Data Structures for Parallel Recursion

by

Jacob Kornerup, Cand. Scient.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1997

Data Structures for Parallel Recursion

Approved by
Dissertation Committee:

To Kathy

Acknowledgments

This work could not have been done without the continuing support of my advisor, Jayadev Misra. He taught me many aspects of the research process, how to ask the right questions, and how to pursue a problem until the elegant solution "fits on a page". He shared his ideas for the PowerList notation very early in its development, and encouraged me to pursue my ideas about mapping the powerlist notation onto hypercubes. Through the dissertation process he continually shared his ideas with me, and encouraged me to continue my work building on his research. His contributions were essential in the development of the ParList notation; the PowerList descriptions and isomorphism proofs of the binary interconnection networks presented in Chapter 4 are due to him.

Edsger W. Dijkstra generously invited me to join the *Austin Tuesday Afternoon Club* in 1990, where rule 0 reads "Don't make a mess of it". A rule that I have tried to avoid violating in my research. He taught me to think clearly about the problem at hand, to present my solution in a concise manner, and that underneath all the flashy buzzwords computer science is just that, a science.

Jørgen Staunstrup was the advisor on my master's thesis at the University of Århus. He encouraged me to open my eyes to the world and to continue my studies abroad. He has been a constant source of support and a valued source of information ranging from the area of formal verification to the design of VLSI circuits.

Greg Plaxton has been a wonderful source of references and deep insights into the area of parallel algorithms. He was always able to point me to relevant literature no matter what my question was. I also want to thank Mohamed Gouda and Vladimir Lifschitz for serving on my dissertation committee and working with me in scheduling the defense.

This work has benefited by discussions with the following researchers to whom I am grateful for their time and interest: Guy Blelloch, F. Tom Leighton, Ernie Cohen, Roland Backhouse and a handful of anonymous referees, who pointed me towards weaknesses in earlier treatments of the structures. I want to thank the members of the Austin Tuesday Afternoon Club who spent two sessions reading a draft of the derivation of the odd-even sort, and suggested numerous improvements that greatly enhanced its presentation.

I have many fellow students to thank for their help, support and companionship during my "tenure" at UT Austin. First, I want to thank Al Carruth, who was my officemate for many years. He encouraged me to work on mapping PowerList onto hypercubes and helped me when I got stuck in my proofs. Edgar Knapp was the first person I met from UT Austin. He was a very dear friend and mentor to me during my first years in Austin, and gave me many good ideas for research directions and valuable feedback on my early work.

Rajeev Joshi has been extremely helpful with my research. He has always been available to listen to an idea or to read and comment on a rough draft. I have learned to trust his judgment, and I know that he will become a very successful computer scientist one day. He is learning from all the right people. Markus Kaltenbach also helped me by reading and commenting very constructively on past papers and drafts of this dissertation. Will Adams provided many insights into the PowerList notation; he also challenged me to describe the odd-even sort in PowerLists.

Marco Schneider and I have been synchronizing our studies almost down to the minute since we started at UT Austin. Thanks for many wonderful shared experiences, on and off campus.

A number of other students also deserve mention for their comments, feedback and support through this process: Emory Berger, Ken Calvert, Tim Collins, Sarah Chodrow, Ruben Gamboa, Pete Manolios, Kedar Namjoshi, Dave Naumann, Carlos Puchol, Lyn Pierce, Rajaraman Rajmohan, J.R. Rao, Mark Staskauskas, Torsten Suel and Richard Trefler.

I want to thank my brother Jesper, who provided me with the wisdom that quitting my teaching job and focusing on finishing my dissertation was the best investment I could ever make in my future. He was right.

My parents Margot and Peter were a constant source of support and love through the dissertation process. This document has benefited from a careful reading from Peter, who spent a part of his vacation reading and commenting on a draft version. Thanks for everything.

Finally and most importantly, I want to thank my wife Kathy Scherer for all her loving and care. I would not have been able to complete this work if it had not been for her. She shared the experiences from her dissertation process and gave me the support I needed when I was in doubt. I am sorry for the late hours and my absence in the last few months. This dissertation is for you.

Jacob Kornerup

The University of Texas at Austin
December 1997

Data Structures for Parallel Recursion

Publication No. ____________________

Jacob Kornerup, Ph.D.
The University of Texas at Austin, 1997

Supervisor: Jayadev Misra

Parallel programming is still considered a difficult and error-prone activity. There are no universally accepted abstractions that both capture the essence of most parallel architectures, and are useful to the programmer in expressing parallel computations.

In this dissertation we present three data structures: PowerList, ParList, and PList that can be used to describe parallel computations in a succinct manner. We define an algebra along with the inductive definition of each structure. Parallel computations are expressed as recursive functions over these definitions. The algebras facilitate equational reasoning allowing functions representing parallel computations to be derived from their specification.

PowerLists are linear structures whose lengths are a power of two. We extend Misra's results [Mis94], by describing a mapping of the PowerList operators to hypercubic architectures and by formally deriving the odd-even transposition sort.

The ParList structure is an extension to the PowerList structure that allows inputs of arbitrary lengths. We present the theory and show how functions and properties of PowerLists can be extended to ParLists.

The PowerList structure is closely related to binary numbers. We generalize PowerLists to PLists by introducing basic constructors that take an arbitrary, positive number of similar PLists and return a single PList. PLists can be used to reason algebraicly about manipulations on mixed-radix representation of natural numbers with minimal use of index notations. Using PLists we describe four generalized interconnection networks and show that they are isomorphic.

