss this problem below.

3. How are the input and output data to be arranged?

Note that these choices are not obvious. Normally, one could assume, e.g., in the application (map f xs) that the function f consumes very little memory and that the list xs is probably long. Under this assumption, it appears sensible to distribute the list xs and to keep a copy of f on every processor that contains a part of xs. However, it may be the case that f is a function which carries along a data base of several hundred megabytes. This can require a distribution of the parts of f, i.e., the data base, to avoid swapping or even memory overflow. Thus, an unsuitable choice of implementation of a skeleton can lead to a substantial loss of performance or even to a failure.

Our new model shares the principles with the polytope model that the space-time mapping is computed with respect to structural parameters, like the problem size, and that the automatic optimization determines values for some decision variables. In the polytope model, these decision variables are the coefficients of an affine space-time mapping, e.g., the entries of a matrix. In our new model, we do not impose restrictions on the algebraic expressions representing the mapping. The tradeoff between the adequacy of the objective function and the quality of the obtainable solution is left to the choice of the user and may depend on the power of available mathematical tools. The implementation of the skeleton is supposed to behave as the space and time functions prescribe, i.e., the parallel execution is controlled by the values of the decision variables determined in the optimization.

For an optimization, we require cost functions that depend on the sizes of data structures, represented by symbolic expressions in structural parameters. Sect. 4.4 is about size inference, which proceeds along the syntactic structure of a function calculating the symbolic size information. Since size inference does not address geometry, the formalism used for computation domains does not affect the size inference itself, but only the constraints and the objective function in the optimization.

There are severe limits on the applicability of automatic optimization:

1. The expressibility of the computation domain and the space-time mapping in terms of structural parameters, which can be computed easily from the input data at run time.

2. The power of mathematical tools for optimization and constraint solving.

The complexity of the symbolic expressions in computation domains forces us to work on different levels of abstraction. We use the following formalisms:

1. Symbolic expressions in algebraic data types to describe computation domains and functions on them.

2. Haskell functions in places where the function itself is not subject to a symbolic manipulation. Given the values of all structural parameters, they compute particular properties of the domain or the space-time mapping. (An example is the function $\boxplus$, which computes a list of all points contained in an index space.) Thus, these functions define the semantics of the algebraic data type expressions.

3. Traditional mathematical notation where the meaning is obvious and a symbolic expression or a Haskell function would burden the reader with too much detail.

4.3.1 Nested structures, data fields and references

In a loop nest, the bounds of the loop at nesting depth j can depend on the iteration variables of the surrounding loops, i.e., of those with a depth of 0 to $j-1$. From a loop nest, one can generate a corresponding index space by scanning, i.e., by a traversal according to the iterations of the loop nest (Ancourt and Irigoin, 1991; Griebl and Lengauer, 1994). The extent of the index space in dimension j can be expressed in terms of the coordinates of the dimensions 0 to $j-1$, as the bounds of the loop nest.

Recursion can be described in a similar way: the properties of an activation of a recursive function at level j of the call tree can be expressed in terms of the node label (Sect. 3.3), which contains the indices of the activations at the levels 0 to $j-1$. The properties are, e.g., the number of recursive calls or the size of the input. However, the difference to nested loop programs is that the depth of recursion is not fixed like the nesting depth of the loop nest. In the following, we develop a formalism that enables the employment of coordinates and the scanning schema in the presence of the unbounded dimensionality of $\mathcal{DC}$.

Let us have a look at the relation between functions and geometry. A vector is isomorphic to a function that maps every element i of the vector's index set to the value indexed by i in the vector. The theory of data fields (Violard, 1994; Lisper, 1996) extends the relation between vectors and functions to the multi-dimensional case. The isomorphism between $\alpha \to \beta$ and β^α has been known for long in mathematics. For our purposes, we instantiate α with $\mathbb{Z}^n$.

To cover unbounded dimensionality, we would like to define a series $(\varphi_n)_{n\in\mathbb{N}}$ of functions, where the type of φ_0 is β and φ_{n+1} maps from $\mathbb{Z}$ to the type of φ_n.

We cannot achieve a finite representation of the type of φ_n this way, because φ_n would be curried with n arguments and n is variable. However, curried and uncurried versions of a function are isomorphic, as proved by the existence of functions `curry` and `uncurry` (Bird, 1998). Thus, we can give φ_n the type $(\mathbb{Z}^n \to \beta)$. The representation of the definition of φ_n is discussed later.

Note the geometrical correspondence of λ-abstraction and application: λ-abstraction corresponds to an embedding of an n-dimensional space into an $(n+1)$-dimensional space and application corresponds to projection. For intuition, let us view $\mathbb{Z}^n$ as an n-dimensional grid and a mapping from $\mathbb{Z}^n$ to β as a grid in which the points are tagged with elements of type β.

In the grids we deal with, the tags correspond to operations of a computation. The terminating computations we consider have a finite number of operations and, thus, our grids are bounded. Like in the polytope model, we have a compact representation of the bounds of a grid. If the points of the grid form a polytope, we can use affine linear expressions; the bounds of a finite union of polytopes can be defined piecewise. For $\mathcal{DC}$, the grid cannot be described this way. Here, we exploit the relation between coordinate vectors and functions. φ_n is a function which takes all coordinate values of a grid point and delivers the tag. For the definition of the lower resp. upper bound in dimension j, we take a function which depends on the coordinate values of the dimensions 0 to j–1. This function is involved in symbolic manipulation, akin to the affine linear expressions in the polytope model. Thus, the function is represented by an algebraic expression.

A major aspect which distinguishes our model from the polytope model is the unbounded dimensionality of computation domains, which complicates their formal description significantly. Computation domains in the polytope model have a fixed dimensionality, because the loop nests they describe have a fixed nesting depth, which equals the domain's dimensionality. The depth of a $\mathcal{DC}$ recursion is not fixed but depends on the problem size. Consider a non-recursive skeleton S for $\mathcal{DC}$ of depth n. We can obtain a skeleton for depth $n+1$ by embedding S into a loop. Each call of S is then used to solve a particular subproblem. By induction, it follows that a $\mathcal{DC}$ recursion corresponds to a loop nest whose nesting depth depends on the depth of recursion.

Because the number of dimensions is not a constant, it is impossible to describe the bounds function by nested λ-abstractions in which each abstraction takes the value of a coordinate. We solve this problem by using the following naming schema in which we refer implicitly to the indices in the lower dimensions. An expression which specifies a bound in dimension j refers to the current index in dimension i $(i<j)$ by $\Rsh i$. Thus, $\Rsh i$ is called a <u>reference</u>.

Again, if the dimensionality were constant, like in the polytope model, all bound expressions could be given by a list in which entry j contains the symbolic expressions for the lower and upper bounds of dimension j. Since this is not the case here, we use the symbol δ as a representative of the current dimension. Bounds of different dimensions can be distinguished by a case analysis on the value of δ or by using δ in an arithmetic expression.

4.3.2 Computation domain

The computation domain consists of an index space and a mapping from this index space to the set of operations. We prefer to use a particular structural parameter which determines the depth of recursion in the program.

In Sect. 4.3.1, we have justified the need for unbounded dimensionality and discussed its problems. An abstract language described by the following algebraic data types has been designed to deal with unbounded dimensionality.

Our definitions are given in Haskell with a beautified syntax; auxiliary definitions of minor importance are only explained in English.

<u>type</u> SPar $= [\mathbb{Z}]$

<u>data</u> OP $= (+) \mid (-) \mid (*) \mid (\uparrow) \mid (\mathsf{IF}) \mid (<) \mid (=)$

<u>data</u> DimExp $= \delta \mid \mathsf{C}\ \mathbb{Z} \mid \Psi\ \mathbb{Z} \mid$ DOp OP [DimExp]

<u>data</u> IndexExp $=$ D DimExp $\mid$ $\Rsh$DimExp $\mid$ IOp OP [IndexExp] $\mid$ Item

<u>type</u> PointOp $=$ (IndexExp $\times$ IndexExp)

1. SPar is the type of structural parameters, which we choose arbitrarily to be a list of integers. Note that, unless we compile for a particular problem size, configuration, etc., the values of this list are unknown at compile time. In the model, they are referred to by their index in the list. The meaning of the positions in the list is not predefined but given by the skeleton developer. E.g., for dcF, we could choose four parameters: one for the depth of recursion and three for the three division degrees in dcF.

2. OP contains a set of operators for symbolic expressions, which cannot be evaluated before the instantiation of the structural parameters is known. $(\uparrow)$ is the maximum operator. (IF) is a ternary operator of a condition and two alternatives.

3. A DimExp expression is used to refer to positions we want to define, e.g.,

$$\mathsf{DOp}\ (\mathsf{IF})\ [\,\mathsf{DOp}\ (<)\ [\,\delta\ ,\mathsf{C}\,5\,],\ x,\ y\,]$$

means that, for all dimensions less than 5, the specification x and, for all other dimensions, the specification y applies. A DimExp expression can be the current dimension (δ), a constant ($\mathsf{C}\ i$), the value of structural parameter i ($\Psi\ i$) or a combination of DimExp expressions *exps* with an operator ($\mathsf{DOp}\ \mathsf{OP}\ exps$).

4. An $\mathsf{IndexExp}$ expression is used to calculate a value that is assigned to positions. It can either be an expression *exp* of type DimExp ($\mathsf{D}\ exp$), a reference to a coordinate in dimension i ($\Rsh i$), where i is determined by a DimExp expression, a combination of $\mathsf{IndexExp}$ expressions *exps* with an operator ($\mathsf{IOp}\ \mathsf{OP}\ exps$) or a source of a data dependence, where the symbol Item is used as a representative for the current operand position. An $\mathsf{IndexExp}$ can depend on a DimExp expression because the value may depend on the coordinate at which we want to place the value. E.g., in an expression which defines the lower bounds, δ represents the coordinate we want to define the bound for, and $\Rsh i$ refers to the value of coordinate i the bound depends on.

5. A $\mathsf{PointOp}$ expression is a pair whose first element contains the operation code and whose second element contains the sources of the operands. In both expressions, the references are bound to the coordinates of the target point, i.e., the point at which the result of the operation resides. In the second expression, Item represents the operand position and δ the requested coordinate of the source point.

Before the definition of the notion computation domain we define the *index space* as a part of it.

Definition 4.5 (Index Space)
An index space $\mathfrak{IS}_\psi$ in the structural parameters ψ is a triple $(\Delta_\psi, \Downarrow_\psi, \Uparrow_\psi)$, whose components have the following meaning:

1. Δ_ψ :: DimExp, the dimensionality of the index space.

2. $\Downarrow_\psi$ :: $\mathsf{IndexExp}$, the lower bounds; references address coordinate values in dimensions lower than the current dimension (δ).

3. $\Uparrow_\psi$ :: $\mathsf{IndexExp}$, the upper bounds, using references like in $\Downarrow_\psi$.

□

We use the algebraic expressions stated above to define the index space in Haskell, with the structural parameters ψ accessible via Ψ. The definition of IndexSpace is by an algebraic data type with the single constructor $\mathfrak{IS}$ and three labeled fields (separated by $\times$).

```
data IndexSpace = 𝔍𝔖{ Δ::DimExp × ⇓::IndexExp × ⇑::IndexExp }
```

Example 4.2 (Index space for the loop nest in Ex. 4.1)
Let us revisit the two-dimensional index space depicted in Fig. 4.2. It includes points not only for the iterations of the loop nest, but also for the input data. Case distinctions are necessary to exclude the points [0,0] and [n,n+1].

The formal definition is_loops is shown below with framed comments. The names n, i and j are taken from Fig. 4.2. n, the only structural parameter, is referred to by (Ψ 0).

```
is_loops ∈ IndexSpace
is_loops = 𝔍𝔖 {
  Δ = C 2,  the index space has two dimensions
  ⇓ = IOp (IF) [ IOp (=) [D δ, D(C 0)],  if is loop on i
                 D (C 0),  then the lower bound (for i) is 0
                 IOp (IF) [ IOp (=) [↱ (C 0), D (C 0)],  else if i=0
                            D (C 1),  then the lower bound (for j) is 1
                            D (C 0) ] ],  else 0
  ⇑ = IOp (IF) [ IOp (=) [D δ, D(C 0)],  if is loop on i
                 IOp (IF) [ IOp (=) [D (Ψ 0), D (C 0)],  then if n=0
                            D (C (−1)),  then the i loop does no iteration
                            D (Ψ 0)],  else the upper bound (for i) is n
                 IOp (IF) [ IOp (=) [↱ (C 0), D (Ψ 0)],  else if i=n
                            D (Ψ 0),  then the upper bound (for j) is n
                            IOp (+) [↱ (C 0), D (C 1)] ] ]  else i+1
}
```

□

Definition 4.6 (Computation domain)
A computation domain $\mathfrak{CD}_\psi$ in the structural parameters ψ is a pair ($\mathfrak{is}_\psi$, $\bowtie_\psi$), whose components have the following meaning:

1. $\mathfrak{is}_\psi$ :: IndexSpace, the index space of the domain. All points whose coordinate in dimension 0 equals the upper bound of dimension 0 deliver values to the environment and are called output points. If the extent of the index space in dimension 0 is 1, the domain does not receive input data but delivers a result which only depends on structural parameters

and customizing functions. Otherwise, all points whose coordinate in dimension 0 equals the lower bound of dimension 0 are input points, i.e., receive their values from the environment. (To maintain the relation to the polytope model, we make an exception for Ex. 4.1.)

2. $\bowtie_\psi$ :: PointOp, the operation at every point of the index space, expressed in terms of references. If these references are substituted with the coordinates of a particular point of the index space, one obtains in the first component the operation code and in the second component the source points of the operands. The result of the operation is assigned to the same point as the operation (*owner-computes rule*). If the point is an input point, a dummy operation, called *input operation*, is assigned to it, which does not perform any work.

□

There is a correspondence between the points of the computation domain and the set of operations performed, where the input values correspond to input operations. We make use of two different views of the computation domain, which is motivated by the owner-computes rule: as a domain of places for values and as a domain of operations which produce these values.

Our definition of a computation domain in Haskell is:

$\underline{\mathtt{data}}$ CompDom $= \mathfrak{CD}\{$ $\mathfrak{is}$::IndexSpace $\times$ $\bowtie$::PointOp $\}$

Next, the meaning of the algebraic expressions, which has been given informally above, is defined formally. The meaning of expressions of type DimExp is defined by function evalDim in terms of the current structural parameters and the current dimension. Function evalInd defines the meaning of expressions of type IndexExp in terms of the current structural parameters, the current coordinate values represented by references, the current dimension and the item number used to enumerate the arguments of the operands. Function appop performs the application of the respective abstract operation and is not shown here.

$$
\begin{aligned}
&\mathsf{evalDim} \in \mathsf{DimExp} \to (\mathsf{SPar}{\times}\mathbb{Z}) \to \mathbb{Z}\\
&\mathsf{evalDim}\ \delta\ (\_, i) = i\\
&\mathsf{evalDim}\ (\mathsf{C}\ i)\ \_ = i\\
&\mathsf{evalDim}\ (\Psi\ i)\ (\psi, \_) = \psi\,!!\,i\\
&\mathsf{evalDim}\ (\mathsf{DOp}\ op\ xs)\ s = \mathsf{appop}\ op\ \ (\mathsf{map}\ (\lambda x \to \mathsf{evalDim}\ x\ s)\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{evalInd} \in \mathsf{IndexExp} \to (\mathsf{SPar}{\times}[\mathbb{Z}]{\times}\mathbb{Z}{\times}\mathbb{Z}) \to \mathbb{Z}\\
&\mathsf{evalInd}\ (\mathsf{D}\ e)\ (\psi, \_, i, \_) = \mathsf{evalDim}\ e\ (\psi, i)\\
&\mathsf{evalInd}\ (\Rsh\ e)\ (\psi, refs, i, \_) = refs\ !!\ (\mathsf{evalDim}\ e\ (\psi, i))\\
&\mathsf{evalInd}\ (\mathsf{IOp}\ op\ xs)\ s = \mathsf{appop}\ op\ (\mathsf{map}\ (\lambda x \to \mathsf{evalInd}\ x\ s)\ xs)\\
&\mathsf{evalInd}\ \mathsf{Item}\ (\_, \_, \_, it) = it
\end{aligned}
$$

For simplicity, we sometimes omit the application of evalDim and evalInd as well as the constructors C and D, if these algebraic data type expressions are used embedded in traditional mathematical notation. E.g., we write just the integer constant i instead of (evalInd (D (C i)) _). In these cases, the structural parameters (ψ), references (*refs*), etc., which would influence the evaluation of evalDim and evalInd, are meant to refer to the (single) index space in question.

The algebraic expressions constitute the index space as follows. The set of points belonging to the index space is computed by function $\boxplus$. First, the dimensionality is obtained by evaluating Δ on the structural parameters Ψ. Then, function φ traverses the dimensions and collects all points.

```
type Point = [ℤ]

⊞ ∈ IndexSpace → SPar → [Point]
⊞ (𝔍𝔖 {Δ,⇓,⇑}) ψ
 = let  dims = evalDim Δ (ψ,0)
        φ 0 = [[]]
        φ d | d>0 =  concat
                     [ map (λy → x++[y]) [  evalInd ⇓ (ψ, x, d−1, 0) ..
                                            evalInd ⇑ (ψ, x, d−1, 0)]
                     |  x ← φ(d−1)]
   in φ dims
```

Since ψ is only used to instantiate the elements of $\mathfrak{IS}$, we write, instead of ($\boxplus\ \mathfrak{IS}\ \psi$), in the sequel ($\boxplus\ \mathfrak{IS}_\psi$) or even ($\boxplus\ \mathfrak{CD}_\psi$).

Let us revisit the nested loop program from Sect. 4.1 and its index space definition is_loops. The dimensionality is 2; thus, the result is $\varphi\,2$. $\varphi\,2$ is defined in terms of $\varphi\,1$, which itself requires $\varphi\,0$. $\varphi\,0$ delivers the empty list as only value for x in $\varphi\,1$. $\varphi\,1$ evaluates the lower bounds ($\Downarrow$) and the upper bounds ($\Uparrow$) in dimension 0 (loop variable i). The result of $\varphi\,1$ is therefore the list $[0\,..\,n]$ in the case that $n>0$ and the list $[0\,..\,-1]$ ($=[\,]$) in the case $n=0$, according to the specification in is_loops. $\varphi\,2$ computes, for every x in $[0\,..\,n]$, the sequence in the bounds in dimension 1 (loop variable j) and appends every element to the coordinate values x determined before. Thus, we obtain the following results:

n	$\varphi\,1$	$\varphi\,2$, result
0	`[]`	`[]`
1	`[0,1]`	`[[0,1],[1,0],[1,1]]`
2	`[0,1,2]`	`[[0,1],[1,0],[1,1],[1,2],[2,0],[2,1],[2,2]]`

4.3.3 Example: Index space for dcF

Let us have a look at the divide phase of an implementation of skeleton dcF, with a degree of input data division of 2 and a degree of problem division of 3, as, e.g., in the Karatsuba polynomial multiplication. The computation domain for a recursion depth of 2 is depicted in Fig. 4.3. Later, when we come to the iterative versions of the skeletons, we discuss the structure of dcF in more detail (Sect. 4.7.6.3). Here, we only want to demonstrate how an index-based computation domain for a $\mathcal{DC}$ skeleton can be defined.

A distinguished dimension is dimension 0, which must be mapped to time. The schema allows for any recursion depth, i.e., any number of dimensions.

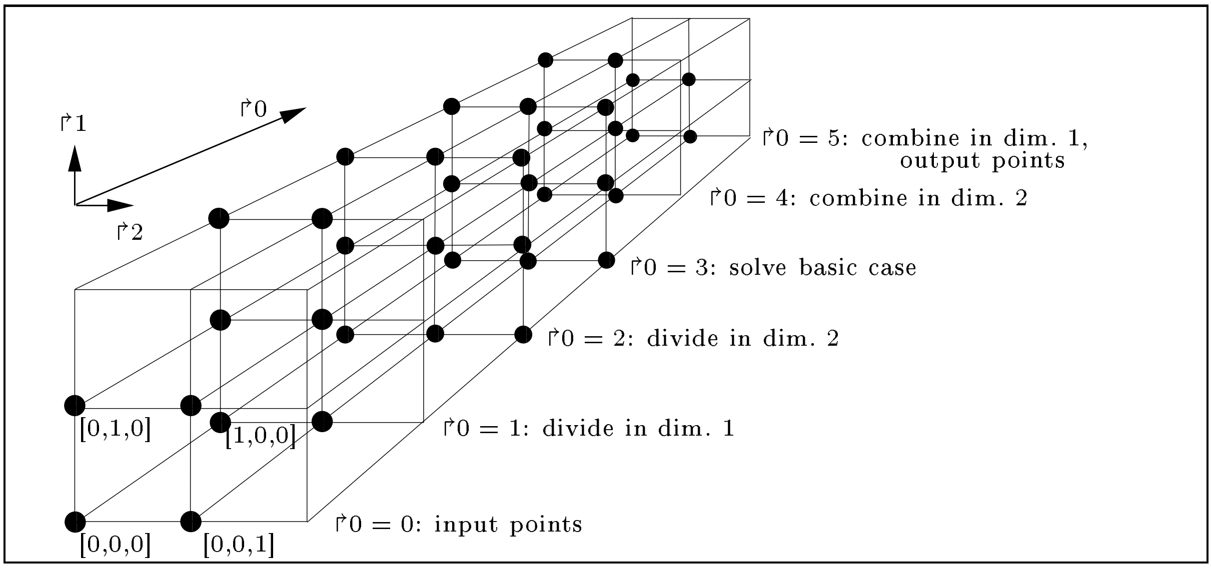


Figure 4.3: Example index space for (dcF 3 2 2), n=2 (Δ=3)

The definition of the index space for the divide phase is given by the expression is_dcF_div. The dimensionality is the recursion depth (the structural parameter n) plus 1. Therefore, Δ = DOp (+) [Ψ 0, C 1]. The lower bound is 0 in every dimension ($\Downarrow$ = D (C 0)). The upper bound ($\Uparrow$) depends on the dimension. The upper bound of dimension 0 equals the recursion depth. Thus, it is given by D (Ψ 0) and is chosen in the case that the symbol δ for the current dimension is assumed to represent the value 0, which is expressed by

IOp (=) [D δ, D(C 0)].

Otherwise, δ is one of the other dimensions. Then, the extent of dimension δ depends on the level in the $\mathcal{DC}$ call tree, which equals the coordinate in dimension 0 (↱ (C 0)). If the level is smaller than δ, expressed by

IOp (<) [↱ (C 0), D δ]

the dimension δ is used to aggregate two data partitions and the upper bound is 1. Otherwise, it is used to aggregate three problem partitions and the upper bound is 2.

is_dcF_div $\in$ IndexSpace
is_dcF_div = $\mathfrak{IS}$ {
  Δ = DOp (+) [Ψ 0, C 1], $\boxed{\text{the dimensionality is } n{+}1}$
  $\Downarrow$ = D(C 0), $\boxed{\text{all lower bounds are 0}}$
  $\Uparrow$ = IOp (IF)
    [IOp (=) [D δ, D(C 0)], $\boxed{\text{if dimension=0}}$
      D (Ψ 0), $\boxed{\text{then the upper bound is } n}$
      IOp (IF) [IOp (<) [↱ (C 0), D δ], $\boxed{\text{else if coordinate 0} < \text{dimension}}$
        D(C 1), $\boxed{\text{then the upper bound is 1}}$
        D(C 2)]] $\boxed{\text{else 2}}$
}

For simplicity, we have decided against an index space definition for dcF in full generality. The basic and combine phase can be added by a case distinction. The three division degrees can be specified by three additional structural parameters, say $\Psi 1$ to $\Psi 3$. Function $\boxplus$, applied to is_dcF_div, delivers for $n=2$ the following set of points, which can be found in Fig. 4.3 in the first three vertical slices.

```
[[0,0,0],[0,0,1],          [0,1,0],[0,1,1],
 [1,0,0],[1,0,1],          [1,1,0],[1,1,1],          [1,2,0],[1,2,1],
 [2,0,0],[2,0,1],[2,0,2],[2,1,0],[2,1,1],[2,1,2],[2,2,0],[2,2,1],[2,2,2]]
```

4.3.4 Dependence graph

The space-time mapping method requires a formalization of the flow of data between the points of the index space. In a *dependence graph*, the nodes represent points of the index space and the (directed) edges reflect data dependences.

The dependence graph of a computation domain $\mathfrak{CD}_\psi$ is stored in a compact form in $\bowtie_\psi$, whose second component contains, for each target point, the source points of the dependences.

Definition 4.7 (Dependence graph)
The dependence graph $(N_\psi, \prec_\psi)$ of a computation domain $\mathfrak{CD}_\psi$ is a DAG in which the nodes N_ψ $(= \boxplus\ \mathfrak{CD}_\psi)$ represent the points in the index set, and the set $\prec_\psi \subseteq (N_\psi \times N_\psi)$ of edges is the transitive closure of the relation formed by the set $\widehat{\prec}_\psi \subseteq (N_\psi \times N_\psi)$ of direct dependences:

$$\widehat{\prec}_\psi \stackrel{\text{def}}{=} \{\, (a,b) \in (N_\psi \times N_\psi) \mid (\mathit{operation\_at}\ \ b)\ \text{`}\mathit{depends\_on}\text{`}\ \ a\}$$

where

$$\mathit{operation\_at}\ \ b \stackrel{\text{def}}{=} \bowtie_\psi(\bigcirc_{i=0}^{\Delta_\psi-1} [\Rsh i := b\,!!\,i])$$

$$(\oplus, \mathit{args})$$
$$\text{`}\mathit{depends\_on}\text{`}\ \ a \stackrel{\text{def}}{=} \bigvee_{i=0}^{\mathsf{arity}\oplus-1}(\bigwedge_{j=0}^{\Delta_\psi-1}((\mathit{args}[\mathsf{Item} := i])[\delta := j]) = a\,!!\,j)$$

The function *operation_at* computes, from a compact description $\bowtie_\psi$ of the set of all operations, the current operation at a particular point b. Therefore, a composition $(\bigcirc_{i=0}^{\Delta_\psi-1})$ of substitutions replaces, for every i, reference i by the coordinate value in dimension i of b. The function '*depends_on*' checks whether the value located at point a is the source of a data dependence whose target is the operation $(\oplus, \mathit{args})$, where $\oplus$ is the operation code and *args* a compact description of the set of all possible arguments. Function arity delivers the number of arguments of an operation, if given its code. $\bigvee_{i=0}^{\mathsf{arity}\oplus-1}$ computes a reduction with logical *or* $(\vee)$, while providing i as an index of an operand. The symbol Item, which stands for the current operand position, has to be replaced by this index. Similarly, $\bigwedge_{j=0}^{\Delta_\psi-1}$ computes a reduction with logical *and* $(\wedge)$, while providing j as a coordinate of the source point. δ has to be replaced by the coordinate value in dimension j of the source point requested. $\square$

We take the notion of a *dependence vector* from the polytope model and use it in the proof of Lemma 4.13.

Definition 4.8 (Dependence vector)
If $(a, b) \in \prec_\psi$, then the difference $b - a$ of the target and the source point of the dependence is called the dependence vector. □

4.3.5 Volume of an index space

The domain of the space-time mappings is an index space of arbitrary dimensionality. In contrast, the codomain is the product of a *fixed* number of time and processor dimensions. It is impossible to refer to the indices of all domain dimensions explicitly, as is done in the polytope model. Instead, a recursive definition serves to access the coordinate values in every dimension in such a way that the points in a part of the index space can be enumerated, i.e, ordered in a one-dimensional space. Although spaces of any finite dimensionality are possible as targets, we concentrate here on a simple enumeration in one dimension, for simplicity.

The enumeration of the points of an n-dimensional index space is based recursively on the enumeration of the points in all $(n-1)$-dimensional subspaces. More precisely: a point that is located in subspace i, where it has the offset j, receives the number $(\sum_{k=0}^{i-1} volume_k + j)$, where $volume_k$ is the total amount of points in subspace k. Note that the dimensionality n can be chosen arbitrarily. For a particular index space, it depends on the structural parameters.

To keep the number of substitutions small, the formal counterpart of *volume*, volume, takes four parameters: the first dimension of the subspace (m), the last dimension of the subspace (n), the lower bounds $(\Downarrow)$ and the upper bounds $(\Uparrow)$. The volume of the index space $\mathfrak{IS}(\Delta, \Downarrow, \Uparrow)$ is obtained by $\mathsf{volume}(0, \Delta{-}1, \Downarrow, \Uparrow)$.

1. In the basic case $(m = n{+}1)$, the subspace is zero-dimensional and consists only of a single point:

$$\mathsf{volume}(n{+}1, n, \Downarrow, \Uparrow) = 1$$

2. In the recursive case $(m \leq n)$, the volume is computed by summing the volumes of all subspaces. To determine the range of coordinates in the bound expressions $\Downarrow$ and $\Uparrow$, the symbol δ is substituted by the current dimension m. (Provided that we keep the number of dimensions fixed, which we cannot do here, this would be equivalent to selecting element m from a list of bound expressions.) The subspace is identified by its coordinate value in the current dimension m. Thus, $\Lsh m$ is substituted in the bound expressions for the recursive use by the current value of i.

$$\mathsf{volume}(m, n, \Downarrow, \Uparrow) = \sum_{i=\Downarrow[\delta:=m]}^{\Uparrow[\delta:=m]} \mathsf{volume}(m{+}1, n, \Downarrow[\Lsh m := i], \Uparrow[\Lsh m := i])$$

4.3.6 The position of a point in an index space

Based on the function volume, the position of a point in a subspace can now be given precisely by a function named position. position takes five parameters. The first parameter p is a point given by a list of coordinates. The four remaining parameters have the same meaning as the four parameters of the volume function. Thus, the position of point p in index space $\mathfrak{IS}(\Delta, \Downarrow, \Uparrow)$ is given by $\mathsf{position}(p, 0, \Delta-1, \Downarrow, \Uparrow)$.

1. The only point in the zero-dimensional space is located at position 0.

$$\mathsf{position}(p, n+1, n, \Downarrow, \Uparrow) = 0$$

2. In the recursive case ($m \leq n$), the position of the point is given by the sum of the volumes of the subspaces containing the points of smaller numbers and the position of the point in its own subspace.

$$\mathsf{position}(p, m, n, \Downarrow, \Uparrow) =$$

$$\sum_{i=\Downarrow[\delta:=m]}^{p\,!!\,m-1} \mathsf{volume}(m+1, n, \Downarrow[\vec{\ }m := i], \Uparrow[\vec{\ }m := i])$$

$$+\ \mathsf{position}(p, m+1, n, \Downarrow[\vec{\ }m := p\,!!\,m], \Uparrow[\vec{\ }m := p\,!!\,m])$$

4.3.7 Schedule and allocation

Definition 4.9 (Space-time mapping)
A space-time mapping is a one-to-one mapping of points of a computation domain $\mathfrak{CD}_\psi$ to the Cartesian product of discrete time and space. The time component of this mapping is called the *schedule*, the space component the *allocation*. □

As illustrated by Darte and Vivien (1994), a multi-dimensional schedule (Feautrier, 1992) can be useful. In the case of $\mathcal{DC}$, the multiplicity of dimensions is due to a nesting of skeletons. Consider a skeleton A that unfolds recursively and calls a skeleton B at every level, e.g., as presented in Sect. 3.5. In this case, the first time component reflects the state of A, while the second component reflects the state of B. However, a multi-dimensional schedule does not solve the problem of a necessary reduction of the unbounded dimensionality of the computation domain. This is achieved by the use of function position.

Definition 4.10 (Schedule)
For a set ψ of structural parameters, we denote the schedule (mapping to time) with τ_ψ, where $\tau_\psi \in (\boxplus\, \mathfrak{CD}_\psi) \to \mathbb{N}^{t_\psi}$ and $t_\psi \in \mathbb{N}$, the dimensionality of time. We impose the lexicographic ordering ($<_{\mathtt{lex}}$) on the codomain of the schedule: if $x, y \in \mathbb{N}^{t_\psi}$ then

$$(x <_{\mathtt{lex}} y) \iff (\bigvee_{k=0}^{t_\psi-1} (\bigwedge_{i=0}^{k-1} x_i = y_i \ \wedge\ x_k < y_k)).$$

□

Example 4.3 (Schedule for Ex. 4.1)
The schedule depicted in Fig. 4.2 is $time[i,j] = i+j-1$. Since i corresponds to $\delta = 0$ and j to $\delta = 1$, the schedule is: $\tau_\psi\ p = [(p\,!!\,0) + (p\,!!\,1) - 1]$. □

Example 4.4 (Schedule for dcF)
Take the example domain $\mathfrak{CD}_\psi$ for the index space is_dcF_div from Sect. 4.3.3.

1. A sequential, one-dimensional schedule is

$$\tau_\psi\ p = [\mathsf{position}(p, 0, \Delta_\psi - 1, \Downarrow_\psi, \Uparrow_\psi)].$$

2. The free schedule (parallel, one-dimensional) is

$$\tau_\psi\ p = [p\,!!\,0].$$

Here, all points which have the same coordinate in dimension 0 are scheduled at the same time. The problem with this schedule is that it requires an unlimited number of processors.

3. A sequential, two-dimensional schedule is

$$\tau_\psi\ p = [p\,!!\,0, \mathsf{position}(p, 1, \Delta_\psi - 1, \Downarrow_\psi[\vec{\ulcorner} 0 := p\,!!\,0], \Uparrow_\psi[\vec{\ulcorner} 0 := p\,!!\,0])].$$

4. A parallel, two-dimensional schedule with a choice for $P \in \mathbb{N}$ $(0 < P < \Delta_\psi - 1)$ is

$$\tau_\psi\ p = [p\,!!\,0, \mathsf{position}(p, 1, \Delta_\psi - 1 - P, \Downarrow_\psi[\vec{\ulcorner} 0 := p\,!!\,0], \Uparrow_\psi[\vec{\ulcorner} 0 := p\,!!\,0])].$$

Here, all dimensions aside from $\Delta_\psi - P$ to $\Delta_\psi - 1$ are mapped to time. If the maximum extent in every dimension does not exceed k, k^P processors are sufficient to satisfy the required number of computation resources.

□

Definition 4.11 (Causality)
We say that a schedule τ_ψ is <u>causal</u> if, and only if,

$$(\forall\ (a, b) \in \prec_\psi\ :\ \tau_\psi\ a <_{\mathtt{lex}} \tau_\psi\ b).$$

If a data dependence exists from a to b, then a must be executed before b, i.e., the schedule of a must be lexicographically smaller than the schedule of b. In the above example for dcF, we always obtain a causal schedule if we map the levels of the call graph to the first dimension (most significant with respect to $<_{\mathtt{lex}}$) of the schedule, because all data dependences in dcF have a positive value in this dimension. □

We use the notion of a *topology* for the connectivity graph of a network of processors.

Definition 4.12 (Topology)
A topology is a directed graph whose set of nodes corresponds to (for simplicity, we say *is*) the set of processors, and which contains an arrow from any processor a to any other processor b if, and only if, there is a hardware facility for processor a to send a value to processor b without any other processor being involved. The nodes are identified with the numbers from 0 to the number of processors minus 1. The edges are represented by an IndExp expression using the references $\Lsh 0$ and $\Lsh 1$. The index expression delivers the number of *data items* that can be transmitted per time step from processor $\Lsh 0$ to processor $\Lsh 1$. If there exists no connection, the value is 0. A data item is the maximum amount of data produced by a point in the computation domain at the level of resolution of the real processor. For simplicity, one sometimes must abstract from communication bandwidth. This can be modeled by adding a symbol for an unlimited number of data items in the index expressions. $\Box$

Definition 4.13 (Allocation)
For a set ψ of structural parameters and a topology $(P,\_)$, we denote the allocation (mapping to space, *placement*) with σ_ψ, where $\sigma_\psi \in (\boxplus\ \mathfrak{CD}_\psi) \to P$. $\Box$

Example 4.5 (Allocation for Ex. 4.1)
If, in Fig. 4.2, n is not instantiated, the schedule $time[i,j] = i+j-1$ incurs an unbounded degree of parallelism because the number of processors must be chosen dependent on n. An allocation could be, e.g., $place[i,j] = i$ or $\sigma_\psi\ p = [p\,!!\,0]$. $\Box$

Example 4.6 (Schedule and allocation for dcF)
Let us revisit the scheduling example for dcF from above.

1. The free schedule is

$$\tau_\psi\ p = [p\,!!\,0].$$

A corresponding allocation, which depends on the schedule, could be

$$\sigma_\psi\ p = [\mathsf{position}(p, 1, \Delta_\psi - 1, \Downarrow_\psi[\Lsh 0 := p\,!!\,0], \Uparrow_\psi[\Lsh 0 := p\,!!\,0])].$$

2. An allocation for the parallel, two-dimensional schedule

$$\tau_\psi\ p = [p\,!!\,0, \mathsf{position}(p, 1, \Delta_\psi - 1 - P, \Downarrow_\psi[\Lsh 0 := p\,!!\,0], \Uparrow_\psi[\Lsh 0 := p\,!!\,0])]$$

could be

$$\sigma_\psi\ p = [\mathsf{position}(p, \Delta_\psi - P, \Delta_\psi - 1, \Downarrow_\psi[\Lsh 0 := p\,!!\,0], \Uparrow_\psi[\Lsh 0 := p\,!!\,0])].$$

Because the only dimension which is referred to as a reference is dimension 0, we can choose a completely synchronous implementation (both time components implemented by two outermost sequential loops) or a partly asynchronous implementation (the first time component as the outermost loop, the space component as the middle loop and the second time component as the innermost loop).

□

Definition 4.14 (Legal space-time mapping)
Let $\mu_\psi\, x = (\tau_\psi\, x, \sigma_\psi\, x)$ be a space-time mapping for $\mathfrak{CD}_\psi$. μ_ψ is called legal, if, and only if,

1. τ_ψ is causal and
2. μ_ψ is one-to-one: $(\forall x, y \in \boxplus\, \mathfrak{CD}_\psi : \mu_\psi\, x = \mu_\psi\, y \implies x = y)$

□

Causality ensures that every variable used already carries its value. The one-to-one property ensures two conditions:

1. A processor does not have to perform more than one operation at a time, i.e., it can do all the work to make the expected result available in time.
2. Each value can be determined uniquely by taking the time when and the processor where it is produced.

A legal space-time mapping is a prerequisite for a correct execution. In the previous examples, we always have chosen legal space-time mappings.

If communication restrictions have to be considered, we can use a refinement of the causality, which we call *feasibility*.

Definition 4.15 (Dilation)
The dilation from point a to b in a topology is the minimum number of links that have to be traversed to send a message from a to b. □

Example 4.7 (Feasibility problem 1: dilation)
Assume that there are two points p and q in a computation domain, that there exists a direct dependence from p to q and that q is scheduled one time step after p, which is sufficient for causality. However, the allocation assigns p and q to two different processors, which do not have a direct link. Then, the datum produced by p requires more than one time step to arrive at q, i.e., it is not available when needed. A time step contains the time needed for communicating across all direct links plus the time for the local computation. □

Definition 4.16 (Congestion)
The congestion is the maximum number of items that have to be transmitted via a physical link at the same time step in order to implement a particular parallel computational schema in time. □

Example 4.8 (Feasibility problem 2: congestion)
Assume that (1) there are three points x_0, x_1 and x_2 in the computation domain, (2) x_i is scheduled at time step i and allocated to processor i, (3) the only way to get from processor 0 to 2 is via processor 1, (4) the capacity of the link from processor 1 to 2 equals the maximum of the values produced at x_0 and x_1, and (5) x_2 requires both the values of x_0 and of x_1.

The dilation is not a problem here since the communication of each data element in isolation can be performed in time. On the other hand, for the values of x_0 and x_1 to arrive at processor 2 in time, both must take the link from processor 1 to processor 2 at the same time step, which is not possible because the link capacity is exceeded. This problem is due to congestion.
□

These feasibility problems complicate the space-time mapping dramatically. Contrary to the computation domain, where the dependences are given, the route a datum takes through a network is irrelevant for the computation itself. Thus, decision variables have to be introduced in the optimization which specify whether a datum requires a particular link, to assure a consistent routing.

4.3.8 Composition of computation domains

This section presents the three basic kinds of composition: *sequential*, *parallel* and *nested*. Particular applications in the larger context of skeletons are presented in Sect. 4.5, since the estimation of their possibilities and difficulties benefit from knowledge about size inference and the Γ-calculus presented in Sect. 4.4.

Our higher-order functional language $\mathcal{HDC}$ provides the glue that connects all skeletons in the program to a working implementation. However, the structure in the program can be identified more easily if skeletons are connected in the model of the computation domain, as far as possible. We remind the reader that computation domains cannot be composed as simply as their corresponding skeletons, due to the intended degrees of freedom in the geometrical layout.

In a functional language, the potential for a composition is established by a match of types. If geometry comes into play, we have to match shapes of the interfacing parts of two domains, e.g., by choosing particular dimensions for a consistent data distribution. We do not present a formalism to describe geometry here, like the theory of data fields does (Violard, 1994; Lisper, 1996). However, the necessary restructuring operations that are required, e.g., a change of basis, can be described by substitution of references.

We have chosen one of many ways to perform the composition, which we think is both simple and productive, since the $\mathsf{map}/\circ$ law

$$\mathsf{map}\ (g \circ f) = \mathsf{map}\, g\ \circ\ \mathsf{map}\, f$$

carries over to the composition of domains.

Parallel and sequential composition both require that the points in the domains involved are all at the same level of resolution, which implies that the involved domains refer to skeletons with the same instantiation of polymorphic types.

Nesting of one domain into another requires that the points of the inner domain have a finer resolution of time and space than the points of the outer domain. We do not define recursive nesting formally but develop computation domains, e.g., for $\mathcal{DC}$, manually with the help of equational reasoning.

4.3.8.1 Sequential composition

Sequential composition is defined by the associative operator $(\circ)$.

Definition 4.17 (Sequential composition $(\circ)$)

$(\circ) \in (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$
$(g \circ f)\ x = g\ (f\ x)$
□

Let the computation domain of f be $\mathfrak{CD}_\phi$ and the one of g be $\mathfrak{CD}_\psi$. We are looking for a computation domain of $g \circ f$ and use the operator $(\widehat{\circ})$ for the domain composition. In order to define a sequential composition of the computation domains, the following properties have to be satisfied:

1. The domains must have the same dimensionality, i.e., $\Delta_\phi = \Delta_\psi$. If this is not the case, the domain with the smaller dimensionality must be embedded into one with the larger dimensionality.

2. The shape of the projection of the output points of domain $\mathfrak{CD}_\phi$ in its last coordinate of dimension 0 equals that of the input points of domain $\mathfrak{CD}_\psi$ in its first coordinate of dimension 0.

Then, the resulting domain $\mathfrak{CD}_\theta = \mathfrak{CD}_\psi \mathbin{\widehat{\circ}} \mathfrak{CD}_\phi$ has the following properties:

1. It computes the function $g \circ f$.

2. $\Delta_\theta = \Delta_\phi$.

3. The domains are concatenated in dimension 0 such that the first coordinate of $\mathfrak{CD}_\psi$ and the last coordinate of $\mathfrak{CD}_\phi$ coincide and the operation is taken from domain $\mathfrak{CD}_\phi$, overriding the input operation in domain $\mathfrak{CD}_\psi$, which has now become useless.

4. The bound expressions are given by a case distinction depending on the coordinate in dimension 0.

4.3.8.2 Parallel composition

Here, parallel composition exploits potential parallelism and does not exclude a purely sequential implementation. In our style of functional programming, parallel computations are mostly not stated explicitly but result from the absence of data dependences. The availability of an explicit operator which expresses parallelism permits a source-to-source transformation which emphasizes the use of parallelism.

Contrary to the imperative style, our parallelism operator $(\|)$ does not connect possibly interfering computations but referentially transparent functions. Thus, the result is a list of the results of the parallel computations. To achieve the associativity of the operator $(\|)$, the functions take the same input and their result is wrapped into a list. $(\|)$ is *not* commutative!

Definition 4.18 (Parallel composition $(\|)$)

$(\|) \in (\alpha{\rightarrow}[\beta]) \rightarrow (\alpha{\rightarrow}[\beta]) \rightarrow (\alpha{\rightarrow}[\beta])$
$(f \; \| \; g) \; x = f \; x \mathbin{+\!\!+} g \; x$

□

Parallel composition has the following properties:

Law 4.1 (Neutral element of $(\|)$)
$e_{\|} = \mathsf{const}\,[\,]$
□

Law 4.2 (Associativity of $(\|)$)
$(f \parallel g) \parallel h = f \parallel (g \parallel h)$
□

Law 4.3 (Rightward distribution of $(\circ)$ over $(\|)$)
$(f \parallel g) \circ h = (f \circ h) \parallel (g \circ h)$
□

Law 4.4 (Leftward pseudo-distribution of $(\circ)$ over $(\|)$)
$\mathsf{map}\; h \;\circ\; (f \parallel g) = (\mathsf{map}\; h \;\circ\; f) \;\parallel\; (\mathsf{map}\; h \;\circ\; g)$
□

Law 4.5 (Multiple $(\|)$ expressed by map)
$\|_{i=0}^{n-1} f_i \;\; = \;\; \lambda x \rightarrow \mathsf{map}\; (\lambda f \rightarrow f \; x) \; [\, f_0, ..., f_{n-1} \,]$
□

We use the operator $(\widehat{\|})$ to work on computation domains. As before, let us consider two computation domains $\mathfrak{CD}_\phi$ for f and $\mathfrak{CD}_\psi$ for g. Then the resulting domain $\mathfrak{CD}_\theta = \mathfrak{CD}_\phi \mathbin{\widehat{\|}} \mathfrak{CD}_\psi$ has the following properties.

1. It computes the function $f \parallel g$.
2. $\Delta_\theta = \Delta_\phi$.
3. The extent in dimension 1 is the sum of the maximum extents of $\mathfrak{IS}_\psi$ and $\mathfrak{IS}_\phi$ in dimension 1. $\mathfrak{CD}_\psi$ is shifted along dimension 1 such that both index spaces do not overlap.

4. The bounds in dimension 0 are constructed using the maximum extent of both plus 1. The additional coordinate is used as an interface, so the input required by both parts is taken once only. For the index space with the smaller extent in dimension 0, identity nodes are inserted, which ensure that all output points have the same coordinate in dimension 0.

5. In dimensions greater 1, the bound expressions are given by a case distinction depending on the coordinate in dimension 1 which specifies membership in either $\mathfrak{CD}_\phi$ or $\mathfrak{CD}_\psi$.

4.3.8.3 Nested composition

If a skeleton f calls a skeleton g, then we name the computation domain $\mathfrak{CD}_\theta$ for the instantiation of domain $\mathfrak{CD}_\phi$ (for f) with domain $\mathfrak{CD}_\psi$ (for g) a nested composition. The properties of the composition depend strongly on the function f. Sect. 4.5 presents the consequences of different choices of f on the computation domain, and Tab. 4.1 tabulates their effect on time and space.

We can at least assume that the relation $\Delta_\theta \leq \Delta_\phi + \Delta_\psi$ holds, because the instantiation can cause expansion of points in the outer domain in at most Δ_ψ dimensions. Note that, in the instantiation, a refinement of the input and output points may occur, which requires also a refined environment. If the environment is not to be changed, computation domains which perform adaptations have to be inserted.

4.4 Size inference

Size inference derives symbolic information about the result returned by a function from structural variables which represent the symbolic information of its arguments. Thus, it is a prerequisite for an automatic optimization.

Our goal is to improve compiler technology and to determine automatically a space-time mapping at compile time, if possible. We are interested in the following symbolic information about a function:

1. the size of the result – in the case of nested lists, a compact description of all levels,

2. the number of operations performed in a particular situation,

3. the length of the longest path in the DAG of a function (Sect. 5.3.10) if all calls are expanded,

4. the number of steps for a given number of processors, if the communication cost is disregarded – this can be estimated from the number of operations and the path length, using Brent's theorem (Quinn, 1994).

Our first approach to size inference considered an introduction of sizes in the type and a definition of type inference rules with constraints. However, this makes type inference undecidable which we could not accept. Fortunately, we encountered a three step approach by Hughes et al. (1996). Quoting them, the steps are:

- **Hindley-Milner Inference.** We check that the program is type correct in terms of ordinary types.
- **Size Inference.** We typecheck the program with our sized type system under the assumption that all type signatures are correct.
- **Constraint solving.** We solve all the constraints and verify that our inferred types match the type signatures.

We have adapted this idea, since it permits us to perform a successful type and size inference and leaves decidability problems to mathematical tools available for constraint solving. However, our calculus is restricted to finite lists instead of arbitrary data types and, thus, is much simpler.

In a previous publication of ours (Herrmann and Lengauer, 1998), the size inference computes an abstract version of a function, which takes the same number of arguments and has the same structure as the original function, but the operations it performs are abstract counterparts of the original operations. However, the rules presented there are too pessimistic because they can destroy useful structural information; an extreme example is the rule for concat. The reason is that structural information can be subject to complex computations.

The concept of size inference in this thesis is based on partial evaluation at compile time, using the single list constructor Γ and the rules presented in Figs. 4.4–4.6. With this representation, the entire structural information is preserved and can be extracted from every part of the $\mathcal{HDC}$ program. Whether the information is really useful depends then only on two aspects:

1. Is the information free of input data values? In this case, all free variables can be turned into structural parameters, like the problem and data division degrees in the skeleton dcF.
2. Can the expressions be simplified to a form in which the evaluation at run time is fast? E.g., for the **concat-Γ** rule in Fig. 4.6, this either requires simple expressions, such that the sum and the maximum computation disappear, or that the computation of sum and maximum can be performed quickly at run time, e.g., by a parallel scan and red. It is unlikely that, in the latter case, statically parallelized code will still be faster than dynamically parallelized code if a lot of auxiliary information has to be computed at run time. Simplification is strongly limited due to problems of computability. In general, the elimination of recursion in size expressions is not possible. We present a heuristic approach by an example in Sect. 4.4.2.

4.4.1 A single-constructor representation for lists

We have a special interest in applying the Γ-calculus to lists, for the following reasons:

intr-Γ $\{xs \text{ is a list}\}$	$\dfrac{xs}{\Gamma\ (xs\,!!)\ (\#xs)}$
elim-Γ.0	$\dfrac{\#\ (\Gamma\ \_\ n)}{n}$
elim-Γ.1	$\dfrac{\Gamma\ f\ n\ !!\ i}{\underline{\texttt{if}}\ 0 \le i < n\ \underline{\texttt{then}}\ f\ i\ \underline{\texttt{else}}\ \bot}$

Figure 4.4: Introduction/Elimination of Γ

null-Γ	$\dfrac{\mathsf{null}\ (\Gamma\ \_\ n)}{n = 0}$
nil-Γ	$\dfrac{[\,]}{\Gamma\ (\mathsf{const}\ \bot)\ 0}$
cons-Γ $\{\underline{\text{fresh}}\ i\}$	$\dfrac{x\ :\ \Gamma\ f\ n}{\Gamma\ (\lambda i \to \underline{\texttt{if}}\ i{=}0\ \underline{\texttt{then}}\ x\ \underline{\texttt{else}}\ f\ (i{-}1))\ (n{+}1)}$
head-Γ	$\dfrac{\mathsf{head}\ (\Gamma\ f\ n)}{\underline{\texttt{if}}\ n{>}0\ \underline{\texttt{then}}\ f\ 0\ \underline{\texttt{else}}\ \bot}$
tail-Γ	$\dfrac{\mathsf{tail}\ (\Gamma\ f\ n)}{\underline{\texttt{if}}\ n{>}0\ \underline{\texttt{then}}\ \Gamma\ (f \circ (+1))\ (n{-}1)\ \underline{\texttt{else}}\ \bot}$

Figure 4.5: Lists defined with Γ

1. The list is the aggregate structure we have chosen as a carrier for possibly distributed data. The transformations generate a form that simplifies access to the lengths of the list and its sublists in terms of structural parameters. The space-time mapping can make use of these lengths.

2. In functional programming, a list often appears as an intermediate data structure, which can cause huge overhead if it is large but just a few elements of it are actually required. Some rules can eliminate intermediate data structures, e.g., $[a, b, c]\,!!\,1$ can be replaced by b. A related optimization technique is *deforestation* (Wadler, 1988; Peyton Jones and Santos, 1998).

Fig. 4.4 contains introduction and elimination rules for Γ. The first argument of Γ is the function that maps an index to an element, the second is the size of the list. **intr-Γ** introduces Γ. There are two rules for elimination: **elim-Γ.0**

take-Γ

$$\frac{\mathsf{take}\ k\ (\Gamma\ f\ n)}{\Gamma\ f\ ((k{\downarrow}n){\uparrow}0)}$$

drop-Γ

$$\frac{\mathsf{drop}\ k\ (\Gamma\ f\ n)}{\Gamma\ (f \circ (+((k{\downarrow}n){\uparrow}0)))\ (n-((k{\downarrow}n){\uparrow}0))}$$

map-Γ

$$\frac{\mathsf{map}\ g\ (\Gamma\ f\ n)}{\Gamma\ (g \circ f)\ n}$$

red-Γ
$\{\underline{\text{fresh}}\ i\}$

$$\frac{\mathsf{red}\ \oplus\ e\ (\Gamma\ f\ n)}{\underline{\texttt{if}}\ n=0\ \underline{\texttt{then}}\ e\ \underline{\texttt{else}}\ \bigoplus_{i=0}^{n-1}(f\ i)}$$

scan-Γ
$\{\underline{\text{fresh}}\ i,j\}$

$$\frac{\mathsf{scan}\ \oplus\ e\ (\Gamma\ f\ n)}{\Gamma\ (\lambda j \rightarrow \underline{\texttt{if}}\ j=0\ \underline{\texttt{then}}\ e\ \underline{\texttt{else}}\ \bigoplus_{i=0}^{j-1}(f\ i))\ (n+1)}$$

conditional-Γ

$$\frac{\underline{\texttt{if}}\ p\ \underline{\texttt{then}}\ \Gamma\ f\ m\ \underline{\texttt{else}}\ \Gamma\ g\ n}{\Gamma\ (\underline{\texttt{if}}\ p\ \underline{\texttt{then}}\ f\ \underline{\texttt{else}}\ g)\ (\underline{\texttt{if}}\ p\ \underline{\texttt{then}}\ m\ \underline{\texttt{else}}\ n)}$$

append-Γ
$\{\underline{\text{fresh}}\ i\}$

$$\frac{\Gamma\ f\ m\ {+\!\!+}\ \Gamma\ g\ n}{\Gamma\ (\lambda i \rightarrow \underline{\texttt{if}}\ i<m\ \underline{\texttt{then}}\ f\ i\ \underline{\texttt{else}}\ g\ (i-m))\ (m+n)}$$

concat-Γ
$\{\underline{\text{fresh}}\ k,p\}$

$$\frac{\mathsf{concat}\ (\Gamma\ (\lambda i \rightarrow \Gamma\ (\lambda j \rightarrow e)\ n)\ m)}{\begin{array}{c}\Gamma\ (\lambda k \rightarrow ((e[i := \mu\ k\ n])[j := k - \sum_{i=0}^{\mu\ k\ n-1} n]))\ (\sum_{i=0}^{m-1} n) \\ \underline{\texttt{where}}\ \mu\ k\ n\ =\ \uparrow\{\ p \in \mathbb{N} \mid \sum_{i=0}^{p-1} n \leq k\ \}\end{array}}$$

Figure 4.6: Combinators on Γ

delivers the length of the list defined by Γ and **elim-Γ.1** (just **elim-Γ** in (Herrmann et al., 1999b)) accesses a particular element. With these rules, lists can be defined formally without recursion.

Fig. 4.5 demonstrates that a single constructor (Γ), instead of [] and (:), is sufficient to build lists. This is done by expressing the predicate, the constructors and the selectors for lists by Γ-expressions. Here, we assume that a type of unbounded integers is available. In practice, a subrange of the integers, with which one can address every bit of the entire virtual memory of the parallel machine (64-bit integers at present), serves for a sufficient replacement of a plain list representation.

In Fig. 4.6, a set of rules is presented that specify the result of combinator applications to lists in Γ-form. The Γ-representation is useful for partial evaluation and optimization at compile time, as shown in Ex. 4.9. A compiler can either deliver an error message where ⊥ occurs, or eliminate run-time checks to achieve a higher performance if the program is proven correct.

Example 4.9 (Optimizing transformation with Γ)

$\mathsf{map}\ f\ [a\,..\,b]\ \texttt{!!}\ j$
$\quad = \boxed{\textbf{intr-}\Gamma}$
$\mathsf{map}\ f\ (\Gamma\ ([a\,..\,b]\texttt{!!})\ (\#[a\,..\,b]))\ \texttt{!!}\ j$
$\quad = \boxed{\text{properties of arithmetic sequence}}$
$\mathsf{map}\ f\ (\Gamma\ (a+)\ ((b-a+1)\uparrow 0))\ \texttt{!!}\ j$
$\quad = \boxed{\textbf{map-}\Gamma}$
$\Gamma\ (f \circ (a+))\ ((b-a+1)\uparrow 0)\ \texttt{!!}\ j$
$\quad = \boxed{\textbf{elim-}\Gamma\textbf{.1}}$
$\underline{\texttt{if}}\ 0 \le j \wedge j < ((b-a+1)\uparrow 0)\ \underline{\texttt{then}}\ (f \circ (a+))\ j\ \underline{\texttt{else}}\ \bot$
$\quad = \boxed{\text{definition } \circ}$
$\underline{\texttt{if}}\ 0 \le j \wedge j < ((b-a+1)\uparrow 0)\ \underline{\texttt{then}}\ f\ (a+j)\ \underline{\texttt{else}}\ \bot$
$\quad = \boxed{\text{arithmetic/logic}}$
$\underline{\texttt{if}}\ 0 \le j \wedge j < b-a+1\ \underline{\texttt{then}}\ f\ (a+j)\ \underline{\texttt{else}}\ \bot$
$\quad = \boxed{\text{original program is correct}}$
$f\ (a+j)$

□

4.4.2 Size inference in the presence of recursion

One has to distinguish between size inference of recursive and of non-recursive functions. Size inference of non-recursive functions can easily be done fully automatically with the rules of the Γ-calculus, following the syntactic structure of the function's body.

We recommend to stay away from user-defined recursive functions that are critical with respect to performance and try to use predefined skeletons instead. These skeletons can be delivered together with a size function, which has been supplied by the developer of the skeleton.

For a recursive function, the Γ-rules deliver a recursive size function. Syntactically simplified, it can be expressed in terms of algebraic, conditional recurrence equations, and given to a constraint solver. Unfortunately, in general, the success of constraint solving in the presence of non-linear equations and integer requirements on solutions is not guaranteed. This motivates the use of heuristics. Although some amount of expert knowledge is required (interpolation, induction and equational reasoning) such heuristics could be implemented to work automatically, without guarantee of success, of course. Here, we present a small example, inspired by a paper of Hughes et al. (1996).

The Γ-calculus not only computes the size of the result, but also gives a defining equation for every element of the result list. This is necessary to keep track of the sizes of inner structures by referring to their indices in the surrounding structures. A case, where this is actually required is the size inference for triangular matrix inversion in Herrmann and Lengauer (1998). (We present a triangular matrix in Γ-form in Sect. 4.5.3).

Example 4.10 (Size inference of reverse)
Consider the function reverse′, defined as follows:

$$\begin{array}{l}
\mathsf{reverse}' \in [\alpha] \rightarrow [\alpha] \\
\mathsf{reverse}'\ xs = \underline{\mathtt{if}}\ \mathsf{null}\ xs\ \ \underline{\mathtt{then}}\ [\,] \\
\qquad\qquad\qquad\qquad\qquad\ \underline{\mathtt{else}}\ \underline{\mathtt{let}}\ \ ys = \mathsf{reverse}'\ (\mathsf{tail}\ xs) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \underline{\mathtt{in}}\ \ \ ys +\!\!+ [\mathsf{head}\ xs]
\end{array}$$

In Γ-notation, the definition converts to:

$$\begin{array}{l}
\mathsf{reverse}''\ (\Gamma\ f\ m) \\
\ = \underline{\mathtt{if}}\ m{=}0\ \ \underline{\mathtt{then}}\ \ \Gamma\ (\mathsf{const}\ \bot)\ 0 \\
\qquad\qquad\quad \underline{\mathtt{else}}\ \underline{\mathtt{let}}\ \ \Gamma\ g\ n = \mathsf{reverse}''\ (\Gamma\ (f \circ (+1))\ (m{-}1)) \\
\qquad\qquad\qquad\quad\ \ \underline{\mathtt{in}}\ \ \ \Gamma\ (\lambda i \rightarrow \underline{\mathtt{if}}\ i{<}n\ \underline{\mathtt{then}}\ g\ i\ \underline{\mathtt{else}}\ f\ 0)\ (n{+}1)
\end{array}$$

The problem we encounter is the recursive call to reverse″. Dependence analysis shows that the second argument of Γ in the result does not depend on the first argument of Γ in the input of reverse″. Thus, we can first try to find a solution for the second component. We call the function on this component revsize.

$$\begin{array}{l}
\mathsf{revsize} \in \mathbb{N} \rightarrow \mathbb{N} \\
\mathsf{revsize}\ m = \underline{\mathtt{if}}\ m{=}0\ \ \underline{\mathtt{then}}\ 0 \\
\qquad\qquad\qquad\qquad \underline{\mathtt{else}}\ \underline{\mathtt{let}}\ \ n = \mathsf{revsize}\ (m{-}1) \\
\qquad\qquad\qquad\qquad\qquad\quad\ \underline{\mathtt{in}}\ \ \ n{+}1
\end{array}$$

The solution can be found easily: revsize $m = m$. Now, we can concentrate on the first component of Γ in the result of reverse″, say revfun, using the solution for the second component:

$$\begin{array}{l}
\mathsf{revfun} \in ((\mathbb{N}{\rightarrow}\alpha){\times}\mathbb{N}){\rightarrow}(\mathbb{N}{\rightarrow}\alpha) \\
\mathsf{revfun}\ (f, m) = \underline{\mathtt{if}}\ m{=}0\ \ \underline{\mathtt{then}}\ \ \mathsf{const}\ \bot \\
\qquad\qquad\qquad\qquad\quad \underline{\mathtt{else}}\ \underline{\mathtt{let}}\ \ g = \mathsf{revfun}\ (f \circ (+1), m{-}1) \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \underline{\mathtt{in}}\ \ \ \lambda i \rightarrow \underline{\mathtt{if}}\ i{<}m{-}1\ \underline{\mathtt{then}}\ g\ i\ \underline{\mathtt{else}}\ f\ 0
\end{array}$$

The simplicity with which f is modified for the recursive call encourages us to match the argument of f against an affine linear expression in the parameters m and i with coefficients a, b and c:

$$\mathsf{revfun}'\ (a, b, c)\ (f, m) = \underline{\mathtt{if}}\ m{=}0\ \underline{\mathtt{then}}\ \mathsf{const}\ \bot\ \underline{\mathtt{else}}\ \lambda i \rightarrow f\ \ (a{*}m{+}b{*}i{+}c)$$

We perform this match by a case distinction according to the conditional in revfun:

1. $i < m{-}1 \implies am + bi + c = a(m{-}1) + bi + c + 1 \implies a = 1$
2. $i = m{-}1 \implies am + b(m{-}1) + c = 0 \implies b = -a \wedge c = b$
   (by comparing coordinates in the vector space $(m, 1)$)

Under the assumption that the expression is really affine, the solution is:

$\mathsf{revfun}''\ (f, m) = \underline{\texttt{if}}\ m{=}0\ \underline{\texttt{then}}\ \mathsf{const}\ \bot\ \underline{\texttt{else}}\ \lambda i \to f(m{-}i{-}1)$

Now, we have to check by equational reasoning whether the assumption is valid. We do this by induction on m:

$\boxed{\text{(i) basic case:}}$

$\mathsf{revfun}\ (f, m)$
$\quad = \boxed{m = 0}$
$\mathsf{const}\ \bot$
$\quad = \boxed{m = 0}$
$\underline{\texttt{if}}\ m{=}0\ \underline{\texttt{then}}\ \mathsf{const}\ \bot\ \underline{\texttt{else}}\ \lambda i \to f\ (m{-}i{-}1)$

$\boxed{\text{(ii) inductive case:}}$

$\mathsf{revfun}\ (f, m)$
$\quad = \boxed{m > 0}$
$\underline{\texttt{let}}\ g = \mathsf{revfun}\ (f \circ (+1),\ m{-}1)$
$\underline{\texttt{in}}\ \lambda i \to \underline{\texttt{if}}\ i{<}m{-}1\ \underline{\texttt{then}}\ g\ i\ \underline{\texttt{else}}\ f\ 0$
$\quad = \boxed{\text{induction hypothesis, fresh } j}$
$\underline{\texttt{let}}\ g = \lambda j \to (f \circ (+1))((m{-}1){-}j{-}1)$
$\underline{\texttt{in}}\ \lambda i \to \underline{\texttt{if}}\ i{<}m{-}1\ \underline{\texttt{then}}\ g\ i\ \underline{\texttt{else}}\ f\ 0$
$\quad = \boxed{\text{substitution of } g}$
$\lambda i \to \underline{\texttt{if}}\ i{<}m{-}1\ \underline{\texttt{then}}\ (f \circ (+1))((m{-}1){-}i{-}1)\ \underline{\texttt{else}}\ f\ 0$
$\quad = \boxed{\text{definition } \circ \text{ and arithmetic}}$
$\lambda i \to \underline{\texttt{if}}\ i{<}m{-}1\ \underline{\texttt{then}}\ f\ (m{-}i{-}1)\ \underline{\texttt{else}}\ f\ (m{-}(m{-}1){-}1)$
$\quad = \boxed{\text{constraint } 0 \le i < m}$
$\lambda i \to \underline{\texttt{if}}\ i{<}m{-}1\ \underline{\texttt{then}}\ f\ (m{-}i{-}1)\ \underline{\texttt{else}}\ f\ (m{-}i{-}1)$
$\quad = \boxed{\text{conditional}}$
$\lambda i \to f\ (m{-}i{-}1)$
$\quad = \boxed{m > 0}$
$\underline{\texttt{if}}\ m{=}0\ \underline{\texttt{then}}\ \mathsf{const}\ \bot\ \underline{\texttt{else}}\ \lambda i \to f\ (m{-}i{-}1)$

The result of our size inference is the following representation $\mathsf{reverse}'''$ in the Γ-calculus:

$\mathsf{reverse}'''\ (\Gamma\ f\ m) = \Gamma\ (\lambda i \to f\ (m{-}i{-}1))\ m$

In list comprehension syntax, it is:

$\mathsf{revresult} \in [\alpha] \to [\alpha]$
$\mathsf{revresult}\ xs = \underline{\texttt{let}}\ \ m = \#xs$
$\qquad\qquad\qquad \underline{\texttt{in}}\ \ [\ xs\,\mathbf{!!}(m{-}i{-}1) \mid i{\leftarrow}[0\,..\,m{-}1]\ ]$

□

4.5 The computation domain of skeletons

This section describes how skeletons can be assigned a computation domain, to make the operations accessible for a space-time mapping.

4.5.1 Motivation for the restriction to skeletons

In the ideal case, the computation domain of the entire program is the composition of the domains of the program functions. The computational structure is given by a set of DAGs (one for each function), in which nodes representing function calls refer to the DAG of the callee. Unfortunately, the combination of recursion and conditionals depending on run-time values makes this construction too complex and a parallelization based on it inefficient. Consider the case that some run-time value determines the choice of one of two algorithms, in order to solve a problem. Let both algorithms be expressed in terms of different skeletons. Then, it is obviously better to choose the appropriate parallel implementation at run time than to provide a complicated structure that combines both algorithms.

As a consequence, we define computation domains not for the entire program but just for appropriate skeletons or for combinations of skeletons. In fact, a global static parallelization of the program is not really necessary. It is sufficient to identify the parts of the computation which require large amount of work, to parallelize them well and to combine these parallelized parts to form the entire target program.

4.5.2 How to construct the computation domain

The construction of a computation domain of a skeleton does not follow uniquely from its parts, but it involves manual choices with respect to a set of premises. One kind of premise is that the shape of the output data of a customizing function depends somehow regularly on the shape of the input data; the precise kind of regularity must be specified for the particular construction. As mentioned before, we distinguish between structural parameters, customizing functions and input data.

Although skeletons are defined as higher-order functions in our language $\mathcal{HDC}$, they are implemented by first-order functions, using an encoding of closures by data structures, e.g., for the customizing functions. Therefore, it is sufficient to discuss first-order programs here. Contrary to the input data, closures for customizing functions are not distributed explicitly. It is assumed that customizing functions can be applied at every point in the index space.

We have to distinguish between the definition of a skeleton in Haskell, which may be recursive, and its implementation. The implementation can be expressed in terms of sequential and parallel combinators and, thus, its computation domain be of sufficient regularity for an efficient parallelization. Then, the use of skeletons in the program, although appearing to be recursive, does not incur recursive calls in the implementation.

The way in which transformations are performed is inspired by geometric considerations, but the correctness of the transformation is justified solely by laws of equational reasoning without reference to geometric properties.

4.5.3 The Γ skeleton

The Γ skeleton takes a customizing function f and an integer n and delivers the list $[f\ 0, ..., f\ (n-1)]$. Γ may be eliminated if succeeding computations require only single elements of the list, but let us here concentrate on the case that the entire list is actually to be computed. To develop a computation domain, we have to consider properties of the arguments f and n. Let us assume that f does not require much space, i.e., f does not carry along a large closure. Thus, it appropriate not to model memory requirements for f in the computation domain. Also, we assume n to be instantiated by an expression in structural parameters.

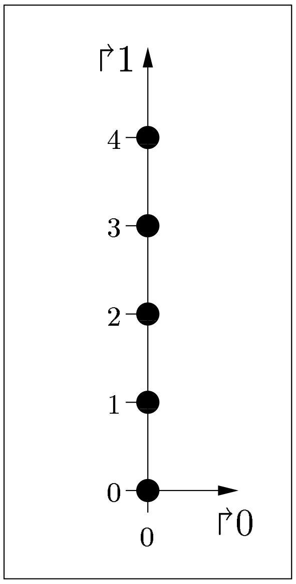


Figure 4.7: Index space for $(\Gamma\ f\ 5)$

Under these assumptions, the set of input points of the computation domain is empty, because there is no input data that depends on run-time values. Dimension 0 of the index space has an extension of 1, as can be seen in Fig. 4.7. This single slice accommodates the computations of all elements, which can be done in parallel, and is identical with the output points of the domain. Dimension 1 has the extent n; each coordinate corresponds to one element of the result list. The existence of other dimensions depends on the index space of the list elements. If Γ is nested, dimension 2 exists; its extent can depend on the coordinate values of dimension 1. An example is a triangular matrix with n rows, in which the value at position $[i, j]$ is $i+j$:

$$\Gamma\ \ (\lambda i \rightarrow \Gamma\ (\lambda j \rightarrow i+j)\ (i+1))\ \ n$$

The corresponding computation domain has an extent of n in dimension 1 and of $\Rsh 1 + 1$ in dimension 2.

4.5.4 The map skeleton

The map skeleton takes a customizing function f and applies it to every element of a list, say xs. If we expressed map in terms of Γ, the construction of the computation domain for Γ would obscure information about the distribution of xs and involve overhead due to closure decoding and reference counting. Thus, we take xs to be input data instead and the length of xs as additional compile-time information invisible to the user.

The computation domain for map differs from the one for Γ in an additional coordinate in dimension 0 which contains the input points, one for each element of the list. The extent of dimension 1, for the input points ($\vec{0} = 0$) and for the output points ($\vec{0} = 1$), is n. Like for Γ, the existence of further dimensions depends on the knowledge of the structure of the list elements. map has the property that the only dependence vector (see the arrows in Fig. 4.8) has the length 0 in dimension 1, which permits an independent application of function f to all elements of the list. If map is nested, dimension 2 exists and the only dependence vector has length 0 also in this dimension.

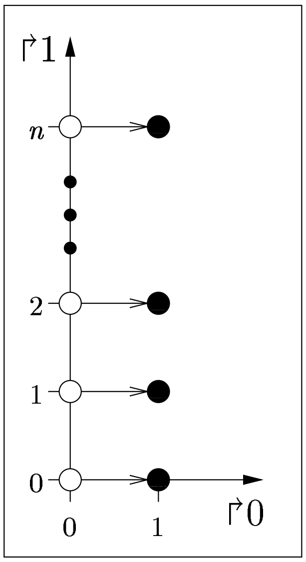


Figure 4.8: Index space for (map f)

4.5.5 Parallel composition ($\|$)

Revisiting parallel composition from Sect. 4.3.8.2, the output list of both operands is distributed across dimension 1. Since the computation domains are appended in dimension 1 too, the associativity of the parallel composition of index spaces corresponds to the associativity of the operator ($\|$) in programs.

4.5.6 Sequential composition ($\circ$)

According to Sect. 4.3.8.1, a composition $g \circ f$ is constructed such that the output points of f's domain have the same distribution as the input points of g's domain. Then the index spaces for f and g can be combined as depicted in Fig. 4.9. (Violard (1994) applied a similar technique to data fields.) We shift the index space of g along dimension 0 such that the last position of f coincides with the first position of g.

For a basis in the development of computation domains, one should at least agree on a set of common layouts for data structures. Assume, e.g., that lists are spread across dimension 1 consecutively, starting at position 0. Then the computation domain of (map g $\circ$ map f) equals the computation domain of map ($g \circ f$), using the construction for map above.

4.5.7 The for skeleton

In the polytope model, each level of a loop nest corresponds to a dimension of the index space of the computation domain. The iterations of the loop nest are

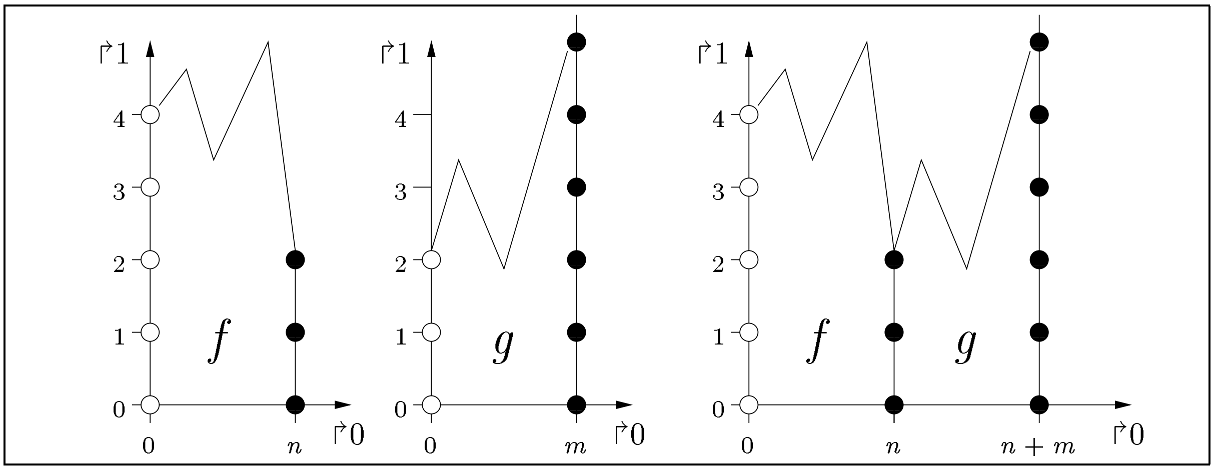


Figure 4.9: Composition of domains, f, g, $g \circ f$

scanning the index space by enumerating its points (Ancourt and Irigoin, 1991; Griebl and Lengauer, 1994).

In the functional setting of this thesis, the for skeleton takes the number n of iterations and a function f which represents the loop body. To mimic the nesting of loops in the polytope model, we construct a computation domain for (for n f) as follows. Dimension 0 has an extent of n+2: one position for the input, one position for each iteration and one position for an identity operation at the end to fit the output format. At coordinate value $k+1$ in dimension 0 (iteration k of for), a copy of the computation domain of (f k) is laid out orthogonally, while the indexing in the subspace for f is changed: a dimension referred to by ↱i before is now referenced by ↱$(i+1)$. This change is formally defined by a substitution of references. If two for loops are nested, the outer iterates across dimension 0 and the inner across dimension 1. One might view dimension 1 as a refinement of dimension 0.

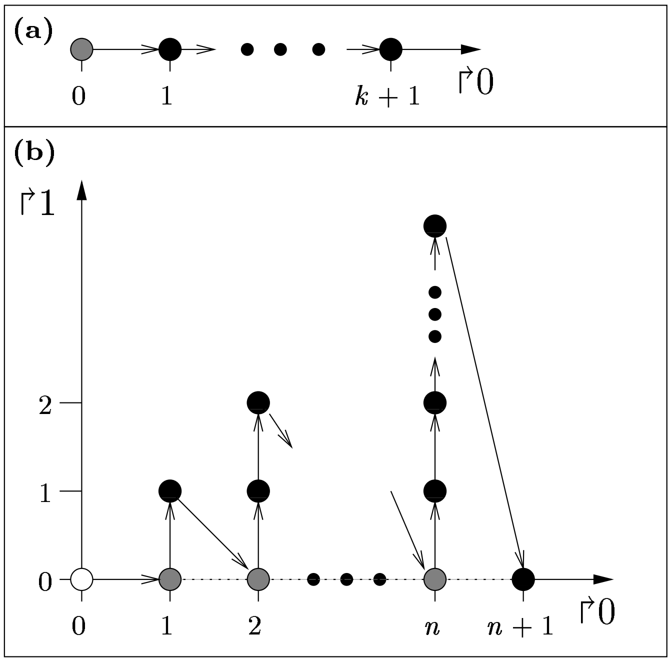


Figure 4.10: Index space for (for n f) **(b)** under the assumption of space **(a)** for (f k)

We encounter problems due to the data layout. Take the following example. A loop nest produces a list which is subsequently to be consumed by an application of map. Dimension 1 of the nest is reserved for the inner loop and, thus, the list delivered is distributed across dimension 2. On the other hand, the application of map expects a distribution across dimension 1. In this case, a computation domain has to be inserted which performs a reshaping of the data. Communication is not required here but the space-time mapping has to deal with a change from two-dimensional to one-dimensional time.

In the iterative versions of the $\mathcal{DC}$ skeletons, the following variants of the for loop are used.

Definition 4.19 (Variants of the for skeleton)
$\mathsf{for} \in \mathbb{N} \to (\alpha{\to}\alpha) \to \alpha \to \alpha$
$\mathsf{for}\ n\ f = \underline{\mathtt{if}}\ n{=}0\ \underline{\mathtt{then}}\ \mathsf{id}\ \underline{\mathtt{else}}\ f \circ \mathsf{for}\ (n{-}1)\ f$

$\mathsf{forup},\ \mathsf{fordown} \in \mathbb{N} \to (\mathbb{N}{\to}\alpha{\to}\alpha) \to \alpha \to \alpha$
$\mathsf{forup}\ \ n\ f\ x = \mathsf{snd}\ (\mathsf{for}\ n\ (\lambda\ (i,x) \to (i{+}1, f\ i\ x))\ (0,x))$
$\mathsf{fordown}\ n\ f\ x = \mathsf{snd}\ (\mathsf{for}\ n\ (\lambda\ (i,x) \to (i{-}1, f\ i\ x))\ (n{-}1,x))$

$\mathsf{scanforup} \in \mathbb{N} \to (\mathbb{N}{\to}\alpha{\to}\alpha) \to \alpha \to [\alpha]$
$\mathsf{scanforup}\ n\ f\ x = [\ \mathsf{forup}\ j\ f\ x \mid j{\leftarrow}[0\,..\,n]\ ]$
□

The for skeleton just applies a function several times. The other functions provide an iteration counter as an additional, first argument to f. forup counts this argument up, while fordown counts it down. scanforup collects all intermediate results of forup.

4.5.8 The while skeleton

while loops differ from for loops mainly in the fact that the number of iterations is unknown in advance. Griebl and Lengauer (1994) added an auxiliary loop variable to while loops to refer to its iterations. We can concentrate the uncertainty of while in a compile-time function *iterations* with codomain $\mathbb{N} \cup \{\infty\}$, which delivers the number of iterations if given the predicate, body and input of the while loop. Thus, the computation domain of a while loop differs from the one of a for loop only by the occurrence of this function, which we carry along without evaluation. We have two alternatives for the implementation of the while skeleton:

1. dynamic space-time mapping and a black-box implementation which preserves the potential for a static space-time mapping of each individual iteration,

2. combining the implementation of the while loop with the implementation of its body, e.g., by a signaling schema as proposed by Griebl and Lengauer (1995).

Definition 4.20 (The while skeleton)
$\mathsf{while} \in (\alpha{\to}\mathbb{B}) \to (\alpha{\to}\alpha) \to \alpha \to \alpha$
$\mathsf{while}\ p\ f\ x = \underline{\mathtt{if}}\ p\ x\ \underline{\mathtt{then}}\ \mathsf{while}\ p\ f\ (f\ x)\ \underline{\mathtt{else}}\ x$
□

skeleton call	*time*	*space*
$(\Gamma\ f\ n, \_)$	$\uparrow_{i=0}^{n-1} time(f,i) \uparrow 1$	$\sum_{i=0}^{n-1} space(f,i)$
$(f \ \Vert\ g,\ x)$	$(time(f,x) \uparrow time(g,x)) + 1$	$space(f,x)\ +\ space(g,x)$
$(\mathsf{map}\ f,\ xs)$	$\uparrow_{i=0}^{\#xs-1} time(f, xs\,!!\,i) \uparrow 2$	$\sum_{i=0}^{\#xs-1} space(f, xs\,!!\,i)$
$(g\ \circ\ f,\ x)$	$time(f,x) + time(g, f\ x) - 1$	$space(f,x) \uparrow space(g, f\ x)$
$(\mathsf{for}\ n\ f,\ x)$	$\sum_{i=0}^{n-1} time(f, \mathsf{for}\ i\ f\ x) + 2$	$\uparrow_{i=0}^{n-1} space(f, \mathsf{for}\ i\ f\ x) \uparrow 0$
$(\mathsf{while}\ p\ f, x)$	$time(\mathsf{for}\ n\ f, x)$ `where` $n = iterations(p,f,x)$	$space(\mathsf{for}\ n\ f, x)$ `where` $n = iterations(p,f,x)$

Table 4.1: Recursive *time*/*space* formulas

4.5.9 Time and space consumption

Of course, the information about time and space consumption requires knowledge of the space-time mapping. In our setting, computation domains can be composed which differ widely in shape. Thus, the space-time mapping is very likely to be defined piecewise and its calculation is an expensive part of the compilation process. Therefore, it is useful to estimate the time and space consumption in advance, assuming a default space-time mapping. Depending on the actual mapping, this estimation may differ extremely from the time and space consumption of the final implementation.

Time and space expressions for skeletons are recursive; the basic case of the recursion depends on the chosen resolution. The metric is given in units. Each primitive operation must be given a number of units of time and space. The amount of space depends on the size of the input and can be large even for simple functions, e.g., for the identity on lists. We assume the following mapping: dimension 0 is mapped to time, all other dimensions are mapped to space. For the for and the while skeleton, we make the exception that a dimension, which was 0 before being renamed (Sect. 4.5.7), is mapped to time too.

The time and space functions are presented in Tab. 4.1. They take a pair (f,x) as input. f is the skeleton instantiation and x is the input data. Abstract information about f and x can be used to express time or space more precisely, exploiting facts like $time(p, q\ x) = 2 \cdot time(p,x)$ for a particular combination of p and q. Obviously, the size inference of Sect. 4.4 can help in the simplification of the expressions, e.g., by replacing $\#xs$ by an expression in structural parameters.

4.6 Transforming $\mathcal{DC}$ recursions into iterations

This section describes our transformation of $\mathcal{DC}$ recursion into iteration, which enables the transition from a computation domain of unbounded dimensionality to a computation domain of fixed dimensionality. The results are used for the derivation of the loop-based versions of the different $\mathcal{DC}$ skeletons in Sect. 4.7. As we discuss in Sect. 2.1, the transformation of recursion into iteration is not new. Following the motives stated in Sect. 2.2, we prefer to develop our own set of derivations in Haskell.

In Sect. 4.6.1, we discuss how the call tree model of Sect. 3.3 can be employed for the transformation. Sect. 4.6.2 is about the elimination of linear recursion as a prerequisite. In Sect. 4.6.3, we apply an encoding of the call tree which permits a separation of the $\mathcal{DC}$ recursion into a vertical component (along the levels of the call tree) and a horizontal component (across a particular level). The recursion in both components is linear. In our previous work, this idea was expressed by a loop program (Herrmann and Lengauer, 1995), later developed for a skeleton `dc3` related to `dcC` (Herrmann and Lengauer, 1997a/1999) and is now derived for the general $\mathcal{DC}$ class `dcA`. The result of Sect. 4.6.2 can be applied to express the outer vertical component by sequential loops. The inner horizontal component can be expressed by a parallel loop.

4.6.1 Use of the call tree model

Obviously, call graphs provide a natural basis for the construction of computation domains. For $\mathcal{DC}$, the call graph consists of the call tree with its mirror image. Thus, we concentrate on the call tree to keep it simple. In Sect. 3.3, we have seen that the initial problem is located at the root and that the subproblems of a node are assigned to the node's subtrees. The leaves of the tree correspond either to the basic cases or to cases in which the number of subproblems is 0.

Our idea is to traverse the call tree level by level, starting from the root, until a level is reached which does not contain any further node. Each level of the call tree is itself traversed from left to right. Thus, every node of the call tree is visited exactly once. At each node, the predicate *istrivial* is checked. If it is not satisfied, the divide function is called and nodes which represent the subproblems are generated and placed at the next level of the call tree. Otherwise nothing happens with the node in this phase. After this process, the levels of the call tree are traversed in the opposite order, finishing with the level of the root. Again, each level is traversed from left to right. If the predicate is satisfied, the basic function is applied, otherwise the combine function, taking the subproblem solutions located at the children at the level below.

Intuitively, it should be clear that this procedure will work, but the problem of choosing an appropriate representation of the call tree remains. The standard tree representation with constructors for branch nodes and leaves is not appropriate, since it does not easily facilitate the traversal of a particular level of the call tree. It is obvious that the entire tree and not just a single level must be stored, because the application of the combine function requires the availability of the input data. Since the $\mathcal{HDC}$ compiler provides a representation of lists that enables independent access to its elements, a promising approach is to represent each level of the tree by a list. Each element of this list is itself a list of nodes, i.e., of those that share the same parent. This way to maintain the structure makes it possible to combine the solution instances that belong together. If no subproblem instance exists because the predicate is satisfied or as a result of the divide function, this list is empty.

In previous work (Herrmann and Lengauer, 1997a), we began with the elimination of recursion from the skeleton dc2, which is a simple predecessor of dcB that does not use input data in the combine function. For the loop implementation, we even restricted ourselves to dc3 (the predecessor of dcC), i.e., fixed the number of subproblems.

Now, these restrictions only serve to impose a static space-time mapping; they are no longer required for a loop program implementation.

4.6.2 Elimination of linear recursion

Both the vertical and the horizontal component of the call tree traversal can be expressed by linearly recursive functions. Additionally, our encoding of the call tree stores the input data of the linearly recursive function. Therefore, it is possible to employ a simple, restricted form of linear recursion which does not provide the input data after the recursive call. In this section, we concentrate on a transformation of this restricted form of linear recursion to a composition of iterative combinators, which have a correspondence to loop templates in the imperative setting.

Let us define our concepts of *restricted linear recursion* and *double linear iteration* and then state their semantic equivalence. Our notion of linear recursion is polymorphic. It can be instantiated to $\mathcal{DC}$ recursion by supplying customizing functions, which are defined on an appropriate encoding of the levels of the call tree.

Definition 4.21 (Restricted linear recursion)

$\mathsf{linrec} \in (\alpha\to\mathbb{B}) \to (\alpha\to\beta) \to (\alpha\to\alpha) \to (\beta\to\beta) \to \alpha\to \beta$
$\mathsf{linrec}\ p\ b\ d\ c = r$
  $\underline{\mathtt{where}}\ r\ x = \underline{\mathtt{if}}\ p\ x\ \ \underline{\mathtt{then}}\ b\ x$
                  $\underline{\mathtt{else}}\ c\ (r\ (d\ x))$

□

Definition 4.22 (Double linear iteration)

$\mathsf{liniter} \in (\alpha\to\mathbb{B}) \to (\alpha\to\beta) \to (\alpha\to\alpha) \to (\beta\to\beta) \to \alpha\to \beta$
$\mathsf{liniter}\ p\ b\ d\ c\ x = \ \underline{\mathtt{let}}\ (q,n) = \mathsf{while}\ (\mathsf{not} \circ p \circ \mathsf{fst})\ (\lambda\ (q,n)\to(d\ q, n+1))\ (x,0)$
            $\underline{\mathtt{in}}\ \ \mathsf{for}\ n\ c\ (b\ q)$

□

The computation domain of double linear iteration can be specified by a composition of the domains of while and for, parameterized in the implicit structural parameter n. The while/for loop enumerates the levels of the $\mathcal{DC}$ call tree downwards/upwards while dividing/combining problem instances.

Theorem 4.6 (linrec=liniter)
Restricted linear recursion is equivalent to double linear iteration.

Proof by induction on the program structure of linrec.

$\boxed{\text{(i) basic case:}}$

linrec $p\ b\ d\ c\ x$
$= \boxed{p\ x = \mathsf{True}}$
$b\ x$
$= \boxed{\text{definition for}}$
for $0\ c\ (b\ x)$
$= \boxed{\texttt{let}\ \text{substitution}}$
<u>let</u> $(q, n) = (x, 0)$ <u>in</u> for $n\ c\ (b\ q)$
$= \boxed{p\ x = \mathsf{True}}$
<u>let</u> (q, n) = while (not $\circ\ p\ \circ$ fst) $(\lambda\ (q, n) \to (d\ q, n{+}1))\ (x, 0)$
<u>in</u> for $n\ c\ (b\ q)$
$= \boxed{\text{folding liniter}}$
liniter $p\ b\ d\ c\ x$

$\boxed{\text{(ii) inductive case:}}$

linrec $p\ b\ d\ c\ x$
$= \boxed{p\ x = \mathsf{False}}$
c (linrec $p\ b\ d\ c\ (d\ x)$)
$= \boxed{\text{induction hypothesis}}$
c (liniter $p\ b\ d\ c\ (d\ x)$)
$= \boxed{\text{unfolding liniter}}$
<u>let</u> (q, n) = while (not $\circ\ p\ \circ$ fst) $(\lambda\ (q, n) \to (d\ q, n{+}1))\ (d\ x, 0)$
<u>in</u> c (for $n\ c\ (b\ q)$)
$= \boxed{\text{definition for}}$
<u>let</u> (q, n) = while (not $\circ\ p\ \circ$ fst) $(\lambda\ (q, n) \to (d\ q, n{+}1))\ (d\ x, 0)$
<u>in</u> for $(n{+}1)\ c\ (b\ q)$
$= \boxed{\text{index shift}}$
<u>let</u> (q, n) = while (not $\circ\ p\ \circ$ fst) $(\lambda\ (q, n) \to (d\ q, n{+}1))\ (d\ x, 1)$
<u>in</u> for $n\ c\ (b\ q)$
$= \boxed{p\ x = \mathsf{False}}$
<u>let</u> (q, n) = while (not $\circ\ p\ \circ$ fst) $(\lambda\ (q, n) \to (d\ q, n{+}1))\ (x, 0)$
<u>in</u> for $n\ c\ (b\ q)$
$= \boxed{\text{folding liniter}}$
liniter $p\ b\ d\ c\ x$

□

4.6.3 Expressing $\mathcal{DC}$ recursion by linear recursion

Next, we instantiate linrec to implement the dcA skeleton, using the representation of the call tree as described above. We generalize dcA to a form mdcA that applies dcA to all nodes at a particular level of the call tree and preserves the input data of the levels above. Lemma 4.9 (mdcA/linrec) states that mdcA is equivalent to mdcAlinrec which is an instantiation of linrec.