Contents

Chapter 1

Introduction

"The arguments for parallel operation are only valid provided one applies them to the steps which the built in or wired in programming of the machine operates. Any steps which are controlled by the operator, who sets up the machine, should be set up only in a serial fashion. It has been shown over and over again that any departure from this procedure results in a system which is far too complicated to use."

The above statement was made by J. P. Eckert, Jr. in 1946 [Eck46] (as cited in [Kel89]) in an argument for parallel data transfer and arithmetic in the computers of the EDVAC's generation. The statement gave a rather pessimistic outlook for the spread of parallel programming as performed by the operator. Since 1946 a number of innovations such as high level languages, compilers, theories and logics have made parallel programming an easier and more reliable task. But, even with these advances, it can be argued that Eckert's statement can be accepted today. Parallel programming is still difficult and error prone in comparison with sequential programming, where tools and methodologies are available to produce correct and reliable programs. This does not mean that sequential programs are error free and can be delivered on schedule, but today it is well understood how to write such programs correctly.

There are a number of success stories in parallel programming. Most of the so called "grand challenge problems" such as large-scale simulations and modeling problems can only be solved by exploiting parallelism. This is usually done on parallel hardware sold by the super-computing industry. An industry that has grown as government and defense agencies and corporate research facilities have increased their demand for computational power. Parallel programming is also necessitated by dedicated parallel hardware such as digital signal processors and imaging hardware. These applications for parallel programming tend to stay close to the "built in or wired in programming" mentioned by Eckert, i.e., they are performed by highly trained experts. A pessimist may still adopt the view that non-expert programmers who attempt to write a parallel program, will produce "a system which is far too complicated to use." We will not advocate such a pessimistic view in this dissertation, but the premise does stand that parallel programming is difficult, and methodologies need to be developed to make the task simpler and more reliable. This dissertation is an attempt to present such a methodology, a simple way to express parallel computations while allowing formal verifications of their correctness.

This dissertation is written with a firm belief in the use of *Formal Methods*, that is the use of tools and techniques based on mathematical models of computations and architectures in the development of software. Combining this belief with an identification of the main tasks in software development, we identify the following three stages in the software development process:

Specification: Constructing a formal description of the problem to be solved. All participants in the development process should be in agreement that this description captures the essence of the problem. The specification should be amenable to a formal proof that can verify that a program meets it.

Program construction: The task of creating an artifact that meets the specification of the problem and can be implemented efficiently on the target architecture.

Implementation The mapping of the the program onto the computing resources available for its execution. This should be done using tools or techniques that have been formally verified or has been under intense scrutiny by a large pool of users. A commonly used compiler is an example of such a tool.

In sequential programming the implementation phase is usually handled by a compiler capable of producing optimized code for the target architecture. Such compilers are possible since most sequential architectures are similar at higher levels of abstraction, and differences at the lower levels can be exploited by the compiler to generate efficient code for each architecture. The similar architectures have the added advantage that high level languages can be designed with ease of expression in mind without sacrificing efficient compilation strategies. This gives programmers of sequential systems the luxury of focusing on the correctness and abstract complexity of their programs, letting the compiler implement their programs efficiently and correctly.

Parallel architectures are very different from each other, even at higher levels of abstraction[1]. One approach to designing a parallel programming language is to let the language reflect a particular architecture, and aim for efficient compilations to the architecture. This approach may be successful for writing programs for a specific architecture, for example signal processing software for customized hardware. This is not an attractive solution, since most programming problems are not phrased in terms of concrete architectures, leaving the programmer with the wrong set of abstractions. Another approach is to abstract away from most architectural considerations, and build a programming language around abstract notions of parallelism. This approach allows the programmer to concentrate on "what needs to be done"

[1]We will discuss some of these differences in Section 1.3.2

rather than "how it can be done on the architecture." We take the second approach in this dissertation, by presenting three formalisms for expressing parallel programs as recursive functions. These formalisms are designed to utilize the symmetry found in most parallel architectures by emphasizing symmetric programming constructs.

A significant amount of parallel software is now written in sequential programming languages extended with "parallelism constructs." These constructs allow the programmer to specify how selected portions of the program can be executed in parallel, and how data are assigned to physical processors. Examples of such languages are FORTRAN-90 [ANSI90], C* [RS87], *Lisp [Las86], multiC [Wav92] and many others. The use of such languages is close to an acceptance of Eckert's statement, since the programmer is only trusted with a sequential language.

Some compilers are designed with this acceptance in mind. They examine the sequential parts of the program in an attempt to discover independent threads of control that can be parallelized. So far there are no success stories where such compilers were able to identify significant amounts of parallelism from sequential programs, except where the independence between subcomputations (i.e., parallelism) is obvious even in a sequential description.

1.1 Synchronous Parallel Programming

In this dissertation we focus on synchronous parallel programming, characterized by architectures where a collection of processing elements operate in a "lock step" manner. A synchronous parallel programming notation should capture the fact that many things happen at once during a parallel execution. Take as an example the problem of incrementing each element of a vector of numbers by one. The increment operations are independent, and can be performed in any order including simultaneous application. On a parallel machine where the number of available processors exceeds the length of the vector, this operation can be achieved in a single step, if each element can be accessed directly by its dedicated processor. This example falls into a particular category of parallel computations: a function is applied independently to each element of a linear structure. It is important to be able to capture these computations in a parallel programming notation, but we do not stop with such constructs; they can already be expressed in sequential languages with extensions for parallelism.