The function unconcat used in mdcAlinrec inverts applications of concat, given a list of the lengths of the concatenated sublists. Here, we give a tail-recursive definition of unconcat which is especially useful for the proof.

Definition 4.23 (unconcat)
unconcat $\in [\mathbb{N}] \to [\alpha] \to [[\alpha]]$
unconcat [ ] _ = [ ]
unconcat $(n{:}ns)$ xs = take n xs : unconcat ns (drop n xs)
□

The following law states the relation between concat and unconcat, proven in App. A.1.

Law 4.7 (unconcat/concat)
unconcat (map # xss) (concat xss) = xss
□

We can replace the recursive form of unconcat by a form which is especially convenient for a parallelization, using prefixsum, i.e., scan $(+)$ 0.

Definition 4.24 (unconcatprefixsum)
unconcatprefixsum ns xs =
 let bs = prefixsum ns
 in $[\,[\,xs\,!!\,k \mid k{\leftarrow}[bs\,!!\,i\,..\,bs\,!!\,(i{+}1){-}1]\,] \mid i{\leftarrow}[0\,..\,\#bs\,-2]]$
□

Law 4.8 (unconcat/prefixsum)
unconcat $=$ unconcatprefixsum
The proof can be found in App. A.2. □

The function mdcA — map of dcA — applies dcA to every element at a particular level of the call tree.

Definition 4.25 (mdcA)
mdcA $\in (\alpha{\to}\mathbb{B}){\to}(\alpha{\to}\beta){\to}(\alpha{\to}[\alpha]){\to}(\alpha{\to}[\beta]{\to}\beta){\to}[[[\alpha]]]{\to}([\beta]{\times}[[[\alpha]]])$
mdcA tr ba di co $(xs{:}xss)$ = (map (dcA tr ba di co) (concat xs), xss)
□

The following lemma provides insight into the correspondence between $\mathcal{DC}$ recursion and linear recursion and constitutes the basis for Theorem 4.11. Its idea is to traverse the levels of the $\mathcal{DC}$ tree by linear recursion. Therefore, mdcA incorporates two extensions of dcA: (1) it applies to a set of problem instances instead of only one (by map) and (2) it carries along the input data *xss* of the levels above the current level. The availability of *xss* permits us to use the restricted form of linear recursion presented before, linrec, which can be expressed easily by loop combinators.

Lemma 4.9 (mdcA/linrec)
mdcA can be expressed in terms of linrec, i.e., it holds mdcA = mdcAlinrec with the following definition:

```
mdcAlinrec istrivial basic divide combine x = linrec tr ba di co x where
  tr (xs:xss)    = null xs
  ba (xs:xss)    = ([], xss)
  di (xs:xss)    = map  (λx → if istrivial x then [] else divide x)
                        (concat xs)
                   : xs : xss
  co (fsols, xs:xss)
   = let  sols1 = unconcat (map  (λx → if istrivial x  then 0
                                                        else #(divide x))
                                 (concat xs) ) fsols
          xsols = zip (concat xs) sols1
          cop (x, sols) = if istrivial x  then basic x
                                          else combine x sols
     in (map cop xsols, xss)
```

The proof by equational reasoning can be found in App. A.3.
□

Let us sketch the idea of mdcA. The customizing functions given to linrec perform the work of all nodes at a particular level of the $\mathcal{DC}$ call tree. We use nested list structures to represent the call tree. In the next paragraph, we explain the representation, before we come to the semantics.

The output of mdcA consists of a pair: the first element is the list of problem solutions at a particular level and the second element is the input data passed along. The outer nesting level of the input of mdcA corresponds to the levels of the call tree, in reverse order. It is extended by a further call tree level when proceeding down the tree and is reduced level by level of the call tree on the way back. The middle nesting level of the input contains siblings of problem instances, i.e., those that share the same parent. This structural information is later used for grouping together the solutions of the siblings. The inner nesting level contains single problem instances.

The linear recursion works as follows: *tr* indicates the base of recursion if a level is reached which does not contain a node. In this case, *ba* delivers an empty list of solutions and the input history back. Otherwise, i.e., if the level reached contains at least one node, the collection *xs* of siblings is flattened into a list of individual nodes by concat. Then, the divide function is applied (in non-trivial cases) to every node, leaving a list of siblings for the next level which is added to the input data history. On the way back from the leaves to the root, the partial solutions *fsols* are grouped together with the input data in *xsols* by unconcat and zip, using the sibling structure of the input data history. Then all groups of input data and the corresponding partial solutions are processed as follows: if the predicate *istrivial* is satisfied, the basic function is applied, otherwise the combine function. Note that a check for an empty list of siblings is not sufficient for this distinction, since it is the combine function that has to be applied in cases in which the divide function produces an empty list of subproblems. The result of *co* is a pair of a list of solutions for the level above and, according to the specification of mdcA, the tail of the input data history.

We state the following law:

Law 4.10 (dcA/mdcA)

$$\begin{aligned} & \mathsf{dcA}\ \mathit{istrivial}\ \mathit{basic}\ \mathit{divide}\ \mathit{combine}\ x \\ = {} & \mathsf{fst}\,(\mathsf{mdcA}\ \mathit{istrivial}\ \mathit{basic}\ \mathit{divide}\ \mathit{combine}\ [[[x]]])\ \mathtt{!!}\ 0 \end{aligned}$$

The proof can be found in App. A.4.

□

Now, we can conclude:

Theorem 4.11 (dcA/linrec)

dcA can be expressed by linear recursion traversing the levels of the call tree.

Proof. dcA can be expressed by mdcA, using Law. 4.10, which can itself be expressed in terms of linrec according to Lemma 4.9. □

4.6.4 Approaching the iterative form of dcA

Theorem 4.11 states that dcA can be expressed by linear recursion. Theorem 4.6 demonstrates that linear recursion can be replaced by iteration. Combining these theorems, we obtain an iterative form of dcA. After additional code-improving transformations, which are described in App. B.1, we name it itA and present it in Sect. 4.7.

4.7 Iterative versions of the $\mathcal{DC}$ skeletons

Every part of this section is dedicated to the iterative equivalent of one of the $\mathcal{DC}$ skeletons dcA to dcF; see Tab. 3.1 and Fig. 3.6. For each recursive form dc$i$, we present a single iterative form it$i$ which corresponds to a synchronous loop program and realizes our ASAP/ALAP schedule. Other derivations are also possible; they can even consider particular machine properties as, e.g., in the APM approach (O'Donnell and Rünger, 1997; Ellmenreich et al., 1999; Winstanley, 1999).

An iterative form is derived by applying the restriction of the corresponding dc$i$ form to the previous iterative form and by simplification with respect to the applicability of iterative combinators.

The derivations are performed by means of equational reasoning; see App. B. Every subsection of this section is divided into a part for the iterative functional program, its explanation and its parallelization.

4.7.1 itA

4.7.1.1 Program

Here, we present the functional target program in Haskell which is already very close to an imperative style of programming.

```
itA ∈ (α→𝔹) → (α→β) → (α→[α]) → (α→[β]→β) → α→ β
itA istrivial basic divide combine x =
 let (xss, n, tss, lss) =
     while
       (λ (xss, n, tss, lss) → not (null (xss !! n)) )
       (λ (xss, n, tss, lss) →
              let  ds = [ if (tss !! n !! p) then [] else divide (xss !! n !! p)
                        | p←[0..#(xss !! n) −1] ]
                   ls = map # ds
                   xs = concat ds
                   ts = map istrivial xs
              in (xss++[xs], n+1, tss++[ts], lss++[ls]))
       ([[x]], 0, [[istrivial x]], [])
 in fordown n
     (λ r fsols →
         let bs = prefixsum (lss !! r)
         in  [ if  tss !! r !! p
                   then basic (xss !! r !! p)
                   else combine  (xss !! r !! p)
                                 [ fsols !! k | k←[bs !! p..bs !! (p+1)−1] ]
             | p←[0..#(xss !! r)−1 ] ])
     []
   !! 0
```

A data-parallel C program with annotations for parallel loops can be found in App. C. It was obtained from this functional version by exploiting correspondences between language constructs. E.g., list comprehensions in Haskell correspond to parallel loops which we annotate with `parfor`.

A look at the target program (imperative or functional) reveals the partitioning of the computation domain. The domain is a composition of two subdomains: one for the divide phase (represented in the program by a while loop) and one for the combine phase including the application of the basic function (represented by a fordown loop).

4.7.1.2 Explanation

The derivation of itA includes some code improvements in addition to mdcAlinrec. The data passed down and up the call tree is flattened, which saves memory space and overhead for restructuring. It also enables a simple distribution of data without regarding dependences between the nesting levels. The idea of flattening was taken from Blelloch (1990/1995).

However, we have to preserve the structural information, which we do by tupling. In the divide phase, we use a quadruple (xss, n, tss, lss). xss is a list of lists of input data. The elements at the outer level of nesting correspond to the levels of the call tree, the elements at the inner level to the positions at a particular level of the call tree. n is the depth of recursion. The structure of tss and lss equals the structure of xss. tss carries the values of the applications of *istrivial* to the input data, lss contains the numbers of subproblems. The quadruple is passed on in the divide phase, but not in the combine phase. There, it is only accessed by index, but the list of solutions at the current level of recursion is passed.

The fordown loop counts down the levels of recursion. r corresponds to the level of the call tree. The list *fsols* contains the solutions from the last level, initially [ ]. List bs is used to combine subproblem solutions that belong together. Element i of this list contains the index of the first element of group i in *fsols*. The values of the applications of *istrivial* are taken from tss, and xss provides the input data in the basic or combine function. The corresponding list of subproblem solutions for the combine function is created by copying a range from the entire list of solutions.

4.7.1.3 Parallelization

In the execution of an instance of the itA skeleton, a substantial amount of information is not known at compile time: the number of iterations of the while loop and the extents of the nested lists. Thus, the construction of a computation domain for itA, although possible if symbols are introduced as placeholders for unknown values, would make no sense because there is too much information missing for an automatic optimization.

Therefore, we prefer a parallelization at run time, which can equalize an imbalance of the call tree by a redistribution at every level, using the structural information carried along. However, parts like function prefixsum can still be parallelized statically. Also, the target code after parallelization of itA does not influence the implementation of other program parts, but only the resources of time, space, communication channels, etc. consumed.

The advantage of itA in contrast to dcA is that time and space are cleanly separated. This principle is known from the polytope model. Both the divide and the combine phase consist of an outer sequential and an inner parallel loop. This separation is not present in dcA. In an implementation of dcA, load balancing requires an explicit management of tasks, because the tasks for the subproblems cannot communicate, due to their independence. This management will likely cause large overhead in a distributed-memory system. In contrast, itA provides a global view of all tasks working at the same level. E.g., if these tasks are mapped to a linear chain of processors, processors with an overload can shift their data to a neighbor processor without violating the ordering of the tasks in space at the same level. Rebalancing in other topologies may be more efficient, but requires also more complicated local bookkeeping to restore the ordering. If the ordering of the tasks at each level of the call tree is preserved, modulo shifting operations, a centralized task management is not required.

4.7.2 itB

dcB differs from dcA in that (1) the predicate has been replaced by a counter of the levels of recursion and (2) the functions *divide* and *combine* are parameterized with the number of levels of recursion below the current level.

Since it is not easy for the user to modify the run-time system, he/she can influence the degree of parallelism and the granularity of the basic cases with the counter, using knowledge about the particular algorithm.

4.7.2.1 Program

The following auxiliary lemma exploits the availability of an explicit counter for the levels of recursion by transforming the while loop of itA into a for loop.

Lemma 4.12 (while/for)
$\forall\, mm, nn \in \mathbb{N} : \ \forall \alpha \in \textsc{Type} : \ \forall\, xx \in \alpha : \ \forall\, d \in (\alpha \to \alpha) :$

$$\begin{array}{ll} \mathsf{while} & (\lambda((m,x),n) \to \mathsf{not}\ (m{=}0)) \\ & (\lambda((m,x),n) \to ((m{-}1, d\ x), n{+}1)) \\ & ((mm, xx), nn) \\ = & \\ \multicolumn{2}{l}{((0, \mathsf{for}\ mm\ d\ xx), nn{+}mm)} \end{array}$$

The proof can be found in App. A.5. □

The step from for to forup in itB is made to provide the iteration counter to the customizing function of the loop.

```
itB ∈ (α→β) → (ℕ→α→[α]) → (ℕ→α→[β]→β) → ℕ → α→ β
itB basic divide combine n x =
 let (xss, lss) =
       forup n (λ r (xss, lss) →
                   let ds = [ divide (n−r) (xss !! r !! p)
                            | p←[0..#(xss !! r) −1] ]
                       ls = map # ds
                       xs = concat ds
                   in (xss++[xs], lss++[ls]))
              ([[x]], [ ])
```

```
      fsols = [ basic (xss !! n !! p)
              | p←[0 . . #(xss !! n)−1] ]
 in fordown n
            (λ r fsols →
               let bs = prefixsum (lss !! r)
               in [ combine (n−r) (xss !! r !! p)
                                  [ fsols !! k | k←[bs !! p . . bs !! (p+1)−1] ]
                  | p←[0 . . #(xss !! r) −1] ] )
            fsols
   !! 0
```

4.7.2.2 Explanation

The disappearance of the predicate *istrivial* has the consequence that *tss* is no longer required. Also, the value n can be eliminated from the tuple, because the while loop has been converted into a forup loop which has an internal counter. The only parameters that remain for the divide phase are *xss* and *lss*. The basic function can now be pulled out of the combine phase and applied between the divide and the combine phase.

4.7.2.3 Parallelization

A schedule for itB can be given at compile time, but not an allocation. Thus, the space-time mapping still depends on run-time values. However, all basic cases are applied at the same time, which may be useful for a simple load distribution. Often, the time required for the divide and combine function is insignificant compared to the time for an application of the basic function. In this case, it is sufficient to concentrate the load balancing on the basic phase by performing a redistribution. Note that, as soon as the divide phase is finished, the structure of the call tree is known and, thus, the allocation can be influenced, using the table *lss* computed by itB.

For the computation domain, the extent of dimension 0, i.e., the length of the schedule, can now be stated precisely. It involves $2+2n$ time steps: 1 input, n divide, 1 basic and n combine steps.

4.7.3 itC

4.7.3.1 Program

In addition to dcB, dcC contains a parameter for the degree of problem division. The skeleton implementation does not check or guarantee that the degree in the application program is correct; it expects it. Of course, a compiler can provide a safety mechanism, e.g., by requiring a list pattern of the respective length as a result of the divide function.

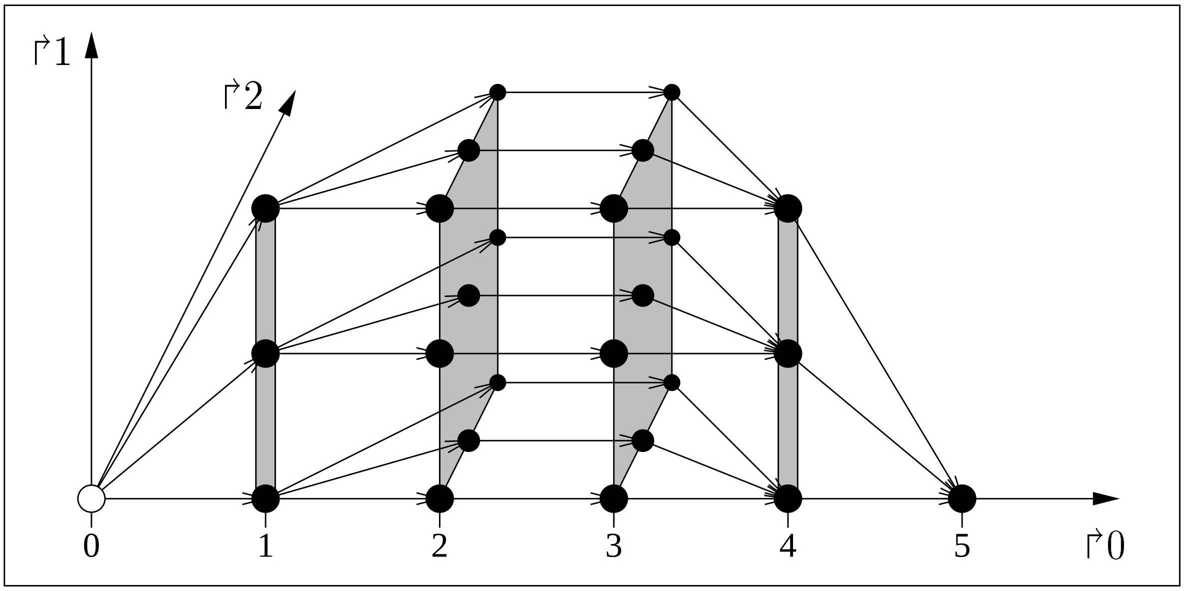


Figure 4.11: Index space for dcC ($k=3$, $n=2$)

```
itC ∈ ℕ → (α→β) → (ℕ→α→[α]) → (ℕ→α→[β]→β) → ℕ → α→ β
itC k basic divide combine n x =
 let  xss = scanforup  n
                      (λ r xs →  [ subs !! i
                                 | p←[0..k^r−1],
                                   let subs = divide (n−r) (xs !! p),
                                   i←[0..k−1] ])
                      [x]
      fsols = [ basic (xss !! n !! p) | p←[0..k^n−1] ]
 in fordown  n
             (λ r fsols →  [ combine  (n−r) (xss !! r !! p)
                                      [ fsols !! (k∗p+i) | i←[0..k−1] ]
                           | p←[0..k^r−1] ] )
             fsols
    !! 0
```

4.7.3.2 Explanation

We do not need the information *lss* any longer because the number of subproblems of every node is fixed to k. Also, we replace the forup loop, which performs the accumulation of the input data *xss* explicitly, by a scanforup loop which does this automatically.

4.7.3.3 Parallelization

The call tree of dcC is a complete k-ary tree of height n. Using the schema of Sect. 3.3, the nodes of this tree can be labeled uniquely with lists of a maximal length of n, carrying elements in the range from 0 to $k-1$. We are interested in unique labels of a fixed length, which can be interpreted as coordinates of points

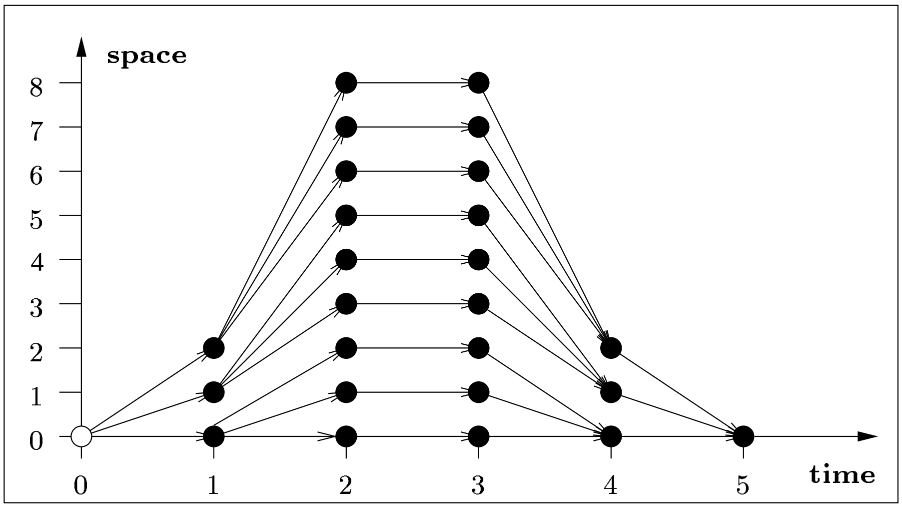


Figure 4.12: Target space for itC ($k=3$, $n=2$, free schedule)

phase	range	extent of space
input	$\Rsh 0 = 0$	1
divide	$1 \leq \Rsh 0 \leq n$	$k^{\Rsh 0}$
basic	$\Rsh 0 = n{+}1$	k^n
combine	$n{+}2 \leq \Rsh 0 \leq 2n{+}1$	$k^{2n-\Rsh 0+1}$

Table 4.2: Extent of the space in itC with free schedule

of an n-dimensional index space. To achieve this, we extend all labels to length n with zeros. Then, we prepend the label with the value of its original length. We use these fixed-size labels in the divide phase in the computation domain. For a problem instance with extended label $(r : xs)$ in the divide phase, the corresponding extended label of its solution instance is $(2n-r+1 : xs)$.

The index space is depicted in Fig. 4.11, for the special case of degree $k=3$ and recursion depth $n=2$. The dimensionality of the index space is $n{+}1$, i.e., 3 here. The shaded areas highlight the levels of the call tree. The arrows depict data dependences. Due to the multi-dimensional index space, the regularity of the data dependences becomes visible: for each coordinate value i in dimension 0, there exists a dimension z_i such that, for all dependences with target point $(i : xs)$ and source point $(i{-}1 : ys)$, xs and ys differ at most in position z_i. This schema is often used in the hypercube model, i.e., for the case $k=2$.

The derivation of itC preserves the free schedule in itB. Due to the limitations of our target language C, we have to stay away from arrays of arbitrary dimensionality. Thus, we present a two-dimensional target space, shown in Fig. 4.12, whose extent in dimension 0 is $2n+2$ (like for itB) and whose extent in dimension 1 is given in Tab. 4.2. $\Rsh i$ refers to the coordinate in dimension i.

A node with label $[b_0, ..., b_{d-1}]$ (without extension), i.e., located at level d of the call tree, is mapped to position $\sum_{i=0}^{d-1} b_i k^{d-i-1}$ in the processor space.

Usually, the number of available processors is not sufficient to implement the free schedule. Instead, some dimensions must be mapped to time. Fig. 4.13

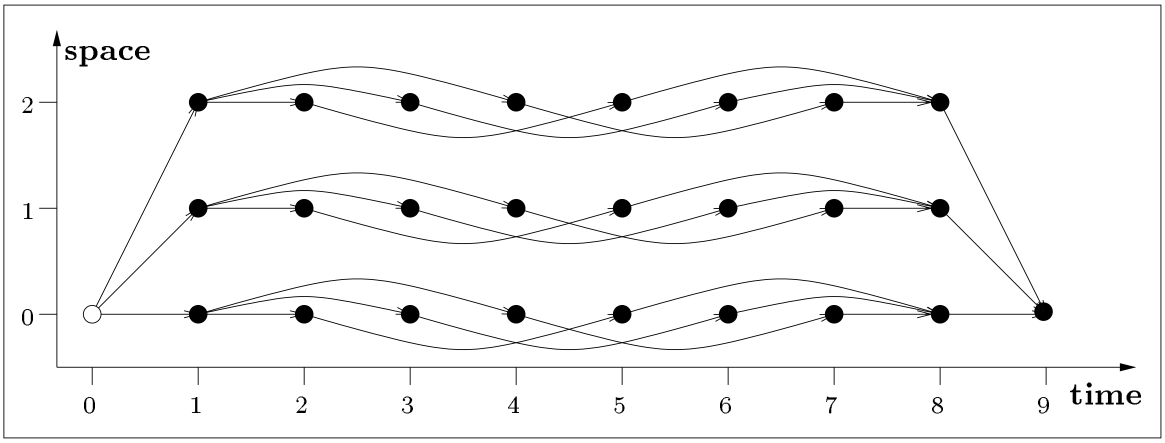


Figure 4.13: Restriction to three processors, breadth-first traversal

space mapped to time	time (τ)	space (σ)
none (free schedule)	$\Rsh 0$	$\sum_{i=1}^{\Rsh 0}(\Rsh i)k^{\Rsh 0-i}$
all but dimension 1	$1+\sum_{i=0}^{\Rsh 0-2} k^i+\sum_{i=2}^{\Rsh 0}(\Rsh i)k^{\Rsh 0-i}$	$\Rsh 1$

Table 4.3: Space-time functions for divide phase of itC

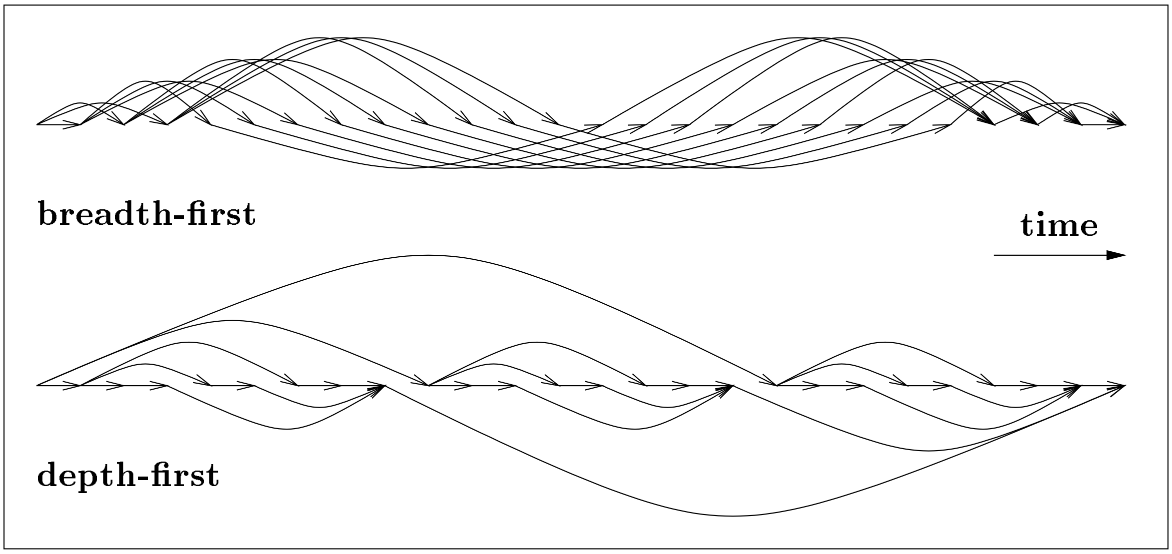


Figure 4.14: Breadth-first vs. depth-first sequential schedule (two levels, $k=3$)

shows the result of the mapping of our example index space, if dimension 2 is mapped to time. Not the maximal extent in dimension 2 is considered, but only the extent required for the current value of coordinate 0. Tab. 4.3 contains example schedules and allocations for the divide phase of itC, for general k and n. Thereby, ↱0 is in the range of 1 to n.

Note that the mapping of space to time just presented preserves the ordering of the free schedule, i.e., it corresponds to a breadth-first traversal of a part of the call tree. However, a depth-first traversal, which does not preserve the ordering, is more efficient with respect to memory consumption. In Fig. 4.14, we depict the two different mappings of a space dimension to time, for $k=3$. The number of arrows crossing a vertical line at a particular time equals the number of problem instances for which data must reside in memory.

4.7.4 itD

4.7.4.1 Program

Program itD is the result of imposing the enumeration of the elements of data structures by dcD on program itC.

```
itD ∈ ℕ→[ℕ]→[ℕ]→([α]→[β])→TDiv α→TCom α β→ℕ→[[α]]→[[β]]
itD k ids ods basic divide combine n x =
 let xss =
       scanforup n
        (λ r xs →
            [ subps !! q
            | p ←[0..k^r−1],
              let subps = let subs = divide (n−r) (xs!!p)
                              in [ [ [ subs !! subprob !! component !! item
                                     | item←[0..(ids!!component)^(n−r−1)−1] ]
                                   | component←[0..#ids −1] ]
                                 | subprob←[0..k−1] ],
              q←[0..k−1]]) [x]
     ys = [ let ys = basic [ xss!!n!!p!!i!!0 | i←[0..#ids −1] ]
             in [ [ys!!j] | j←[0..#ods−1] ]
          | p←[0..k^n−1] ]
 in (fordown n
            (λ r sols →
               [ let sol = combine (n−r) (xss!!r!!p)
                                   [ sols!!(k∗p+i) | i←[0..k−1] ]
                 in [ [ sol !! component !! part !! item
                      | part←[0..ods!!component−1],
                        item←[0..(ods!!component)^(n−r−1)−1] ]
                    | component←[0..#ods −1] ]
               | p ←[0..k^r−1]])
            ys) !! 0
```

4.7.4.2 Explanation

itD differs from itC mainly in the fact that the lists returned by the customizing functions are traversed using an enumeration schema which is guided by the lists *ids* and *ods*. These structural parameters contain the division degrees of the input and output data components. The traversal does not perform useful computational work (it can be skipped in the implementation), but it has the following purposes:

1. It serves as a specification of the extents of the nested structures returned by a customizing function. For the user, it prescribes constraints that must be established by the program, i.e., the program is taken to be defined only on the set of inputs for which the specification, if executed, terminates successfully. For the implementer, it defines the domain for which the implementation must work. Whether the implementation checks that the constraints are satisfied depends on the compiler and is not an issue here.

2. The proof seems simpler if restrictions, which are due to design decisions, are backpropagated to the specification, i.e., are contained implicitly in the functional expressions instead of being imposed by a side condition.

Contrary to itC, the size of the input and output data is known. This means that, in principle, the memory range in which the result of the customizing function is stored can be provided by the skeleton implementation, which enables a stronger reuse of allocated memory and, thus, improves performance. Unfortunately, this has the disadvantage that the code generation becomes more complicated, because invariants concerning the memory management have to be changed, i.e., the changes are not restricted to the implementation of itD.

4.7.4.3 Parallelization

The maximum degree of parallelization of the $\mathcal{DC}$ call tree is already achieved by itC. The skeleton does not induce a parallelization of the customizing functions themselves. Thus, the computation domain and the parallelization do not differ from itC.

4.7.5 itE

4.7.5.1 Program

In itE, every application of a customizing function is prepared for a parallelization by applying operations elementwise. This requires the customizing function to have a slightly different type. It must not deliver the entire structure of the result but only a single element of it, which is determined by additional parameters to the function.

4.7.5.2 Explanation

In both itD and itE, the elements of the results of the customizing functions are enumerated by the list comprehensions with the same extents. The difference is that, in itD, the list structure delivered by a single call is traversed without performing useful work and is skipped in the final implementation. In itE, the (parallel) traversal generates the elements of the result independently of each other.

```
itE ∈ ℕ→[ℕ]→[ℕ]→(ℕ→[α]→β)
      →TDiv_elem α→TCom_elem α β→ℕ→[[α]]→[[β]]
itE k ids ods basic divide combine n x =
 let xss =
        scanforup n
        (λ r xs →
           [ [ [ divide (n−r) subprob component item (xs!!p)
               | item←[0..(ids!!component)^(n−r−1)−1] ]
             | component←[0..#ids −1] ]
           | p ←[0..k^r−1],
             subprob ←[0..k−1] ]) [x]
     ys = [ [ [ basic j [ xss!!n!!p!!i!!0 | i←[0..#ids−1] ] ]
              | j←[0..#ods−1] ]
            | p←[0..k^n−1] ]
 in fordown n
     (λ r sols →
        [ [ [ combine (n−r) component part item (xss!!r!!p)
                          [ sols!!(k*p+j) | j←[0..k−1] ]
            | part←[0..ods!!component−1],
              item←[0..(ods!!component)^(n−r−1)−1] ]
          | component←[0..#ods −1] ]
        | p←[0..k^r−1]])
     ys
 !! 0
```

4.7.5.3 Parallelization

According to the list comprehensions, the index space of the computation domain can be refined by dimensions for the component, the data partition (*part*), etc. These dimensions can be laid out in space, due to the lack of dependences. In the final implementation, the data items need not be wrapped deeply inside a list structure but can be distributed homogeneously across the points of the index space grid.

4.7.6 itF

4.7.6.1 Program

In dcF, there is only a single input and a single output data block. This fact and the omission of the input data in the combine function permit correspondent communications between the subblocks.

```
itF ∈ ℕ→ℕ→ℕ→(α→β)→(ℕ →[α]→α)→(ℕ→[β]→β)→ℕ→[α]→[β]
itF k indeg outdeg basic divide combine n x =
 let xs = forup n
          (λ r xs →
               [ divide subprob  [ xs!!(item+(p*indeg+q)*indeg^(n−r−1))
                                 | q←[0..indeg−1] ]
               |  p ←[0..k^r−1],
                  subprob ←[0..k−1],
                  item←[0..indeg^(n−r−1)−1] ]) x
      ys =  [ basic (xs!!p)
            | p←[0..k^n−1] ]
 in fordown n
    (λ r sols →
         [  combine part  [ sols!!((k*p+j)*outdeg^(n−r−1)+item)
                          | j←[0..k−1] ]
         |  p←[0..k^r−1],
            part←[0..outdeg−1],
            item←[0..outdeg^(n−r−1)−1] ])
    ys
```

4.7.6.2 Explanation

In dcF, we only have a single component in the input and in the output, respectively. Thus, we can eliminate one nesting level in the divide and combine phase. The list comprehensions for the component disappear and list comprehensions for correspondent communication in the argument selection for the divide and combine function appear instead. This is inherited from dcF.

4.7.6.3 Parallelization

The equational transformations do not exploit properties of geometry. Thus, the index spaces for the list comprehensions in itF are of fixed dimensionality, but their extent depends exponentially on the recursion depth. The correspondent communications inspire us to use an extension of the hypercube model. A list of extent l^n, for which $l \in \{k, indeg, outdeg\}$, is represented by n dimensions of extent l in the model. Then, the dependence vectors for every time step are nonzero in only a single space dimension, which makes a mapping to a finite-dimensional mesh, e.g., a 3D grid, especially convenient (Herrmann and Lengauer, 1996).

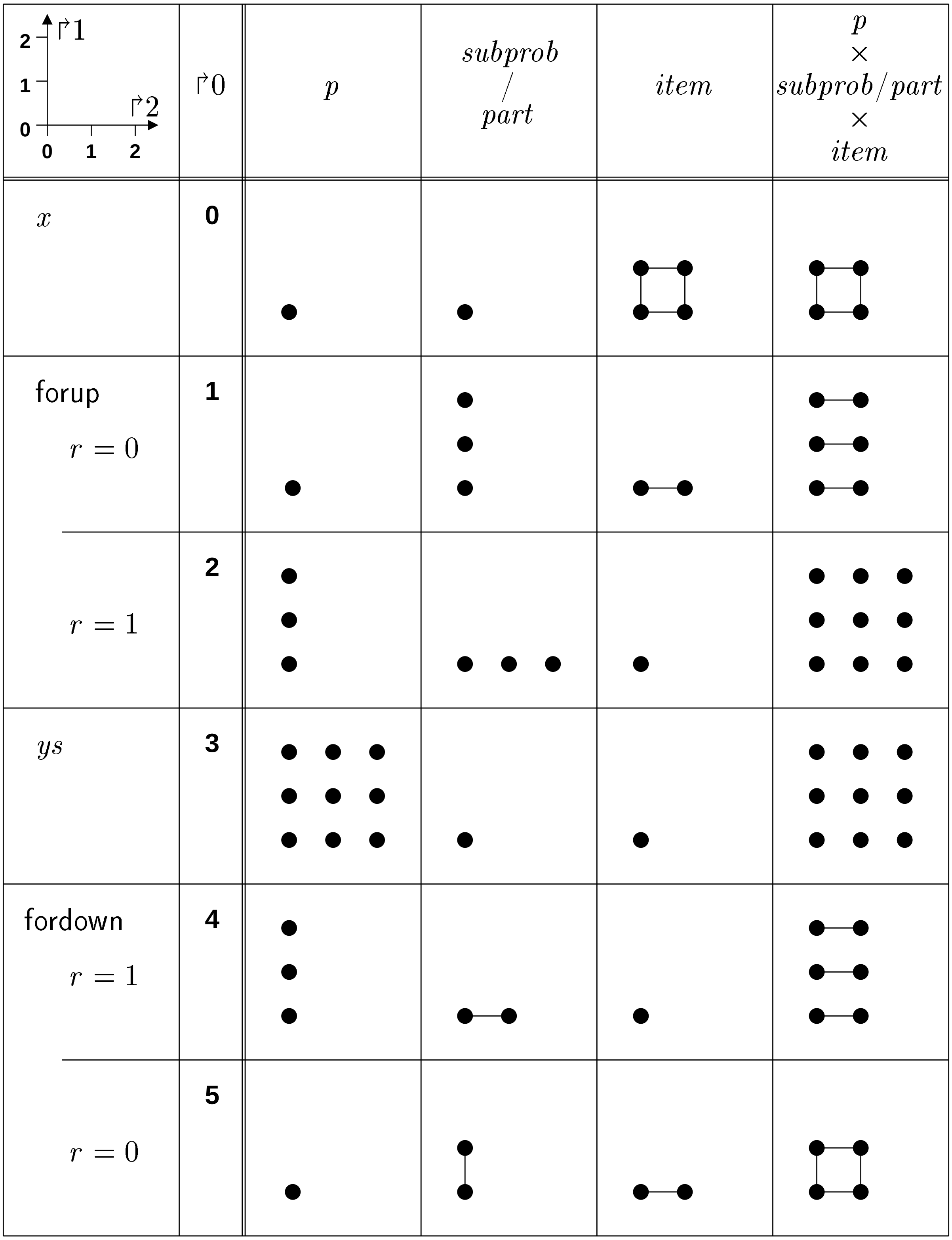


Table 4.4: Problem and data layout for (itF 3 2 2) with $n = 2$

skeleton	time
itA	dynamic
itB, itC, itD	$1+\sum_{i=0}^{n-1} d_i + b + \sum_{i=0}^{n-1} c_i$
itE, itF	$1+nd+b+nc$

Table 4.5: Time for iterative $\mathcal{DC}$ skeletons

skeleton	space
itA,itB	dynamic
itC	k^n
itD, itE	$\uparrow_{degs\in\{ids,ods\}} \uparrow_{i=0}^{n} (\sum_{j=0}^{\#degs-1}(degs\,!!\,j)^i)k^{n-i}$
itF	$\uparrow_{i=0}^{n}(indeg \uparrow outdeg)^i k^{n-i}$

Table 4.6: Space for iterative $\mathcal{DC}$ skeletons

We discuss the index space with the help of Tab. 4.4. The iterators fordown and forup, as well as the variables, refer to the names in itF. The rows of the table refer to coordinates in dimension 0, which must be mapped to time. Due to the choice of $n=2$, we have two other dimensions, which are partly used to lay out problems, parts and data items. The columns show these different kinds and the Cartesian product of them. If we place the entries of the last column along dimension 0, we obtain the index space depicted in Fig. 4.3.

Whether a dimension other than 0 is used for the layout of problems or data depends on the level of recursion, i.e., on the coordinate in dimension 0. Lines between circles indicate dimensions which are used for the layout of data items or parts; the other dimensions are used for the layout of problems, subproblems and solutions. The points in column p represent problem instances or solutions. Communication takes place along dimension $subprob/part$. In the divide phase, this is the dimension which switches from data layout to problem layout, while changing an extent of $indeg$ $(=2)$ in $item$ to k $(=3)$. In the combine phase, it switches from problem layout to data layout, while changing an extent of k $(=3)$ in p to $outdeg$ $(=2)$. In every picture, we set the coordinates of the dimensions not involved to 0. Thus, we achieve that the entry of the last column is the Cartesian product of the entries of the three columns before.

4.7.7 Time and space consumption

Let us assume that dimension 0 is mapped entirely to time and all other dimensions are mapped entirely to space. Then, the time and space consumption is given by Tab. 4.5 resp. Tab. 4.6, in the following parameters:

- n: recursion depth
- k: degree of problem division

- $d/b/c$: maximum execution time of the divide/basic/combine function

Since d or c can vary widely between levels of recursion, they are indexed with the recursion depth i. *indeg*, *ids*, *outdeg* and *ods* refer to the parameters in the particular skeleton. If one removes the quantifications $\uparrow_{i=0}^{n}$ in Tab. 4.6, one obtains the space consumption at the time steps $n-i$ (if *ids* or *indeg* is chosen) resp. $n+i+1$ (if *ods* or *outdeg* is chosen).

4.8 Example: Mapping itF to a multi-dimensional grid

An interesting special case of space-time mapping itF is if the topology is a D-dimensional grid. Special cases of such mappings have been investigated by Goodman and Mou (1991), Mou et al. (1992) and Gorlatch and Bischof (1999).

For simplicity, we choose (itF 3 2 2) with $n=4$ levels of recursion, mapped to a two-dimensional grid of 9×9 processors such that every processor has to compute exactly one basic case. We concentrate on the divide phase here; the schema in the combine phase is analogous.

The data locations and communications in the grid are depicted in Fig. 4.15. Each of the four pictures corresponds to a particular division step. We associate a division step with the discrete time at which the step has terminated, because we apply the owner-computes rule. Thus, $\Rsh 0 = 0$ refers to the reading of the input, which is not of interest for us here. The first division step is assigned to $\Rsh 0 = 1$ and shown in the upper left picture.

The filled circles contain the data allocation at the beginning of a division step. A division step consists of an all-to-all exchange along the wires depicted and of a local computation of the individual part of the divide function. At the end of a step, all connected circles, also the unfilled ones, provide input data for the next division step.

Also for simplicity, the space mapping has been chosen not to depend on coordinate 0, i.e., no redistributions are necessary. The advantage is a high regularity of the data dependences. Unfortunately, the disadvantage of the regularity is that a convenient distribution of the input and output data on border processors is not possible.

However, with our choice, the space mapping preserves useful properties of the data dependences in the index space, with the result that the direction of the data dependences is restricted to a single space dimension at a time. Regardless of the size of the grid, the use of pipelining allows us to apply a communication schema that is free of congestion, as Fig. 4.15 illustrates.

4.8.1 Communication times

For a grid, which has the extent of k^R in every dimension, the communication for any value of $\Rsh 0$ requires at most $2(k-1)k^{R-1}$ parallel steps of sends and receives. Have a look at Fig. 4.15, where $R=2$. In our model, every link can perform a

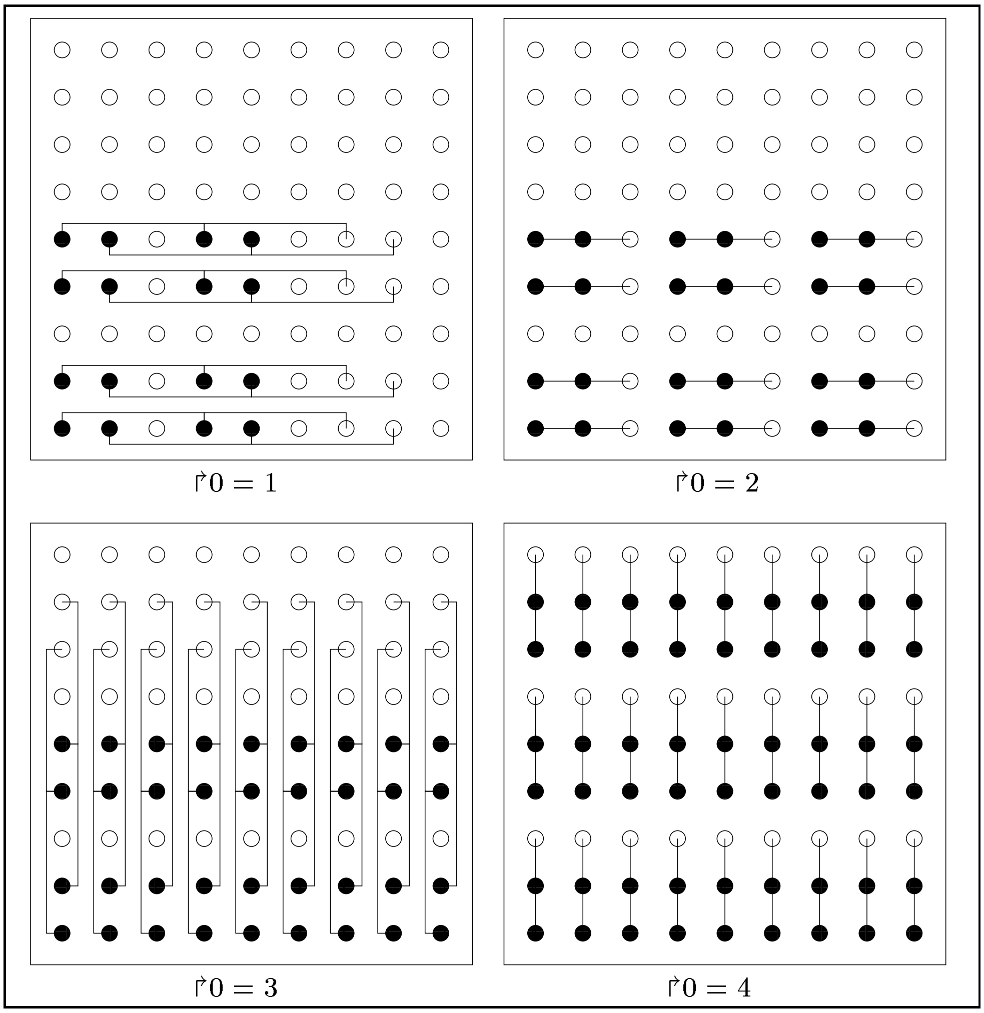


Figure 4.15: Divide phase of (itF 3 2 2), $n=4$ on a 9×9 grid

uni-directional transfer of the data located at a point in one time step. In the upper left picture, the communication requires not more than six steps from left to right and six steps from right to left, in a pipelined fashion. (For simplicity, we withhold a more sophisticated schema with even fewer steps.) The worst case along a particular dimension dominates all other communications along this dimension. Also, there are only D dimensions.

Definition 4.26 (Diameter)
The distance of two nodes in a graph is the length of the shortest path between them. The diameter of a graph is the maximum distance of all pairs of nodes. $\square$

Mou et al. (1992) stated that, for (itF 2 2 2) on grids, the sum of all communication times is proportional to the diameter of the grid. We can generalize this fact by the following lemma:

Lemma 4.13 (Communication times for itF on grids)
Let the index space of (itF k *indeg outdeg*) with recursion depth n be constructed as described in Sect. 4.7.6.3. Let $c_{\text{in}}/c_{\text{out}}$ be the number of real memory cells

required for an element of the input/output list. Further, assume a space mapping to a D-dimensional, hypercubic grid of $p = k^r = k^{R \cdot D}$ ($r \leq n$) processors, in which σ_j, the projection to space dimension j, is defined by:

$$\sigma_j = \sum_{i=0}^{R-1} (\vec{r}(j \cdot R + i + 1)) k^{R-i-1}$$

□

Then, an upper bound for the total amount of communication steps T_{comm} is:

$$T_{\text{comm}} = 2D(\sqrt[D]{p} - 1)(c_{\text{in}} \cdot indeg^{n-r} + c_{\text{out}} \cdot outdeg^{n-r})$$

Proof. The lengths of the dependence vectors are the multiples from 1 to $k-1$ of a power of k. Thus, for the numbers *indeg* or *outdeg* in the range 1 to k, the worst case communication time is for the value k.

The extent of a D-dimensional grid of $p = k^r$ processors is k^R in every dimension. If we concentrate on a single topology dimension, the total amount of communication steps in either the divide and combine phase is $\sum_{i=0}^{R-1} 2(k-1)k^i$, which is $2(k^R - 1)$. If we consider all space dimensions, i.e., an entire divide or combine phase, we obtain $2D(k^R - 1)$, which is $2D(\sqrt[D]{p} - 1)$. However, the communication time also depends on the amount of data transmitted. The amount of input/output data per node which remains constant during the divide/combine phase is $indeg^{n-r}$ resp. $outdeg^{n-r}$.

4.8.2 Computation times

Lemma 4.14 (Computation times for itF)
Let the index space of (itF k *indeg outdeg*) with recursion depth n be constructed as described in Sect. 4.7.6.3. Let $r \leq n$ be the number of index space dimensions mapped to processor space. Let $t_{\text{div}}/t_{\text{bas}}/t_{\text{com}}$ be the amount of real time for an application of the divide/basic/combine function. Then, an upper bound for the pure computation time is:

$$\begin{aligned} T_{\text{comp}} &= r(t_{\text{div}} \cdot indeg^{n-r} + t_{\text{com}} \cdot outdeg^{n-r}) \\ &+ \textstyle\sum_{i=0}^{n-r-1} (t_{\text{div}} \cdot indeg^{n-r-i} + t_{\text{com}} \cdot outdeg^{n-r-i}) k^i \\ &+ t_{\text{bas}} \cdot k^{n-r} \end{aligned}$$

Proof. Every processor has to apply the appropriate function to $indeg^{n-r}$ resp. $outdeg^{n-r}$ points, in every non-local step of the divide and combine phase the processor is involved in. There are r such steps in both the divide and combine phase, which yields the first row in the equation for T_{comp}. Then, there are $n-r$ steps in both phases, in which the divide and combine function have to be applied purely locally, i.e., sequentially, due to a lack of processors. Step i of these has to do the work on k^i problem instances, each of which works on $indeg^{n-r-i}$ resp. $outdeg^{n-r-i}$ data items. Finally, the number of basic cases computed at each node is k^{n-r}.

4.8.3 Use of assumptions that hold for some problems

For a detailed analysis of bitonic merge, Karatsuba's polynomial product and Strassen's matrix multiplication, we can fix some parameters. We take $t_{\text{div}} = t_{\text{bas}} = t_{\text{com}} = 1$ and $c_{\text{in}} = c_{\text{out}} = 1$, which is realistic if simple operations are involved. For all three examples, $\mathit{indeg} = \mathit{outdeg}$. Let $d = \mathit{indeg}$.

Then, the following times result:

$$\begin{aligned}
T_{\text{comm}} &= 4D(\sqrt[D]{p} - 1)d^{n-r} \\
T_{\text{comp}} &= k^{n-r} + 2d^{n-r}(r + \textstyle\sum_{i=0}^{n-r-1}(k/d)^i)
\end{aligned}$$

To solve the summation in T_{comp}, we make a case distinction:

$$\begin{aligned}
d = k &: T_{\text{comp}} = (2n+1)k^{n-r} \\
d < k &: T_{\text{comp}} = \tfrac{1}{k-d}(2(r(k-d) - d)d^{n-r} + (k+d)k^{n-r})
\end{aligned}$$

To concentrate on the governing factors, we investigate the asymptotic behavior, dependent on the input size N. The number p of processors can be chosen dependent on N, i.e., is not a constant. n depends on N and r depends on p and, thus, these values are not constant, either. Constants are the division degrees and the dimensionality D of the topology. Thus, we obtain:

$$T_{\text{comm}} \in \Theta(\sqrt[D]{p} \cdot d^{n-r})$$

$$\begin{aligned}
d = k &: T_{\text{comp}} \in \Theta(n \cdot k^{n-r}) \\
d < k &: T_{\text{comp}} \in \Theta(k^{n-r})
\end{aligned}$$

Now, we use the equations $d^n = N$ and $k^r = p$:

$$T_{\text{comm}} \in \Theta(N \cdot p^{1/D - \log_k d})$$

$$\begin{aligned}
d = k &: T_{\text{comp}} \in \Theta(N \cdot \log N / p) \\
d < k &: T_{\text{comp}} \in \Theta(N^{\log_d k} / p)
\end{aligned}$$

Next, we use these results to obtain the conditions for sublinear execution time and cost optimality.

4.8.3.1 Sublinear execution time

From a theoretical point of view, we are interested in sublinear execution time, because it exceeds the acceleration that can be achieved by the polytope model. However, we do not see a practical relevance, since the number of processors of a particular machine is fixed.

The following conditions must be satisfied to achieve sublinear execution times:

1. Communication: $D > \log_d k$ and $p \in \omega(1)$ (p increases with N)

   This means that a two-dimensional grid is sufficient for bitonic merge, Karatsuba's polynomial multiplication and Strassen's matrix multiplication , because 2 is greater than $\log_2 2$, $\log_2 3$ and $\log_4 7$.

			Sublinearity	Cost optimality	
Algorithm	d	k	$D \geq 2$	$D = 2$	$D = 3$
bitonic merge	2	2	$p \in \omega(\log N)$	$p \in O((\log N)^2)$	$p \in O((\log N)^3)$
Karatsuba	2	3	$p \in \omega(N^{0.58})$	$p \in O(N^{0.67})$	$p \in O(N^{0.83})$
Strassen	4	7	$p \in \omega(N^{0.40})$	$p \in O(N^{0.51})$	$p \in O(N^{0.65})$

Table 4.7: Conditions for example algorithms

2. Computation:

   (a) $d = k$: $p \in \omega(\log N)$

   (b) $d < k$: $p \in \omega(N^{\log_d k - 1})$

Tab. 4.7 presents the conditions for the current examples. Note that, in Strassen's multiplication, we have chosen N to be the number of matrix elements (real input size) instead of the number of rows. In this case, the sequential time complexity of Strassen is in $\theta(N^{1.40..})$, i.e, smaller than for Karatsuba, which is in $\theta(N^{1.58..})$.

4.8.3.2 Cost optimality

Referring to Def. 1.7, the computation is *cost-optimal* if, and only if, the product of parallel computation time and number of processors equals asymptotically the sequential time complexity. Cost optimality or, more precisely, improvements of asymptotic costs can be practically relevant. E.g., the Karatsuba and Strassen algorithms show only a small overhead factor compared to their naïve counterparts, which means that their absolute execution times are superior to the naïve algorithms, already for realistic problem sizes.

The requirements for cost optimality are:

1. Communication:

   (a) $d = k$: $p \in O((\log N)^D)$
   This means, that cost optimality cannot be established for a reasonable number of processors with a grid of fixed dimensionality.

   (b) $d < k$: $p \in O(N^{(\log_d k - 1)/(1/D - \log_k d + 1)})$

2. Computation: None with the proposed schema.

See also Tab. 4.7 for cost optimality.

Chapter 5

Compilation Techniques

This chapter presents a series of compilation techniques that can be employed to translate a functional source program containing skeletons into an imperative target program which makes use of predefined parallel implementations of these skeletons. Large parts of this chapter originate from a technical report on the $\mathcal{HDC}$ compiler developed at our lab (Herrmann et al., 1999b). All parts other than those listed below have been developed solely by the author of this thesis.

The contents of

1. Sects. 5.3.1 and 5.3.2 describe work by Robert Günz (1998). This work is important for the $\mathcal{HDC}$ compiler itself, but it is of minor relevance for the methods proposed here and, therefore, only sketched.

2. Sects. 5.3.6 and 5.3.7 are based on joint work with Christian Schaller (1998). They deal with crucial steps: the elimination of polymorphism and higher-order functions.

3. Sects. 5.3.8–5.3.10, 5.3.12.1 and 5.3.13 are based on joint work with Jan Laitenberger (1999). These sections deal with transformations that deliver simple control structures and perform significant program optimizations.

Several parts of this chapter differ substantially from the $\mathcal{HDC}$ report; they discuss the latest state of the implementation or even techniques which are not yet implemented.

The techniques discussed here are presented according to phases of the $\mathcal{HDC}$ compiler. Several ideas are not original and find application also in other compilers, possibly, in a different combination. Some phases of our compiler perform source-to-source transformations – in an internal representation. We follow the principle of *compilation by transformation*, which has been used successfully in the Glasgow Haskell compiler(GHC) (Haskell, 2000; Peyton Jones and Santos, 1998).

Compilation by transformation eliminates or introduces program structures step by step, in several, but often simple transformation phases. This is in contrast to many compilers for imperative languages whose designers aim to minimize the number of phases. However, because they do not provide code representation at several intermediate steps, they cannot replace compilation techniques easily.

On the one hand, each particular skeleton is specified by a Haskell function, often a recursive one. On the other hand, an implementation benefits from being written in an imperative language which can be instrumented efficiently for parallel execution, e.g., in C with calls of MPI library functions.

In (Herrmann and Lengauer, 1996/1997a), we presented parallel imperative implementations for a few $\mathcal{DC}$ skeletons, but left open the question of how to pass customizing functions to skeletons. In the examples, the customizing functions had been inlined manually. The $\mathcal{DC}$ specifications are recursive and thus, the correspondence between them and their iterative implementations is not trivial. We decided to stay in the Haskell formalism as long as possible, in order to benefit from the strength of equational reasoning (Herrmann and Lengauer, 1997a). The first step was to transform each functional source skeleton ($\mathsf{dc}i$) into an equivalent *functional target* skeleton ($\mathsf{it}i$) by means of equational reasoning. In the second step, $\mathsf{it}i$ was translated to C under preservation of the program structure. Each language construct used has a counterpart in C, e.g., list comprehensions correspond to loops.

For data, there is a correspondence, e.g., of lists on the functional side with arrays on the imperative side, which is based on the shape of the index space and not on the representation. Haskell lists as well as C arrays (vectors) are indexed by natural numbers and their elements all have the same type. Nested C arrays are not constrained to a rectangular shape. They are indexed dimension by dimension, a property which they have in common with lists of lists in Haskell and which differs from multi-dimensional arrays in Haskell or Pascal.

Later, we found a source-to-source transformation method, which replaces every customizing function by an encoding of its closure (Bell et al., 1997). Then, we were able to let a functional language form the glue between the different skeletons in the program, transparent to the user and the skeleton implementer. Because skeletons can be mimicked as definitions in the source language itself, the implementation of nested skeletons results by composition from the implementation of single skeletons. The investigation of nested skeletons, like dcF in dcD for bitonic sort as described in Sect. 3.5, is therefore not necessary in order to obtain a working implementation but is of advantage for finding a good balance between the degree of parallelism in both skeletons at compile time.

We start with a brief description of our language $\mathcal{HDC}$. The currently used skeletons are presented in Sect. 5.2. Then, we describe the $\mathcal{HDC}$ compiler and its phases in Sect. 5.3. Sect. 5.4 discusses the run-time system and Sect. 5.5 points out possibilities of $\mathcal{HDC}$ for the future.

5.1 The language $\mathcal{HDC}$

$\mathcal{HDC}$ is a restriction of Haskell to a subset of the syntax and the type class system. In contrast to Haskell, $\mathcal{HDC}$ does not have a strict semantics. Basically, function application is strict to respect the space-time mapping of skeletons, but aggressive program optimizations are permitted which can even eliminate function applications. To be on the safe side, termination under a strict semantics must always be guaranteed by the programmer.

$\mathcal{HDC}$ was designed with the goal to support the parallelization of $\mathcal{DC}$. The $\mathcal{HDC}$ report (Herrmann et al., 1999b) describes the language in detail. Here, we provide only a brief list of its features:

- An $\mathcal{HDC}$ program is a list of functions, one of which is named parmain. It constitutes the main function.

- $\mathcal{HDC}$ offers the primitive data types Unit (the type which contains exactly one element), Bool (the truth values), Int (32-bit integers) and Double (the floating point numbers). The type combinators in $\mathcal{HDC}$ are _ → _ (function types), [_] (list types), (_ , ..., _) (tuple types) and IO _ (monadic input/output types). In addition, the user can define algebraic data types.

- Expressions in $\mathcal{HDC}$ are variables, constants, function applications, λ-abstractions, conditionals, tuples, lists, patterns, binary infix operators, sectioning, let-expressions, case-expressions, arithmetic sequences and list comprehensions – all akin to Haskell.

  The usual list constructor in functional programming, (:), stands in the way of parallelism by inducing dependences between adjacent list elements. By default, the $\mathcal{HDC}$ compiler implements lists as arrays, transparent to the user. Thus, the user is advised to construct lists by append (++) of nearly equally-sized lists, list comprehensions and other combinators. The access of list elements by the predefined indexing operator (!!) is a constant-time operation in $\mathcal{HDC}$. List comprehensions are used to describe parallel, index-based operations.

5.2 Skeletons in $\mathcal{HDC}$

Skeletons have been used widely for parallel programming. $\mathcal{HDC}$ lends special support to programming with skeletons. Our central goal is to provide skeletons for the $\mathcal{DC}$ paradigm and supply efficient parallel implementations for them.

We divide the skeletons offered by $\mathcal{HDC}$ into five distinct categories. Each of the following subsections lists the implemented skeletons of one category. For each skeleton, we provide the signature, an algorithmic definition in $\mathcal{HDC}$ and an example application. The functions which generate the C code of the skeleton implementations are quite long and, therefore, not presented here.

5.2.1 Skeletons for commonly used functions

The skeletons presented here are commonly used functions which have efficient parallel implementations. They should always be used if they do not hide the presence of a more complex skeleton like one of the $\mathcal{DC}$ skeletons.

5.2.1.1 map

Applies a function to all elements of a list. Repeating Def. 1.10:

$\mathsf{map} \in (\alpha{\to}\beta) \to [\alpha] \to [\beta]$
$\mathsf{map}\ f\ [\,] \quad = [\,]$
$\mathsf{map}\ f\ (x{:}xs) = f\ x : \mathsf{map}\ f\ xs$

$\mathsf{map}\ (+1)\ [0,1,2] = [1,2,3]$

5.2.1.2 red

Uses a function f to reduce a list of values to a single value. It is assumed that f is associative because the reduction is aimed to be implemented via a balanced tree computation. If f is not associative, the implementation is not obliged to implement the semantics of the recursive specification below. For this reason, we avoid the name fold here. In the future, we plan to express associativity by the type class system instead of the informal side condition here.

$\mathsf{red} \in (\alpha{\to}\alpha{\to}\alpha) \to \alpha{\to} [\alpha] \to \alpha$
$\mathsf{red}\ f\ n\ [\,] \quad = n$
$\mathsf{red}\ f\ n\ (x{:}xs) = f\ x\ (\mathsf{red}\ f\ n\ xs)$
$\mathsf{red}\ (+)\ 0\ [1,2,3] = 6$

5.2.1.3 scan

Applies red to all prefixes of the given list.

$\mathsf{scan} \in (\alpha{\to}\alpha{\to}\alpha) \to \alpha{\to} [\alpha] \to [\alpha]$
$\mathsf{scan}\ f\ n\ xs = [\ \mathsf{red}\ f\ n\ (\mathsf{take}\ i\ xs) \mid i{\leftarrow}[0\,..\,\#xs]\ ]$

$\mathsf{scan}\ (+)\ 0\ [1,2,3] = [0,1,3,6]$

5.2.1.4 filter

Filters all elements that satisfy a predicate p.

$\mathsf{filter} \in (\alpha{\to}\mathbb{B}) \to [\alpha] \to [\alpha]$
$\mathsf{filter}\ p\ [\,] \quad = [\,]$
$\mathsf{filter}\ p\ (x{:}xs) = \underline{\mathtt{let}}\ rest = \mathsf{filter}\ p\ xs$
$\qquad \underline{\mathtt{in}}\ \underline{\mathtt{if}}\ p\ x\ \underline{\mathtt{then}}\ x : rest$
$\qquad\qquad \underline{\mathtt{else}}\ rest$

$\mathsf{filter}\ ({>}2)\ [0,5,3,1,5] = [5,3,5]$

5.2.2 $\mathcal{DC}$ skeletons

At present, $\mathcal{HDC}$ provides the general $\mathcal{DC}$ schema, in the form of skeleton dcA, and a skeleton dcF, obtained by successive specializations, which incurs a large amount of data parallelism. In (Herrmann et al., 1999b), dcA is called dc0; dcF without level information in the divide and combine function is called dc4io. Both skeletons are described in Chapter 3.

5.2.3 Skeletons for improved efficiency

These skeletons enrich the functional language with powerful imperative (while) or symbolic (Γ) constructs. Their main purpose is to be plugged in by an automatic optimization. Aside from that, they also can be used by the programmer.

5.2.3.1 while

Takes a predicate p, a function f and an initial input value x and iterates f as long as the predicate on the input of the iteration is True. The while skeleton is useful as a result of an elimination of tail recursion, denoted in the source language. In contrast to tail recursion, while does not increase the recursion stack from iteration to iteration.

Repeating Def. 4.20:

$$\mathsf{while} \in (\alpha{\rightarrow}\mathbb{B}) \rightarrow (\alpha{\rightarrow}\alpha) \rightarrow \alpha \rightarrow \alpha$$
$$\mathsf{while}\ p\ f\ x = \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ \mathsf{while}\ p\ f\ (f\ x)\ \underline{\texttt{else}}\ x$$

$$\begin{aligned} \mathsf{while}\ \ & (\lambda(i,s) \rightarrow i{<}3) \\ & (\lambda(i,s) \rightarrow (i{+}1, s{+}i{*}i)) \\ & (0,0) \\ = {} & (3,5) \end{aligned}$$

5.2.3.2 Γ (sinGen)

Takes a function f and a number n and generates a list of length n whose value at position i is computed by applying f to i. Γ stems from list comprehensions. In (Herrmann et al., 1999b) Γ is named sinGen.

The aim of Γ is to have a short representation for large, regular index sets, e.g., the first 10000 even numbers. To make this work in program optimization (Sect. 5.3.12.2), Γ has to be fused with other combinators as shown in Figs. 4.4–4.6.

$$\Gamma \in (\mathbb{N}{\rightarrow}\alpha) \rightarrow \mathbb{N} \rightarrow [\alpha]$$
$$\Gamma\ f\ n = [\ f\ i \mid i{\leftarrow}[0\,..\,n{-}1]\ ]$$

$$\Gamma\ (\lambda i{\rightarrow}\ i{*}i)\ 17 = [0,1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256]$$

5.2.4 Evaluation control

The skeletons of this section influence the evaluation. Note that some functions like delay, which defers the evaluation of an expression, and force, which enforces the evaluation of a delayed expression, are defined in the $\mathcal{HDC}$ prelude and not implemented by skeletons. Thus, they do not appear here.

5.2.4.1 strict

Strictness is the worst case to be assumed for $\mathcal{HDC}$ functions. Program optimization can cause strictness partly to be eliminated, especially if functions are inlined. However, strictness can be required at some places. Thus, the skeleton strict takes a function f and an argument x and evaluates x before calling f. Program optimizations will not touch the application of f to x. This skeleton is necessary to protect IO-monadic functions against elimination via inlining in the case of a lack of data dependences. Otherwise, output operations could erroneously be eliminated if their return value is not used.

strict $f\ x = f\ x$

5.2.4.2 Parallel composition (||)

The meaning of the expression $(f \parallel g)$ is that the functions f and g are to be applied in parallel to the same argument. Its introduction was inspired by the par function of GpH, but its use is different.

Repeating Def. 4.18 from Sect. 4.3.8.2:

$(\parallel) \in (\alpha{\to}[\beta]) \to (\alpha{\to}[\beta]) \to (\alpha{\to}[\beta])$
$(f \parallel g)\ x = f\ x \mathbin{+\!\!+} g\ x$

5.2.5 Input/Output

For the input and output functions, we cannot provide a purely functional definition. The reason is their interaction with the environment. However, this impurity is hidden by the IO monad.

5.2.5.1 put

Takes a value and delivers an I/O action which returns Unit. In the I/O action, the value is appended to the standard $\mathcal{HDC}$ output channel. Currently, printable values can be of type Int, Double and can also be tuples and lists composed of printable values. In the future, we plan to extend the set of printable values and support parallel output.

put $\in \alpha \to$ IO Unit

5.2.5.2 get

Performs an I/O action in which a value of type α is read from the standard $\mathcal{HDC}$ input channel. Parallel input is considered for the future. The set of readable values is the same as the set of printable values (see put).

get $\in$ IO α

5.3 A compiler for $\mathcal{HDC}$ and its phases

We have developed a compiler for $\mathcal{HDC}$ that supports the integration of skeletons in an imperative, parallel environment. The $\mathcal{HDC}$ compiler generates code in an imperative language – at present, C with MPI calls. The main difference to Haskell is that $\mathcal{HDC}$ is almost strict, in order to facilitate a compile-time parallelization. Two implementational differences to a typical Haskell compiler are that (1) functional arguments are eliminated and (2) list comprehensions are simplified to combinations of parallel skeletons. The reason is that higher-order functions complicate but list comprehensions simplify a static space-time mapping.

Together with the compiler, we also provide an interpreter which analyzes the intermediate code after particular phases of the compilation and reports certain properties of the program to the user, like the free schedule or the average degree of parallelism, i.e., the average number of parallel processors required by the free schedule. Compiler and interpreter are either controlled interactively or by running a Haskell script which must be compiled together with the rest of the system.

The following subsections deal with the phases of the compiler, presented in the same order as in the compilation.

5.3.1 Scanner and parser

The $\mathcal{HDC}$ parser, generated by the tool *happy* (Haskell, 2000), translates the source program into a set of syntax trees, one for each function in the program. Each syntax tree is represented by an algebraic data type in Haskell.

The *layout style* of Haskell is supported, i.e., indentation can be used instead of braces and semicolons to group items at the same level of particular syntactic structures. The user can declare new operators and state their precedence and associativity.

5.3.2 Simplification of pattern matching

In this phase, every nested pattern matching is transformed into a combination of simple pattern matchings by additional case-expressions. After the transformation, each pattern is either a simple variable or a constructor with simple variables as arguments.

lcEmpty	$$\frac{[\ e \mid\ ]}{[\ e\ ]}$$
lcSinGuard $\{g \text{ is a guard}\}$	$$\frac{[\ e \mid g\ ]}{\underline{\texttt{if}}\ g\ \underline{\texttt{then}}\ [e]\ \underline{\texttt{else}}\ [\,]}$$
lcOptGuard $\{g \text{ is a guard},$ $x \notin \text{freevars}(g)\}$	$$\frac{[\ e \mid R, x \leftarrow xs, g, S\ ]}{[\ e \mid R, \underline{\texttt{if}}\ \mathsf{null}\ xs\ \underline{\texttt{then}}\ \mathsf{False}\ \underline{\texttt{else}}\ g, x \leftarrow xs, S\ ]}$$
lcXGen	$$\frac{[\ e \mid R, x \leftarrow xs\ ]}{\mathsf{concat}\ [\ \mathsf{map}\ (\lambda x \rightarrow e)\ xs \mid R\ ]}$$
lcGenGuard $\{g \text{ is a guard}\}$	$$\frac{[\ e \mid R, x \leftarrow xs, g\ ]}{\mathsf{concat}\ [\ \mathsf{map}\ (\lambda x \rightarrow e)\ (\mathsf{filter}\ (\lambda x \rightarrow g)\ xs) \mid R\ ]}$$
lcTwoGuards $\{g_0, g_1 \text{ are guards}\}$	$$\frac{[\ e \mid R, g_0, g_1\ ]}{[\ e \mid R,\ \underline{\texttt{if}}\ g_0\ \underline{\texttt{then}}\ g_1\ \underline{\texttt{else}}\ \mathsf{False}\ ]}$$

Figure 5.1: Desugaring of list comprehensions

5.3.3 Desugaring of list comprehensions

GHC resolves comprehensions completely, up to the construction by [] and (:), by traversing the list of qualifiers (the right-hand side of "|") from left to right. Our goal is to base list comprehensions on (parallel) skeletons. As presented here, our rewrite rules in Fig. 5.1 specify the elimination of the list of qualifiers in the opposite order: from right to left. This has two advantages: (1) nested maps are not intertwined with nested concats which preserves structural information; (2) an efficient filter skeleton is used instead of generating lots of empty lists in cases in which guards fail. The disadvantage is that the rules will become far more complicated if extended to the full capability of Haskell.

The rewrite rules shown in Fig. 5.1 cover all possible list comprehension formats in our restricted language. In the figure, R and S represent finite sequences of qualifiers.

The rules prescribe how to replace every list comprehension by applications of the skeletons concat, map and filter which are supposed to have efficient implementations. The rules are applied until no further application is possible. One should prefer an application of rule **lcOptGuard** to any other rule since its application avoids unnecessary evaluations of a predicate.

Rule **lcEmpty** deals with the case that the sequence of qualifiers has become empty by the other transformations.

Rule **lcSinGuard** describes the treatment of a qualifier list consisting of a single guard g. Depending on the value of the guard, the result is a list of either length 1 or length 0.

An application of rule **lcOptGuard** moves a guard located directly behind a generator forward if it does not depend on the generator variable. The guard may only be evaluated if it is ever reached in the original sequence.

Rule **lcXGen** deals with the case that the last qualifier is a generator. The other qualifiers produce a list of environments. In the comprehension before simplification, the last qualifier replaces each element of the environment list by a sequence of new elements. A new element consists of the one replaced plus one particular binding for the generator variable x. The bindings are taken successively from the list xs. After the transformation, this replacement is performed by the expression on the left side of "|". We reuse the name x of the generator variable for the λ-abstraction to preserve the environment for e. Note that the left side is in the scope of the environment defined by the qualifiers on the right side. Therefore, all free variables of xs are bound to the same values as before.

If a guard appears behind a generator, rule **lcGenGuard** helps to fuse the two. It is similar to rule **lcXGen**, except that the new bindings for the generator variable x, which lead to a failure of the guard g, are eliminated from the list via the filter skeleton before. Giving priority to an application of rule **lcOptGuard** assures that the guard really refers to the variable bound by the generator.

If two guards appear next to each other, they can be simplified to a single guard according to rule **lcTwoGuards**.

5.3.4 λ-lifting, let-elimination

All λ-abstractions and let-expressions are eliminated by introducing auxiliary functions (Johnsson, 1985).

1. λ-lifting is the replacement of a λ-abstraction $(\lambda x \rightarrow e)$ by the partial application of a new top-level function f to the free variables $v_0, ..., v_n$ occurring in e, i.e., f is defined by $(f\ v_0...v_n\ x = e)$. Due to this transformation, a restriction of the types of x and e, the so-called *monomorphism restriction*, is not an issue, i.e., f can be used with different type instantiations.

2. let-elimination is the conversion of the definitions in a let-body to new top-level definitions. For moving a definition D to the top level, not only the free variables appearing on the right-hand side of D have to be considered. In the case of recursion inside the let-body, all free variables that appear on right-hand sides of other definitions, required by D, have to be taken into account.

   It is possible to avoid recomputations of values defined non-recursively by v=a in the body of a let-expression, by abstracting the let-expression without v=a, say e', by v and use the λ-lifted form of $((\lambda v \rightarrow e')\ a)$. This works also if v represents a function because functions are later replaced by an appropriate encoding.

5.3.5 Type checking

The type checker checks and infers types in the Hindley-Milner system (Damas and Milner, 1982). It performs a unification of type expressions in a term algebra, in which the function symbols are type constructors, the constants are type constants and the variables are type variables. We use the Martelli-Montanari rules (Martelli and Montanari, 1982) with a check for *infinite types*, i.e., whether a type variable has to be unified with a different type expression that contains this variable.

A simple type class system is implemented by assigning each type variable a set of possible types. The unification of two type variables then involves an update of both sets with their intersection.

5.3.6 Monomorphization

In this phase, all type variables are eliminated and replaced by the types actually used. This requires the duplication of every function for all concrete types it is instantiated with.

Monomorphization is useful for implementing data structures like tuples of atomic data types without overhead of memory space and indirect element accesses. Thus, the target data representation is similar to that used in imperative languages.

5.3.7 Elimination of functional arguments

In this phase, also called higher-oder elimination (HOE), the program is transformed into a semantically equivalent one that does not contain functional arguments. HOE is of major importance to bridge the gap between the functional style the application programs are written in and the imperative style used for the skeleton implementations.

5.3.7.1 Principles

One major difficulty in the compilation of functional languages with static scoping is the implementation of functional arguments. When a function definition like $(\lambda x \rightarrow x * a)$ is passed as an argument, its free variables (here a) refer to the environment of the caller. Thus, the callee must be supplied with a *closure*, i.e., a reference to the function plus the values of these free variables.

Cousineau et al. (1985) translated ML into a nameless λ-calculus, in which values of free variables can be accessed at run time by a so-called *access* function. Appel (1992) described different methods of how to represent the static environment by records.

Bell et al. (1997) proposed a complete algorithm for transforming a *closed*, higher-order and well-typed functional program into an equivalent first-order program. A program is closed if, and only if, a monomorphic main function exists and every function in the program is reachable from it via the call structure of the program.

Previous algorithms were not able to deal with higher-order functions as results (Reynolds, 1972), recursion (Wadler, 1988) or a combination of functional arguments, e.g., by functional composition (Chin and Darlington, 1996). Therefore, we have decided to use the algorithm of Bell et al. (1997) which, quoting them, has the following properties:

1. If a program is well typed according to the Hindley-Milner algorithm, then the transformation results in a well-typed program equivalent to the original program.

2. The transformation always terminates.

3. The transformation of a closed program results in a first-order program.

Nice properties of the algorithm are that it is able to deal with constructor applications and that it performs only as much type specialization as necessary. The algorithm encodes closures by algebraic data types, according to the following principles:

- A variable of a function type changes only its type because, after the transformation, the variable carries the encoding of the closure.

- A functional argument at the side of the caller is replaced by an element of an algebraic data type in which the function identifier is represented by a constructor. The arguments of the constructor carry the values of the free variables in the argument. These values are taken from the context of the call.

- All locations where a functional argument, say f, is applied are replaced by a call of an *apply function* constructed for the respective function type. The first argument of the apply function is the closure of the functional argument, say $\hat{f}$, the following arguments are the arguments f is applied to in the callee. The apply function decodes $\hat{f}$ and obtains a reference to a function f' and the values of the free variables in the environment where f was defined. Function f' equals f except from the fact that the free variables are abstracted by additional arguments. Thus, in the callee, f' is applied first to the values of the free variables, which restores f, and then to the arguments of the application of f in the callee.

5.3.7.2 Rules

Some of the seven rules, which the original HOE algorithm (Bell et al., 1997) is based on, deal with restricting polymorphism and become obsolete in our monomorphic setting. We use four rules, which are described elsewhere in brief (Herrmann et al., 1999a) and in full with a detailed example (Herrmann et al., 1999b).

After the HOE, all function applications are saturated with arguments, such that the result is not a function. In principle, one could now replace all curried definitions and applications by tuple representations. This is not done by the $\mathcal{HDC}$ compiler for two reasons:

1. The tuples, which are objects of the $\mathcal{HDC}$ language, are, in turn, expressed in terms of pattern matching `case`-expressions, which require curried functions on the right-hand side again.
2. The interpreter can remain simpler if it only has to deal with curried functions.

We state the following convention: after the HOE, any application of an $\mathcal{HDC}$ function has to supply all curried arguments. This schema can be regarded as first-order and is equivalent to a schema of tupled arguments.

Optimizations with Γ at a later compiler phase reintroduce higher-order arguments, e.g., the functional composition introduced by rule **map-Γ** in Fig. 4.6. Of course, one could apply these rules before the HOE but then, they would miss the applications that are enabled by the inlining specializations coming later. Thus, there is need for an incremental HOE.

5.3.8 Elimination of mutual recursion

Elimination of mutual recursion is not necessary for producing code, but useful for inlining optimizations and simplification of the size inference.

The $\mathcal{HDC}$ compiler uses two methods for the removal of mutual recursion in programs: *elimination by inlining* and *elimination by emulation*. Mutual recursion is identified by calculating the *strongly connected components* (SCCs) in the graph of functional dependences. Since there is no circular dependence between SCCs, mutual recursion can be eliminated in each SCC individually.

5.3.8.1 Elimination by inlining

Inlining is the process of replacing the call of a function by the function's body. The inlining process is described in detail in Sect. 5.3.12.1.

Mutual recursion can be eliminated by inlining, if the SCC contains a node f whose removal from the SCC would make the residual graph g acyclic. The set of functions which g represents is, in that case, free of mutual recursion. Thus, it is possible to inline all calls of functions in g in the body of f, until the only recursive calls left are directly recursive (Kaser et al., 1993).

5.3.8.2 Elimination by emulation

If mutual recursion cannot be eliminated by inlining, we propose a general method that is based on the emulation of an SCC by a single function. Let f_i $(1 \leq i \leq n)$, be the functions of an SCC and $m(i)$ the number of arguments of function f_i. The functions f_i can all be united in a single function f' by selecting the appropriate function in a `case`-statement; see Fig. 5.2.

To support different signatures of the functions in the SCC, we create, for each function f_i, a constructor C_i to encode the arguments and a constructor

$$
\begin{array}{l}
\forall\, i \in \mathbb{N} : 1 \leq i \leq n : \\
f_i :: t_{(i,1)} \rightarrow t_{(i,2)} \rightarrow \ldots \rightarrow t_{(i,m(i))} \rightarrow t_{(i,0)} \\
f_i\ arg_{(i,1)}\ arg_{(i,2)} \ldots arg_{(i,m(i))} = body_i
\end{array}
$$

$$
\begin{array}{l}
f' :: \mathsf{Data} \rightarrow \mathsf{Data} \\
f'\ arg = \underline{\texttt{case}}\ arg\ \underline{\texttt{of}} \\
\qquad C_1\ arg_{(1,1)} \ldots arg_{(1,m(1))} \rightarrow CR_{t(1,0)}\ body_1 \\
\qquad \vdots \\
\qquad C_n\ arg_{(n,1)} \ldots arg_{(n,m(n))} \rightarrow CR_{t(n,0)}\ body_n
\end{array}
$$

Figure 5.2: Elimination of mutual recursion

$CR_{t(i,0)}$ for the result. The constructor name is used to select the body of function f_i. Finally, calls to the functions f_i have to be adapted to fit f'. Data denotes the union of all algebraic data types used internally by the compiler.

5.3.8.3 Efficiency issues

Whenever possible, elimination by inlining should take precedence over elimination by emulation. Inlining does not spoil the structure of the program and the resulting intermediate code can usually be optimized more effectively.

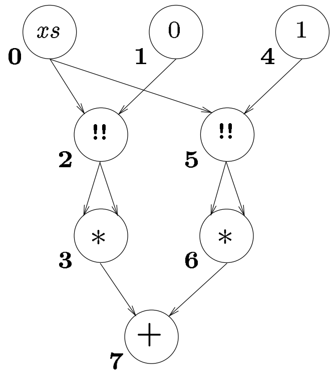
Both methods are expensive if the program contains SCCs with more than three to four functions. Unfortunately, cycles may be introduced by the transformations of earlier compilation phases. The size of the program can grow significantly due to the removal of mutual recursion. Therefore, the user must be able to control this growth, e.g., by annotations or compiler settings. The default setting is to apply elimination by inlining, where possible, and then use the emulation for the remaining SCCs. As stated before, the $\mathcal{HDC}$ programmer should prefer the use of skeletons to user-defined recursive functions in order to keep the amount of recursion low.

5.3.9 `case`-elimination

Complicated forms of pattern matching have already been simplified in a previous compiler phase. Here, we still have to deal with `case`-expressions, for which the left-hand side of each branch either consists of a variable or a constructor with a variable for every argument.

We perform further simplification, because we want to reuse the code optimizations dealing with `if`-expressions in later compiler phases. Thus, we replace every `case`-expression by nested `if`-expressions. Since the arguments of every constructor can only be variables, it is sufficient to compare two constructors, encoded by integers. Each occurrence of such a variable at the right-hand side

sqrsum $\in [\mathbb{Z}] \to \mathbb{Z}$
sqrsum xs = $(xs\,!!\,0) * (xs\,!!\,0) + (xs\,!!\,1) * (xs\,!!\,1)$



0:	xs			:: $[\mathbb{Z}]$
1:	0			:: $\mathbb{Z}$
2:	(!!)	**0**	**1**	:: $\mathbb{Z}$
3:	$(*)$	**2**	**2**	:: $\mathbb{Z}$
4:	1			:: $\mathbb{Z}$
5:	(!!)	**0**	**4**	:: $\mathbb{Z}$
6:	$(*)$	**5**	**5**	:: $\mathbb{Z}$
7:	$(+)$	**3**	**6**	:: $\mathbb{Z}$

Figure 5.3: Example: program, DAG and table

of a case-branch is replaced by an access to the field of the data object selected.

The HOE phase often introduces case-expressions with only one branch. These receive a special treatment: no if-expression is created because the branch always matches.

5.3.10 Generation of intermediate DAG code

The syntax tree of each function is transformed to a DAG to enable sharing of common subexpressions. Each syntax DAG contains a set of expressions with associated numbers, ordered by respecting their dependences. Every subexpression is referenced by its number. Fig. 5.3 shows an example program with its DAG and the corresponding table representation. The direction of the references is inverse to that of the data flow, which is depicted by the arrows in the figure.

The transformation of a syntax tree into a DAG is done by a standard technique called the *value number method* (Aho et al., 1986). It proceeds by a depth-first traversal of the syntax tree, in which the expression at each node receives its number after all of its subexpressions. This number is determined as follows: if there is already a node in the DAG that matches the operation and the operands assigned to the node, then the number of this node is returned; otherwise, a new node is created and its number is returned.

Optimizing transformations (as described below) are performed at this intermediate code level.

5.3.11 Tuple elimination

Tuples are replaced by algebraic data types, one for each occurring tuple type. Each tuple is tagged with the appropriate constructor for its particular type. This simplifies the run-time system and, at the same time, provides fast access to information about the types and sizes of the components of the tuple by looking them up in a table. As a consequence, the memory management functions need not be specialized for each particular type.

However, for tuples in which all elements are of a fixed size, a space-minimal representation is possible which packs all elements in a continuous memory area.

5.3.12 Optimization cycle

Code optimization proceeds in a cycle. Every iteration of the cycle performs three steps in sequence: inlining of functions calls (Sect. 5.3.12.1), rule-based DAG optimizations (Sect. 5.3.12.2) and size inference (Sect. 4.4).

Many of the DAG optimizations are enabled by a previous inlining step. The size inference provides information as a basis for decisions in the next iteration or in the space-time mapping. It has to be applied in every iteration because of changes in the program like expansion or reduction of functions. The cyclic process is finished if an iteration has no effect on the program or the specified limit on the number of iterations is reached.

Every time the transformation of a DAG in an iteration is completed, the body in the corresponding function definition is overwritten with the target DAG.

5.3.12.1 Inline expansion

Here, we continue with a detailed description of *inlining*, which we have sketched in Sect. 5.3.8.1. The inline expansion transforms a source DAG into a corresponding target DAG with possibly expanded calls. First, the nodes of the source DAG are copied successively. If a node representing a function call is reached, a heuristic decision, based on the expected amount of code increase, is made, whether to inline this call or just to copy the call node. In the case of inlining, the copying process switches its source temporarily from the caller to the callee. All nodes of the DAG of the callee are copied. There is no recursive inlining of calls within an iteration, but copied calls may be inlined in the next iteration.

Inlining can enable further optimizations on DAGs:

1. In the DAGs involved in the inlining, common subexpressions are already shared. This sharing is preserved by the inlining. Equivalent expressions in the caller and the callee are shared additionally after the inlining of the call. Thus, inlining may not lead to a duplication of work, but offers the chance to reduce work.

2. The embedding of the body of the callee in the caller enables applications of optimization rules which are based on the special use. Also, partial evaluation and check for value equality can be applied here. The combination of expressions of the caller with expressions of the callee enables optimization rules based on fusion, e.g., if the callee produces a list of which the caller only takes one element.

3. After inlining in the current pass is finished, the DAGs can be simplified. Parameters not used by a called function can be deleted at both the callee's

side and the side of every caller. As a consequence, dead code can possibly be removed from callers. Parameter elimination is not possible if a caller is a skeleton, because a skeleton has a predefined implementation in a different language, e.g., C, which cannot follow these changes.

We apply two major strategies for inlining: *current version inlining* and *original version inlining*.

- Current version inlining.
  The most recent DAG for the called function is inlined. This method requires fewer iterations to get a large amount of inlined code, since DAGs with calls already inlined are used for inlining again. One drawback is that the functions are growing very fast and, therefore, the inlining process may be suppressed after only a few steps.

- Original version inlining.
  The original definition of the function is inlined. This incurs a linear code growth when inlining recursive functions. Original version inlining offers more possibilities for optimization and, therefore, may lead to better results (Kaser et al., 1992).

Kaser et al. (1992) also compare static and profile-based approaches. At present, we do not accumulate or exploit profiling information.

5.3.12.2 Rule-based DAG optimizations

In this phase, the DAG is optimized by applying transformation rules in the direction of the data flow. A simple optimization is, e.g., the exploitation of distributivity by a transformation of $(a*b)+(a*c)$ into $a*(b+c)$. If a + occurs at a node, the optimizer checks whether both references point to a node with a $*$ and, if these nodes share the first operand, replaces the part of the DAG computing $(a*b)+(a*c)$ by a DAG which computes $a*(b+c)$, provided that the subexpressions $a*b$ or $a*c$ are not required elsewhere. Minimization of the entire weighted amount of operations in a DAG is not addressed at the moment.

DAG optimizations can reduce time and space consumption significantly. Assume that, of a large list specified in the program, only a few elements are required. In this case, the list should not be generated entirely but the elements should be computed individually, making use of the function which constructs the list. For instance, an indexed arithmetic sequence $[a\,..\,b]\,!!\,i$ can be replaced by the value $a+i$. The Γ-calculus presented in Sect. 4.4 can be used to perform such optimizations on lists, but it is not implemented yet. One could think of the risk that the sharing of intermediate results in a list computation is lost if some of its elements are computed separately. However, this risk is quite small for the following reasons:

1. Lists produced by skeletons remain because skeleton implementations cannot be transformed by the compiler.

2. If many elements are required, it is likely caused by a generator. In this case, the computation of the list requested is just fused with the computation induced by the generator. DAG optimizations exploit common subexpressions; thus, no duplication of work should occur.

3. Lists produced by non-skeleton function calls can only be eliminated if the function is expanded. In this case, we can exploit common subexpressions, again.

5.3.13 Abstract code generation

The intermediate code is represented by a set of DAGs, one DAG for each function in the program. As described in Sect. 5.3.10, each DAG is represented by a table: each node of the DAG corresponds to a table entry, and each directed edge of the DAG is represented at the entry of the target of the data dependence by the table index of the source of the data dependence.

During the phase of abstract code generation, the interpretation of the DAGs changes: before, they are interpreted with a denotational semantics, afterwards with an operational semantics. A few node types of the DAGs also change: one type of node is eliminated and three other types are introduced.

Under the denotational interpretation, a DAG is evaluated by starting at a distinguished node of the DAG, the *root*, so called because it originated from the root of the syntax tree. The result of the root is considered the result of the function represented by the DAG. If the evaluation of a node requires the result of another node, this node is visited and evaluated. There is a special kind of node for accessing formal parameters. If a node representing a function call is visited, first its actual parameters and then the DAG of the called function are evaluated. `if`-nodes require a special treatment: the value of the condition has to be tested, and then only one of both branches is evaluated.