Many programming problems have solutions that can be characterized as *divide and conquer*. Depending on the size and structure of the problem it is either solved directly, or broken into smaller problems that can be solved recursively; these solutions are then combined to form the answer. The increment problem described above has a simple divide and conquer description: the element of a single-element vector is incremented; a vector whose length is at least two is broken into two disjoint sub vectors, where the increment operation is performed separately on each component; finally, the resulting sub vectors are combined in their original order to form the resulting vector. Note that we presented an abstract solution to the problem. An implementation may choose to apply the increment directly to each element of the vector; this would be consistent with the divide and conquer solution.

We base our theories on functions over linear data structures. We define these structures as symmetrically as possible, in order to equalize the size of the recursive tasks generated by the divide and conquer strategy. This is done to exploit the symmetry present in most parallel architectures. We give two different ways to construct linear structures from a pair of equally sized sub-structures, either by concatenation or by interleaving[2]. The choice between these two constructions give the programmer an extra degree of freedom, and makes it possible to express many parallel programs elegantly.

Our choice of a functional notation is influenced by the successes of sequen-

[2]These constructions will be clarified further when we define the PowerList, ParList, and PList notations in Chapters 2, 3 and 4.

tial functional programming languages. A functional program can often be specified directly by presenting a function that solves the problem. This function may not be the most efficient implementation available, as we saw with the increment problem above. It is often possible to prove that functional descriptions representing more efficient implementations compute the same function. These proofs can often be done using *equational reasoning*, that is through a series of identity preserving steps that transforms one function into the other. Some proofs can be done as *derivations*: starting from the specification function, the efficient implementation emerges through a series of equality preserving steps. Ideally, the choice of the next transformation is given from the shape of the current formula and heuristics that have been developed for the particular problem domain. We will see this technique used throughout this dissertation. In this work we spend our efforts on creating the right abstractions to express certain parallel computations, rather than presenting a complete programming language. We leave the detailed design decisions of a full-fledged programming language to others.

We use algebraic constructors for the data structures as the only way to access their individual elements. It is the author's belief that a main reason why parallel programming is so difficult today is the widespread use of *indexing* notations in parallel programming languages. The programming task becomes a painstaking bookkeeping job, and proofs of correctness become nightmares rather than creative and enlightening activities. Many architectures and algorithms are built on nice algebraic properties; it is our goal to capture these properties as elegantly as possible in the structures and theories we define.

1.2 Basic Definitions and Notations

In this section we introduce the type system, notations and conventions that will be used in the rest of this dissertation.

1.2.1 Types

In this dissertation we define three linear data structures: PowerList, ParList and PList. We introduce a simple type system, to be precise about the types of these structures and the use of types in general. We use the name Type to denote the class of all types of interest[3]. We will not do any manipulations on the class Type as a whole, only on elements of it.

We use the following basic data types:

Type name	Defined as
Nat	The natural numbers
Pos	The positive natural numbers
Real	The real numbers
Com	The complex numbers
Bool	The booleans

In Chapter 4 we define aggregate types such as linear lists (List), strings (String) and sets (Set). We use the *type variables* A, B, C, L, M, X, Y and Z to denote elements of Type and, unless explicitly restricted in the text, these variables are universally quantified over Type. The type of a function is specified by giving the name of the function, its domain and its range. For example, the successor function *succ* on Nat has the type $succ : \mathsf{Nat} \longrightarrow \mathsf{Nat}$. We use $\times$ to denote pairing of types (e.g., $\mathsf{Nat} \times \mathsf{Nat}$ is the type of all pairs on natural numbers). This notation will primarily be used to denote that a function has more than one argument. We use exponentiation of a type by a natural number to denote the multi-way pairing of a type (e.g., Nat^2 denotes the same type as $\mathsf{Nat} \times \mathsf{Nat}$).

We formalize the types of the PowerList, ParList and PList structures by introducing a type function for each structure. These functions take two arguments,

[3]If this definition troubles the reader, then it should be ignored and Type should only be thought of as a name of some abstract collection.

a type and a length, and return the type of all instances of the structure with elements of the given type and length equal to the given length. As an example we define ParList by the type function[4]

$$\mathsf{ParList} : \mathsf{Type} \times \mathsf{Pos} \longrightarrow \mathsf{Type} \tag{1.1}$$

which returns the type containing all ParLists with elements from Type, whose length is as specified by the second argument. For example, ParList.Nat.2 denotes the type of all ParLists of length 2 with natural numbers as elements. We overload the name ParList by having it denote the type of all ParLists (corresponding to ParList.X.n for all X and n). We also use ParList as a name for the algebra we define to prove properties of ParLists.

We will only write expressions that have a correct type as defined by the type constructors for the structure. In the ParList algebra, for example, when we write $p \bowtie q$ it is understood that p and q are similar ParLists (i.e., both members of ParList.X.n for some X and n). When we write $a \triangleright p$ we assume that a is similar to the elements of p. These conventions apply to the PowerList and PList algebras as well. A type that is not a PowerList, ParList or a PList is called a *scalar type*.

1.2.2 Operator Priority

We give different binding powers to binary operators, as prescribed by the table below, to minimize the use of parentheses in expressions. The operators in the table are grouped in decreasing binding power downwards; within a group the operators have the same binding power:

[4] A function like ParList is often called a type constructor [MTH90].

$$.$$

$$\rightarrow \quad \leftarrow \quad \overset{G}{\rightarrow} \quad \overset{G}{\leftarrow}$$

$$\diamond \quad *$$

$$\uparrow \quad \downarrow \quad \oplus \quad \otimes \quad + \quad \odot \quad \star \quad \bullet \quad \div \quad \mathrm{mod} \quad +\!\!+$$

$$\triangleright \quad \triangleleft$$

$$\bowtie \quad | \quad \bowtie^{G} \quad |^{G}$$

$$\circ \quad \Diamond$$

$$\leq \quad < \quad = \quad \neq \quad \prec \quad \sim$$

$$\Rightarrow \quad \Leftarrow$$

$$\wedge \quad \vee$$

$$\equiv$$

$$\times$$

$$\longrightarrow$$

1.2.3 Notation and Proof Style

We will use a notation similar to that presented by Dijkstra and Scholten [DS90], which includes writing function application by an infix, left associative dot "." operator; for example, $f.x$ denotes the value returned by the function f when applied to the argument x . When a function has more than one argument we write a sequence of arguments separated by the dot operator, denoting an implicit currying of the function. For example, $g.a.b$ is the same as $(g.a).b$ by left associativity, since we identify $g : \mathsf{A} \times \mathsf{B} \longrightarrow \mathsf{C}$ with $g : \mathsf{A} \longrightarrow (\mathsf{B} \longrightarrow \mathsf{C})$. By currying we have $g.a : \mathsf{B} \longrightarrow \mathsf{C}$.

Proofs and derivations are written with lines containing formulas separated by lines that have a relational operator and a piece of text (called a "hint") that explains why the preceding and the following line are related by the operator. For instance, a proof of the boolean expression $P \Rightarrow R$ is true could be written as:

Proof of $P \;\Rightarrow\; R$

$$\begin{array}{ll} & R \\ \Leftarrow & \{ \text{ hint explaining why } R \;\Leftarrow\; Q \;\} \\ & Q \\ \equiv & \{ \text{ hint explaining why } Q \;\equiv\; P \;\} \\ & P \end{array}$$

End of Proof

Occasionally we break formulas and hints across more than one line. The indentation of the following line should make it clear that this has taken place. Formulas are labeled with numbers of the form $(m.n)$ where m is the number of the chapter where the formula is introduced and n is a running counter within each chapter. These numbers are used in hints and in the text to refer to the definition of a formula. When a function name is given with such a number in a hint, it refers to the formula that defines the function.

The functional notations we define enjoy the *referential transparency* property [BW88]; that is, variables do not change their values within the context of their definition, and can thus be substituted using a simplified version of the *Rule of Leibnitz*:

$$(\forall x, y, f : x, y \in \mathsf{A} \;\wedge\; f \in (\mathsf{A} \longrightarrow \mathsf{B}) : \;\; x = y \;\Rightarrow\; f.x = f.y \;) \tag{1.2}$$

Above we used the quantified notation from [DS90] where the *dummies* are x, y, f, the *range* is $x, y \in \mathsf{A} \;\wedge\; f \in (\mathsf{A} \longrightarrow \mathsf{B})$ and the *term* is $x = y \;\Rightarrow\; f.x = f.y$. The range restricts the values of the dummies within the quantification; thus, we omit the range if it is obvious from the context. We also use this notation for operators that are associative and commutative. We can write a well-known identity by:

$$(+i : 0 \le i \;\wedge\; i < n : i) \;=\; \frac{n * (n-1)}{2} \tag{1.3}$$

When the operator has an identity element, the range may be empty (i.e., false) and the value of the quantification is the unit element. The quantification in (1.3) has the value 0 when $n = 0$. We use the shorthand $0 \leq i < n$ to specify the range of the natural numbers in (1.3).

We use *lambda abstraction* to define anonymous functions; for example,

$$(\lambda n :: n * n)$$

is the function that returns the square of its argument.

All free variables in formulas are universally quantified over their type, unless restricted in the surrounding text. We use the following conventions for naming variables unless stated otherwise in the context:

$a, b, c, d, e, f, g, h, x, y, z$	Scalars
p, q, r, s, t, u, v, w	PowerLists, ParLists or PLists
i, j, k, m, n	Nat or Pos
l	List
A, B, C, L, M, X, Y, Z	Type

The choice between PowerLists, ParLists or PLists is determined by the enclosing chapter or section.

1.3 Cost Calculus

In this section we develop a cost calculus that enables us to estimate the time used by an algorithm when mapped to a particular architecture. In Chapter 2 we will use this calculus to quantify the running times of PowerList functions mapped onto hypercubes.

Big O notation

In the literature an upper bound of the growth of a function is often given with the "big O" notation, e.g.,

$$\log .n = O(\sqrt{n})$$

which states that the logarithm function grows no faster than the square root function. In general, "big O" is defined by:

$$g.n = O(f.n) \quad \equiv \quad (\exists c, m : c > 0 \;\wedge\; m \geq 0 : (\forall n : n \geq m : g.n \leq c * f.n)) \qquad (1.4)$$

Thus, g has a slower growth than that of f for all arguments beyond a certain point.