Under the operational interpretation, the evaluation proceeds by traversing the table entries in increasing order. If the result of another node is required, it has already been computed and can be looked up in a preceding table entry. The root node is the last entry in the table and contains the result of the function. The problem with the `if`-nodes is that when they are reached (if ever!) both branches already would have been evaluated, also the branch not to be selected. Therefore, a mechanism is implemented to skip nodes belonging to the wrong branch. If a DAG does not contain `if`-expressions, it is used as abstract code without modification.

The DAG nodes in the table are rearranged to avoid unnecessary evaluations due to nested conditionals and to reduce the time frame objects have to reside in memory. The $\mathcal{HDC}$ report (Herrmann et al., 1999b) presents the generation of abstract code in more technical detail and an example.

5.3.14 Space-time mapping

The space-time mapping of $\mathcal{DC}$ has been described in Chapter 4. In the $\mathcal{HDC}$ compiler, only the results of skeleton transformations are available yet. The optimization of $\mathcal{HDC}$ program parts aside from skeletons is much more complicated and still not undertaken.

We believe that space-time mapping is most effective when applied in the individual derivation of parallel skeleton implementations. If the dependence structure of a skeleton is sufficiently regular, e.g., for some kinds of $\mathcal{DC}$, the points of the computation domain can be laid out in time and space at compile time. The size of the computation space and even its dimensionality can depend on the problem size. Despite this fact, the computation domain can be defined without prior knowledge of the problem size, as we have seen in Sect. 4.3.3.

The user is well advised to construct the program by a composition of appropriate skeletons, which have efficient implementations. Note that the generation of each skeleton is done by a Haskell function which is to be delivered by the skeleton implementer. It is up to this function to use the results of the size inference provided or even to call external tools. The task of the $\mathcal{HDC}$ compiler is, at a minimum, to transmit symbolic space-time mapping information via the call structure of the program to the points where it is needed, by making use of the abstract functions delivered by the skeleton implementer.

The DAG nodes are scheduled sequentially by the $\mathcal{HDC}$ compiler in the current version; no parallelization is done here.

5.3.15 Target code generation

The code generation phase first produces C code, which is then compiled with a standard C compiler and linked together with the functions of the $\mathcal{HDC}$ run-time library, which are also written in C. The C code is generated in two phases. First, the abstract code of the user program is translated; see Sect. 5.3.15.1. Second, an appropriate implementation is generated for every skeleton instance used in the program; see Sect. 5.3.15.2.

5.3.15.1 DAG compilation

For every DAG of the abstract code, a C function is generated. The nodes in the DAG are treated separately.

The code for a node can be a jump or a small sequence of statements. In the latter case, the node is associated with a variable of either (1) a basic type, (2) the union of all algebraic data types or (3) the union of all list types. The restricted polymorphism of (2) and (3) at the levels in the type structure below the topmost level is possible due to safe type-casting of pointers in C. The actual type of an object a pointer refers to is recovered as soon as the object is used; otherwise, it is of no interest.

The sequence of statements consists mainly of applications of single operators, memory allocation, reference counting, calls of auxiliary functions and assignment of a value to the node's variable. Variables are reused. Typically, a function with, say, 200 nodes does not require 200 variables but, e.g., 10. This enables large recursion depths, which can occur if explicit recursion is used instead of skeletons. A variable for a list or an algebraic data type only carries a pointer to a dynamic data structure and, thus, permits sharing. Assignments of those structures are only assignments of references. The reference counter is located exactly at the address the pointer points to. Thus, updating the counter when copying a reference is fast.

Jumps occur where inappropriate branches of conditionals have to be skipped. They are implemented using the `if` and `goto` commands of C. Advanced program structures are not required. Thus, a generation of assembler code would also be possible, but exclude portability and optimization by the C compiler.

Every new constructor encountered during the DAG compilation is inserted into a table available to the run-time environment, which carries information about types and sizes of the components of the corresponding object. This information is required if the memory allocated for an object is to be released and for the marshaling, i.e., the packaging of dynamic shared data structures into a byte array for communication.

For every call of a skeleton, the name of the skeleton is stored together with the current types of the arguments. This information is required for the generation of the appropriate skeleton type instances we describe now.

5.3.15.2 Skeleton generation

After all DAGs have been processed, the type instantiations of the skeletons are generated and stored.

$\mathcal{HDC}$ offers a special, very flexible mechanism for the integration of custom-implemented skeletons. For every skeleton, the implementer delivers a Haskell function, say Φ, which is called by the code generator of the $\mathcal{HDC}$ compiler and which produces the actual instance of the skeleton. In the simplest case, the body of function Φ will be just a Haskell string of C target code, but Φ can also prescribe decisions based on type and size information provided by the compiler.

Remember that the C code generated must be monomorphic; this applies also to the implementation of a skeleton. The implementer of Φ has to consider at least the root symbol of the type tree. The implementation must differ, e.g., between lists and integers, but not necessarily between lists of integers and lists of lists. In the latter case, the root of the type tree is in both cases the list type constructor.

To illustrate what a parameterized skeleton implementation may look like, let us sketch an implementation of the `map` skeleton in a parallel model, in which all data is passed along with the control. The details of the run-time system supporting this implementation are presented in Sect. 5.4.4. The `map` skeleton takes a function (really a closure, i.e, the code of a function together with an environment) and a list and applies the function to every list element.

For simplicity, we assume a space-time mapping which allots roughly the same number of list elements to every processor. This mapping is efficient only if all elements of the list require nearly the same amount of work and space.

One might consider the use of collective MPI operations (MPI Forum, 1995) like *broadcast* (to distribute the function closure), *scatter* (to distribute the list among the processors) and *gather* (to collect the results from the processors). This would work for lists of Int, Bool or Double, with a special skeleton implementation for each type. In general, *gather* and *scatter* cannot be used, since they assume that lists are plain and do not contain references to a heap. E.g., if the elements are functions, the list contains just pointers to a shared part of the heap. As a consequence, we have to custom-implement collective operations for $\mathcal{HDC}$, using the MPI primitives *send* and *receive*. For the implementation of these collective operations, the divide-and-conquer approach proves to be useful, too.

5.4 The parallel run-time environment

5.4.1 The model of parallel execution in $\mathcal{HDC}$

We want to provide a platform which does not limit the design choices concerning $\mathcal{DC}$ parallelism, but we are restrictive enough to exclude unstructured fork-join parallelism (Almasi and Gottlieb, 1989). In the interest of generality and scalability, our programming model is SPMD.

Unfolded recursion can be viewed as a call tree in which the root represents the entire problem instance and in which, for each node, child i corresponds to subproblem i. Time proceeds down the tree: each level corresponds to one parallel computation step.

At the start, all processors are assigned to the root of the call tree. Each node partitions its processors blockwise among its children. We call the set of processors belonging to a node a *block* and the set of processors of each of its children a *subblock*. Every block has a distinguished processor we call the *master*, which is responsible for the management of the block. In problem division, the master of a block activates the masters of the subblocks and, if the computations of the subproblems have been finished, the masters of the subblocks become idle.

During the time in which each master processes the problem instance assigned to it, no control message is sent or received by any processor of the master's block across the block's border. Other messages, e.g., for accesses to globally distributed data aggregates, are not restricted. We call this the principle of *control-closed blocks*, in analogy to the principle of *communication-closed layers* (Elrad and Francez, 1982). Its goal is to impose a structure on the communication such that deadlocks or races are avoided.

Another issue is the data layout, i.e., the way in which the data is distributed among the processors. We distinguish three kinds of data layout. Centralized layout is always applied to elements of atomic data types. Only tuples, lists and algebraic data types, which can become large, can also be subject to the other distributions.

- **Centralized data layout:** Input data of a task is passed along with the signal of initiation of the task and output data is passed back with the report on the task's completion. Obviously, this is a good choice if the amount of data is small, even if it incurs some unnecessary communications. For large data, a centralized data layout will lead to unacceptable overhead, due to long data transmission times, or even to memory overflow.

In the remaining two layout styles, instead of passing the data with the control, the information about the location of the data is known at compile time or passed.

- **Hierarchical data layout:** The input and output data of each block is distributed among the processors assigned to the block, as prescribed by the space-time mapping. The default space-time mapping is that the data is distributed in balance. Hierarchical layout is especially convenient for $\mathcal{DC}$ algorithms on large data which does not fit onto a single processor, but only if the data size decreases with the division of the problem, as enforced by the dcF skeleton. Really, the implementation of the dcF skeleton uses only a subset of the block's processors for the distribution of input and output data, according to the data division degrees.

- **Globally distributed data layout:** The input data is distributed according to a space-time mapping. Each intermediate and result value is located on the processor that produced it, according to the *owner-computes rule* (Wolfe, 1995).

  A major difference to the centralized data layout is the data access mechanism: instead of a large volume of data, only a reference to the data is passed. Remote data retrieval is realized by direct access to the remote memory. Remote access can only be reading, not writing, except from organizational accesses like updating reference counts. Using reference counting (Sect. 5.4.3), a data object is preserved at a particular location in memory as long as there exists a reference to it. The principle of control-closed blocks does not rule out access to globally distributed data, if we use the model of a virtual shared memory or one-sided calls in MPI (MPI Forum, 1997), since the synchronization schema is not affected in these cases.

5.4.2 Run-time control over skeletons

The available implementations of a skeleton determine the set of possible space-time mappings that can be chosen in a parallel execution. Thus, skeleton implementations are generated in dependence of the context in which they are called, exploiting type and possibly also symbolic size information (see Sect. 5.3.15.2).

The first argument of every function in the target code, including skeletons, contains a pointer to system information, comprising a description of the master and the part of the topology the processor belongs to.

The skeleton implementer can enrich skeletons with an optional functional argument which contains a mapping strategy coded in $\mathcal{HDC}$. Some of its arguments refer to values of the application program, others refer to values of the current system information. Then, the user with experience in parallel programming is provided with a mechanism to control part of the mapping explicitly. This idea was inspired by the work of Hudak (1986).

5.4.3 Memory management

The memory for the local variables of functions is stored on the stack, which results from a direct translation to C. Lists and algebraic data types, for which a call-by-value parameter passing mechanism would be too inefficient, receive special support by our run-time system. They are passed as pointers to heap-allocated structures. Common subexpressions are shared, which means that the linked structures form a DAG. Since lists are represented by arrays, a common final segment of two different lists cannot be shared.

By default, each processor manages its own local heap using reference counting for release (Deutsch and Bobrow, 1976), which is possible due to the absence of cyclic data structures.

Distributed data structures have to be maintained explicitly by skeleton implementations, using functions of the run-time system for remote memory access. Arguments passed to another processor are marshaled in their entirety. It remains the task of program optimization not to pass a large data object to a function if only a small part of it is actually required.

Every list or algebraic data type consists of a descriptor, which contains a reference counter as well as constructor or length information. The substructures are represented by pointers, if they are lists or algebraic data types themselves, and plain otherwise. The reference counter counts the number of direct references that point to that structure. If the number becomes 0, the reference counters of all substructures are decremented and the cell which represents the top of the structure is released. Reference counting is fast and also has the advantages, that (1) data has not to be moved in memory, which enables a direct access by other processors, and (2) a *destructive update* (in-place modification) can be performed if the number of references is 1. O'Donnell (1985/1993) proposes ways to deal with the element update problem of array-like data structures in functional languages.

Weighted reference counting (Bevan, 1987; Watson and Watson, 1987) can be used in the case of remote access to data structures. A processor P which establishes a reference to a remote data object reserves a certain number of reference counts on the remote processor. Then, P can lend out or consume these counts without the need to update the original counter.

5.4.4 Interaction with skeleton implementations

If the computation is hierarchically divided into subcomputations, the block of processors assigned to this computation is divided into subblocks. Each processor belongs to exactly one subblock. Each subproblem is solved on its own subblock. At the beginning, the block has one master processor, the other processors are *slaves*. In the division of a block, the master turns some of its slaves into new masters, one particular for each subblock, except from the subblock the master manages itself.

Let us now revisit the implementation of the map skeleton from Sect. 5.3.15.2. If the list does not contain at least two elements or the block has only one processor, map must be computed sequentially. (map *may* be computed sequentially if parallelization does not pay off according to a strategy chosen by the skeleton implementer.) Otherwise the block is divided into two subblocks; let us call them the *left* subblock and the *right* subblock. The left subblock computes map on the left part and the right subblock on the right part of the list. The master of the block, say, L, retains the responsibility for the left subblock and sends the packed closure of the function to be mapped and the right part of the list to another distinguished processor, say, R responsible for the right subblock. Computation proceeds recursively until the left and right subblock are united again and R gives back control for its part to L.

Now, let us have a closer look at the call mechanism. Processor L is the one on which the map skeleton is called. Thus, it receives all formal arguments via a function call. Processor R is activated by L with an index of the actual skeleton instance. R uses this index to look up the pointer to the *slave skeleton*. This skeleton does not receive the application data via a function call but awaits the data from L. L sends the data to R, both L and R call the master skeleton with their particular subproblem, R returns into the slave skeleton and sends its result back to L. Note that Φ of Sect. 5.3.15.2 has to generate two skeleton instances here: the one for the master and the one for the slave!

5.5 Outlook

Aside from the parser, which is produced by the tool *happy*, the current implementation of the $\mathcal{HDC}$ compiler has been developed from scratch. We decided so, because we did not want our implementation decisions to depend on the availability of existing software. Inspired by the implementation of the language Eden (Breitinger et al., 1997), our current point of view is that the Glasgow Haskell compiler (GHC) could, possibly, be used to extend the capabilities of $\mathcal{HDC}$ to a larger part of Haskell. We do not want to go as far to use the code generation of GHC, because of the following reasons:

- Referring to Finne et al. (1999), we assume *stable pointers* to be the current exchange mechanism for passing functional closures of GHC compiled code via a C interface. With this schema, it is still not possible to exchange functional closures between different heaps of a distributed memory machine. Note that Glasgow Parallel Haskell (GpH) uses a single, distributed heap, but not separate heaps. The advantage of the schema implemented by GpH is global graph sharing, the disadvantage is the large set of data dependences between processors.

- Our target language, C+MPI, is implemented on a wider range of systems than the special libraries which the Haskell compilers require. This turned out to be useful for producing code for the transputer.

However, we possibly could use parts of the front end of GHC:

1. It would be the task of our modification of the GHC parser to convert annotated types and objects into legal Haskell source code. E.g., if a (:) operator is annotated to have the standard Haskell list implementation, nothing changes. Otherwise, (:) has to be converted to fit a predefined $\mathcal{HDC}$ list type and conversion functions have to be applied to the operands and the result. List comprehensions can cause some complications here.

2. The GHC compilation stops with the generation of the intermediate code in the Haskell core language. This intermediate code contains polymorphic higher-order functions and, thus, has to enter the $\mathcal{HDC}$ compilation before the monomorphization phase.

By these decisions, most parts of the existing $\mathcal{HDC}$ compiler still remain, as the current state of the $\mathcal{HDC}$ compiler provides an implementation of polymorphic higher-order functions. The connection with GHC can add the programming comfort of Haskell, especially the type class and instance declarations.

We see the need to provide a code generation which is tailored for massive parallel execution. Especially, a tradeoff between sharing of data and a sufficient amount of independence has to be made. At present, we eliminate higher-order functions and use an imperative, call-by-value function call mechanism because we do not see how space-time mapping of skeletons can be combined well with lazy graph reduction.

Hammond (1999) pointed out the danger of a potential loss of sharing in the case of several independent heaps. However, this is critical only if multiple copies of a large data object are received by a processor via different communications. We consider the use of stamps that indicate the processor and the time of generation to identify data objects. Then, every processor has to maintain a mapping of stamps to locations in its own memory to enable a reuse.

Chapter 6

Experimental Results

Our experimental results demonstrate that the methods presented provide the potential for significant speedups in some cases, but also that the techniques used have to be improved to provide good results for a large class of algorithms.

Preliminary experiments with skeletons written in C have been performed by Musiol (1996), in cooperation with the author of this thesis. The speedup results obtained for the parallelization of a special form of dcF were very encouraging. However, the comfort of programming with C skeletons by using their implementation directly is low since the user has to worry about memory management, encoding of customizing functions and data type adaptations.

The introduction of the functional programming style into the skeleton framework introduces some overhead, due to a larger amount of procedure calls, memory management operations and copying. For an estimation of the quality of the code produced by the $\mathcal{HDC}$ compiler, we provide, in the first two examples, also the sequential execution times of the program compiled by GHC. In the first example, we also compare with handwritten C programs.

We have used two parallel machines: a 1024-processor transputer network Parsytec GCel-1024 (Inmos T805 30MHz CPU, 4MB of memory) and a cluster of six workstations (Sun UltraSPARC-IIi 300MHz CPU, 128–256MB of main memory), connected by a 10-Mb Ethernet.

The transputer network was chosen as a machine designed for parallel computation. The fact that it is roughly ten years old is irrelevant for our studies of the relative speedup. We are mainly interested in the impact of the program and its parallelization on the speedup and, thus, a machine with a good communication behavior, like the transputer network, is most appropriate. However, one has to be aware of the fact that the communication-by-computation ratio of some modern parallel machines is worse than that of the transputer network and, thus, will result in smaller speedups. A few experiments made within our group with the Karatsuba example on the Siemens hpcLine confirm this. (These results are not presented here.)

The experiments on the workstation cluster demonstrate that moderate speedup results can even be gained with the machine equipment of a standard working environment. Due to variations in the execution times, the numbers presented for the workstation cluster are taken as the arithmetic mean of the numbers measured in three program executions.

Now, let us explain the structure of the tables with which we present the results. Columns labeled "abs" contain absolute execution times in seconds. In columns labeled "rel", we list the relative speedup, which is the ratio of the absolute time on one processor by the time on the particular number of processors chosen.

The column labeled "#p" refers either to the number of processors used, or contains, only in the sequential case, the label "user" or "real".

A row containing "user" refers to the amount of CPU time the application process consumed. If it contains "real", it refers to the wall-clock time difference between start and end of the process. Both times include I/O.

A row labeled with a number n contains the execution time of a parallel execution on n processors, measured as wall-clock time difference with the MPI timer `MPI_Wtime()`, on the only processor which performs I/O. The times for I/O are not included, but the distribution of the input and the collection of the output data are.

The row label "GHC" refers to code compiled with the Glasgow Haskell Compiler (Haskell, 2000), V.4.06 with the optimization flag `-O2`. If not stated explicitly, everything else was compiled with the current $\mathcal{HDC}$ compiler, with the default optimization settings, and subsequently with the GNU C compiler (V.2.7.2.3), with optimization flag `-O2`.

Between the rows of the tables, the machine, compiler, skeleton and processor number vary; between the columns, the program and input vary.

In each of the following sections, we present one application example: Karatsuba's polynomial multiplication (Sect. 6.1), the n queens problem (Sect. 6.2), the maximum independent set problem (Sect. 6.3), sorting (Sect. 6.4) and the computation of the convex hull of a set of points (Sect. 6.5). The first two examples have been investigated on both machines and compared with GHC. The other examples were only executed on our transputer network.

The programs we used for the experiments differ from those presented in Chapter 3, since their purpose is not to provide an elegant description of the algorithm, but to avoid expressions that the $\mathcal{HDC}$ compiler does not translate efficiently at present. For a comparison, GHC is given a program with similar simplifications. The programs used in the experiments can be found in App. D.

Sect. 6.6 compares the different speedup results obtained.

6.1 Karatsuba's polynomial multiplication

In (Lengauer et al., 1997), we presented different ways to parallelize polynomial multiplication if the problem is divided into four subproblems. In contrast, the Karatsuba algorithm reduces a multiplication of polynomials of sizes $n = 2^m$ ($m > 0$) to three multiplications of polynomials of sizes 2^{m-1} (Aho et al., 1974; Herrmann and Lengauer, 1999). Thus, the depth of recursion is m and the number of basic cases created is 3^m. This number determines the complexity, which is $\Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3})$, where $\log_2 3 \approx 1.58$. We expect the sequential execution time to increase by a factor of 3 when doubling the operand size. The experiments with this algorithm are using a program based on skeleton dcA and a program based on dcF.

To demonstrate the difficulty of outperforming an imperative program with a functional program, consider the following central part of a loop program. It multiplies two polynomials whose coefficients are stored in the arrays a and b and delivers the coefficients of the result in array c.

```
for(i=0;i<2*n;i++)
   c[i] = 0;
for(i=0;i<n;i++)
   for(j=0;j<n;j++)
      c[i+j] += a[i]*b[j];
```

We refer to the execution time of this program in the rows labeled "C_lo". A handwritten sequential C program for the Karatsuba algorithm is given in App. D.1.4 and referred to in the rows labeled "C_ka". It outperforms the loop program for n$\geq$4096 on the Sparc, because of its superior time complexity. On the transputer, C_ka is superior to C_lo not below the problem size 16384. We guess that this effect is caused by the small size of the transputer cache, which is a bottleneck for function calls.

As can be seen in Tab. 6.1, on the transputer, for a problem size of 32768, the dcF code is faster than the handwritten C program (C_ka) using 27 processors, by a factor of more than 2. The loop program (C_lo) is already outperformed with 9 processors.

We show only experiments with numbers of processors which are powers of 3, since this is convenient for the parallel execution. For dcF, choosing a different number would give the same result as choosing the largest power of 3 smaller than this number. For dcA, a number of processors different from a power of 3 can even lead to a slow down, since the current version of the $\mathcal{HDC}$ compiler is not yet able to choose parallelism only where it is efficient. The dcF skeleton was just developed to exploit the parallelism in the customizing functions in a controlled way.

The program compiled with GHC does not make use of skeletons because this would slow down the computation. For the GHC-compiled code, it was necessary to establish a heap of size 100MB/300MB for a problem of size 4096/8192. Possibly, the low speed of this code is due to problems of the evaluation strategy. The program given to GHC is quite simple but it is not straightforward how the list operations can be made more efficient. For the problem sizes 16384 and 32768, the GHC-compiled code would require a heap size of more than 600MB, which our environment could not provide. Also, for a problem size of 32768, the dcA program requires more memory than the 4MB of the T805.

Fig. 6.1 displays the speedups for dcA and dcF. dcF is superior to dcA by

machine		operand size							
prog	# p	4096		8192		16384		32768	
		abs	rel	abs	rel	abs	rel	abs	rel
cluster									
C_lo	user	1.9		7.5		30.0		122	
	real	2.0		7.5		30.1		123	
C_ka	user	1.6		4.8		14.3		43.1	
	real	1.6		6.9		14.5		43.3	
GHC	user	34.7		104		out of heap space			
	real	40.7		1102					
dcA	user	46.9		135		407		1219	
	real	48.1		137		413		1227	
	1	46.7	1.00	135	1.00	408	1.00	1225	1.00
	3	18.6	2.51	47.8	2.82	147	2.78	445	2.75
dcF	user	20.2		59.8		177		543	
	real	23.7		61.4		179		548	
	1	22.3	1.00	59.3	1.00	177	1.00	545	1.00
	3	7.8	2.86	20.7	2.86	62.6	2.83	186	2.93
transputer									
C_lo	1	85.3		341		1365		5460	
C_ka	1	137		411		1235		3705	
dcA	1	3650	1.00	10943	1.00	32796	1.00		
	3	1229	2.97	3674	2.98	11015	2.98	memory	
	9	416	8.77	1239	8.83	3703	8.86		
	27	144	25.3	425	25.7	1258	26.1	allocation	
	81	54.5	67.0	154	71.1	443	74.0		
	243	25.8	141	65.1	168	174	188	failed	
	729	26.8	136	43.6	251	89.7	366		
dcF	1	1582	1.00	4749	1.00	14258	1.00	42806	1.00
	3	536	2.95	1605	2.96	4833	2.95	14395	2.97
	9	186	8.51	552	8.60	1638	8.70	4873	8.78
	27	67.3	23.5	194	24.5	568	25.1	1678	25.5
	81	28.0	56.5	75.1	63.2	210	67.9	598	71.6
	243	14.8	107	34.9	136	88.1	162	235	182
	729	MPI error: truncated msg.				51.0	280	118	363

Table 6.1: Execution times for Karatsuba's polynomial multiplication

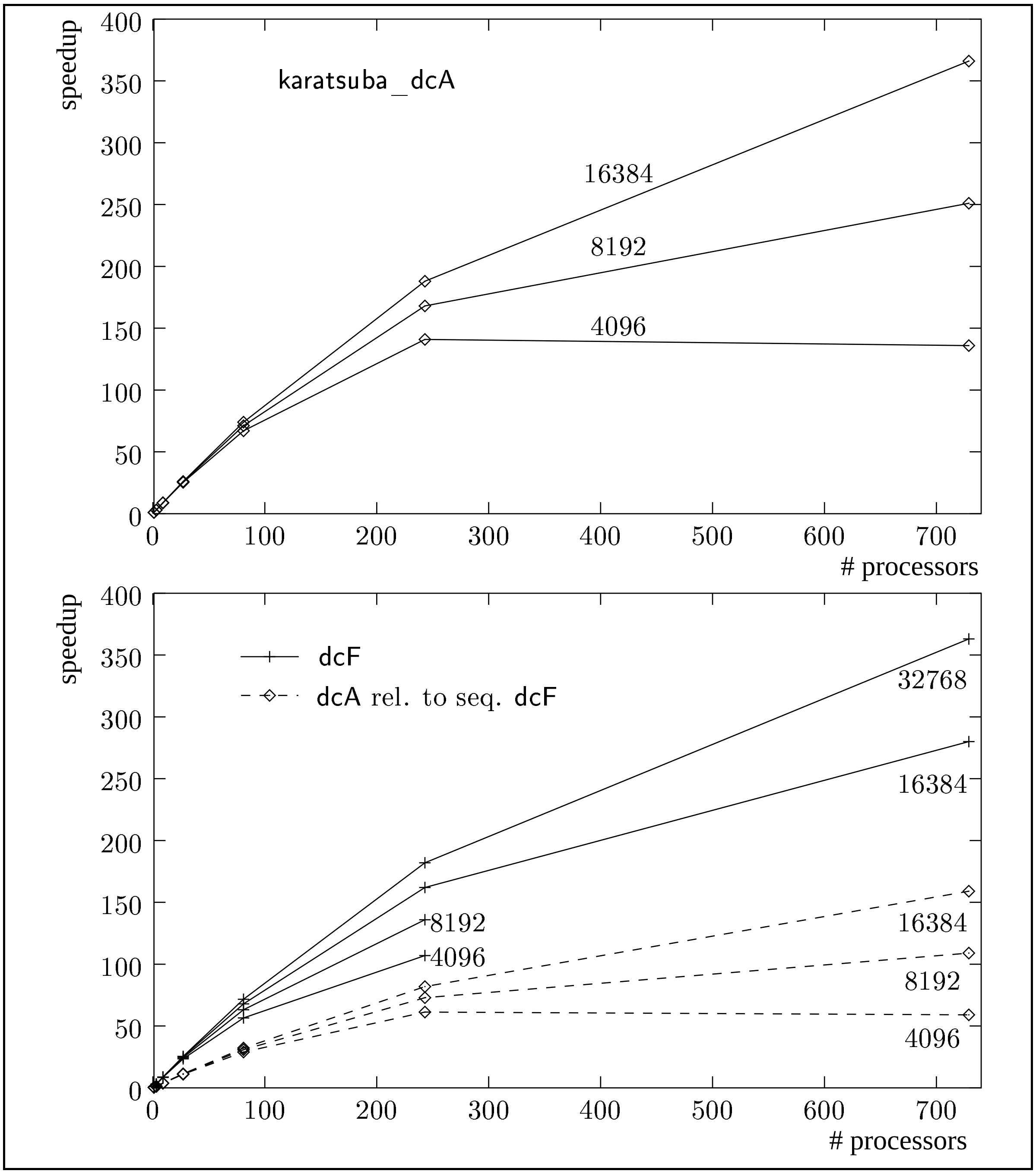


Figure 6.1: Relative speedup for karatsuba (transputer)

a factor of roughly 2. However, this factor was already present in the one-processor times where the dcF implementation benefits from destructive array updates in C.

We expect that the increase in speedup would be superior for dcF if the I/O data were distributed. In the current version, this is not the case. Thus, the time for the initial distribution of data and the time for the final collection of it, which are more expensive than the applications of the divide and combine function, are measured also for dcF. However, if the I/O data of dcF were distributed, a comparison of the times for dcA and dcF would be dubious.

6.2 n queens problem

The program for the n queens problem was taken from the Hugs distribution (Haskell, 2000) and computes the actual placements. It was adapted to fit skeleton dcA for $\mathcal{HDC}$ compilation. (The n queens problem even fits skeleton dcB, but dcB is not implemented yet.) The parallelization of the list comprehensions in program queens, presented in Ex. 3.5, leads to bad results. To avoid this parallelization, we transformed the list comprehensions manually. The result (queens_dcA) is based on a sequential version of map (seqmap).

In our earlier work, we observed slowdowns for two or three processors (Herrmann and Lengauer, 2000). This was caused by the following problem: if the number of processors is smaller than the number of subproblems, further division is done in sequence. However, by proceeding down the call tree, the number of possible placements for a queen shrinks. Beyond a particular level, the number of subproblems becomes small enough and parallel execution ensues. This is inefficient, because many times small-sized tasks are created.

To avoid this, we use the definition of dcA in the source language, based on a parallel map. The implementation of map was designed to distribute the problems equally among the processors. In contrast, the implementation of dcA was originally designed to distribute the processors among the problems, which does not hurt as long as enough processors are available.

The results of the version based on recursion and map are displayed in Tab. 6.2 and Fig. 6.2. We can see that an efficiency of at least 50% can be achieved, for 10/11/12 queens upto 32/64/128 processors.

The experiments show superlinear speedups for 12 queens on the transputer network. We guess that this is due to cache effects, as mentioned in Sect. 6.1.

For the future, we plan an implementation of the itA form of Sect. 4.7.1, which should provide (1) good load balance, (2) control over granularity and (3) efficient memory use.

6.3 Maximum independent set

The program for the computation of the maximum independent set, adapted from Ex. 3.4, is applied to four regular graphs with long paths that have to

machine		q = # queens							
prog	# p	10		11		12		13	
		abs	rel	abs	rel	abs	rel	abs	rel
cluster									
GHC	user			5.8		34.6		218	
	real			5.9		35.0		220	
dcA	user			17.6		94.2		539	
by	real	times		19.5		98.4		544	
rec.	1			17.5	1.00	94.2	1.00	539	1.00
map	2	too		10.1	1.73	50.3	1.87	297	1.81
	3			8.1	2.16	48.8	1.93	251	2.15
	4	small		5.7	3.07	27.6	3.41	191	2.82
	5			5.5	3.18	25.8	3.65	141	3.82
	6			5.7	3.07	28.9	3.26	158	3.41
transputer									
dcA	1	308	1.00	1712	1.00	15581	1.00		
by	2	154	2.00	909	1.88	6070	2.57		
rec.	3	152	2.03	720	2.38	5564	2.80		
map	4	93.3	3.30	447	3.83	2765	5.64		
	5	93.3	3.30	447	3.83	2765	5.64		
	6	62.4	4.94	447	3.83	2809	5.55	times	
	8	62.3	4.94	295	5.80	1739	8.96		
	12	32.7	9.42	151	11.3	850	18.3	too	
	16	32.4	9.51	149	11.5	833	18.7		
	32	19.2	16.0	79.4	21.6	472	33.0	large	
	64	12.1	25.5	50.3	34.0	298	52.3		
	128	8.12	37.9	38.0	45.1	212	73.5		
	256	8.69	35.4	27.1	63.2	140	111		
	512	9.58	32.2	22.3	76.8	101	154		

Table 6.2: Execution times for queens_dcA

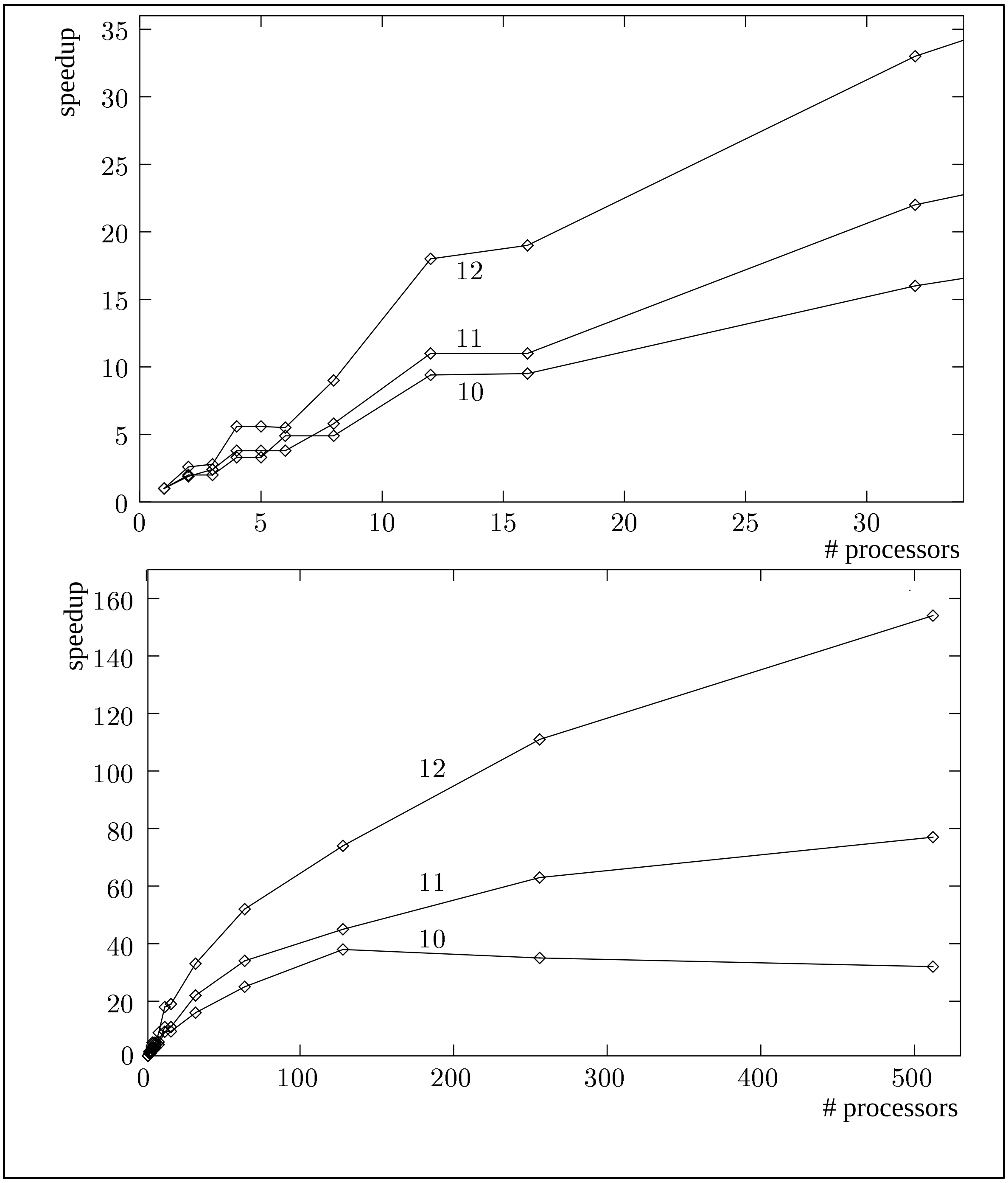


Figure 6.2: Relative speedup for queens_dcA (transputer)

# p	ring-30		chord-30/5		torus-5×5		HC-5D	
	abs	rel	abs	rel	abs	rel	abs	rel
1	316	1.00	189	1.00	28.1	1.00	152	1.00
2	201	1.57	148	1.27	21.7	1.29	119	1.28
4	116	2.72	118	1.60	15.9	1.77	88.5	1.72
8	66.9	4.72	94.8	1.99	11.5	2.44	66.0	2.30
16	38.9	8.12	55.8	3.38	7.7	3.65	50.7	3.00
32	22.6	14.0	32.9	5.74	5.2	5.40	36.6	4.15
64	13.5	23.4	17.9	10.6	3.6	7.81	27.3	5.57

Table 6.3: Execution times for max_independent_set_dcA (transputer)

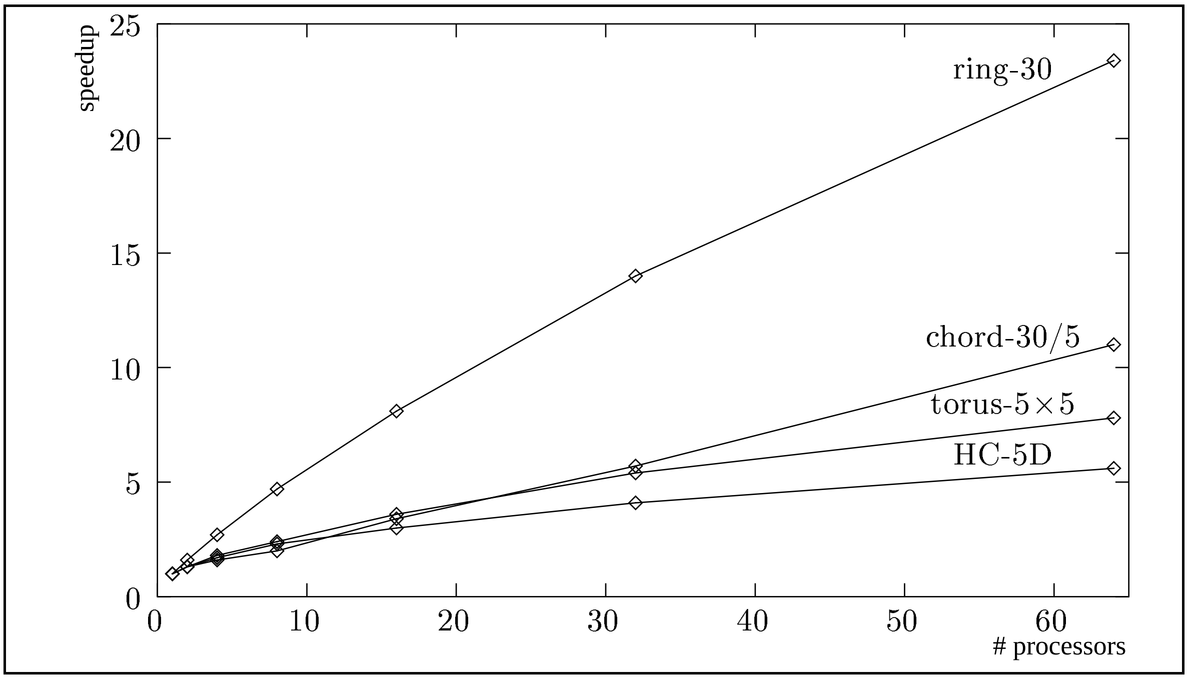


Figure 6.3: Relative speedup for max_independent_set_dcA (transputer)

be traversed: a ring with 30 nodes (ring-30), a chordal ring with 30 nodes and chords of length 5 (chord-30/5), a 5×5 torus (torus-5×5) and a hypercube with 5 dimensions (HC-5D). The results are listed in Tab. 6.3. The low speedups depicted in Fig. 6.3 are due to a load imbalance, exposed by the different number of memory allocations of the processors (not shown here). A redistribution of work at particular levels of the call tree seems to be a solution for this problem.

In contrast to the n queens algorithm, which delivers all solutions, the maximum independent set algorithm delivers only a single solution. Therefore, the branch-and-bound paradigm is probably more appropriate for it than $\mathcal{DC}$.

6.4 Sorting

We present the results for two sorting algorithms: (1) qsort_dcA, a modified version of qsort from Sect. 3.4.1; (2) a modified version of bitonic sort from

# p	qsort_dcA		bitonicOpt	
	abs	rel	abs	rel
1	101	1.00	101	1.00
2	70.7	1.43	249	0.41
4	59.4	1.70	232	0.44
8	53.1	1.90	174	0.58
16	46.7	2.16	126	0.80
32	40.8	2.48	88.1	1.15
64	39.8	2.54	66.3	1.52
128	39.2	2.58	58.6	1.72
256	39.0	2.59	53.0	1.91
512	39.4	2.56	49.7	2.03

Table 6.4: Execution times for sorting 32768 numbers (transputer)

Sect. 3.5, named bitonicOpt, since bitonic sort is extremely inefficient if executed sequentially. The modification is that bitonicOpt calls qsort_dcA in the sequential case. The results for qsort_dcA depend strongly on the particular input data. The one chosen was generated by the following output action in Haskell: writeFile "input" (show [$(i*i)$‘mod‘10007 | $i\leftarrow$[0..32767]])
For bitonicOpt, the depth of recursion is controlled by an additional number at the beginning of the input file.

The results on the transputer network are shown in Tab. 6.4; the speedup development is depicted in Fig. 6.4. Unfortunately, the speedups we obtain are very bad, which is due to the fact that the sequential execution is already very fast. However, programs which are developed manually for particular machines are able to achieve good speedups even for sorting (Heidelberger et al., 1990; Brown and Xiong, 1993).

Although qsort_dcA is superior to bitonicOpt with the input data chosen, it can behave significantly worse in the case of load imbalance. This imbalance cannot occur when using bitonicOpt.

The bitonic sort provides a high potential for speedup. We believe that the future dcD version of bitonic sort with distributed I/O will outperform qsort_dcA, if the communication network manages the large amount of messages without a significant slow down.

6.5 Convex hull

The convex hull of a given set of points in n-dimensional space is the smallest polytope which contains all given points. Usually, this polytope is represented by the set of its vertices. For simplicity, we consider only two-dimensional space here. The algorithm we have chosen is *quickhull* (Atallah, 1999). Our function quickhull_dcA contains two calls of a $\mathcal{DC}$ algorithm called subhull, which constructs a clockwise traversal of a part of the hull, given a starting and a final point. quickhull_dcA calls subhull, for a construction of (1) the upper part and

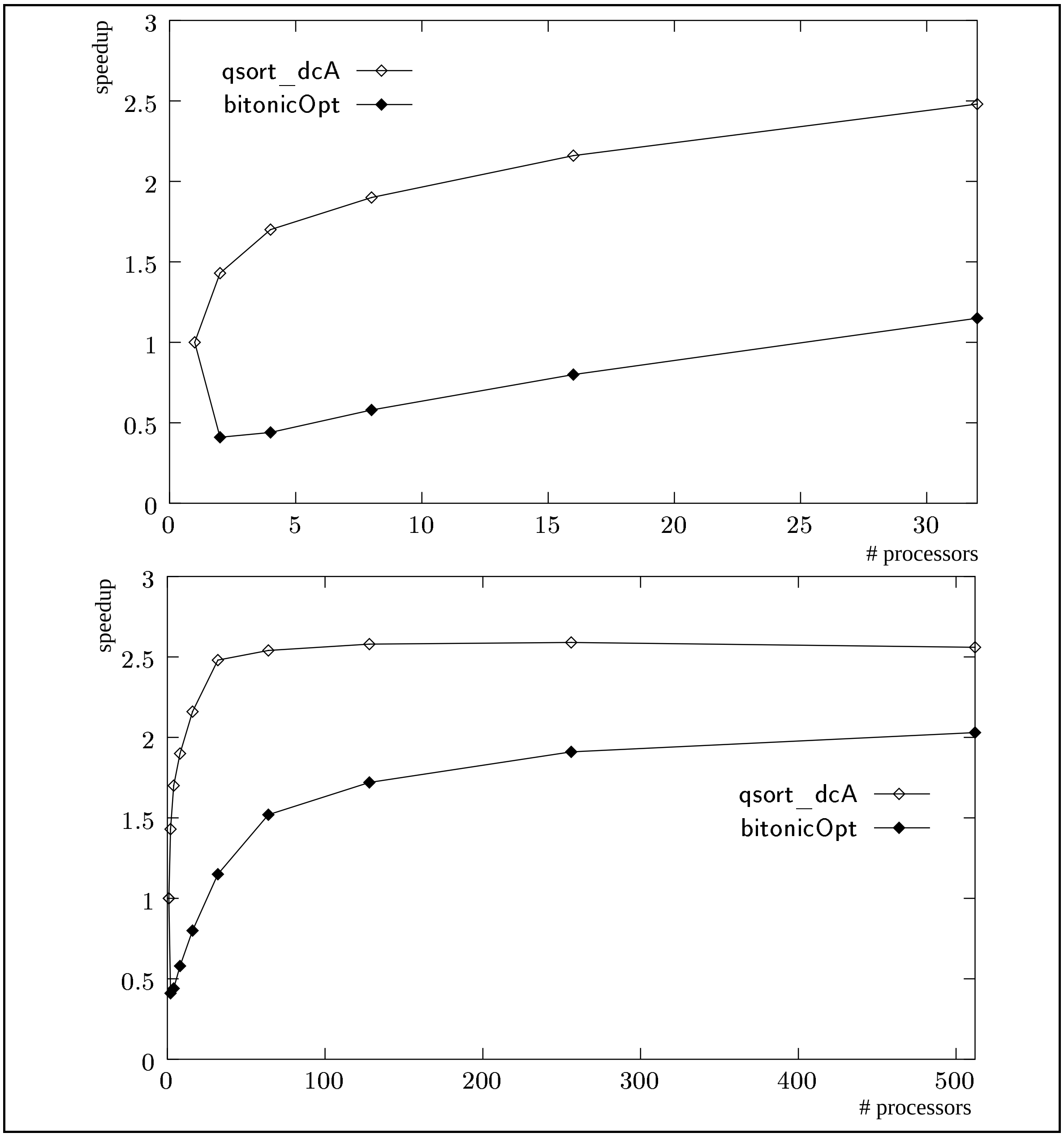


Figure 6.4: Relative speedup for sorting (transputer)

# p	20000/2		30000/2		30000/5	
	abs	rel	abs	rel	abs	rel
1	123	1.00	184	1.00	147	1.00
2	80.3	1.53	117	1.57	99.1	1.48
4	62.0	1.98	93.7	1.96	83.2	1.77
8	55.3	2.22	79.3	2.32	77.5	1.90
16	49.7	2.47	76.3	2.41	72.1	2.04
32	48.2	2.55	77.6	2.37	70.1	2.10
64	47.9	2.57	71.7	2.57	69.8	2.11

Table 6.5: Execution times for quickhull_dcA (transputer)

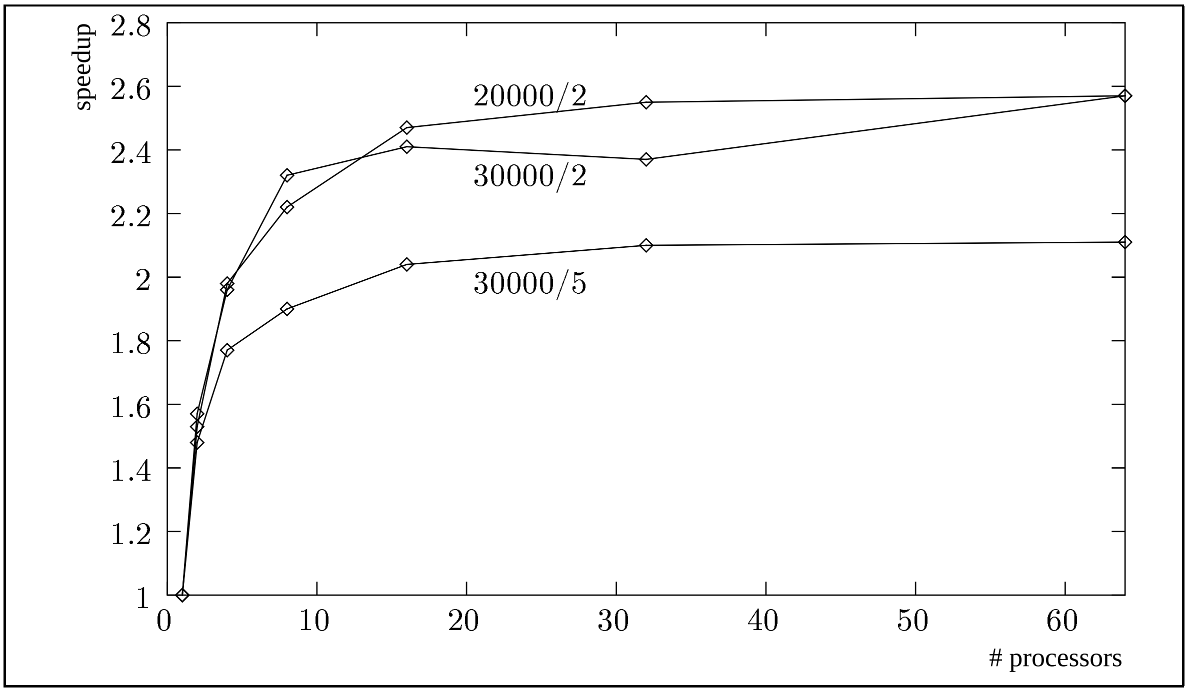


Figure 6.5: Relative speedup for quickhull_dcA (transputer)

(2) the lower part of the hull, with the leftmost and the rightmost point and vice versa.

subhull examines the set L of points which are left of the line from point a to point b; see Fig. 6.6. If L is empty, the part of the hull consists of just this line. Otherwise the point of the set L, which has the largest distance from a selected point on the line, say m, is a point on the hull. In this case, subhull calls itself twice, with the construction of a part of the hull (1) from point a to point m and (2) from point m to point b.