While the concept of bounding the growth of a function is important, this notation is problematic due to the use of the equality operator, a symmetric relation. This prevents equational reasoning using the "big O" notation. Most good expositions of the notation make cautionary remarks about its use. Cormen, Leiserson and Rivest acknowledge this potential abuse of notation in their textbook [CLR90]. They define O as a function that maps natural valued functions to sets of natural valued functions, and state complicated rules to interpret equalities containing O.

An alternative to this approach is to view O as a relation on natural valued functions: $\mathcal{O}$: $(\mathsf{Nat} \longrightarrow \mathsf{Real}) \times (\mathsf{Nat} \longrightarrow \mathsf{Real}) \longrightarrow \mathsf{Bool}$

$$g \,\mathcal{O}\, f \quad \equiv \quad (\exists c, m : c > 0 \;\wedge\; m \geq 0 : (\forall n : n \geq m : g.n \leq c * f.n)) \qquad (1.5)$$

As a relation $\mathcal{O}$ is reflexive and transitive, but neither symmetric nor anti-symmetric. Viewed this way $\mathcal{O}$ introduces a *preorder* on the set of functions in $(\mathsf{Nat} \longrightarrow \mathsf{Nat})$

The relation $\mathcal{O}$ provides an upper bound for the "growth" of a function. Its dual Ω provides a lower bound for the growth of a function:

$$g \,\Omega\, f \quad \equiv \quad (\exists c, m : c > 0 \;\wedge\; m \geq 0 : (\forall n : n \geq m : g.n \geq c * f.n)) \qquad (1.6)$$

It is simple to prove

Lemma 1

$$f\,\mathcal{O}\,g \quad\equiv\quad g\,\Omega\,f$$

Proof Lemma

$f\,\mathcal{O}\,g$

$=\;$ { Definition of $\mathcal{O}$ (1.5) }

$(\exists c, m : c > 0 \;\wedge\; m \geq 0 : (\forall n : n \geq m : f.n \leq c * g.n))$

$=\;$ { Arithmetic, $c > 0$ }

$(\exists c, m : c > 0 \;\wedge\; m \geq 0 : (\forall n : n \geq m : 1/c * f.n \leq g.n))$

$=\;$ { Change of dummy $d := c$, arithmetic }

$(\exists d, m : d > 0 \;\wedge\; m \geq 0 : (\forall n : n \geq m : g.n \geq d * f.n))$

$=\;$ { Definition of Ω (1.6) }

$g\,\Omega\,f$

End of Proof

From this it follows that only one of $\mathcal{O}$ and Ω is needed.

The relation Θ gives both an upper and a lower bound on the growth of a function, it is defined by:

$$f\,\Theta\,g \equiv (\exists c, d, m : c > 0 \wedge d > 0 \wedge m \geq 0 : (\forall n : n \geq m : c{*}g.n \leq f.n \wedge f.n \leq d{*}g.n)) \tag{1.7}$$

The relation Θ can be replaced by:

$f\,\Theta\,g$

$\equiv\;$ { Θ (1.7) }

$(\exists c, d, m : c > 0 \wedge d > 0 \wedge m \geq 0 : (\forall n : n \geq m : c * g.n \leq f.n \;\wedge\; f.n \leq d * g.n))$

$\equiv\;$ { predicate calculus, and arithmetic }

$\quad(\exists d, m : d > 0 \;\wedge\; m \geq 0 : (\forall n : n \geq m : f.n \leq d * g.n))$

$\wedge\;(\exists c, m : c > 0 \;\wedge\; m \geq 0 : (\forall n : n \geq m : c * g.n \leq f.n))$

$\equiv\;$ { definitions of $\mathcal{O}$ (1.5) and Ω (1.6) }

$$
\begin{array}{ll}
 & f\,\mathcal{O}\,g \ \wedge\ f\,\Omega\,g \\
\equiv & \{\ \text{Lemma (1)}\ \} \\
 & f\,\mathcal{O}\,g \ \wedge\ g\,\mathcal{O}\,f
\end{array}
$$

These two derivations establish that the relation $\mathcal{O}$ is fundamental for stating complexity bounds.

To facilitate proofs we introduce a modified less-than relation on functions over the naturals $\lesssim : (\mathsf{Nat} \longrightarrow \mathsf{Real}) \times (\mathsf{Nat} \longrightarrow \mathsf{Real}) \longrightarrow \mathsf{Bool}$, denoting that above some threshold the left function yields values below those of the right function:

$$f \lesssim g \quad \equiv \quad (\exists n :: (\forall k : k \geq n : f.k \leq g.k)) \qquad (1.8)$$

The functions **one** : $\mathsf{Nat} \longrightarrow \mathsf{Nat}$ and **id** : $\mathsf{Nat} \longrightarrow \mathsf{Nat}$ are defined by

$$\mathbf{one} \quad = \quad (\lambda n :: 1) \qquad (1.9)$$

$$\mathbf{id} \quad = \quad (\lambda n :: n) \qquad (1.10)$$

They are used to measure the complexity of the functions we define. The following lemma is useful in solving recurrences over Nat