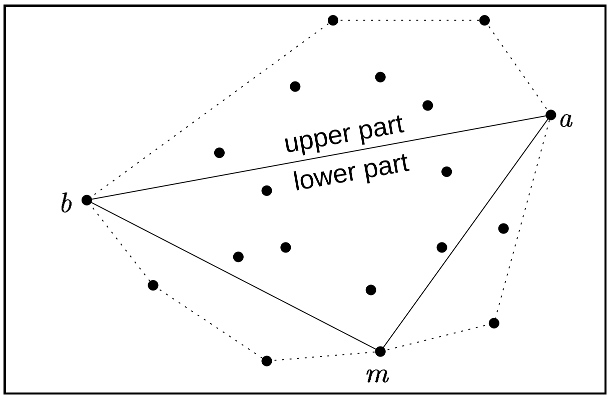


Figure 6.6: subhull on lower part

In the input data chosen, the n

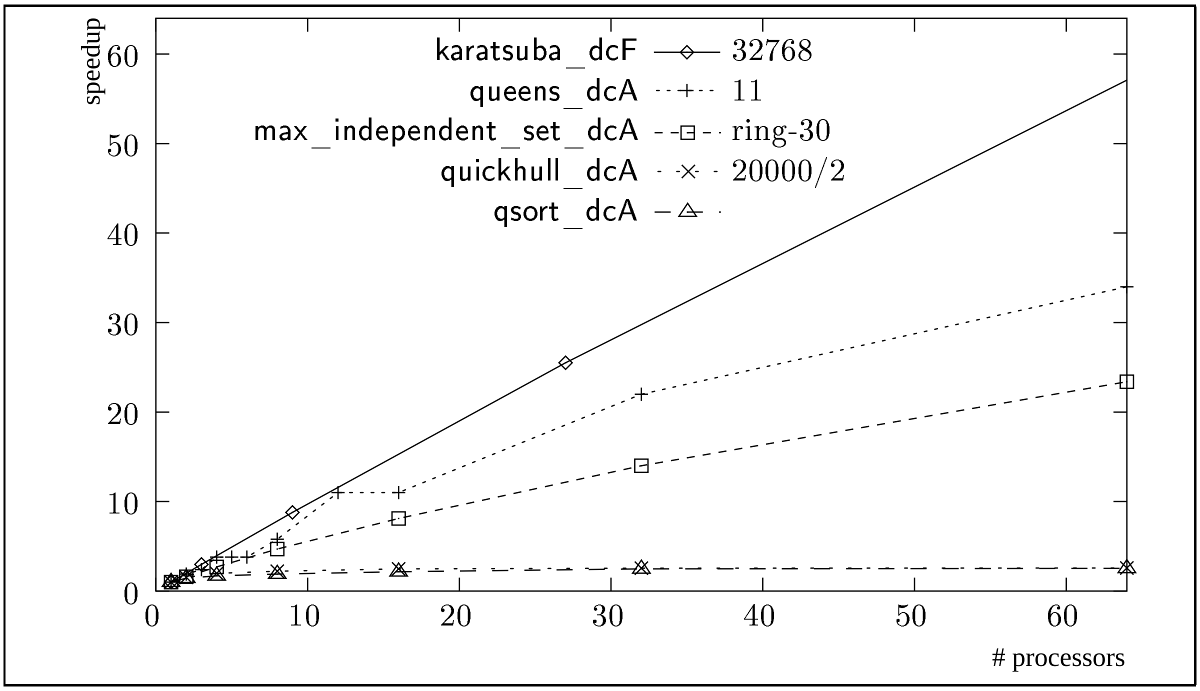


Figure 6.7: Speedup comparisons

points are distributed equally among k concentric circles, such that the hull consists of n/k points, which are located on the circle with the largest radius. In Tab. 6.5 and Fig. 6.5, we label the particular problem with n/k.

The quickhull algorithm incurs a similar risk of load imbalance as the quicksort algorithm (Atallah, 1999).

6.6 Comparison of the algorithms

In Fig. 6.7, we compare the speedups achievable for algorithms that describe the solution of different problems. For each problem discussed in this chapter, we choose the algorithm with the input that shows the best relative speedup, except for three cases: (1) we prefer the dcF version of the Karatsuba algorithm to the dcA version, since the absolute execution times are better; (2) we do not choose the parallel execution in queens_dcA that has superlinear speedup; (3) we choose qsort_dcA instead of bitonicOpt, because of the superior absolute execution time. We make these exceptions in order to present a realistic view of the potential for parallelization of the problems discussed.

The comparison confirms that the cost complexity of the algorithm has a dominating effect on the speedup. The examples karatsuba_dcF, queens_dcA and max_independent_set_dcA have a complexity substantially above linearity and show good speedups. The other examples have a complexity of $\Theta(n \cdot \log n)$ and show almost no speedup.

The speedup results for the n queens problem and the maximum independent set problem can still be improved. There is potential in the balance of load and the control of granularity.

Chapter 7

Conclusions

This thesis delivers as its main result evidence that the appropriate use of skeletons, especially $\mathcal{DC}$ skeletons, can guarantee significant speedups in the parallel implementation of some functional programs. The crucial point is that the parallel execution schema of every skeleton is defined operationally, i.e., with a strong emphasis on the state of the parallel machine, including communication and synchronization. In this point, our approach differs from other functional language implementations, in which the skeleton implementation is defined in the language itself.

The use of a skeleton that fits the problem well can already yield a relative speedup which is near the theoretical optimum (the number of processors) as, e.g., for Karatsuba's polynomial multiplication. On the other hand, one has to pay the price of an overhead, due to the need to adapt the algorithm for the skeleton, but this pays off for processor numbers beyond a certain size.

Sect. 7.1 discusses the conditions under which high speedups can be achieved. Here, we present only general, informal results. Quantitative results for the implementation of itF on grids can be found in Sect. 4.8. Most of these conditions do not affect the application programmer but the programmers of compilers and parallel skeleton implementations. Sect. 7.2 reflects on the experiences we made with the use of skeletons. Sect. 7.3 reviews the use of space-time mappings. The essence of the experimental results obtained with the current version of our compiler is presented in Sect. 7.4. Future perspectives are addressed in Sect. 7.5.

7.1 Factors of influence on the speedup

7.1.1 Cost complexity of the algorithm chosen

The $\mathcal{DC}$ paradigm permits a simple specification of some efficient algorithms, e.g., Karatsuba's polynomial multiplication and Strassen's matrix multiplication, which are difficult to implement by loop programs. The loop implementations of the naïve algorithms for these problems may achieve good relative speedups, but they behave badly for high problem sizes, due to their inherently high cost complexity. We have demonstrated that an efficient parallelization of the improved algorithms via the use of $\mathcal{DC}$ is possible.

It is a commonly held view that $\mathcal{DC}$ is connected with the solution of problems in the class $\mathcal{NC}$, i.e., those problems which can be processed with a polynomial number of processors in polylogarithmic time. Our definition of dcA is not restricted to this class, as our experiments with the exponential time algorithms for n queens and maximum independent set show.

Exponential-time algorithms have a high potential for speedup, since the ratio of communication by computation time is low. The challenge is to establish high granularity and load balance.

7.1.2 Compiler and run-time system

The compiler, the skeleton implementations and the run-time system have to provide powerful facilities and techniques.

1. Problem-specific skeletons have to be designed and implemented. The design is guided by the following considerations:
   (a) Identification of a common dependence structure of the algorithms in a class.
   (b) Focus on a fast reduction of the amount of time and space of subcomputations by the $\mathcal{DC}$ schema.
   (c) Choice of data representations which allow for a schema that is as regular as possible.
2. Space-time mapping techniques must lead to an efficient parallel implementation of skeletons, with the following properties:
   (a) Control over which functions are to be parallelized and to what extent.
   (b) Avoidance of data sharing between parallel tasks.
   (c) Sharing of data in sequential computations.
   (d) Aggregation of communications.
   (e) Space-efficient sequential computations, e.g., traversing the $\mathcal{DC}$ call tree depth-first instead of breadth-first.
3. The functional style incurs many sources of potential inefficiency in the produced code. Sophisticated compilation techniques are necessary:
   (a) Desugaring of let-expressions which preserves common subexpressions, instead of a pure lifting.
   (b) Inlining in combination with rule-based optimizations.
   (c) Avoidance of copying.
   (d) Branch optimization.

   (e) Release of data structures immediately after the last use, e.g., by reference counting.
   (f) In-place update of single-threaded structures.

4. Compiled functional programs make extreme use of the run-time system. The following services should receive special support:

   (a) Memory allocation and release. A large fraction of all operations requires memory allocation. The standard functions for memory allocation and release are too slow.
   (b) Fast marshaling and unmarshaling of heap-allocated structures.
   (c) Flexible collective communications. The communication primitives provided by libraries, like, e.g., MPI, do not permit to gather and scatter different-size portions of a memory area.

Several of these factors became subject to closer investigation since we identified them as causes of efficiency loss. One example is the sharing of data on one processor: we started to consider a flattened implementation of data structures, following Blelloch (1993), but recognized that, at least in our higher-order setting, the absence of sharing incurs an unacceptably high memory consumption. Thus, we discarded the idea of flattening, in general. However, in some special cases, flattening is still useful, e.g., for tuples of static data types.

Because optimizations of static data structures and of the space-time mapping are not implemented yet, the intended behavior of the compiler was achieved by the following manipulations:

1. We enforced a restriction of parallelization to dcA by choosing an appropriate number of processors, e.g., a power of 3 in the Karatsuba example or by excluding the map skeleton from the parallelization.

2. Instead of using let, whose desugaring does not exploit common subexpressions yet, the $\mathcal{HDC}$ sequencing operator $\twoheadrightarrow$ has been used occasionally.

3. Tuples are not flattened, which causes high memory overhead for long lists of pairs. Until they are implemented in general, we use a distinguished type IntPair as a container for a pair of Int and access this container by the functions fstPair, sndPair and packPair.

7.1.3 The application programmer

Under the premise that algorithm, compiler, run-time system and skeleton implementations are time- and space-efficient, the following responsibilities still remain for the application programmer:

1. Choose a set of candidate skeletons which appear appropriate for the underlying paradigm of the algorithm, e.g. $\mathcal{DC}$.

2. Select a skeleton which exploits as much of the algorithmic structure as possible, even if small adaptations of the data, e.g., tupling, are necessary. Avoid an overhead of operations due to adaptations which cannot be bounded by a constant factor.

3. Avoid explicit recursion, since the function call introduces control dependences. Use predefined skeletons instead.

4. Stay away from introducing large chains of data dependences, as occur when building up long lists with the (:) constructor.

5. Use higher-order functions to emphasize structure (generalization), not to introduce dependences, e.g., by continuation passing (Appel, 1992).

6. Beware of the eager evaluation schema in $\mathcal{HDC}$, i.e., avoid expensive computations whose result is possibly not used. The functions delay and force of the $\mathcal{HDC}$ prelude could be used to defer evaluation, where it pays off.

7.2 Skeletons

The work spent on skeletons will have major influence on the efficient parallel implementation of functional programs. A skeleton need not be just a schema with an implementation in isolation, but can come together with knowledge of its properties and possibilities for transformation and different parallel implementations.

Skeletons are even useful to accelerate a sequential computation because their implementation can use memory space more efficiently than recursive functions can. It turned out that a dedicated sequential implementation for *small* skeletons like map, red and filter can accelerate the program execution. On the other hand, applications of these small skeletons have to be parallelized with care because they often spend less time in a sequential execution than their parallel execution requires.

Large skeletons like dcA can mostly be parallelized without risk, because they cover an entire algorithm which is supposed to consume a considerable amount of operations. The use of a large skeleton provides a view on large parts of the program and enables an analysis of time and space consumption, if the space-time mapping of the skeleton is known, of course. This means that the parallelization of the program can be controlled by changing parameters of the space-time mapping.

7.3 Space-time mapping

The original goal of space-time mapping (Lengauer, 1993) is to optimize the execution time, number of processors, communications, etc., with respect to a program and a machine of fixed interconnections.

We encounter problems in modeling and decidability:

1. In the polytope model, the optimizations assume an ideal grid as a target architecture. This is justified, e.g., in systolic design. In contrast, the developers of parallel machines are interested in fast communication times and not in providing a topology with simple properties.

2. The mathematical description of the optimization problems requires integer variables and is not affine linear, i.e., it is undecidable, in general. Therefore, the optimization must be based on heuristics.

7.4 Experimental results

The experimental results shown in Chapter 6 are preliminary, because they are based on a compiler which is not yet finished concerning the implementation of many optimization methods. However, some important conclusions can be drawn:

1. The higher-order functional style, if used moderately, is no impediment to an efficient parallelization.

2. Skeletons have a great influence on the speedup. In the experiments for the Karatsuba example, presented in Tab. 6.1, the use of dcF is significantly superior to the use of dcA.

3. In contrast to imperative languages like C, where the programmer is responsible for the use of memory and the flow of control, small drawbacks in a compiler of a functional language, like a loss of sharing or space leaks, can have horrible consequences for the results, as we recognized in preliminary experiments.

We were faced with the following impediments for good speedups:

1. The sequential algorithm has a time complexity of $O(n \cdot (\log n)^k)$ for a small natural k and input size n. These algorithms are already very fast if processed sequentially, which becomes apparent in the examples of quickhull and sorting.

   If the input resp. output data is located on a single processor, the distribution of data requires linear time. The constant factor of this time has to be significantly smaller than $(\log n)^k$, which is not easy to achieve if communication over a slow network or marshaling of dynamic data structures is required. The solution of this problem is a distributed allocation of input and output data, in which the time for communication in the first step is divided by the number of processors. This is not implemented in our compiler yet.

   We believe that, e.g., for sorting, a straight-forward parallelization of the divide and combine function is not sufficient to yield good speedups, but that a distinguished skeleton has to be implemented which exploits properties of the real machine.

2. The granularity of tasks is too small. We improved our results for the n queens problem by a change of the skeleton, from dcA without granularity control to a parallel recursive map. In other cases, we prevent some applications of map from being parallelized where it does not pay off.

3. The load is not balanced. This occurred for maximum independent set in all cases we observed. In fact, the algorithm itself decomposes the graph asymmetrically. The algorithm for solving the n queens problem also incurs an imbalance, albeit less extreme.

   We plan to exploit the structure of the iterative versions itA/itB of the skeletons dcA/dcB, to permit a balancing of tasks at each individual level of the call tree. The maximum independent set problems and the n queens problem are computationally expensive enough to warrant a balancing at run time. However, for algorithms for which symbolic size information is available, it would be superior to do the balancing at compile time.

7.5 Future

Our future plans consider mainly the completion of the $\mathcal{HDC}$ compiler and the skeleton library:

1. Extension of the library of skeleton implementations.

2. Integration of the GHC front end into the $\mathcal{HDC}$ compiler, in order to cover almost all features of Haskell, aside from non-strictness.

3. The extension of the code generation and the run-time system to support a distributed data layout and parallel I/O. Here, we are looking forward to support by the MPI-2 library concerning one-sided calls and collective file operations (MPI Forum, 1997).

4. Partial evaluation and size inference in combination with the Γ constructor. Here, the rules of Figs. 4.4–4.6 can be used. We hope that, in some cases, it will be possible to infer useful information for granularity control and load balance.

Appendix A

Proofs

A.1 Law 4.7 (unconcat/concat)

$\boxed{\mathsf{unconcat}\ (\mathsf{map}\ \#\ xss)\ (\mathsf{concat}\ xss)\ =\ xss}$

Proof by induction on the structure of xss.

$\boxed{\text{(i) basic case:}}$

$\mathsf{unconcat}\ (\mathsf{map}\ \#\ [\,])\ (\mathsf{concat}\ [\,])$
$\quad = \boxed{\text{definition } \mathsf{map}}$
$\mathsf{unconcat}\ [\,]\ (\mathsf{concat}\ [\,])$
$\quad = \boxed{\text{definition } \mathsf{unconcat}}$
$[\,]$

$\boxed{\text{(ii) inductive case:}}$

$\mathsf{unconcat}\ (\mathsf{map}\ \#\ (xs{:}xss))\ (\mathsf{concat}\ (xs{:}xss))$
$\quad = \boxed{\text{definition } \mathsf{map}}$
$\mathsf{unconcat}\ (\#xs : \mathsf{map}\ \#\ xss)\ (\mathsf{concat}\ (xs{:}xss))$
$\quad = \boxed{\text{definition } \mathsf{concat}}$
$\mathsf{unconcat}\ (\#xs : \mathsf{map}\ \#\ xss)\ (xs \mathbin{+\!\!+} \mathsf{concat}\ xss)$
$\quad = \boxed{\text{definition } \mathsf{unconcat}}$
$\mathsf{take}\ (\#xs)\ (xs \mathbin{+\!\!+} \mathsf{concat}\ xss)$
$: \mathsf{unconcat}\ (\mathsf{map}\ \#\ xss)\ (\mathsf{drop}\ (\#xs)\ (xs \mathbin{+\!\!+} \mathsf{concat}\ xss))$
$\quad = \boxed{\text{property } \mathsf{take}, \text{ property } \mathsf{drop}}$
$xs : \mathsf{unconcat}\ (\mathsf{map}\ \#\ xss)\ (\mathsf{concat}\ xss)$
$\quad = \boxed{\text{induction hypothesis}}$
$xs : xss$

A.2 Law 4.8 (unconcat/prefixsum)

$\boxed{\textsf{unconcatprefixsum} \;=\; \textsf{unconcat}}$

Proof by induction on the structure of the first argument.

$\boxed{\text{(i) basic case:}}$

$\textsf{unconcatprefixsum}\ [\,]\ [\,]$
$\quad = \boxed{\text{unfolding } \textsf{unconcatprefixsum}}$
$\underline{\texttt{let}}\ bs = \textsf{scanl}\ (+)\ 0\ [\,]$
$\underline{\texttt{in}}\ [\ [\ [\,]\,!!\,k \mid k\leftarrow[bs\,!!\,i\,..\,bs\,!!\,(i{+}1){-}1]\ ] \mid i\leftarrow[0\,..\,\#bs\ {-}2]]$
$\quad = \boxed{\text{substituting } bs}$
$[\ [\ [\,]\,!!\,k \mid k\leftarrow[[0]\,!!\,i\,..\,[0]\,!!\,(i{+}1){-}1]\ ] \mid i\leftarrow[0\,..\,\#[0]\ {-}2\ ]]$
$\quad = \boxed{\text{evaluation}}$
$[\ \perp \mid i\leftarrow[0\,..\,({-}1)]\ ]$
$\quad = \boxed{\text{empty generator}}$
$[\,]$
$\quad = \boxed{\text{folding } \textsf{unconcat}}$
$\textsf{unconcat}\ [\,]\ [\,]$

$\boxed{\text{(ii) inductive case:}}$

$\textsf{unconcatprefixsum}\ (n{:}ns)\ xs$
$\quad = \boxed{\text{unfolding } \textsf{unconcatprefixsum}}$
$\underline{\texttt{let}}\ bs = \textsf{scanl}\ (+)\ 0\ (n{:}ns)$
$\underline{\texttt{in}}\ [\ [\ xs\,!!\,k \mid k\leftarrow[bs\,!!\,i\,..\,bs\,!!\,(i{+}1){-}1]\ ] \mid i\leftarrow[0\,..\,\#bs\ {-}2]]$
$\quad = \boxed{\text{splitting generator}}$
$\underline{\texttt{let}}\ bs = \textsf{scanl}\ (+)\ 0\ (n{:}ns)$
$\underline{\texttt{in}}\ \ [\ xs\,!!\,k \mid k\leftarrow[bs\,!!\,0\,..\,bs\,!!\,1{-}1]\ ]$
$\quad :\ [\ [\ xs\,!!\,k \mid k\leftarrow[bs\,!!\,i\,..\,bs\,!!\,(i{+}1){-}1]\ ] \mid i\leftarrow[1\,..\,\#bs\ {-}2]]$
$\quad = \boxed{\text{evaluation}}$
$[\ xs\,!!\,k \mid k\leftarrow[0\,..\,n{-}1]\ ]$
$:\ \ \underline{\texttt{let}}\ bs = \textsf{scanl}\ (+)\ 0\ (n{:}ns)$
$\quad \underline{\texttt{in}}\ [\ [\ xs\,!!\,k \mid k\leftarrow[bs\,!!\,i\,..\,bs\,!!\,(i{+}1){-}1]\ ] \mid i\leftarrow[1\,..\,\#bs\ {-}2]]$
$\quad = \boxed{\text{index shift}}$
$[\ xs\,!!\,k \mid k\leftarrow[0\,..\,n{-}1]\ ]$
$:\ \ \underline{\texttt{let}}\ bs1 = \textsf{scanl}\ (+)\ n\ ns$
$\quad \underline{\texttt{in}}\ [\ [\ xs\,!!\,k \mid k\leftarrow[bs1\,!!\,i\,..\,bs1\,!!\,(i{+}1){-}1]\ ] \mid i\leftarrow[0\,..\,\#bs1\ {-}2]]$

$= \boxed{\text{propery take, initial element scanl}}$

take n xs
: let $bs1$ = scanl $(+)$ 0 ns
  in [[$xs\,!!\,(k{+}n)$ | $k \leftarrow [bs1\,!!\,i\,..\,bs1\,!!\,(i{+}1){-}1]$] | $i \leftarrow [0\,..\,\#bs1\;-2]$]

$= \boxed{\text{property drop}}$

take n xs
: let $bs1$ = scanl $(+)$ 0 ns
  in [[(drop n xs)$\,!!\,k$ | $k \leftarrow [bs1\,!!\,i\,..\,bs1\,!!\,(i{+}1){-}1]$] | $i \leftarrow [0\,..\,\#bs1\;-2]$]

$= \boxed{\text{folding unconcatprefixsum}}$

take n xs : unconcatprefixsum ns (drop n xs)

$= \boxed{\text{induction hypothesis}}$

take n xs : unconcat ns (drop n xs)

$= \boxed{\text{folding unconcat}}$

unconcat $(n{:}ns)$ xs

A.3 Lemma 4.9 (mdcA/linrec)

$\boxed{\text{mdcAlinrec} \;=\; \text{mdcA}}$

Proof by induction on the remaining levels of recursion, i.e., the difference between the maximum and the current length of xss in (mdcAlinrec p b d c xss).

To simplify the proof, we use the following auxiliary definitions:

(predicate)
mdcAp $(xs{:}xss)$ = null xs

(basic function)
mdcAb $(xs{:}xss)$ = $([\,], xss)$

(divide function)
mdcAd p d $(xs{:}xss)$ = map $(\lambda x \rightarrow$ if p x then $[\,]$ else d $x)$ (concat xs)
                     : xs : xss

(combine function)
mdcAc p b d c $(fsols, xs{:}xss)$ =
 let $sols1$ = unconcat (map $(\lambda x \rightarrow$ if p x then 0 else $\#(d\;x))$ (concat xs))
                $fsols$
     $xsols$ = zip (concat xs) $sols1$
     cop $(x, sols)$ = if p x then b x else c x $sols$
 in (map cop $xsols$, xss)

$\boxed{\text{(i) basic case:}}$

mdcAlinrec $p\ b\ d\ c$ ($[\,]$:xss)
  = $\boxed{\text{unfolding mdcAlinrec}}$
linrec mdcAp mdcAb (mdcAd $p\ d$) (mdcAc $p\ b\ d\ c$) ($[\,]$:xss)
  = $\boxed{\text{unfolding linrec}}$
if mdcAp ($[\,]$:xss)
  then mdcAb ($[\,]$:xss)
  else (mdcAc $p\ b\ d\ c$) (linrec mdcAp mdcAb (mdcAd $p\ d$) (mdcAc $p\ b\ d\ c$)
    (mdcAd $p\ d$ ($[\,]$:xss)))
  = $\boxed{\text{definition mdcAp}}$
if null $[\,]$
  then mdcAb ($[\,]$:xss)
  else (mdcAc $p\ b\ d\ c$) (linrec mdcAp mdcAb (mdcAd $p\ d$) (mdcAc $p\ b\ d\ c$)
    (mdcAd $p\ d$ ($[\,]$:xss)))
  = $\boxed{\text{null } [\,] = \text{True}}$
mdcAb ($[\,]$:xss)
  = $\boxed{\text{definition mdcAb}}$
($[\,]$, xss)
  = $\boxed{\text{property map}}$
(map (dcA $p\ b\ d\ c$) $[\,]$, xss)
  = $\boxed{\text{definition mdcA}}$
mdcA $p\ b\ d\ c$ ($[\,]$:xss)

$\boxed{\text{(ii) inductive case:}}$ (assumption: $xs \neq [\,]$)

mdcAlinrec $p\ b\ d\ c$ (xs:xss)
  = $\boxed{\text{unfolding mdcAlinrec}}$
linrec mdcAp mdcAb (mdcAd $p\ d$) (mdcAc $p\ b\ d\ c$) (xs:xss)
  = $\boxed{\text{unfolding linrec}}$
if mdcAp (xs:xss)
  then mdcAb (xs:xss)
  else mdcAc $p\ b\ d\ c$ (linrec mdcAp mdcAb (mdcAd $p\ d$) (mdcAc $p\ b\ d\ c$)
    (mdcAd $p\ d$ (xs:xss)))
  = $\boxed{\text{definition mdcAp}}$
if null xs
  then mdcAb (xs:xss)
  else mdcAc $p\ b\ d\ c$ (linrec mdcAp mdcAb (mdcAd $p\ d$) (mdcAc $p\ b\ d\ c$)
    (mdcAd $p\ d$ (xs:xss)))

$$= \boxed{\mathsf{null}\ xs = \mathsf{False}}$$

$$\begin{array}{l}\mathsf{mdcAc}\ p\ b\ d\ c\ (\mathsf{linrec}\ \ \mathsf{mdcAp}\ \mathsf{mdcAb}\ (\mathsf{mdcAd}\ p\ d)\ (\mathsf{mdcAc}\ p\ b\ d\ c)\\ \qquad\qquad\qquad\qquad (\mathsf{mdcAd}\ p\ d\ (xs{:}xss)))\end{array}$$

$$= \boxed{\text{folding mdcAlinrec}}$$

$$\mathsf{mdcAc}\ p\ b\ d\ c\ (\mathsf{mdcAlinrec}\ p\ b\ d\ c\ (\mathsf{mdcAd}\ p\ d\ (xs{:}xss)))$$

$$= \boxed{\text{unfolding divide}}$$

$$\begin{array}{l}\mathsf{mdcAc}\ p\ b\ d\ c\\ \quad (\mathsf{mdcAlinrec}\ p\ b\ d\ c\ (\ \mathsf{map}\ \ (\lambda x \rightarrow \underline{\mathtt{if}}\ p\ x\ \underline{\mathtt{then}}\ [\,]\ \underline{\mathtt{else}}\ d\ x)\\ \qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{concat}\ xs)\\ \qquad\qquad\qquad\qquad\quad :\ xs\ :\ xss))\end{array}$$

$$= \boxed{\text{induction hypothesis (remaining levels of recursion decreased)}}$$

$$\begin{array}{l}\mathsf{mdcAc}\ p\ b\ d\ c\\ \quad (\mathsf{mdcA}\ p\ b\ d\ c\ (\ \mathsf{map}\ \ (\lambda x \rightarrow \underline{\mathtt{if}}\ p\ x\ \underline{\mathtt{then}}\ [\,]\ \underline{\mathtt{else}}\ d\ x)\\ \qquad\qquad\qquad\qquad\qquad (\mathsf{concat}\ xs)\\ \qquad\qquad\qquad\quad :\ xs\ :\ xss))\end{array}$$

$$= \boxed{\text{unfolding mdcA}}$$

$$\begin{array}{l}\mathsf{mdcAc}\ p\ b\ d\ c\\ \quad (\mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (\mathsf{concat}\ (\mathsf{map}\ \ (\lambda x \rightarrow \underline{\mathtt{if}}\ p\ x\ \underline{\mathtt{then}}\ [\,]\ \underline{\mathtt{else}}\ d\ x)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{concat}\ xs))),\\ \quad\ xs{:}xss)\end{array}$$

$$= \boxed{\mathsf{map}\ f\ (\mathsf{concat}\ zss) = \mathsf{concat}\ (\mathsf{map}\ (\mathsf{map}\ f)\ zss)}$$

$$\begin{array}{l}\mathsf{mdcAc}\ p\ b\ d\ c\\ \quad (\mathsf{concat}\ (\mathsf{map}\ \ (\mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c))\\ \qquad\qquad\qquad\quad (\mathsf{map}\ (\lambda x \rightarrow \underline{\mathtt{if}}\ p\ x\ \underline{\mathtt{then}}\ [\,]\ \underline{\mathtt{else}}\ d\ x)\ (\mathsf{concat}\ xs))),\\ \quad\ xs{:}xss)\end{array}$$

$$= \boxed{\mathsf{map}\ g\ (\mathsf{map}\ f\ zs) =\ \mathsf{map}\ (g \circ f)\ zs}$$

$$\begin{array}{l}\mathsf{mdcAc}\ p\ b\ d\ c\\ \quad (\mathsf{concat}\ \ (\mathsf{map}\ \ (\quad \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\\ \qquad\qquad\qquad\qquad\ \circ\ (\lambda x \rightarrow \underline{\mathtt{if}}\ p\ x\ \underline{\mathtt{then}}\ [\,]\ \underline{\mathtt{else}}\ d\ x))\\ \qquad\qquad\qquad\quad (\mathsf{concat}\ xs)),\\ \quad\ xs{:}xss)\end{array}$$

$$= \boxed{\circ\ /\ \text{conditional}}$$

$$\begin{array}{l}\mathsf{mdcAc}\ p\ b\ d\ c\\ \quad (\mathsf{concat}\ \ (\mathsf{map}\ \ (\lambda x \rightarrow \underline{\mathtt{if}}\ p\ x\ \underline{\mathtt{then}}\ [\,]\ \underline{\mathtt{else}}\ \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x))\\ \qquad\qquad\qquad\quad (\mathsf{concat}\ xs)),\\ \quad\ xs{:}xss)\end{array}$$

$$= \boxed{\text{unfolding combine}}$$

$$\begin{array}{l}
\underline{\texttt{let}}\ \ \mathit{sols1} \\
\qquad = \mathsf{unconcat} \\
\qquad\quad (\mathsf{map}\ \ (\lambda x \to \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ 0\ \underline{\texttt{else}}\ \#(d\ x))\ (\mathsf{concat}\ xs)) \\
\qquad\quad (\mathsf{concat}\ (\mathsf{map}\ \ (\lambda x \to \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ [\,]\ \underline{\texttt{else}}\ \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x)) \\
\qquad\qquad\qquad\qquad (\mathsf{concat}\ xs))) \\
\qquad \mathit{xsols} = \mathsf{zip}\ (\mathsf{concat}\ xs)\ \mathit{sols1} \\
\qquad \mathit{cop}\ (x, \mathit{sols}) = \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ b\ x\ \underline{\texttt{else}}\ c\ x\ \mathit{sols} \\
\underline{\texttt{in}}\ (\mathsf{map}\ \mathit{cop}\ \mathit{xsols}, \mathit{xss})
\end{array}$$

$$= \boxed{\#zs = \#(\mathsf{map}\ f\ \ zs)}$$

$$\begin{array}{l}
\underline{\texttt{let}}\ \ \mathit{sols1} \\
\qquad = \mathsf{unconcat} \\
\qquad\quad (\mathsf{map}\ \ (\lambda x \to \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ 0\ \underline{\texttt{else}}\ \#(\mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x))) \\
\qquad\qquad\quad (\mathsf{concat}\ xs)) \\
\qquad\quad (\mathsf{concat}\ (\mathsf{map}\ \ (\lambda x \to \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ [\,]\ \underline{\texttt{else}}\ \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x)) \\
\qquad\qquad\qquad\qquad (\mathsf{concat}\ xs))) \\
\qquad \mathit{xsols} = \mathsf{zip}\ (\mathsf{concat}\ xs)\ \mathit{sols1} \\
\qquad \mathit{cop}\ (x, \mathit{sols}) = \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ b\ x\ \underline{\texttt{else}}\ c\ x\ \mathit{sols} \\
\underline{\texttt{in}}\ (\mathsf{map}\ \mathit{cop}\ \mathit{xsols}, \mathit{xss})
\end{array}$$

$$= \boxed{\#/\ \text{conditional}}$$

$$\begin{array}{l}
\underline{\texttt{let}}\ \ \mathit{sols1} \\
\qquad = \mathsf{unconcat} \\
\qquad\quad (\mathsf{map}\ \#\ (\mathsf{map}\ \ (\lambda x \to \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ [\,]\ \underline{\texttt{else}}\ \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x)) \\
\qquad\qquad\qquad\qquad (\mathsf{concat}\ xs))) \\
\qquad\quad (\mathsf{concat}\ \ (\mathsf{map}\ \ (\lambda x \to \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ [\,]\ \underline{\texttt{else}}\ \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x)) \\
\qquad\qquad\qquad\qquad (\mathsf{concat}\ xs))) \\
\qquad \mathit{xsols} = \mathsf{zip}\ (\mathsf{concat}\ xs)\ \mathit{sols1} \\
\qquad \mathit{cop}\ (x, \mathit{sols}) = \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ b\ x\ \underline{\texttt{else}}\ c\ x\ \mathit{sols} \\
\underline{\texttt{in}}\ (\mathsf{map}\ \mathit{cop}\ \mathit{xsols}, \mathit{xss})
\end{array}$$

$$= \boxed{\mathsf{unconcat/concat}}$$

$$\begin{array}{l}
\underline{\texttt{let}}\ \ \mathit{sols1} = \mathsf{map}\ \ (\lambda x \to \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ [\,]\ \underline{\texttt{else}}\ \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x)) \\
\qquad\qquad\qquad\quad (\mathsf{concat}\ xs) \\
\qquad \mathit{xsols} = \mathsf{zip}\ (\mathsf{concat}\ xs)\ \mathit{sols1} \\
\qquad \mathit{cop}\ (x, \mathit{sols}) = \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ b\ x\ \underline{\texttt{else}}\ c\ x\ \mathit{sols} \\
\underline{\texttt{in}}\ (\mathsf{map}\ \mathit{cop}\ \mathit{xsols}, \mathit{xss})
\end{array}$$

$$= \boxed{\text{substitution of } \mathit{sols}1}$$

$$\begin{array}{l}
\underline{\texttt{let}}\ \ \mathit{xsols} = \mathsf{zip}\ \ (\mathsf{concat}\ xs) \\
\qquad\qquad\qquad (\mathsf{map}\ \ (\lambda x \to \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ [\,]\ \underline{\texttt{else}}\ \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x)) \\
\qquad\qquad\qquad\qquad (\mathsf{concat}\ xs)) \\
\qquad \mathit{cop}\ (x, \mathit{sols}) = \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ b\ x\ \underline{\texttt{else}}\ c\ x\ \mathit{sols} \\
\underline{\texttt{in}}\ (\mathsf{map}\ \mathit{cop}\ \mathit{xsols}, \mathit{xss})
\end{array}$$

$$= \boxed{\text{fresh } z \implies \mathsf{zip}\ zs\ (\mathsf{map}\ f\ \ zs) = \mathsf{map}\ (\lambda z \to (z,\ f\ \ z))\ zs}$$

$$\begin{array}{l}
\underline{\texttt{let}}\ \ xsols = \mathsf{map}\ \ (\lambda x \to (x,\ \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ [\,]\ \underline{\texttt{else}}\ \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x)))\\
\qquad\qquad\qquad\quad (\mathsf{concat}\ xs)\\
\qquad cop\ (x, sols) = \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ b\ x\ \underline{\texttt{else}}\ c\ x\ sols\\
\underline{\texttt{in}}\ (\mathsf{map}\ cop\ xsols, xss)\\
\quad = \boxed{\text{substitution of } cop}\\
\underline{\texttt{let}}\ \ xsols = \mathsf{map}\ \ (\lambda x \to (x,\ \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ [\,]\ \underline{\texttt{else}}\ \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x)))\\
\qquad\qquad\qquad\quad (\mathsf{concat}\ xs)\\
\underline{\texttt{in}}\ (\mathsf{map}\ (\lambda(x, sols) \to \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ b\ x\ \underline{\texttt{else}}\ c\ x\ sols)\ xsols, xss)\\
\quad = \boxed{\text{substitution of } xsols}\\
(\mathsf{map}\ \ (\qquad (\lambda(x, sols) \to \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ b\ x\ \underline{\texttt{else}}\ c\ x\ sols)\\
\qquad\quad\ \circ\ (\lambda x \to (x,\ \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ [\,]\ \underline{\texttt{else}}\ \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x))))\\
\qquad (\mathsf{concat}\ xs), xss)\\
\quad = \boxed{\text{property } \circ}\\
(\mathsf{map}\ \ (\lambda x \to \underline{\texttt{if}}\ p\ x\ \ \underline{\texttt{then}}\ b\ x\\
\qquad\qquad\qquad\qquad \underline{\texttt{else}}\ c\ x\ (\underline{\texttt{if}}\ p\ x\ \ \underline{\texttt{then}}\ [\,]\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \underline{\texttt{else}}\ \mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x)))\\
\qquad (\mathsf{concat}\ xs), xss)\\
\quad = \boxed{\text{conditional}}\\
(\mathsf{map}\ \ (\lambda x \to \underline{\texttt{if}}\ p\ x\ \underline{\texttt{then}}\ b\ x\ \underline{\texttt{else}}\ c\ x\ (\mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (d\ x)))\\
\qquad (\mathsf{concat}\ xs), xss)\\
\quad = \boxed{\text{folding } \mathsf{dcA}}\\
(\mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (\mathsf{concat}\ xs),\ xss)\\
\quad = \boxed{\text{folding } \mathsf{mdcA}}\\
\mathsf{mdcA}\ p\ b\ d\ c\ (xs{:}xss)
\end{array}$$

A.4 Lemma 4.10 (dcA/mdcA)

$$\boxed{\mathsf{dcA}\ p\ b\ d\ c\ x\ \ =\ \ \mathsf{fst}\ (\mathsf{mdcA}\ p\ b\ d\ c\ [[[x]]])\ !!\ 0}$$

Proof.

$$\begin{array}{l}
\mathsf{dcA}\ p\ b\ d\ c\ x\\
\quad = \boxed{\text{property } \mathsf{map}}\\
\mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ [x]\ \ \texttt{!!}\ 0\\
\quad = \boxed{\text{property } \mathsf{concat}}\\
\mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (\mathsf{concat}\ [[x]])\ \ \texttt{!!}\ 0\\
\quad = \boxed{\text{definition } \mathsf{fst}}\\
\mathsf{fst}\ (\mathsf{map}\ (\mathsf{dcA}\ p\ b\ d\ c)\ (\mathsf{concat}\ [[x]]),\ [\,])\ \texttt{!!}\ 0\\
\quad = \boxed{\text{folding } \mathsf{mdcA}}\\
\mathsf{fst}\ (\mathsf{mdcA}\ p\ b\ d\ c\ [[[x]]])\ \ \texttt{!!}\ 0
\end{array}$$

A.5 Lemma 4.12 (while/for)

$\forall\, mm, nn \in \mathbb{N} : \ \forall\, \alpha \in \textsc{Type} : \ \forall\, xx \in \alpha : \ \forall\, d \in (\alpha \to \alpha) :$

$$
\begin{array}{ll}
\mathsf{while} & (\lambda((m,x),n) \to \mathsf{not}\ (m{=}0)) \\
 & (\lambda((m,x),n) \to ((m{-}1,\, d\ \ x), n{+}1)) \\
 & ((mm, xx), nn) \\
= & \\
\multicolumn{2}{l}{((0, \mathsf{for}\ \ mm\ \ d\ \ xx), nn{+}mm)}
\end{array}
$$

Proof by induction on mm.