Lemma 2

$$(\exists c : c \in \mathsf{Nat} : (\lambda n :: f.(n+1)) \lesssim (\lambda n :: f.n + c)) \quad \Rightarrow \quad f\ \mathcal{O}\ \mathbf{id} \qquad (1.11)$$

$$(\lambda n :: f.(n+1)) \lesssim f \quad \Rightarrow \quad f\ \mathcal{O}\ \mathbf{one} \qquad (1.12)$$

$$(\exists c : c \in \mathsf{Nat} : f \lesssim (\lambda n :: c)) \quad \Leftarrow \quad f\ \mathcal{O}\ \mathbf{one} \qquad (1.13)$$

1.3.1 Parallel Algorithm Complexity

To motivate the cost calculus we introduce some basic concepts from parallel algorithm complexity theory as presented in the literature (e.g., [KR90, JáJ92]). Assume that we are studying a parallel algorithm that solves a particular problem, parameterizable by the size of the input n. Let $S.n$ denote the time complexity of the best known sequential algorithm that solves a problem of size n, and let $P.n$ be the time complexity of the algorithm that we are studying. According to Brent's Scheduling Principle [Bre74]

$$P.n \geq S.n/p \tag{1.14}$$

where p is the number of processors used by the parallel algorithm. An operational interpretation of Brent's Scheduling Principle is that any parallel algorithm can be simulated on a uniprocessor machine, by having the single processor in turn act as each of the p parallel processing elements. In p steps the single processor simulates one step of the parallel algorithm. This is a simplified simulation, since the intermediate states produced by the simulation may corrupt the sequential simulation. This can be remedied by replicating the values of the variables read by the parallel step before each step is simulated by the uniprocessor.

For a parallel algorithm it is not always the case that all processors are doing something useful in each step. Consider the problem of computing the sum of the elements of a list. This can be done in parallel by placing the elements of the list in the leaves of a binary tree. At each step the processor assigned to a node of the tree performs the addition of the values stored at its children, if these values have been computed. In a balanced binary tree the number of additions performed at the leaves is half the length of the list, whereas only one addition is performed at the root. Due to the data dependency inherent to the problem, the addition at the root cannot be performed before the other additions in the tree have been performed. This means that at the last step of the computation only one processor will be active.

To measure how efficiently a parallel algorithm is utilizing the processors we introduce the notion of the *cost*, $C.p$, of a parallel algorithm using p processors, defined by:

$$C.p.n = P.n * p$$

A parallel algorithm is considered *optimal* if

$$C.p \;\mathcal{O}\; S$$

Next, we define the reflexive and transitive relation $\mathcal{P}$

$$f\,\mathcal{P}\,g \quad\equiv\quad (\exists k : k \geq 0 : f\,\mathcal{O}\,(g * \log^k)) \tag{1.15}$$

that states that f grows no faster than the product of g and some logarithmic polynomial. A parallel algorithm is considered to be *efficient* if

$$C.p \;\mathcal{P}\; S \tag{1.16}$$

For the problem of computing the sum of a list, using a tree structure, we have the following relationships (see the proof below for explanations):

$$\mathbf{id} \;\mathcal{O}\; S \tag{1.17}$$

$$P \;\mathcal{P}\; \mathbf{one} \tag{1.18}$$

$$C.p \;\mathcal{O}\; P * \mathbf{id} \tag{1.19}$$

From the above we can prove that this scheme is efficient:

$$C.p \;\mathcal{P}\; S \tag{1.20}$$

Proof of (1.20)

$C.p$

$\mathcal{O}$ { p is at most input size (1.19) }

$P * \mathbf{id}$

$\mathcal{O}$ { summing is linear (1.17); monotonicity of $*$ under $\mathcal{O}$ (1.21) }

$P * S$

$\mathcal{P}$ { tree has logarithmic height (1.18); monotonicity of $*$ under $\mathcal{P}$ (1.22) }

one $* S$

Θ { property of **one** }

S

End of Proof

In the proof we used the following monotonicity properties of multiplication with respect to $\mathcal{O}$ and $\mathcal{P}$:

$$f \, \mathcal{O} \, h \quad \Rightarrow \quad g * f \;\mathcal{O}\; g * h \tag{1.21}$$

$$f \, \mathcal{P} \, h \quad \Rightarrow \quad g * f \;\mathcal{P}\; g * h \tag{1.22}$$

(1.20) follows from the derivation above and the following "transitivity" properties:

$$f \, \mathcal{O} \, g \;\wedge\; g \, \mathcal{P} \, h \quad \Rightarrow \quad f \, \mathcal{P} \, h \tag{1.23}$$

$$f \, \mathcal{P} \, g \;\wedge\; g \, \Theta \, h \quad \Rightarrow \quad f \, \mathcal{P} \, h \tag{1.24}$$

1.3.2 Parallel Computation Models

The most studied model of a parallel architecture is the Parallel Random Access Machine (PRAM). A PRAM consists of a set of processing elements (processors) each with access to their private memories and to a shared memory. A computational step of a PRAM algorithm consists of the following sequence of operations performed by each processor: read a single data value from either the shared or private memory, perform a single operation, and write a value to either memory. There are different variations on the PRAM model that specify whether more than one processor can read/write to the same memory location in the same step and how conflicts are resolved. In this work we will only study the CREW PRAM, that permits more than one processor to read from the same memory location in a step, but requires that all writes be to separate locations.

The PRAM models are idealized machines in the sense that they ignore the fact that most parallel architectures do not have a shared memory, but instead rely on communication channels between processors with private memory. Such an architecture can be viewed abstractly as a graph where the nodes are the processing elements and the edges are the communication channels. Several properties of this graph are important in characterizing the architecture:

Diameter the maximal distance between two nodes. A measure of how many "hops" a message may have to endure.