$\boxed{\text{(i) basic case:}}$

$$
\begin{array}{ll}
\mathsf{while} & (\lambda((m,x),n) \to \mathsf{not}\ (m{=}0)) \\
 & (\lambda((m,x),n) \to ((m{-}1,\, d\ \ x), n{+}1)) \\
 & ((0, xx), nn)
\end{array}
$$

$= \boxed{\text{unfolding } \mathsf{while}}$

$\underline{\texttt{if}}\ (\mathsf{not}\ (0{=}0))\ \underline{\texttt{then}}\ \bot\ \underline{\texttt{else}}\ ((0, xx), nn)$

$= \boxed{\text{condition}}$

$((0, xx), nn)$

$= \boxed{\text{folding } \mathsf{for}}$

$((0, \mathsf{for}\ \ 0\ \ d\ \ xx), nn)$

$\boxed{\text{(ii) inductive case:}}$

$$
\begin{array}{ll}
\mathsf{while} & (\lambda((m,x),n) \to \mathsf{not}\ (m{=}0)) \\
 & (\lambda((m,x),n) \to ((m{-}1,\, d\ \ x), n{+}1)) \\
 & ((mm{+}1, xx), nn)
\end{array}
$$

$= \boxed{\text{unfolding } \mathsf{while}}$

$$
\begin{array}{lll}
\underline{\texttt{if}}\ (\mathsf{not}\ (mm{+}1{=}0)) & \underline{\texttt{then}}\ \mathsf{while} & (\lambda((m,x),n) \to \mathsf{not}\ (m{=}0)) \\
 & & (\lambda((m,x),n) \to ((m{-}1,\, d\ \ x), n{+}1)) \\
 & & ((mm,\, d\ \ xx), nn{+}1) \\
 & \underline{\texttt{else}}\ \bot &
\end{array}
$$

$= \boxed{\text{condition}}$

$$
\begin{array}{ll}
\mathsf{while} & (\lambda((m,x),n) \to \mathsf{not}\ (m{=}0)) \\
 & (\lambda((m,x),n) \to ((m{-}1,\, d\ \ x), n{+}1)) \\
 & ((mm,\, d\ \ xx), nn{+}1)
\end{array}
$$

$= \boxed{\text{induction hypothesis}}$

$((0, \mathsf{for}\ \ mm\ \ d\ \ (d\ \ xx)), nn{+}1{+}mm)$

$= \boxed{\text{folding } \mathsf{for}}$

$((0, \mathsf{for}\ \ (mm{+}1)\ \ d\ \ xx), nn{+}mm{+}1)$

Appendix B

Derivations

B.1 itA

For simplification, we make the following auxiliary definition:

$$
\begin{array}{l}
\mathsf{finaldividephase}\ \mathit{istrivial}\ \mathit{divide}\ x \\
\quad = \mathsf{while} \\
\qquad (\lambda(\mathit{xss}, n, \mathit{tss}, \mathit{lss}) \rightarrow \mathsf{not}\ (\mathsf{null}\ (\mathit{xss}\,!!\,n))) \\
\qquad (\lambda(\mathit{xss}, n, \mathit{tss}, \mathit{lss}) \rightarrow \\
\qquad\qquad \underline{\mathtt{let}}\ \ \mathit{ds} = [\ \underline{\mathtt{if}}\ (\mathit{tss}\,!!\,n\,!!\,p)\ \underline{\mathtt{then}}\ [\,]\ \underline{\mathtt{else}}\ \mathit{divide}\ (\mathit{xss}\,!!\,n\,!!\,p) \\
\qquad\qquad\qquad\qquad \mid p \leftarrow [0\,..\,\#(\mathit{xss}\,!!\,n)\ -1]\ ] \\
\qquad\qquad\qquad \mathit{ls} = \mathsf{map}\ \#\ \mathit{ds} \\
\qquad\qquad\qquad \mathit{xs} = \mathsf{concat}\ \mathit{ds} \\
\qquad\qquad\qquad \mathit{ts} = \mathsf{map}\ \mathit{istrivial}\ \mathit{xs} \\
\qquad\qquad \underline{\mathtt{in}}\ (\mathit{xss} +\!\!+ [\mathit{xs}], n+1, \mathit{tss} +\!\!+ [\mathit{ts}], \mathit{lss} +\!\!+ [\mathit{ls}])) \\
\qquad ([[x]], 0, [[\mathit{istrivial}\ x]], [\,])
\end{array}
$$

$$
\begin{array}{l}
\mathsf{itA}\ \mathit{istrivial}\ \mathit{basic}\ \mathit{divide}\ \mathit{combine}\ x \\
\quad = \boxed{\text{specification itA}} \\
\mathsf{dcA}\ \mathit{istrivial}\ \mathit{basic}\ \mathit{divide}\ \mathit{combine}\ x \\
\quad = \boxed{\text{Law 4.10 (dcA/mdcA)}} \\
\mathsf{fst}\ (\mathsf{mdcA}\ \mathit{istrivial}\ \mathit{basic}\ \mathit{divide}\ \mathit{combine}\ [[[x]]])\ !!\ 0 \\
\quad = \boxed{\text{Theorems 4.9 (mdcA/linrec) and 4.6 (linrec/liniter)}} \\
\underline{\mathtt{let}}\ \ \mathit{tr}\ (\mathit{xs}{:}\mathit{xss}) = \mathsf{null}\ \mathit{xs} \\
\qquad \mathit{ba}\ (\mathit{xs}{:}\mathit{xss}) = ([\,], \mathit{xss}) \\
\qquad \mathit{di}\ (\mathit{xs}{:}\mathit{xss}) = \quad \mathsf{map}\ \ (\lambda x \rightarrow \underline{\mathtt{if}}\ \mathit{istrivial}\ x\ \underline{\mathtt{then}}\ [\,]\ \underline{\mathtt{else}}\ \mathit{divide}\ x) \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{concat}\ \mathit{xs}) \\
\qquad\qquad\qquad\quad :\ \mathit{xs}\ :\ \mathit{xss}
\end{array}
$$

```
    co (fsols, xs:xss) =
       let  sols1 =  unconcat  (map  (λx → if istrivial x  then 0
                                                            else #(divide x))
                                      (concat xs))
                                 fsols
            xsols = zip (concat xs) sols1
            cop (x, sols) = if istrivial x then basic x else combine x sols
       in (map cop xsols, xss)
in fst (liniter tr ba di co [[[x]]]) !! 0
  = [unfolding liniter]
let (q, n) = while  (not ∘ (λ(xs:xss) → null xs) ∘ fst)
                    (λ(xs:xss, n) → (map  (λx → if istrivial x  then []
                                                                else divide x)
                                           (concat xs)
                                     : xs : xss,
                                     n+1))
                    ([[[x]]], 0)
in  fst  (for n  (λ(fsols, xs:xss) →
                   let  sols1 =  unconcat
                                 (map  (λx → if istrivial x  then 0
                                                             else #(divide x))
                                       (concat xs))
                                 fsols
                        xsols = zip (concat xs) sols1
                        cop (x, sols) = if istrivial x  then basic x
                                                        else combine x sols
                   in (map cop xsols, xss))
                 ((λ(xs:xss) → ([], xss)) q))
      !! 0
  = [reversing the construction of xss]
let (xss, n) = while  (λ(xss, n) → not (null (xss !! n)))
                      (λ(xss, n) →
                        (let  cxs = concat (xss !! n)
                         in   xss ++ [map  (λx → if istrivial x  then []
                                                                 else divide x)
                                            cxs ],
                         n+1))
                      ([[[x]]], 0)
in fst (for n  ( λ(fsols, xss) →
                 let  cxs = concat (last xss)
                      sols1 =  unconcat
                               (map  (λx → if istrivial x  then 0
                                                           else #(divide x))
                                     cxs)  fsols
                      xsols = zip cxs sols1
```

```
                    cop (x, sols) = if istrivial x  then basic x
                                                    else combine x sols
                in (map cop xsols, init xss))
               ([], init xss))  !! 0
   = [introduction fordown]
let (xss, n) = while  (λ(xss, n) → not (null (xss !! n)))
                      (λ(xss, n) →
                        (let cxs = concat (xss !! n)
                         in xss ++ [map  (λx → if istrivial x  then []
                                                               else divide x)
                                         cxs],
                         n+1))
                      ([[[x]]], 0)
in (fordown n (λ r fsols →
                  let  cxs = concat (xss !! r)
                       sols1 =  unconcat
                                 (map  (λx → if istrivial x  then 0
                                                             else #(divide x))
                                       cxs) fsols
                       xsols = zip cxs sols1
                       cop (x, sols) = if istrivial x  then basic x
                                                       else combine x sols
                   in  map cop xsols)
              [] ) !! 0
   = [tupling with tss and lss]
let (xss, n, tss, lss)
     = while
        (λ(xss, n, tss, lss) → not (null (xss !! n)) )
        (λ(xss, n, tss, lss) →
           let  cxs = concat (xss !! n)
                ds =  [ if (tss !! n !! p)  then []
                                            else divide (cxs !! p)
                       | p←[0..#cxs −1] ]
                ls = map # ds
                ts = map istrivial (concat ds)
           in (xss++[ds], n+1, tss++[ts], lss++[ls]))
        ([[[x]]], 0, [[istrivial x]], [])
in (fordown n (λ r fsols →
                  let  sols1 = unconcat (lss !! r) fsols
                       xsols = zip (concat (xss !! r)) sols1
                       cop (x, sols) = if istrivial x  then basic x
                                                       else combine x sols
                  in  map cop xsols)
              [] ) !! 0
```

```
    = [ flattening xss, substitution of xsols ]
let (xss, n, tss, lss) = finaldividephase istrivial divide x
in (fordown n (λ r fsols →
                  let  sols1 = unconcat (lss !! r) fsols
                       cop (x, sols) = if istrivial x  then basic x
                                                       else combine x sols
                  in  map cop (zip (xss !! r) sols1))
              [] )
   !! 0
    = [ Law 4.8 (unconcat/scan) ]
let (xss, n, tss, lss) = finaldividephase istrivial divide x
in (fordown n (λ r fsols →
                  let  bs = prefixsum (lss !! r)
                       sols1 =  [ [ fsols !! k | k←[bs !! p . . bs !! (p+1)−1] ]
                                | p←[0 . . #bs −2] ]
                       cop (x, sols) = if istrivial x  then basic x
                                                       else combine x sols
                  in  map cop (zip (xss !! r) sols1))
              [] )
   !! 0
    = [ introduction of a list comprehension ]
let (xss, n, tss, lss) = finaldividephase istrivial divide x
in (fordown n (λ r fsols →
                  let  bs = prefixsum (lss !! r)
                       sols1 =  [ [ fsols !! k | k←[bs !! p . . bs !! (p+1)−1] ]
                                | p←[0 . . #bs −2] ]
                       cop (x, sols) = if istrivial x  then basic x
                                                       else combine x sols
                  in  [ cop ((zip (xss !! r) sols1) !! j)
                      | j←[0 . . #sols1 −1 ] ])
              [] )
   !! 0
    = [ substitution of cop ]
let (xss, n, tss, lss) = finaldividephase istrivial divide x
in (fordown n (λ r fsols →
                  let  bs = prefixsum (lss !! r)
                       sols1 =  [ [ fsols !! k | k←[bs !! p . . bs !! (p+1)−1] ]
                                | p←[0 . . #bs −2] ]
                  in   [ if  istrivial (xss !! r !! p)
                             then basic (xss !! r !! p)
                             else combine (xss !! r !! p) (sols1 !! p)
                       | p←[0 . . #bs −2 ] ])
              [] )
   !! 0
```

```
   = [substitution of sols1, #bs]
let (xss, n, tss, lss)  = finaldividephase istrivial divide x
in (fordown n (λ r fsols →
                   let bs = prefixsum (lss !! r)
                   in [ if  tss !! r !! p
                             then basic (xss !! r !! p)
                             else combine  (xss !! r !! p)
                                             [ fsols !! k | k←[bs !! p . . bs !! (p+1)−1] ]
                        | p←[0 . . #(xss !! r)−1 ] ])
              [] )
   !! 0
```

B.2 itB

```
itB basic divide combine mm x
   = [specification itB]
dcB basic divide combine mm x
   = [Lemma 3.2: dcB_by_dcA]
dcA  ((=0) ∘ fst) (basic ∘ snd) (λ(m, y) → map (λz→(m−1, z)) (divide m y))
     (uncurry combine) (mm, x)
   = [specification itA]
itA  ((=0) ∘ fst) (basic ∘ snd) (λ(m, y) → map (λz→(m−1, z)) (divide m y))
     (uncurry combine) (mm, x)
   = [final form itA]
let (xss, nn, tss, lss) =
      while
        (λ (xss, r, tss, lss) → not (null (xss !! r)) )
        (λ (xss, r, tss, lss) →
            let  ds = [ if  (tss !! r !! p)
                             then [ ]
                             else  (λ(m, y) → map (λz→(m−1, z)) (divide m y))
                                      (xss !! r !! p)
                        | p←[0 . . #(xss !! r)−1] ]
                 ls = map # ds
                 xs = concat ds
                 ts = map ((=0) ∘ fst) xs
            in (xss++[xs], r+1, tss++[ts], lss++[ls]))
        ([[(mm, x)]], 0, [[mm=0]], [ ])
in fordown nn
        (λ r fsols →
            let bs = prefixsum (lss !! r)
```

```
      in [ if  tss!!r!!p
               then (basic ∘ snd) (xss!!r!!p)
               else  (λ(m, y) → (combine m y))
                     (xss!!r!!p)
                     [ fsols!!k | k←[bs!!p . . bs!!(p+1)−1] ]
         | p←[0 . . #(xss!!r)−1] ] )
    []
  !! 0
```

= $\boxed{\text{regrouping the tuple, separating the first component from } xss\,!!\,n\,!!\,p\text{: } m}$

```
let ((_, (xss, tss, lss)), nn) =
      while (λ ((m, (xss, tss, lss)), n) → not (null (xss!!n)) )
            (λ ((m, (xss, tss, lss)), n) →
               let  ds = [ if  (tss!!n!!p)  then []
                                            else divide m (xss!!n!!p)
                         | p←[0 . . #(xss!!n)−1] ]
                    ls = map # ds
                    xs = concat ds
                    ts = map (const (m=1)) xs
               in ((m−1, (xss++[xs], tss++[ts], lss++[ls])), n+1))
            ((mm, ([[x]], [[mm=0]], [ ])), 0)
    m = [mm, mm−1 . . 0]
in (fordown nn
     (λ r fsols →
        let bs = prefixsum (lss!!r)
        in [ if  tss!!r!!p
                 then basic (xss!!r!!p)
                 else  combine (m!!r) (xss!!r!!p)
                       [ fsols!!k | k←[bs!!p . . bs!!(p+1)−1] ]
           | p←[0 . . #(xss!!r)−1] ] )
     [ ])
     !! 0
```

= $\boxed{\text{simplifying and separating } tss}$

```
let ((_, (xss, _, lss)), nn) =
       while  (λ ((m, (xss, ts, lss)), n) → not (null (xss!!n)) )
              (λ ((m, (xss, ts, lss)), n) →
                 let  ds = [ if  ts!!n  then []
                                        else divide m (xss!!n!!p)
                           | p←[0 . . #(xss!!n)−1] ]
                      ls = map # ds
                      xs = concat ds
                 in ((m−1, (xss++[xs], ts++[m=1], lss++[ls])), n+1))
              ((mm, ([[x]], [mm=0], [ ])), 0)
    (ts, _) = while (λ(ts, n) → not (null (xss!!n)))
                    (λ(ts, n) → (ts++[mm−n=1], n+1))
                    ([mm=0], 0)
```

```
in (fordown nn
      (λ r fsols →
          let bs = prefixsum (lss !! r)
          in [ if  ts !! r then basic (xss !! r !! p)
                            else combine (mm−r) (xss !! r !! p)
                                     [ fsols !! k | k←[bs !! p . . bs !! (p+1)−1] ]
               | p←[0 . . #(xss !! r)−1] ] )
      []) !! 0
   = [ elimination of ts ]
let ((_, (xss, lss)), nn) =
       while (λ ((m, (xss, lss)), n) → not (null (xss !! n)) )
             (λ ((m, (xss, lss)), n) →
                 let  ds = [ if  mm=n then [] else divide m (xss !! n !! p)
                             | p←[0 . . #(xss !! n)−1] ]
                 in ((m−1, (xss++[concat ds], lss++[map # ds])), n+1))
             ((mm, ([[x]], [])), 0)
in (fordown nn
      (λ r fsols →
          let bs = prefixsum (lss !! r)
          in [ if  mm=r then basic (xss !! r !! p)
                            else combine (mm−r) (xss !! r !! p)
                                     [ fsols !! k | k←[bs !! p . . bs !! (p+1)−1] ]
               | p←[0 . . #(xss !! r)−1] ] )
    []) !! 0
   = [ exploiting overrun-tolerance of extending lists at the end ]
let ((m1, (xss1, lss1)), nn1) =
       while (λ ((m, (xss, lss)), n) → not (null (xss !! n)) )
             (λ ((m, (xss, lss)), n) →
                 let  ds = [ if  mm=n then [] else divide m (xss !! n !! p)
                             | p←[0 . . #(xss !! n)−1] ]
                 in ((m−1, (xss++[concat ds], lss++[map # ds])), n+1))
             ((mm, ([[x]], [])), 0)
    ((m, (xss, lss)), nn) =
       while (λ ((m, (xss, lss)), n) → m≥0)
             (λ ((m, (xss, lss)), n) → ((m−1, (xss++[[]], lss++[[]])), n+1))
             ((m1, (xss1, lss1)), nn1)
in (fordown nn
    (λ r fsols → let bs = prefixsum (lss !! r)
                 in [ if  mm=r
                           then basic (xss !! r !! p)
                           else  combine (mm−r) (xss !! r !! p)
                                     [ fsols !! k | k←[bs !! p . . bs !! (p+1)−1] ]
                      | p←[0 . . #(xss !! r)−1] ] )
      []) !! 0
   = [ fusing the while loops, regarding special case mm = 0 ]
```

```
let ((_, (xss, lss)), nn) =
     while
       (λ ((m, (xss, lss)), n) → not (m=0))
       (λ ((m, (xss, lss)), n) →
           let  ds = [ divide m (xss !! n !! p)
                     | p←[0 . . #(xss !! n)−1] ]
           in ((m−1, (xss++[concat ds], lss++[map # ds])), n+1))
       ((mm, ([[x]], [ ])), 0)
in (fordown (mm+1)
       (λ r fsols →
           let bs = prefixsum (lss !! r)
           in [ if  r=mm
                    then basic (xss !! r !! p)
                    else combine (mm−r) (xss !! r !! p)
                             [ fsols !! k | k←[bs !! p . . bs !! (p+1)−1] ]
                | p←[0 . . #(xss !! r)−1] ] )
       [ ]) !! 0
   = [Lemma 4.12 (while/for), unfolding fordown]
let (_, (xss, lss)) =
      for mm (λ (m, (xss, lss)) →
                  let ds = [ divide m (xss !! (mm−m) !! p)
                           | p←[0 . . #(xss !! (mm−m))−1] ]
                  in (m−1, (xss++[concat ds], lss++[map # ds])))
         (mm, ([[x]], [ ]))
in fordown mm
         (λ r fsols → let bs = prefixsum (lss !! r)
                      in [ combine (mm−r) (xss !! r !! p)
                                 [ fsols !! k | k←[bs !! p . . bs !! (p+1)−1] ]
                         | p←[0 . . #(xss !! r)−1] ] )
            [basic (xss !! mm !! p) | p←[0 . . #(xss !! mm)−1] ]
  !! 0
   = [replacing for by forup]
let  (xss, lss) =
       forup mm (λ r (xss, lss) →
                      let  ds = [ divide (mm−r) (xss !! r !! p)
                                | p←[0 . . #(xss !! r)−1] ]
                      in (xss++[concat ds], lss++[map # ds]))
                  ([[x]], [ ])
     fsols = [ basic (xss !! mm !! p) | p←[0 . . #(xss !! mm)−1] ]
in (fordown  mm ( λ r fsols →
                     let bs = prefixsum (lss !! r)
                     in [ combine  (mm−r) (xss !! r !! p)
                                     [ fsols !! k | k←[bs !! p . . bs !! (p+1)−1] ]
                        | p←[0 . . #(xss !! r)−1] ] )
          fsols) !! 0
```

B.3 itC

```
itC k basic divide combine n x
    =  specification itC
dcC k basic divide combine n x
    =  definition dcC
dcB basic (λ r x → [divide r x !! i | i←[0..k−1]]) combine n x
    =  specification itB
itB basic (λ r x → [divide r x !! i | i←[0..k−1]]) combine n x
    =  final form itB
let (xss, lss) =
        forup  n  (λ r (xss, lss) →
                        let  ds = [ [ divide (n−r) (xss!!r!!p) !! i
                                      | i←[0..k−1] ]
                                    | p←[0..#(xss!!r)−1] ]
                             ls = map # ds
                             xs = concat ds
                        in (xss++[xs], lss++[ls]))
              ([[x]], [ ])
     fsols = [ basic (xss!!n!!p)
             | p←[0..#(xss!!n)−1] ]
in fordown  n
              (λ r fsols →
                   let bs = prefixsum (lss!!r)
                   in  [ combine  (n−r) (xss!!r!!p)
                                  [ fsols!!q | q←[bs!!p..bs!!(p+1)−1] ]
                       | p←[0..#(xss!!r)−1] ] )
              fsols
   !! 0
    =  passing k
let (xss, lss) =
        forup  n
              (λ r (xss, lss) →
                   let  ds = [ [ divide (n−r) (xss!!r!!p) !! i
                                 | i←[0..k−1] ]
                               | p←[0..#(xss!!r)−1] ]
                        ls = map (const k) ds
                        xs = [ ds!!p!!i
                             |  p←[0..#(xss!!r)−1],
                                i←[0..k−1] ]
```

```
                in (xss++[xs], lss++[ls]))
             ([[x]], [ ])
     fsols = [ basic (xss !! n !! p)
             | p←[0 . . #(xss !! n)−1] ]
in fordown  n
             (λ r fsols →
                let bs = prefixsum (lss !! r)
                in  [ combine (n−r) (xss !! r !! p)
                          [ fsols !! q | q←[bs !! p . . bs !!(p+1)−1] ]
                    | p←[0 . . #(xss !! r)−1] ] )
             fsols
  !! 0
  = substitution of ds
let  (xss, lss) =
       forup  n
             (λ r (xss, lss) →
                let  ls = map (const k) [0 . . #(xss !! r)−1]
                     xs =  [ divide (n−r) (xss !! r !! p)  !!  i
                           |  p←[0 . . #(xss !! r)−1],
                              i←[0 . . k−1] ]
             in (xss++[xs], lss++[ls]))
             ([[x]], [ ])
     fsols = [ basic (xss !! n !! p)
             | p←[0 . . #(xss !! n)−1] ]
in fordown  n
             (λ r fsols →
                let bs = prefixsum (lss !! r)
                in  [ combine  (n−r) (xss !! r !! p)
                           [ fsols !! q | q←[bs !! p . . bs !!(p+1)−1] ]
                    | p←[0 . . #(xss !! r)−1] ] )
             fsols
  !! 0

  = substitution of ls
let  xss =
       forup  n
             (λ r xss →
                let xs =  [ divide (n−r) (xss !! r !! p)  !!  i
                          | p←[0 . . k^r−1],  i←[0 . . k−1] ]
                in xss++[xs])
             [[x]]
     fsols = [ basic (xss !! n !! p) | p←[0 . . k^n−1] ]
in fordown  n
             (λ r fsols →
                let bs = prefixsum (map (const k) [0 . . k^r−1])
```

$$
\begin{array}{l}
\qquad\qquad\qquad\quad \underline{\mathtt{in}}\ \ [\ \mathit{combine}\ \ (n{-}r)\ (\mathit{xss}\,!!\,r\,!!\,p) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\ \mathit{fsols}\,!!\,q \mid q{\leftarrow}[\mathit{bs}\,!!\,p\,..\,\mathit{bs}\,!!\,(p{+}1){-}1]\ ] \\
\qquad\qquad\qquad\qquad \mid p{\leftarrow}[0\,..\,k\,\hat{}\,r{-}1]\ ]\ ) \\
\qquad\qquad\quad \mathit{fsols} \\
\quad !!\ 0 \\
\quad = \boxed{\textsf{forup/scanforup, elimination of prefixsum}} \\
\underline{\mathtt{let}}\ \ \mathit{xss} = \textsf{scanforup}\ \ n \\
\qquad\qquad\qquad\qquad\quad (\lambda\ r\ \mathit{xs} \rightarrow\ [\ \mathit{subs}\,!!\,i \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\ p{\leftarrow}[0\,..\,k\,\hat{}\,r{-}1], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \underline{\mathtt{let}}\ \mathit{subs} = \mathit{divide}\ (n{-}r)\ (\mathit{xs}\,!!\,p), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad i{\leftarrow}[0\,..\,k{-}1]\ ]) \\
\qquad\qquad\qquad\qquad\quad [x] \\
\qquad \mathit{fsols} = [\ \mathit{basic}\ (\mathit{xss}\,!!\,n\,!!\,p) \mid p{\leftarrow}[0\,..\,k\,\hat{}\,n{-}1]\ ] \\
\underline{\mathtt{in}}\ \textsf{fordown}\ \ n \\
\qquad\qquad\quad (\lambda\ r\ \mathit{fsols} \rightarrow\ [\ \mathit{combine}\ \ (n{-}r)\ (\mathit{xss}\,!!\,r\,!!\,p) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\ \mathit{fsols}\,!!\,(k{*}p{+}i) \mid i{\leftarrow}[0\,..\,k{-}1]\ ] \\
\qquad\qquad\qquad\qquad\quad \mid p{\leftarrow}[0\,..\,k\,\hat{}\,r{-}1]\ ]\ ) \\
\qquad\qquad\quad \mathit{fsols} \\
\quad !!\ 0
\end{array}
$$

B.4 itD

```
itD k ids ods basic divide combine n x
   =  specification itD
dcD k ids ods basic divide combine n x
   =  definition dcD
let
 ba xs
  = let ys = basic [ xs!!i!!0 | i←[0..#ids−1] ]
    in  [ [ys!!j] | j←[0..#ods−1] ]
 di level inputs
   = let subs = divide level inputs
     in [ [ [ subs !! subprob !! component !! item
              | item←[0..(ids!!component)^(level−1)−1] ]
            | component←[0..#ids −1] ]
          | subprob←[0..k−1] ]
 co level inputs partsols
   = let sol = combine level inputs partsols
     in [ [ sol !! component !! part !! item
            | part←[0..ods!!component−1],
              item←[0..(ods!!component)^(level−1)−1] ]
          | component←[0..#ods −1] ]
in dcC k ba di co n x
   =  final form itC
let xss =
     scanforup n
       (λ r xs →
          [ subps !! q
          | p ←[0..k^r−1],
            let subps = let subs = divide (n−r) (xs!!p)
                        in  [ [ [ subs !! subprob !! component !! item
                                  | item←[0..(ids!!component)^(n−r−1)−1] ]
                                | component←[0..#ids −1] ]
                              | subprob←[0..k−1] ],
            q←[0..k−1]]) [x]
    ys = [ (λxs → let ys = basic [ xs!!i!!0 | i←[0..#ids−1] ]
                  in  [ [ys!!j] | j←[0..#ods−1] ])
           (xss!!n!!p)
         | p←[0..k^n−1] ]
in fordown n
      (λ r sols →
         [ let sol = combine (n−r) (xss!!r!!p)
                                 [ sols!!(k∗p+i) | i←[0..k−1] ]
```

```
          in [ [ sol !! component !! part !! item
               | part←[0..ods!!component−1],
                 item←[0..(ods!!component)^(n−r−1)−1] ]
             | component←[0..#ods −1] ]
         | p←[0..k^r−1]])
    ys
  !! 0
   = [simplification ys]
let xss =
      scanforup n
        (λ r xs →
           [ subps !! q
           | p ←[0..k^r−1],
             let subps = let subs = divide (n−r) (xs!!p)
                         in  [ [ [ subs !! subprob !! component !! item
                                 | item←[0..(ids!!component)^(n−r−1)−1] ]
                               | component←[0..#ids −1] ]
                             | subprob←[0..k−1] ],
             q←[0..k−1]]) [x]
    ys = [ let ys = basic [ xss!!n!!p!!i!!0 | i←[0..#ids−1] ]
           in  [ [ys!!j] | j←[0..#ods−1] ]
         | p←[0..k^n−1] ]
in fordown n (λ r sols →
                [ let sol = combine (n−r) (xss!!r!!p)
                                    [ sols!!(k*p+i) | i←[0..k−1] ]
                  in [ [ sol !! component !! part !! item
                       | part←[0..ods!!component−1],
                         item←[0..(ods!!component)^(n−r−1)−1] ]
                     | component←[0..#ods −1] ]
                | p←[0..k^r−1]])
           ys
    !! 0
```

B.5 itE

```
itE k ids ods basic divide combine n x
    = [specification itE]
dcE k ids ods basic divide combine n x
    = [definition dcE]
let  ba x = [ basic q x | q←[0..#ods −1] ]
     di level inputs
       = [ [ [ divide level subprob component item inputs
               | item←[0..(ids !! component)^(level−1)−1] ]
             | component←[0..#ids −1] ]
           | subprob←[0..k−1] ]
     co level inputs partsols
       = [ [ [ combine level component part item inputs partsols
               | item←[0..(ods !! component)^(level−1)−1] ]
             | part←[0..ods !! component−1] ]
           | component←[0..#ods −1] ]
in dcD k ids ods ba di co n x
    = [final form itD, list comprehension indexing]
let  xss =
      scanforup n
       (λ r xs →
          [ subps !! i
          | p ←[0..k^r−1],
            let subps = [ [ [ divide (n−r) subprob component item (xs !! p)
                                | item←[0..(ids !! component)^(n−r−1)−1] ]
                              | component←[0..#ids −1] ]
                            | subprob←[0..k−1] ],
            i←[0..k−1]]) [x]
     ys = [ [ [ basic j [ xss !! n !! p !! i !! 0 | i←[0..#ids−1] ] ]
            | j←[0..#ods−1] ]
          | p←[0..k^n−1] ]
in fordown n
    (λ r sols →
       [ [ [ combine   (n−r) component part item (xss !! r !! p)
                       [ sols !! (k∗p+j) | j←[0..k−1] ]
           | part←[0..ods !! component−1],
             item←[0..(ods !! component)^(n−r−1)−1] ]
         | component←[0..#ods −1] ]
       | p←[0..k^r−1]])
    ys !! 0
```

$= \boxed{\text{simplification}}$

```
let xss =
      scanforup n
      (λ r xs →
          [ [ [ divide (n−r) i component item (xs!!p)
                | item←[0..(ids!!component)^(n−r−1)−1] ]
              | component←[0..#ids −1] ]
          | p ←[0..k^r−1],
            i ←[0..k−1] ]) [x]
    ys = [ [ [ basic j [ xss!!n!!p!!i!!0 | i←[0..#ids−1] ] ]
             | j←[0..#ods−1] ]
           | p←[0..k^n−1] ]
in fordown n
   (λ r sols →
       [ [ [ combine (n−r) component part item (xss!!r!!p)
                         [ sols!!(k*p+j) | j←[0..k−1] ]
           | part←[0..ods!!component−1],
             item←[0..(ods!!component)^(n−r−1)−1] ]
         | component←[0..#ods −1] ]
       | p←[0..k^r−1]])
   ys !! 0
```

B.6 itF

itF k *indeg outdeg basic divide combine* n x

$= \boxed{\text{specification itF}}$

dcF k *indeg outdeg basic divide combine* n x

$= \boxed{\text{definition dcF}}$

```
let ba _ x = basic (x!!0)
    di level subprob _ item x
      = divide subprob [ x!!0!!(item+i*indeg^(level−1))
                       | i←[0..indeg−1] ]
    co _ _ part item _ sol = combine part [ sol!!j!!0!!item
                                          | j←[0..k−1] ]
in dcE k [indeg] [outdeg] ba di co n [x] !! 0
```

$= \boxed{\text{final form itE}}$

```
let xss =
    scanforup n
     (λ r xs →
        [ [ [ divide subprob  [ xs!!p!!0!!(item+q*indeg^(n−r−1))
                              | q←[0..indeg−1] ]
            | item←[0..([indeg]!!component)^(n−r−1)−1] ]
          | component←[0..#[indeg] −1] ]
        | p ←[0..k^r−1],
          subprob ←[0..k−1] ]) [[x]]
   ys = [ [ [ basic (xss!!n!!i!!0!!0) ]
            | j←[0..#[outdeg]−1] ]
          | i←[0..k^n−1] ]
in fordown n
    (λ r sols →
       [ [ [ combine part [ sols!!(k*p+j)!!0!!item
                          | j←[0..k−1] ]
           | part←[0..[outdeg]!!component−1],
             item←[0..([outdeg]!!component)^(n−r−1)−1] ]
         | component←[0..#[outdeg] −1] ]
       | p←[0..k^r−1]])
    ys
 !! 0 !! 0
```

$= \boxed{\text{simplification}}$

```
let xs =
    forup n
     (λ r xs →
        [ [ [ divide subprob [ xs!!p!!0!!(item+q*indeg^(n−r−1))
                             | q←[0..indeg−1] ]
            | item←[0..indeg^(n−r−1)−1] ]
          | component←[0] ]
        | p ←[0..k^r−1],
          subprob ←[0..k−1] ]) [[x]]
   ys = [ [ [ basic (xs!!i!!0!!0) ] ]
          | i←[0..k^n−1] ]
in fordown n
    (λ r sols →
       [ [ [  combine part  [ sols!!(k*p+j)!!0!!item
                            | j←[0..k−1] ]
           | part←[0..outdeg−1],
             item←[0..outdeg^(n−r−1)−1] ]
         | component←[0] ]
       | p←[0..k^r−1]])
    ys
 !! 0 !! 0
```

```
    = [nesting level reduction]
let xs =
    forup n
     (λ r xs →
        [ [ divide subprob [ xs!!p!!(item+q*indeg^(n-r-1))
                           | q←[0..indeg-1] ]
          | item←[0..indeg^(n-r-1)-1] ]
        | p ←[0..k^r-1],
          subprob ←[0..k-1] ]) [x]
   ys = [ [ basic (xs!!i!!0) ]
        | i←[0..k^n-1] ]
in fordown n
    (λ r sols →
       [ [  combine part [ sols!!(k*p+j)!!item
                         | j←[0..k-1] ]
         | part←[0..outdeg-1],
           item←[0..outdeg^(n-r-1)-1] ]
       | p←[0..k^r-1]])
    ys !! 0
    = [nesting level reduction]
let xs =
    forup n
     (λ r xs →
        [ divide subprob [ xs!!(item+(p*indeg+q)*indeg^(n-r-1))
                         | q←[0..indeg-1] ]
        | p ←[0..k^r-1],
          subprob ←[0..k-1],
          item←[0..indeg^(n-r-1)-1] ]) x
   ys = [ basic (xs!!i)
        | i←[0..k^n-1] ]
in fordown n
    (λ r sols →
       [ combine part [ sols!!((k*p+j)*outdeg^(n-r-1)+item)
                      | j←[0..k-1] ]
       | p←[0..k^r-1],
         part←[0..outdeg-1],
         item←[0..outdeg^(n-r-1)-1] ])
    ys
```

Appendix C

Target Skeleton itA in C

Here, we want to give the reader an idea of the correspondence between the iterative Haskell skeleton itA containing list comprehensions and a loop program in C.

The skeleton itA can easily be translated into C, if a parallel loop (`parfor`) is available, the customizing functions do not carry an environment, memory is never released and the overhead of a fully polymorphic implementation is taken into account.

Because our skeleton implementation for itA does not make these restrictive assumptions, the code looks much more complicated than what we present here.

The `parfor` loop behaves like the usual `for` loop with the additional assertion that the semantics is not affected by any interleaving of the iterations.

Before we present the skeleton implementation, we state the code for some auxiliary functions.

C.1 Auxiliary functions

The function `initList` allocates memory for a list of a particular length. `concat` flattens a list of lists. `snoc` appends an element at the end of a list. `prefixsum` takes a list of integers and delivers the prefix sum of this list.

```
#define parfor for

#define index(xs,i)  ((((List)(xs))->elem)[i])
#define len(xs)      (((List)(xs))->length)

typedef enum {False,True} Bool;

struct LIST { int length;
              void **elem; };

typedef struct LIST *List;
```

```
void *initList(int n)
{
  void *l;
  l=(void *)(malloc(sizeof(struct LIST)));
  len(l)=n;
  ((List)l)->elem=(void *)(malloc(n*sizeof(void *)));
  return(l);
}

void *concat(void *l)
{
  void *r, *e;
  int sumlen=0;
  int i,j,k;
  for(i=0;i<len(l);i++)
    sumlen += len(index(l,i));
  r = initList(sumlen);
  k=0;
  for(i=0;i<len(l);i++) {
    e=index(l,i);
    for(j=0;j<len(e);j++)
      index(r,k++) = index(e,j); }
  return(r);
}

void *snoc(void *ls, void *l)
{
  void *r;
  int i;
  r = initList(len(ls)+1);
  parfor(i=0;i<len(ls);i++)
    index(r,i)=index(ls,i);
  index(r,len(ls))=l;
  return(r);
}

void *prefixsum(void *ls)
{
   void *r;
   int i;
   r = initList(len(ls)+1);
   (int)index(r,0) = 0;
   for(i=0;i<len(ls);i++)
      (int)(index(r,i+1)) = (int)(index(r,i)) + ((int)index(ls,i));
   return(r);
}
```

C.2 Skeleton code

```
void *itA(void * (*istrivial)(), void * (*basic)(), void * (*divide)(),
          void * (*combine)(), void *x)
{
    void *xss, *tss, *lss, *cxs, *ds, *ls, *xs, *ts, *bs, *fsols,
         *newfsols, *sols;
    int n=0, p, r, i, j, count, lev;
    xs = initList(1);
    index(xs,0)=x;
    xss=initList(1);
    index(xss,0)=xs;
    lss=initList(0);
    ts = initList(1);
    index(ts,0)=istrivial(x);
    tss=initList(1);
    index(tss,0) = ts;
    while(len(index(xss,n))!=0) {
        ds = initList(len(index(xss,n)));
        ls = initList(len(index(xss,n)));
        parfor(p=0;p<len(index(xss,n));p++) {
            if(((Bool)(index(index(tss,n),p))) == True)
               index(ds,p) = initList(0);
            else
               index(ds,p) = divide(index(index(xss,n),p));
            index(ls,p) = (void *)len(index(ds,p));}
        xs = concat(ds);
        ts = initList(len(xs));
        parfor(p=0;p<len(xs);p++)
           index(ts,p) = istrivial(index(xs,p));
        xss = snoc(xss,xs); tss = snoc(tss,ts); lss = snoc(lss,ls);
        n++; }
    fsols = initList(0);
    for(r=n-1;r>=0;r--) {
        bs = prefixsum(index(lss,r));
        xs = index(xss,r);
        newfsols = initList(len(xs));
        parfor(i=0;i<len(xs);i++) {
            if(((Bool)index(index(tss,r),i)) == True)
                index(newfsols,i) = basic(index(xs,i));
            else {
                count = ((int)index(bs,i+1)) - ((int)index(bs,i));
                sols=initList(count);
                parfor(j=0;j<count;j++) {
                    index(sols,j) = index(fsols,j+((int)index(bs,i))); }
                index(newfsols,i) = combine (index(xs,i),sols); } }
        fsols=newfsols; }
    return(index(fsols,0));
}
```

Appendix D

Example Programs

D.1 Karatsuba's polynomial product

D.1.1 Haskell

```
left  xs = take (#xs ‘div‘ 2) xs
right xs = drop (#xs ‘div‘ 2) xs

karatsuba_hs ∈ [ℤ] → [ℤ] → [ℤ]
karatsuba_hs xs ys =
  if #xs = 1
    then [0, (xs !! 0)∗(ys !! 0)]
    else let xhs  = left xs
             xls  = right xs
             yhs  = left ys
             yls  = right ys
             hs   = karatsuba_hs xhs yhs
             ls   = karatsuba_hs xls yls
             ms   = karatsuba_hs (zipWith (+) xhs xls) (zipWith (+) yhs yls)
             mhls = zipWith3 (λ m h l → m−h−l) ms hs ls
             q0   = left hs
             q1   = zipWith (+) (right hs) (left mhls)
             q2   = zipWith (+) (right mhls) (left ls)
             q3   = right ls
         in  q0 ++ q1 ++ q2 ++ q3

main ∈ IO ()
main = do  s ← readFile "input"
           let  (a, rest):_ = reads s    ∈ [([ℤ]×String)]
                (b, _):_    = reads rest ∈ [([ℤ]×String)]
                c = karatsuba_hs a b
           writeFile "output" (show c)
           return ()
```

D.1.2 $\mathcal{HDC}$ program based on dcA

```
import HDCPrelude

karatsuba_dcA ∈ [ℤ] → [ℤ] → [ℤ]
karatsuba_dcA x y =
  let { trivial z  = #z  < 2 ;
        basic   z  = [0, fstPair (z!!0) ∗ sndPair (z!!0)] ;
        divide  z  = let {  a = map fstPair z;
                            b = map sndPair z;
                            h = left z;
                            l = right z }
                     in [h, l, zipWith  packPair
                                        (zipWith (+) (left a) (right a))
                                        (zipWith (+) (left b) (right b))] ;
        combine _ z
           = (λ[h, l, m] →  let mid = zipWith (−) m (zipWith (+) h l)
                            in ( left h ++
                                 zipWith (+) (right h) (left mid) ++
                                 zipWith (+) (left l) (right mid) ++
                                 right l)) z }
  in dcA trivial basic divide combine (zipWith packPair x y)


parmain ∈ IO Unit
parmain =  get >>= λxs →
           get >>= λys →
           put (karatsuba_dcA (xs∈[ℤ]) (ys∈[ℤ]))
```

D.1.3 $\mathcal{HDC}$ program based on dcF

```
import HDCPrelude

karatsuba_dcF ∈ [ℤ] → [ℤ] → [ℤ]
karatsuba_dcF a b =
  let  basic xy
        = packPair 0 (fstPair xy ∗ sndPair xy)
       divide i hl
        = if i<2  then hl!!i
                  else packPair  (fstPair (hl!!0) + fstPair (hl!!1))
                                 (sndPair (hl!!0) + sndPair (hl!!1))
       combine i [h, l, m]
        = if  i=0
          then packPair  (fstPair h)
                         (fstPair l + sndPair m − sndPair h − sndPair l)
```

else packPair (sndPair h + fstPair m − fstPair h − fstPair l)
(sndPair l)
in (dcF 3 2 2 *basic divide combine*
(ilog2 ($\#a$))
(zipWith packPair a b)) $\twoheadrightarrow$ ($\lambda z \to$
map fstPair z ++ map sndPair z)

parmain $\in$ IO Unit
parmain = get $\gg\!=$ $\lambda a \to$
get $\gg\!=$ $\lambda b \to$
put ((karatsuba_dcF a b) $\in$ $[\mathbb{Z}]$)

D.1.4 C program

```
#include <stdio.h>
#include "mpi.h"

void karatsuba(int size, int *a, int *b, int *l, int *h)
{
    int i;
    int *mh, *ml;
    int *ma, *mb;
    if (size==1) { h[0]=0; l[0]=a[0]*b[0]; return;
                 }
 karatsuba(size/2,a,b,l,l+size/2);
 karatsuba(size/2,a+size/2,b+size/2,h,h+size/2);
   ma = (int*)malloc(size*sizeof(int)/2);
   mb = (int*)malloc(size*sizeof(int)/2);
   for(i=0;i<size/2;i++) {
      ma[i] = a[i]+a[i+size/2];
      mb[i] = b[i]+b[i+size/2];
   }
   mh = (int*)malloc(size*sizeof(int)/2);
   ml = (int*)malloc(size*sizeof(int)/2);
 karatsuba(size/2,ma,mb,ml,mh);
   free(ma); free(mb);
   for(i=0;i<size/2;i++) {
      ml[i] -= (h[i]+l[i]);
      mh[i] -= (h[i+size/2]+l[i+size/2]);
   }
   for(i=0;i<size/2;i++) {
      l[i+size/2] += ml[i];
      h[i] += mh[i];
   }
   free(mh); free(ml);
   return;
}
```

```
int main(int argc,char **argv)
{
   double time_start, time_end;
   int *a, *b, *c;
   int n, i, j;
   FILE *in, *out;
   n = atoi(argv[1]);
   printf("%d\n",n);
   a = (int*)malloc(n*sizeof(int));
   b = (int*)malloc(n*sizeof(int));
   c = (int*)malloc(2*n*sizeof(int));
   in=fopen("input","r");
   fscanf(in,"[%d",&a[n-1]);
   for(i=1;i<n;i++)
       fscanf(in,",%d",&a[n-i-1]);
   fscanf(in,"][%d",&b[n-1]);
   for(i=1;i<n;i++)
       fscanf(in,",%d",&b[n-i-1]);
   fscanf(in,"]");
   fclose(in);
   MPI_Init(&argc,&argv);
   time_start = MPI_Wtime();
   karatsuba(n,a,b,c,c+n);
   time_end = MPI_Wtime();
   printf("time: %fs\n",time_end-time_start); fflush(stdout);
   MPI_Finalize();
   out=fopen("output","w");
   fprintf(out,"[%d",c[2*n-1]);
   for(i=2*n-2;i>=0;i--)
       fprintf(out,",%d",c[i]);
   fprintf(out,"]");
   fclose(out);
}
```

D.2 n queens problem

D.2.1 Haskell

$\mathsf{dcA} \in (\alpha{\to}\mathbb{B}){\to}(\alpha{\to}\beta){\to}(\alpha{\to}[\alpha]){\to}(\alpha{\to}[\beta]{\to}\beta){\to}(\alpha{\to}\beta)$
$\mathsf{dcA}\ p\ b\ d\ c = r$
$\quad \underline{\mathtt{where}}\ r\ x = \underline{\mathtt{if}}\ p\ x\ \ \underline{\mathtt{then}}\ b\ x$
$\qquad\qquad\qquad\qquad \underline{\mathtt{else}}\ c\ x\ (\mathsf{map}\ r\ (d\ x))$

$\mathsf{queens\_hs} \in \mathbb{Z} \to [[\mathbb{Z}]]$
$\mathsf{queens\_hs}\ n = \mathsf{dcA}\ \mathit{istrivial}\ \mathit{basic}\ \mathit{divide}\ \mathit{combine}\ (n, ([\,], [0\,..\,n{-}1]))\ \underline{\mathtt{where}}$
$\quad \mathit{istrivial}\ (n, \_) = n{=}0$
$\quad \mathit{basic}\quad (\_, (\mathit{placed}, \_)) = [\mathit{placed}]$

```
  divide (n, (placed, remain))
   = map  (λi→(n−1, (placed++[i], filter (≠i) remain)))
          (filter  (λj→ not (any  (λk →  (#placed − k)
                                          = abs (j − placed !! k))
                                     [0..#placed −1]))
                remain)
  combine _ = concat



main ∈ IO ()
main = do
         s ← readFile "input"
         let  (n, rest):_ = reads s    ∈ [(ℤ×String)]
               r = queens_hs n
         writeFile "output" (show r)
         return ()
```

D.2.2 $\mathcal{HDC}$ program based on dcA

Note that here dcA is not based on the C implementation of the dcA skeleton but on the user-defined function in App. D.2.1.

```
abs ∈ ℤ → ℤ
abs x = if x<0 then (−x) else x

queens_dcA ∈ ℤ → [[ℤ]]
queens_dcA n
 = dcA istrivial basic divide combine (n, ([], [0..n−1])) where
    istrivial (n, _) = n=0
    basic     (_, (placed, _)) = [placed]
    divide (n, (placed, remain))
      = seqmap  (λi→(n−1, (placed++[i], filter (≠i) remain)))
                (filter  (λj→ not (any ( λk → (#placed − k)
                                              = abs (j − placed !! k))
                                         [0..#placed −1]))
                      remain)
    combine _ = concat

parmain ∈ IO Unit
parmain =  get >>= λn →
           put (queens_dcA n)
```