Degree the maximal number of edges incident to a node. The lower the degree the easier it is to physically realize the design in hardware.

There are many different proposals for topologies for parallel architectures, among them are (from [McC91]):

Topology	Degree	Diameter
1D array (ring)	2	$p/2$
Shuffle-exchange	3	$2 * \log p$
Cube-connected-cycles	3	$(5/2) * \log p$
2D mesh of trees	3	$2 * \log p$
3D mesh of trees	3	$2 * \log p$
2D array (toroidal)	4	$\sqrt{p}$
Butterfly (wrapped)	4	$2 * \log p$
de Bruin	4	$\log p$
3D array (toroidal)	6	$\sqrt[3]{p}$
Pyramid	9	$\log p$
Hypercube	$\log p$	$\log p$
PRAM	p	1

Note that we can view a PRAM as the complete p-graph. The literature contains many results that show that one architecture can simulate another with a *slowdown* described by a function f; that is, an algorithm that runs in time $P.n$ on the second architecture runs in time $f.n * P.n$ under the simulation on the first architecture. If $f\,\mathcal{O}\,\mathbf{one}$ then we say that the first architecture is at least as powerful as the second; thus, we can describe a partial order that ranks the computational power of the different architectures.

A realistic complexity model for parallel algorithms needs to consider the communication costs that are inherent in most architectures. One such proposal is *LogP* [CKP^{+}93], that models an architecture abstractly with four parameters to specify: the computing bandwidth, the communication bandwidth, the communication delay, and the efficiency of coupling communication and computation. For our purposes such a model is too complex, since we want to avoid working with architectures at this level of detail.

Chapter 2

Powerlists

In this chapter we present the PowerList data structure and its theory [Mis94] illustrated with examples of parallel algorithms expressed as functions over PowerList. We provide a cost calculus that allows us to quantify the time that implementations of the PowerList notation may take on particular parallel architectures and show an efficient mapping of the PowerList operators to hypercubes [Kor94, Kor95]. Finally, we study how different sorting algorithms can be expressed in the PowerList notation, focusing on a derivation of the odd-even transposition sort [Kor97a, Kor97c].

2.1 Introduction

Functional programming languages typically employ *lists* where the basic constructors (adding or removing a single element) allow for sequential processing of the list elements. The PowerList notation [Mis94] uses balanced division of lists in order to allow for parallel processing.

A PowerList is a linear data structure whose elements are all of the same data type. The length of a PowerList is always a power of two. The smallest PowerList has length one; it is called *a singleton* and is written as $\langle a \rangle$ where a is the element of the singleton. PowerLists of equal length with elements from the same data type are called *similar*. Two similar PowerLists, p, q, can be combined into a PowerList of twice their lengths in two different ways:

- $p \mid q$ (pronounced p "tie" q) is the PowerList that consists of the elements of p (in order) followed by the elements of q (also in order);
- $p \bowtie q$ (pronounced p "zip" q) is the PowerList that consists of alternately taking elements from p and q (in order) starting with p.

We introduce the type-function $\mathsf{PowerList} : \mathsf{Type} \times \mathsf{Nat} \longrightarrow \mathsf{Type}$ that takes two arguments, a type (say X) and a natural number (say k) and returns the type of all PowerLists with elements of type X and length equal to 2^k. for example, $\mathsf{PowerList.Nat}.2$ is the type of all PowerLists of length 2^2 containing natural numbers as elements. The statement that p and q are similar is equivalent to saying that p and q both belong to $\mathsf{PowerList.X}.n$ for some X and n. The types of the PowerList constructors are as follows:

$$\langle \_ \rangle \; : \mathsf{X} \longrightarrow \mathsf{PowerList.X}.0 \tag{2.1}$$

$$\_ \mid \_ \; : \mathsf{PowerList.X}.n \times \mathsf{PowerList.X}.n \longrightarrow \mathsf{PowerList.X}.(n+1) \tag{2.2}$$

$$\_ \bowtie \_ \; : \mathsf{PowerList.X}.n \times \mathsf{PowerList.X}.n \longrightarrow \mathsf{PowerList.X}.(n+1) \tag{2.3}$$

The functions $\mathit{length} : \mathsf{PowerList.X}.n \longrightarrow \mathsf{Pos}$ and $\mathit{loglen} : \mathsf{PowerList.X}.n \longrightarrow \mathsf{Nat}$ are defined by

$$(\forall p : p \in \mathsf{PowerList.X}.n : \mathit{length}.p = 2^n) \tag{2.4}$$

$$(\forall p : p \in \mathsf{PowerList.X}.n : \mathit{loglen}.p = n) \tag{2.5}$$

The following axioms define the PowerList algebra:

$$\mathit{loglen}.p > 0 \;\Rightarrow\; (\exists u, v : p = u \mid v) \tag{2.6}$$

$$\mathit{loglen}.p > 0 \;\Rightarrow\; (\exists u, v :: p = u \bowtie v) \tag{2.7}$$

$$\langle a \rangle = \langle b \rangle \quad \equiv \quad a = b \tag{2.8}$$

$$p \mid q = u \mid v \quad \equiv \quad p = u \ \wedge \ q = v \tag{2.9}$$

$$p \bowtie q = u \bowtie v \quad \equiv \quad p = u \ \wedge \ q = v \tag{2.10}$$

$$\langle a \rangle \mid \langle b \rangle \quad = \quad \langle a \rangle \bowtie \langle b \rangle \tag{2.11}$$

$$(p \mid q) \ \bowtie \ (u \mid v) \quad = \quad (p \bowtie u) \ \mid \ (q \bowtie v) \tag{2.12}$$

We often refer to Axiom (2.12) by saying that zip and tie *commute* (Richard Bird [Bir89] calls this property *abide*).