D.3 Maximum independent set ($\mathcal{HDC}$)

```
elem ∈ Ord α ⇒ α→ [α] → 𝔹
elem x xs = any (=x) xs

adjacent (_, edges) u v = v∈(edges !! u)

neighborhood (nodes, edges) v =
  filter (λn → adjacent (nodes, edges) v n) nodes

remove_node ∈ ([ℤ]×[[ℤ]]) → ℤ → ([ℤ]×[[ℤ]])
remove_node (nodes, edges) u =
  ( filter (λx→x≠u) nodes,
    seqmap  (λi→if i=u then [] else filter (λx→x≠u) (edges !! i))
            [0..#edges −1])

select_node (nodes, edges) =
  (red max 1 (seqmap # edges) ↠ (λmaxval →
  (while (λi → #(edges !! (nodes !! i)) ≠ maxval) (+1) 0 ↠ (λmaxpos →
  (nodes !! maxpos)))))

max_independent_set_dcA ∈ ([ℤ]×[[ℤ]]) → [ℤ]
max_independent_set_dcA xs = dcA istrivial basic divide combine xs where
 istrivial (_, edges) = all (λx → #x =0) edges
 basic (nodes, _)  = nodes
 divide g = [  remove_node g (select_node g),
               foldl  (λ a b → remove_node a b)
                      (remove_node g (select_node g))
                      (neighborhood g (select_node g)) ]
 combine g [n1, n2] = if  #n1 > #n2
                          then n1
                          else select_node g : n2

parmain ∈ IO Unit
parmain =  get ≫= λx → put (max_independent_set_dcA x)
```

D.4 Convex hull ($\mathcal{HDC}$)

```
minindex ∈ Ord α⇒ [α] → ℤ
minindex xs =
 let (_, p, _) = while  (λ (i, pos, v) → (i<#xs))
                        (λ (i, pos, v) → if xs !! i < v  then (i+1, i, xs !! i)
                                                          else (i+1, pos, v))
                        (1, 0, xs !! 0)
 in p
```

```
maxindex ∈ Ord α⇒ [α] → ℤ
maxindex xs =
 let (_, p, _) = while  (λ (i, pos, v) → (i<#xs))
                        (λ (i, pos, v) → if xs !! i > v  then (i+1, i, xs !! i)
                                                          else (i+1, pos, v))
                        (1, 0, xs !! 0)
 in p

left_of ∈ IntPair → IntPair → IntPair → 𝔹
left_of a b c =
 let  ax = fstPair a
      ay = sndPair a
      bx = fstPair b
      by = sndPair b
      cx = fstPair c
      cy = sndPair c
 in (bx−ax)∗(cy−ay) > (by−ay)∗(cx−ax)

distancesq ∈ IntPair → IntPair → IntPair → Double
distancesq a b c =
 let  ax = fstPair a
      ay = sndPair a
      bx = fstPair b
      by = sndPair b
      cx = fstPair c
      cy = sndPair c
      sq x = x∗x
      mu = sqrt (  fromInt (sq (cx−ax) + sq (cy−ay)) /
                   fromInt (sq (bx−ax) + sq (by−ay)) )
      sx = fromInt ax + mu ∗ fromInt (bx−ax)
      sy = fromInt ay + mu ∗ fromInt (by−ay)
      ds = sq (fromInt cx − sx) + sq (fromInt cy − sy)
 in ds


subhull ∈ ([IntPair]×IntPair×IntPair) → [IntPair]
subhull xs = dcA istrivial basic divide combine xs
 where  istrivial (leftof, _, _) = #leftof < 2
        basic (leftof, u, _) = u : leftof
        divide (leftof, u, v)
         =  maxindex (seqmap (distancesq u v) leftof) ↠ λi →
            leftof !! i ↠ λw →
            [  (filter (left_of u w) leftof, u, w),
               (filter (left_of w v) leftof, w, v)]
        combine  _ [as, bs] = as++bs
```

```
quickhull_dcA ∈ [(ℤ×ℤ)] → [(ℤ×ℤ)]
quickhull_dcA ps =
 seqmap (λ (x, y) → packPair x y) ps  ↠ λxs →
 seqmap fstPair xs                    ↠ λx0s →
 xs !! (minindex x0s)                 ↠ λxmin →
 xs !! (maxindex x0s)                 ↠ λxmax →
 seqmap  (λx → (fstPair x, sndPair x))
         (concat (map subhull [  (filter (left_of xmin xmax) xs, xmin, xmax),
                                 (filter (left_of xmax xmin) xs, xmax, xmin)]))

parmain ∈ IO Unit
parmain =  get ≫= λxs → put (quickhull_dcA (xs∈[(ℤ×ℤ)]))
```

D.5 qsort_dcA ($\mathcal{HDC}$)

```
qsort_dcA ∈ [ℤ] → [ℤ]
qsort_dcA ys =
 let  istrivial xs  = #xs < 2
      basic xs      = xs
      pivot xs      = ((xs!!0)+(xs!!(#xs ‘div‘ 2))+(xs!!(#xs −1))) ‘div‘3
      divide xs     = [filter (< pivot xs) xs, filter (> pivot xs) xs]
      combine xs [ys, zs] = ys ++ (filter (= pivot xs) xs) ++ zs
 in dcA istrivial basic divide combine ys

parmain ∈ IO Unit
parmain =  get ≫= λx → put (qsort_dcA (x∈[ℤ]))
```

D.6 Optimized bitonic sort ($\mathcal{HDC}$)

```
bitonicOpt ∈ (ℤ×[ℤ]) → [ℤ]
bitonicOpt (n, xs) = dcA istrivial basic divide combine (n, xs) where
   istrivial (n, _) = n=0
   basic (_, x)     = qsort_dcA x
   divide (n, x)    = [(n−1, left x), (n−1, right x)]
   combine _ [x, y]
    = let  b x = x
           d i [x, y] = if i=0 then min x y else max x y
           c i xs = xs!!i
      in dcF 2 2 2 b d c (1+ ilog2(#x)) (x ++ reverse y)

parmain ∈ IO Unit
parmain = get ≫= λn → get ≫= λx → put (bitonicOpt (n∈ℤ, x∈[ℤ]))
```

Appendix E

Skeleton Types

(table on the next page)

skeleton	degree of problem division	degree(s) of input data decomposition	degree(s) of output data composition	predicate *istrivial*	*basic* function	*divide* function	*combine* function	recursion depth	input data	output data
dcA				$\alpha \to \mathbb{B}$	$\alpha \to \beta$	$\alpha \to [\alpha]$	$\alpha \to [\beta] \to \beta$		α	β
dcB					$\alpha \to \beta$	$\mathbb{N} \to \alpha \to [\alpha]$	$\mathbb{N} \to \alpha \to [\beta] \to \beta$	$\mathbb{N}$	α	β
dcC	$\mathbb{N}$				$\alpha \to \beta$	$\mathbb{N} \to \alpha \to [\alpha]$	$\mathbb{N} \to \alpha \to [\beta] \to \beta$	$\mathbb{N}$	α	β
dcD	$\mathbb{N}$	$[\mathbb{N}]$	$[\mathbb{N}]$		$[\alpha] \to [\beta]$	$\mathbb{N} \to [[\alpha]] \to [[[\alpha]]]$	$\mathbb{N} \to [[\alpha]] \to [[[\beta]]] \to [[[\beta]]]$	$\mathbb{N}$	$[[\alpha]]$	$[[\beta]]$
dcE	$\mathbb{N}$	$[\mathbb{N}]$	$[\mathbb{N}]$		$\mathbb{N} \to [\alpha] \to \beta$	$\mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to [[\alpha]] \to \alpha$	$\mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N} \to [[\alpha]] \to [[[\beta]]] \to \beta$	$\mathbb{N}$	$[[\alpha]]$	$[[\beta]]$
dcF	$\mathbb{N}$	$\mathbb{N}$	$\mathbb{N}$		$\alpha \to \beta$	$\mathbb{N} \to [\alpha] \to \alpha$	$\mathbb{N} \to [\beta] \to \beta$	$\mathbb{N}$	$[\alpha]$	$[\beta]$

Bibliography

Bruce Abramson and Mordechai (Moti) Yung. Divide and conquer under global constraints: A solution to the n-queens problem. *J. Parallel and Distributed Computing*, 6(3):649–662, June 1989.

Klaus Achatz and Wolfram Schulte. Architecture independent massive parallelization of divide-and-conquer algorithms. In Bernhard Möller, editor, *Mathematics of Program Construction (MPC'95)*, Lecture Notes in Computer Science 947, pages 97–127. Springer-Verlag, 1995.

Klaus Achatz and Wolfram Schulte. Massive parallelization of divide-and-conquer algorithms over powerlists. *Science of Computer Programming*, 26: 59–78, 1996.

Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Series in Computer Science and Information Processing. Addison-Wesley , 1974.

Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

Selim G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, 1989.

George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* Series in Computer Science and Engineering. Benjamin/Cummings, 2nd edition, 1994.

G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conf. Proc.*, Vol. 30, pages 483–485. Thompson Books, April 1967.

Margarita Amor López. *Divide and Conquer Algorithms onto Mesh Computers.* PhD thesis, Dept. of Electronics and Systems, Univ. de Santiago de Compostela, Spain, 1997. In Spanish.

Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. *ACM SIGPLAN Notices*, 26(7), 39–50, 1991.

Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

Mikhail Atallah, editor. *Algorithms and Theory of Computation Handbook.* CRC Press, 1999.

Tom Axford. The divide-and-conquer paradigm. In Lydia Kronsjö and Dean Shumsheruddin, editors, *Advances in Parallel Algorithms*, Chapter 2, pages 26–65. Blackwell, 1992.

Bruno Bacci, Marco Danelutto, Salvatore Orlando, Susanna Pelagatti and Marco Vanneschi. P^3L: A structured high-level programming language, and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995.

John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, 21(8):613–641, Aug. 1978.

John Backus. The algebra of functional programs: Function-level reasoning, linear equations, and extended definitions. In J. Diaz and I. Ramos, editors, *Formalization of Programming Concepts*, Lecture Notes in Computer Science 107, pages 1–43. Springer-Verlag, 1981.

Hendrik P. Barendregt. *The Lambda Calculus.* Studies in Logic, Vol. 103. North-Holland, 1984.

Michel Barreteau and Paul Feautrier. Automatic mapping of scans and reductions. In Vincent van Dongen, editor, *Proc. High Performance Computing Symp. '95*, pages 15–26, July 1995.

Kenneth E. Batcher. Sorting networks and their applications. In *Proc. 1968 Spring Joint Computer Conf.*, Vol. 32, pages 307–314. AFIPS Press, 1968.

Jeffrey M. Bell, Françoise Bellegarde and James Hook. Type-driven defunctionalization. *ACM SIGPLAN Notices*, 32(8):25–37, 1997.

Jon L. Bentley. Multidimensional divide-and-conquer. *Comm. ACM*, 23(4): 214–229, 1980.

Kenneth A. Berman and Jerome L. Paul. *Fundamentals of Sequential and Parallel Algorithms.* PWS Publishing Company, Boston, 1996.

D.I. Bevan. Distributed garbage collection using reference counting. In J.W. de Bakker, A.J. Nijman and P.C. Treleaven, editors, *PARLE: Parallel Architectures and Languages Europe, Vol. II*, Lecture Notes in Computer Science 259, pages 176–187. Springer-Verlag, 1987.

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

Richard S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.

Richard S. Bird. *Introduction to Functional Programming using Haskell.* Series in Computer Science. Prentice-Hall Europe, 2nd edition, 1998.

Guy E. Blelloch. scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, 1989.

Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.

Guy E. Blelloch. Prefix sums and their applications. In John H. Reif, editor, *Synthesis of Parallel Algorithms*, Chapter 1, pages 35–60. Morgan Kaufmann, 1993.

Guy E. Blelloch. NESL: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, Dept. of Computer Science, Carnegie Mellon Univ., 1995.

Guy E. Blelloch. Programming parallel algorithms. *Comm. ACM*, 39(3):85–97, 1996.

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.

Eerke A. Boiten, A. Max Geerling and Helmuth A. Partsch. Transformational derivation of (parallel) programs using skeletons. Technical Report 93-20, Katholieke Univ. Nijmegen, Sept. 1993.

George H. Botorog and Herbert Kuchen. Algorithmic skeletons for adaptive multigrid methods. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'95)*, Lecture Notes in Computer Science 980, pages 27–41. Springer-Verlag, 1995.

George H. Botorog and Herbert Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proc. Fifth IEEE Int. Symp. on High Performance Distributed Computing*, pages 243–252. IEEE Computer Society Press, 1996.

Tore A. Bratvold. A skeleton-based parallelising compiler for ML. In Rinus Plasmeijer and Marko van Eekelen, editors, *Proc. 5th Int. Workshop on the Implementation of Functional Languages (IFL'93)*, pages 23–33. Katholieke Univ. Nijmegen, 1993.

Tore A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs.* PhD thesis, Dept. Computing and Electrical Engineering, Heriot-Watt Univ. Edinburgh, Nov. 1994.

Silvia Breitinger, Ulrike Klusik and Rita Loogen. An implementation of Eden on top of Concurrent Haskell. In Werner Kluge, editor, *Implementation of Functional Languages (IFL'96)*, Lecture Notes in Computer Science 1268, pages 141–161. Springer-Verlag, 1997.

Silvia Breitinger, Rita Loogen and Yolanda Ortega-Mallén. Towards a declarative language for parallel and concurrent programming. In David N. Turner, editor, *Proc. 1995 Glasgow Workshop on Functional Programming (GFPW'95)*, Electronic Workshops in Computing. Springer-Verlag, July 1995. http://www.ewic.org.uk.

Theodore Brown and Renbing Xiong. A parallel quicksort algorithm. *J. Parallel and Distributed Computing*, 19(2):83–89, Oct. 1993.

Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, Jan. 1977.

David J. Busvine. *Detecting Parallel Structures in Functional Programs.* PhD thesis, Dept. Computing and Electrical Engineering, Heriot-Watt Univ. Edinburgh, Oct. 1993.

David C. Cann. Retire Fortran? A debate rekindled. *Comm. ACM*, 35(8): 81–89, 1992.

B. Carpentieri and Zhijing George Mou. Compile-time transformations and optimization of parallel divide-and-conquer algorithms. *ACM SIGPLAN Notices*, 26(10):19–28, 1991.

Wei-Ngan Chin and John Darlington. A higher order removal method. *Lisp and Symbolic Computation*, 9(4):287–322, 1996.

Wei-Ngan Chin and Siau-Cheng Khoo. Tupling Functions with Multiple Recursion Parameters. In Patrick Cousot, Moreno Falaschi, Gilberto Filè and Antoine Rauzy, editors, *Static Analysis, Third Int. Workshop (WSA'93)*, Lecture Notes in Computer Science 724, pages 124–140. Springer-Verlag, 1993.

Silvia Ciarpaglini, Marco Danelutto, Laura Folchi, Carlo Manconi and Susanna Pelagatti. ANACLETO: A template-based p3l compiler. In *Proc. Seventh Parallel Computing Workshop (PCW'97)*, pages P2-F-1–7. Australian National Univ., Aug. 1997.

Volker Claus and Andreas Schwill. *Duden Informatik.* Dudenverlag, 2. Auflage, 1993. In German.

Fabien Coelho, Cécile Germain and Jean-Louis Pazat. State of the art in compiling HPF. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, Lecture Notes in Computer Science 1132, pages 104–133. Springer-Verlag, 1996.

Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

Murray I. Cole. On dividing and conquering independently. In Christian Lengauer, Martin Griebl and Sergei Gorlatch, editors, *Euro-Par'97: Parallel Processing*, Lecture Notes in Computer Science 1300, pages 634–637. Springer-Verlag, 1997.

Richard Cole. Parallel merge sort. *SIAM J. Computing*, 17(4):770–785, Aug. 1988.

Jean-François Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. Parallel Programming*, 23(2):191–219, 1995.

James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

G. Cousineau, P.-L. Curien and M. Mauny. The categorial abstract machine. In Jean-Pierre Jouannaud, editor, *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA'85)*, Lecture Notes in Computer Science 201, pages 50–64. Springer-Verlag, 1985.

William J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Trans. on Computers*, 39(6):775–785, 1990.

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.

John Darlington and Rod M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6:41–60, 1976.

John Darlington, Anthony J. Field, Peter G. Harrison, Paul H.J. Kelly, David W.N. Sharp, Qian Wu and Ronald L. While. Parallel programming using skeleton functions. In Arndt Bode, Mike Reeve and Gottfried Wolf, editors, *PARLE'93: Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 694, pages 146–160. Springer-Verlag, 1993.

John Darlington, Yi-ke Guo, Hing Win To and Jin Yang. Parallel skeletons for structured composition. *ACM SIGPLAN Notices*, 30(8):19–28, 1995.

Alain Darte and Frédéric Vivien. Automatic parallelization based on multi-dimensional scheduling. Research Report RR1994-24, LIP, ENS Lyon, Sept. 1994.

I.P. de Guzmán, Peter G. Harrison and E. Medina. Pipelines for divide-and-conquer functions. *The Computer Journal*, 36(3):254–268, 1993.

L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Comm. ACM*, 19(9):522–526, 1976.

Nils Ellmenreich, Christian Lengauer and Martin Griebl. Application of the polytope model to functional programs. In Larry Carter and Jeanne Ferrante, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 1863, pages 219–235. Springer-Verlag, 2000.

Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2:155–173, 1982.

Martin S. Feather. A survey and classification of some program transformation approaches and techniques. In Lambert G.L.T. Meertens, editor, *Program Specification and Transformation*, pages 165–195. North-Holland, 1987.

Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, 1992.

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.

Yosee Feldman and Ehud Shapiro. Spatial machines: A more realistic approach to parallel computation. *Comm. ACM*, 35(10):60–73, 1992.

John T. Feo, David C. Cann and Rodney R. Oldehoeft. A report on the Sisal language project. *J. Parallel and Distributed Computing*, 10(4):349–366, Dec. 1990.

Raphael A. Finkel and Udi Manber. DIB - a distributed implementation of backtracking. *ACM Trans. on Programming Languages and Systems*, 9(2): 235–256, April 1987.

Sigbjørn Finne, Daan Leijen, Erik Meijer and Simon L. Peyton Jones. Calling hell from heaven and heaven from hell. *ACM SIGPLAN Notices*, 34(9):114–125, 1999.

Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

Geoffrey C. Fox. Parallel computing comes of age: Supercomputer level parallel computations at Caltech. *Concurrency: Practice and Experience*, 1(1):63–103, Sept. 1989.

Luis A. Galán, Cristóbal Pareja and Ricardo Peña. Functional skeletons generate process topologies in Eden. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs (PLILP'96)*, Lecture Notes in Computer Science 1140, pages 289–303. Springer-Verlag, 1996.

Michael R. Garey and David S. Johnson. *Computers and Intractability*. W.H. Freeman & Co., 1979.

Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

A. Max Geerling. Formal derivation of SIMD parallelism from non-linear recursive specifications. In Bruno Buchberger and Jens Volkert, editors, *Parallel Processing: CONPAR94–VAPP VI*, Lecture Notes in Computer Science 854, pages 136–147. Springer-Verlag, 1994.

Alfons Geser and Sergei Gorlatch. Parallelizing functional programs by generalization. *J. Functional Programming*, 9(6):649–673, Nov. 1999.

Alan Gibbons and Ridha Ziani. The balanced binary tree technique on mesh-connected computers. *Information Processing Letters*, 37:101–109, Jan. 1991.

J.C. Glas, R.F.H. Hofman and W.G. Vree. Parallelization of branch-and-bound algorithms in a functional programming environment. In Herbert Kuchen and Rita Loogen, editors, *Proc. 4th Int. Workshop on the Parallel Implementation of Functional Languages (IFL'92)*. Univ. Aachen, 1992.

Allen Goldberg, Peter Mills, Lars Nyland, Jan Prins, John Reif and James Riely. Specification and development of parallel algorithms with the Proteus system. In Guy E. Blelloch, K. Mani Chandy and Suresh Jagannathan, editors, *Specification of Parallel Algorithms*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 18, pages 383–399. AMS Press, 1994.

Marc Goodman and Zhijing George Mou. $\mathcal{DC}$ transpose: A method for reducing communication in divide-and-conquer algorithms on mesh-based computers. In *Proc. Third IEEE Symp. on Parallel and Distributed Processing*, pages 218–221. IEEE Computer Society Press, 1991.

Sergei Gorlatch. From transformations to methodology in parallel program development: A case study. *Microprocessing and Microprogramming*, 41:571–588, 1996/a.

Sergei Gorlatch. Stages and transformations in parallel programming. In M. Kara, J.R. Davy, D. Goodeve and J. Nash, editors, *Abstract Machine Models*, pages 147–161. IOS Press, 1996/b.

Sergei Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte and Yves Robert, editors, *Euro-Par'96: Parallel Processing, Vol. II*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996/c.

Sergei Gorlatch. $\mathcal{N}$-graphs: Scalable topology and design of balanced divide-and-conquer algorithms. *Parallel Computing*, 23:687–698, 1997/a.

Sergei Gorlatch. Optimizing compositions of scans and reductions in parallel program derivation. Technical Report MIP-9711, Fakultät für Mathematik und Informatik, Univ. Passau, May 1997/b.

Sergei Gorlatch. Programming with divide-and-conquer skeletons: A case study of FFT. *J. Supercomputing*, 12(1-2):85–97, 1998.

Sergei Gorlatch and Holger Bischof. A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Processing Letters*, 8(4):447–458, 1998.

Sergei Gorlatch and Christian Lengauer. Parallelization of divide-and-conquer in the Bird-Meertens formalism. *Formal Aspects of Computing*, 7:663–682, 1995.

Sergei Gorlatch and Susanna Pelagatti. A transformational framework for skeletal programs: Overview and case study. In José D.P. Rolim et al., editors, *Parallel and Distributed Processing: Workshops at IPPS/SPDP'99*, Lecture Notes in Computer Science 1586, pages 123–137. Springer-Verlag, 1999.

Martin Griebl and Christian Lengauer. On the space-time mapping of while-loops. *Parallel Processing Letters*, 4(3):221–232, 1994.

Martin Griebl and Christian Lengauer. A communication scheme for the distributed execution of loop nests with while loops. *Int. J. Parallel Programming*, 23(5):471–496, 1995.

Martin Griebl and Christian Lengauer. Classifying loops for space-time mapping. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte and Yves Robert, editors, *Euro-Par'96: Parallel Processing, Vol. I*, Lecture Notes in Computer Science 1123, pages 467–474. Springer-Verlag, 1996.

Robert Günz. Automatische Verifikation von Gleichheitsbeweisen in Haskell. Diploma thesis, Fakultät für Mathematik und Informatik, Univ. Passau, Aug. 1998. In German.

John L. Gustafson. Reevaluating Amdahl's law. *Comm. ACM*, 31(5):532–533, 1988.

Robert H. Halstead JR. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

Per Hammarlund and Björn Lisper. On the relation between functional and data parallel programming languages. In *Proc. Sixth Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 210–222. ACM Press, June 1993.

Kevin Hammond. *Parallel SML: A Functional Language and its Implementation in Dactl.* Research Monographs in Parallel and Distributed Computing. Pitman, 1991.

Kevin Hammond. Personal communication at the IFL'98 conference, Sept. 1998.

Kevin Hammond. Personal communication at the Dagstuhl seminar on High Level Parallel Programming, April 1999.

Chris Hankin. *Lambda Calculi, A Guide for Computer Scientists.* Clarendon Press, Oxford, 1994.

Peter G. Harrison. Linearisation: An optimisation for nonlinear functional programs. *Science of Computer Programming*, 10:281–318, 1988.

Peter G. Harrison and Hessam Khoshnevisan. A new approach to recursion removal. *Theoretical Computer Science*, 93:91–113, 1992.

Haskell: http://www.haskell.org, 2000.

Philip Heidelberger, Alan Norton and John T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Trans. on Computers*, 39(1):133–138, 1990.

P.C. Held and A.C.J. Kienhuis. div, floor, ceil, mod and step functions in nested loop programs and linearly bounded lattices. In Marc Moonen and Francky Catthoor, editors, *Algorithms and Parallel VLSI Architectures III*, pages 271–282. Elsevier Science BV, 1995.

Christoph A. Herrmann, Jan Laitenberger, Christian Lengauer and Christian Schaller. Static parallelization of functional programs: Elimination of higher-order functions & optimized inlining. In Patrick Amestoy, Philippe Berger, Michel Daydé, Iain Duff, Valérie Frayssé, Luc Giraud and Daniel Ruiz, editors, *Euro-Par'99: Parallel Processing*, Lecture Notes in Computer Science 1685, pages 930–934. Springer-Verlag, 1999/a.

Christoph A. Herrmann and Christian Lengauer. Notes on the space-time mapping of divide-and-conquer recursions. In *Proc. GI/ITG FG PARS'95*, PARS Mitteilungen Nr.14, pages 132–139. Gesellschaft für Informatik e.V., 1995.

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.

Christoph A. Herrmann and Christian Lengauer. Parallelization of divide-and-conquer by translation to nested loops. Technical Report MIP-9705, Fakultät für Mathematik und Informatik, Univ. Passau, March 1997/a.

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: Implementations, Logics, and Programs (PLILP'97)*, Lecture Notes in Computer Science 1292, pages 95–109. Springer-Verlag, 1997/b.

Christoph A. Herrmann and Christian Lengauer. Size inference of nested lists in functional programs. In Kevin Hammond, Tony Davie and Chris Clack, editors, *Proc. 10th Int. Workshop on the Implementation of Functional Languages (IFL'98)*, pages 347–364. Univ. College London, Sept. 1998.

Christoph A. Herrmann and Christian Lengauer. Parallelization of divide-and-conquer by translation to nested loops. *J. Functional Programming*, 9(3): 279–310, May 1999.

Christoph A. Herrmann and Christian Lengauer. The $\mathcal{HDC}$ compiler project. In Alain Darte, Georges-André Silber and Yves Robert, editors, *Proc. Eighth Int. Workshop on Compilers for Parallel Computers (CPC 2000)*, pages 239–253. LIP, ENS Lyon, 2000.

Christoph A. Herrmann, Christian Lengauer, Robert Günz, Jan Laitenberger and Christian Schaller. A compiler for $\mathcal{HDC}$. Technical Report MIP-9907, Fakultät für Mathematik und Informatik, Univ. Passau, May 1999/b.

W. Daniel Hillis. *The Connection Machine.* ACM Distinguished Dissertations. MIT Press, 1985.

W. Daniel Hillis and Guy L. Steele JR. Data parallel algorithms. *Comm. ACM*, 29(12):1170–1183, 1986.

C. Anthony R. Hoare. Algorithm 63, partition; Algorithm 64, quicksort. *Comm. ACM*, 4(7):321, July 1961.

Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proc. 14th Ann. IEEE Symp. on Logic in Computer Science (LICS'99)*, pages 464–473. IEEE Computer Society Press, July 1999.

Ellis Horowitz and Alessandro Zorat. Divide-and-conquer for parallel processing. *IEEE Trans. on Computers*, 32(6):582–585, 1983.

Zhenjiang Hu, Masato Takeichi and Hideya Iwasaki. Diffusion: Calculating efficient parallel programs. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, NS-99-1, pages 85–94. BRICS, 1999.

A.J. Huang and Zhijing George Mou. Parallel partition expansion for the solution of arbitrary recurrences. In Quentin F. Stout, editor, *Proc. 1992 Int. Conf. on Parallel Processing. Vol. III: Algorithms & Applications*, pages 109–113. CRC Press, Aug. 1992.

Chua-Huang Huang, Jeremy R. Johnson and R.W. Johnson. Generating parallel programs from tensor product formulas: A case study of Strassen's matrix multiplication algorithm. In Quentin F. Stout, editor, *Proc. 1992 Int. Conf. on Parallel Processing, Vol. III: Algorithms & Applications*, pages 104–108. CRC Press, Aug. 1992.

Chua-Huang Huang and Christian Lengauer. The automated proof of a trace transformation for a bitonic sort. *Theoretical Computer Science*, 46:261–284, 1986.

Paul Hudak. Para-functional programming. *IEEE Computer*, 19:60–71, Aug. 1986.

Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

Paul Hudak. *The Haskell School of Expression.* Cambridge University Press, 2000.

Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5):T1–T53, May 1992.

John Hughes. Why functional programming matters. *Computer Journal*, 32(2): 98–107, 1989.

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.

Joseph Jájá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

C. Barry Jay, D.G. Clarke and J.J. Edwards. Exploiting shape in parallel programming. In *Proc. IEEE Second Int. Conf. on Algorithms and Architectures for Parallel Processing*, pages 295–302. IEEE, 1996.

C. Barry Jay and Milan Sekanina. Shape checking of array programs. In J. Harland, editor, *Proc. Australasian Theory Seminar on Computing*, Vol. 19, *Australian Computer Science Communications*, pages 113–121, 1997.

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, pages 190–203. Springer-Verlag, 1985.

A. Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962. In Russian.

Richard M. Karp, Raymond E. Miller and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

Owen Kaser, C.R. Ramakrishnan and Shaunak Pawagi. A new approach to inlining. Technical Report 92/06, Computer Science Dept., SUNY at Stony Brook, NY, Sept. 1992.

Owen Kaser, C.R. Ramakrishnan and Shaunak Pawagi. On the conversion of indirect to direct recursion. *ACM Letters on Programming Languages and Systems*, 2(1–4):151–164, 1993.

Paul H.J. Kelly. *Functional Programming for Loosely-coupled Multiprocessors.* Pitman, 1989.

Donald E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, 2nd edition, 1998.

Jacob Kornerup. Mapping a functional notation for parallel programs onto hypercubes. *Information Processing Letters*, 53:153–158, 1995.

Jacob Kornerup. Odd-even sort in powerlists. *Information Processing Letters*, 61:15–24, 1997/a.

Jacob Kornerup. Parlists – A generalization of powerlists (extended version). Technical Report CS-TR-97-15, Univ. Texas, Austin, June 1997/b.

Jacob Kornerup. Plists: Taking powerlists beyond base two. In Sergei Gorlatch, editor, *First Int. Workshop on Constructive Methods for Parallel Programming (CMPP'98)*, pages 102–116. Technical Report MIP-9805, Fakultät für Mathematik und Informatik, Univ. Passau, May 1998.

B. Kumar, Chua-Huang Huang, P. Sadayappan and R.W. Johnson. A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction. *Scientific Programming*, 4(4):275–289, 1995.

K.G. Kumar and David B. Skillicorn. Data parallel geometric operations on lists. *Parallel Computing*, 21:447–459, 1995.

Jan Laitenberger. Codeoptimierung für strikte funktionale Programme. Diploma thesis, Fakultät für Mathematik und Informatik, Univ. Passau, June 1999. In German.

Leslie Lamport. The parallel execution of DO loops. *Comm. ACM*, 17(2):83–93, 1974.

Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.

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.

Christian Lengauer, Sergei Gorlatch and Christoph A. Herrmann. The static parallelization of loops and recursions. *J. Supercomputing*, 11(4):333–353, Dec. 1997.

Björn Lisper. Single-assignment semantics for imperative programs. In Eddy Odijk, Martin Rem and Jean-Claude Syre, editors, *PARLE'89: Parallel Architectures and Languages Europe, Vol. II: Parallel Languages*, Lecture Notes in Computer Science 366, pages 321–334. Springer-Verlag, 1989/a.

Björn Lisper. *Synthesizing Synchronous Systems by Static Scheduling in Space-Time.* Lecture Notes in Computer Science 362. Springer-Verlag, 1989/b.

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.

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.

Virginia Lo, Sanjay Rajopadhye, Jan Arne Telle and Xiaoxiong Zhong. Parallel divide and conquer on meshes. *IEEE Trans. on Parallel and Distributed Systems*, 7(10):1049–1058, 1996.

Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. on Programming Languages and Systems*, 4(2):258–282, April 1982.

Ernst W. Mayr and Ralph Werchner. Divide-and-conquer algorithms on the hypercube. *Theoretical Computer Science*, 162:283–296, 1996.

D.L. McBurney and M.R. Sleep. Transputer-based experiments with the ZAPP architecture. In J.W. de Bakker, A.J. Nijman and P.C. Treleaven, editors, *PARLE: Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 258, Vol. I: Parallel Architectures, pages 242–259. Springer-Verlag, 1987.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine. Part I. *Comm. ACM*, 3(4):184–204, 1960.

Geoff P. McKeown, Vic J. Rayward-Smith and H.J. Turpin. Branch-and-bound as a higher-order function. *Annals of Operations Research*, 33:379–402, 1991.

Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Trans. on Programming Languages and Systems*, 16(6):1737–1767, Nov. 1994.

Zhijing George Mou. Divacon: A parallel language for scientific computing based on divide-and-conquer. In *Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation*, pages 451–461. IEEE Computer Society Press, Oct. 1990/a.

Zhijing George Mou. *A Formal Model for Divide-and-Conquer and Its Parallel Realization.* PhD thesis. Technical Report YALEU/DCS/RR-795. Dept. Computer Science, Yale Univ., May 1990/b.

Zhijing George Mou, Cornel Constantinescu and Timothy J. Hickey. Divide-and-conquer on three-dimensional meshes. In Wouter Joosen and Elie Milgrom, editors, *Parallel Computing: From Theory to Sound Practice*, pages 344–355. IOS Press, 1992.

Zhijing George Mou and Paul Hudak. An algebraic model for divide-and-conquer and its parallelism. *J. Supercomputing*, 2(3):257–278, 1988.

Zhijing George Mou and Xiaojing Wang. Optimal mappings of m dimensional FFT communication to k dimensional mesh for arbitrary m and k. In Arndt Bode, Mike Reeve and Gottfried Wolf, editors, *PARLE'93: Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 694, pages 104–119. Springer-Verlag, 1993.

MPI Forum. *MPI: A Message-Passing Interface Standard.* Univ. of Tennessee at Knoxville, 1995.

MPI Forum. *MPI-2: Extensions to the Message-Passing Interface.* Univ. of Tennessee at Knoxville, 1997.

Marian Musiol. Implementierung von parallelem Divide-and-Conquer auf Gittertopologien. Diploma thesis, Fakultät für Mathematik und Informatik, Univ. Passau, May 1996. In German.

David Nassimi and Sartaj Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. on Computers*, 28(1):2–7, 1979.

John T. O'Donnell. An architecture that efficiently updates associative aggregates in applicative programming languages. In Jean-Pierre Jouannaud, editor, *Proc. Conf. on Functional Programming Languages and Computer Architecture (FPCA'85)*, Lecture Notes in Computer Science 201, pages 164–189. Springer-Verlag, 1985.

John T. O'Donnell. Data parallel implementation of extensible sparse functional arrays. In Arndt Bode, Mike Reeve and Gottfried Wolf, editors, *PARLE'93: Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 694, pages 68–79. Springer-Verlag, 1993.

John T. O'Donnell. Bidirectional fold and scan. In *Functional Programming: Glasgow 1993*, Workshops in Computing, pages 193–200. Springer-Verlag, 1994/a.

John T. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, Sept. 1994/b.

John T. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In Pieter H. Hartel and Rinus Plasmeijer, editors, *Functional Programming Languages in Education*, Lecture Notes in Computer Science 1022, pages 195–214. Springer-Verlag, 1995.

John T. O'Donnell. Lecture on functional programming at the University of Passau, July 1997.

John T. O'Donnell and Gudula Rünger. A methodology for deriving parallel programs with a family of parallel abstract machines. In Christian Lengauer, Martin Griebl and Sergei Gorlatch, editors, *Euro-Par'97: Parallel Processing*, Lecture Notes in Computer Science 1300, pages 662–669. Springer-Verlag, 1997.

Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

Helmuth A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

Helmuth A. Partsch and Peter Pepper. A family of rules for recursion removal. *Information Processing Letters*, 5(6):174–177, Dec. 1976.

V. Paúl Pauca, Xiaobai Sun, Siddhartha Chatterjee and Alvin R. Lebeck. Architecture-efficient Strassen's matrix multiplication: A case study of divide-and-conquer algorithms. Technical Report CS-1998-06, Dept. Computer Science, Duke Univ., May 1998.

Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

Susanna Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs*. PhD thesis 11/93, Dept. Computer Science, Univ. Pisa, March 1993.

Susanna Pelagatti. Compiling and supporting skeletons on MPP. In *Proc. Third Working Conf. on Massively Parallel Programming Models (MPPM'97)*, pages 140–150. IEEE Computer Society Press, 1997.

Susanna Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

Peter Pepper. Deductive derivation of parallel programs. In Robert Paige, John Reif and Ralph Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 1–53. Kluwer, 1993.

Peter Pepper, Jürgen Exner and Mario Südholt. Functional development of massively parallel programs. In Dines Bjørner, Manfred Broy and Igor V. Pottosin, editors, *Formal Methods in Programming and Their Applications*, Lecture Notes in Computer Science 735, pages 217–238. Springer-Verlag, 1993.

Rózsa Péter. *Recursive Functions in Computer Theory*. Ellis Horwood, 1981.

Frans J. Peters. Tree machines and divide-and-conquer algorithms. In Wolfgang Händler, editor, *CONPAR'81*, Lecture Notes in Computer Science 111, pages 25–36. Springer-Verlag, 1981.

Simon L. Peyton Jones, Andrew Gordon and Sigbjørn Finne. Concurrent Haskell. *POPL'96: The 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages*, pages 295–308. ACM Press, 1996.

Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conf.*, Keele, July 1993.

Simon L. Peyton Jones and John Hughes, editors. Haskell 98: A non-strict, purely functional language. Technical report, http://haskell.org, Feb. 1999.

Simon L. Peyton Jones and André L.M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, Sept. 1998.

Michael J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, 2nd edition, 1994.

Patrice Quinton. The systematic design of systolic arrays. In Françoise Fogelman-Soulié, Yves Robert and Maurice Tchuenté, editors, *Automata Networks in Computer Science*, Chapter 9, pages 229–260. Manchester University Press, 1987. Also: Technical Reports 193 and 216, IRISA (INRIA-Rennes), 1983.

Fethi A. Rabhi. Exploiting parallelism in functional languages: A "paradigm-oriented" approach. In John R. Davy and Peter M. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, pages 118–139. Oxford University Press, 1995.

Fethi A. Rabhi and Gordon A. Manson. Divide-and-conquer and parallel graph reduction. *Parallel Computing*, 17(3):189–205, 1991.

Sanjay V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3:88–105, 1989.

Xavier Redon and Paul Feautrier. Detection of recurrences in sequential programs with loops. In Arndt Bode, Mike Reeve and Gottfried Wolf, editors, *PARLE'93: Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 694, pages 132–145. Springer-Verlag, 1993.

Karl Rüdiger Reischuk. *Einführung in die Komplexitätstheorie.* Leitfäden und Monographien der Informatik. B.G. Teubner Stuttgart, 1990. In German.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conf.*, pages 717–740. ACM Press, 1972.

H. Gordon Rice. Recursion and iteration. *Comm. ACM*, 8(2):114–115, Feb. 1965.

Christian Schaller. Elimination von Funktionen höherer Ordnung in Haskell-Programmen. Diploma thesis, Fakultät für Mathematik und Informatik, Univ. Passau, Sept. 1998. In German.

Jacob T. Schwartz, Robert B.K. Dewar, E. Dubinsky and Edmund Schoenberg. *Programming with Sets: An Introduction to Setl.* Springer-Verlag, 1986.

Robert Sedgewick. *Algorithms.* Addison-Wesley, 2nd edition, 1988.

Yuji Shinano, Masahiro Higaki and Ryuichi Hirabayashi. An interface design for general parallel branch-and-bound algorithms. In Afonso Ferreira, José Rolim, Yousef Saad and Tao Yang, editors, *Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96)*, Lecture Notes in Computer Science 1117, pages 277–284. Springer-Verlag, 1996.

Stephen K. Skedzielewski. Sisal. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Chapter 4. ACM Press Frontier Series, 1991.

David B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, 1990.

David B. Skillicorn. The Bird-Meertens formalism as a parallel model. In Janusz S. Kowalik and Lucio Grandinetti, editors, *Software for Parallel Computation*, NATO ASI Series F: Computer and Systems Sciences, Vol. 106, pages 120–133. Springer-Verlag, 1992.

David B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *J. Parallel and Distributed Computing*, 28(1):65–83, July 1995.

Neil J.A. Sloane. AT&T integer sequence A000170 (n queens), June 2000. http://www.research.att.com/~njas/sequences/index.html.

Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

Dan Suciu and Val Tannen. Efficient compilation of high-level data parallel algorithms. In *Proc. 6th Ann. Symp. on Parallel Algorithms and Architectures*, pages 57–66. ACM Press, June 1994.

Simon Thompson. *Haskell – The Craft of Functional Programming.* Int. Computer Science Series. Addison-Wesley, 1996.

Phil W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl and Simon L. Peyton Jones. Algorithm + strategy = parallelism. *J. Functional Programming*, 8(1):23–60, 1998.

David A. Turner. Elementary strong functional programming. In Pieter H. Hartel and Rinus Plasmeijer, editors, *Functional Programming Languages in Education*, Lecture Notes in Computer Science 1022, pages 1–13. Springer-Verlag, 1995.

Eric Violard. A mathematical theory and its environment for parallel programming. *Parallel Processing Letters*, 4(3):313–328, 1994.

Eric Violard, Stéphane Genaud and Guy-René Perrin. Refinement of data parallel programs in Pei. In Richard S. Bird and Lambert Meertens, editors, *Algorithmic Languages and Calculi*, Chapter 5, pages 107–131. Chapman & Hall, 1997.

Eric Violard and Guy-René Perrin. Pei: A language and its refinement calculus for parallel programming. *Parallel Computing*, 18:1167–1184, 1992.

Philip Wadler. Deforestation: Transforming programs to eliminate trees. In Harald Ganzinger, editor, *ESOP'88*, Lecture Notes in Computer Science 300, pages 344–358. Springer-Verlag, 1988.

Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In J.W. de Bakker, A.J. Nijman and P.C. Treleaven, editors, *PARLE: Parallel Architectures and Languages Europe, Vol. II*, Lecture Notes in Computer Science 259, pages 432–443. Springer-Verlag, 1987.

Herbert S. Wilf. *Algorithms and Complexity.* Prentice-Hall, 1986.

John H. Williams. On the development of the algebra of functional programs. *ACM Trans. on Programming Languages and Systems*, 4(4):733–757, Oct. 1982.

Noel Winstanley. Parallel programming by transformation. In Patrick Amestoy, Philippe Berger, Michel Daydé, Iain Duff, Valérie Frayssé, Luc Giraud and Daniel Ruiz, editors, *Euro-Par'99: Parallel Processing*, Lecture Notes in Computer Science 1685, pages 858–865. Springer-Verlag, 1999.

Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1995.

C. Xavier and S.S. Iyengar. *Introduction to Parallel Algorithms.* John Wiley & Sons, 1998.

Index

Top-Level Definitions

List of Figures

List of Tables

Legend of Symbols

$\mathfrak{CD}$ / $\mathfrak{IS}$	computation domain / index space
$\mathsf{Ord}\ \alpha$	elements of type α can be compared
$\log x$ / $\log_b x$	natural logarithm of x / logarithm of x to base b
$\mathbb{B}$ / $\mathbb{N}$ / $\mathbb{Z}$	the Booleans / naturals / integers
α / β / γ	type variables
δ	the actual dimension in a compact description
$\lambda x \to y$	lambda-abstraction: function with argument x and result y
$\alpha \to \beta$	type of a function with domain α and codomain β
$x \leftarrow xs$	list comprehension generator, takes successively x from xs
σ / τ	space/time function (allocation/schedule)
Δ	dimensionality of an index space
Γ	single generator function
Ψ	list of structural parameters
$\lfloor x \rfloor$	greatest integer not exceeding x
$[\ x\ ]$	list with a single element x
$[\ a, b, c\ ]$	list with elements a, b, and c
$(e)_i$	series of expressions e indexed by i
(a, b, c)	Cartesian product of elements a, b and c
$[\,]$	empty list
$[a\,..\,b]$	arithmetic sequence: list containing the integers from a to b
$[\ e \mid q\ ]$	list comprehension with element description e and qualifiers q
$(e[x := y])$	substitute in e every free occurrence of x by y
$(e[\delta\ := y])$	substitute in e every occurrence of δ by y
$(e[\mathsf{Item} := y])$	substitute in e every occurrence of Item by y
$(e[\Rsh i := y])$	substitute in e every reference to coordinate i by y
$\Rsh i$	coordinate value of dimension i in a compact description
$g \circ f$	functional composition
$\bigcirc_{i=a}^{b}\ (f\ i)$	functional composition $f\ a \circ ... \circ f\ b$
$\downarrow$ / $\uparrow$	minimum/maximum operator
$\Downarrow$ / $\Uparrow$	compact description of the lower/upper bounds of an index set
$\#$	length of a list
$\boxplus$	function which constructs a concrete index space
$:$	adding an element at the front of a list
$\mathbf{!!}$	indexing a list
$+\!\!+$	append of two lists
$\alpha \times \beta$	Cartesian product of types α and β
$\bowtie$	compact description of the operations in a domain
$\mathsf{=}$	equal by definition in the Haskell semantics
$\overset{\text{def}}{=}$	equal by mathematical definition
$=$	(a) comparison operator in Haskell, (b) proposition of equality in mathematical expressions
$<_{\texttt{lex}}$	lexicographically smaller
$\prec$	data dependence relation