There is no way to directly address a particular element of a PowerList. The only way to access the elements of a PowerList is to break it down using $\bowtie$ and $\mid$ as *deconstructors*, i.e., by using Axioms (2.6) and (2.7). For expository reasons we overload the singleton notation to write concrete PowerLists in examples, e.g. $\langle 0\ 1\ 2\ 3 \rangle$ is the PowerList containing the first four natural numbers. This notation is not part of the theory itself and will not be used in derivations.

Let $\otimes : \mathsf{Y} \times \mathsf{Y} \longrightarrow \mathsf{Y}$ be a binary operator, defined on the scalar type Y. We lift $\otimes$ to operate on $\mathsf{PowerList.Y}$ in an "element-wise" fashion, i.e., $\otimes : \mathsf{PowerList.Y}.n \times \mathsf{PowerList.Y}.n \longrightarrow \mathsf{PowerList.Y}.n$, with the following laws

$$\langle a \rangle \otimes \langle b \rangle \quad = \quad \langle a \otimes b \rangle \tag{2.13}$$

$$(p \mid q) \otimes (u \mid v) \quad = \quad (p \otimes u) \ \mid \ (q \otimes v) \tag{2.14}$$

$$(p \bowtie q) \otimes (u \bowtie v) \quad = \quad (p \otimes u) \ \bowtie \ (q \otimes v) \tag{2.15}$$

Note that only one of (2.14) and (2.15) is needed, as one can be proven by structural induction from the other. Note also the similarity between (2.12) and (2.14) (or (2.15)); we often refer to (2.14) (and (2.15)) by saying that $\mid$ and $\otimes$ (or $\bowtie$ and $\otimes$) commute. As an example we look at addition over natural numbers:

$$\langle 4\ 7\ 9\ 3 \rangle + \langle 2\ 5\ 8\ 4 \rangle = \langle 6\ 12\ 17\ 7 \rangle$$

Relations over scalar types are lifted in a similar fashion, to relations on PowerLists of the same type. Let $\triangle$ be a relation defined on a data type X, i.e.,

$\triangle : \mathsf{X} \times \mathsf{X} \longrightarrow \mathsf{Bool}$, and let $p, q, u, v \in \mathsf{PowerList}.\mathsf{X}.n$ and $x, y \in \mathsf{X}$, we define the lifting of $\triangle$ by:

$$\langle x \rangle \ \triangle \ \langle y \rangle \quad \equiv \quad x \ \triangle \ y \tag{2.16}$$

$$(p \bowtie q) \ \triangle \ (u \bowtie v) \quad \equiv \quad (p \triangle u) \ \wedge \ (q \triangle v) \tag{2.17}$$

i.e., two PowerLists are related by $\triangle$ iff all elements are related pairwise. This extends Axiom (2.10) from the PowerList algebra, which defines $=$ on PowerLists constructed using $\bowtie$ in this way. It is worth noting that the terms $p \neq q$ and $\neg(p = q)$ are not necessarily identical for non-singleton PowerLists p and q. We could have used $|$ in defining $\triangle$ over PowerLists, since a simple consequence of (2.17) is

$$(p \mid q) \ \triangle \ (u \mid v) \quad \equiv \quad (p \triangle u) \ \wedge \ (q \triangle v) \tag{2.18}$$

Functions on PowerLists are defined using pattern matching known from functional programming languages such as ML [MTH90] and Miranda$^{\mathrm{TM}}$ [Tur86]. It follows from the PowerList axioms that a singleton can be deconstructed uniquely, and similarly that a non-singleton PowerList can be deconstructed uniquely using both $\bowtie$ and $|$. We can define the permutation function $rev : \mathsf{PowerList}.\mathsf{X}.n \longrightarrow \mathsf{PowerList}.\mathsf{X}.n$ that returns the PowerList where the order of the elements of the argument PowerList are reversed:

$$rev.\langle a \rangle \quad = \quad \langle a \rangle \tag{2.19}$$

$$rev.(u \mid v) \quad = \quad rev.v \mid rev.u \tag{2.20}$$

rev can also be defined using zip as the deconstructor:

$$rev.(u \bowtie v) = rev.v \bowtie rev.u \tag{2.21}$$

It is a simple exercise to show that *rev* is an *involution*, i.e., its own inverse. As an example of applying *rev* we have:

$$rev.\langle 0\ 1\ 2\ 3 \rangle = \langle 3\ 2\ 1\ 0 \rangle$$

Next, we define the function $sum : \mathsf{PowerList}.\mathsf{Y}.n \longrightarrow \mathsf{Y}$ that computes the sum of the elements of a PowerList. We assume that the addition operator $\oplus : \mathsf{Y} \times \mathsf{Y} \longrightarrow \mathsf{Y}$ is associative:

$$\begin{aligned} sum.\langle a \rangle &= a && (2.22) \\ sum.(p \mid q) &= sum.p \oplus sum.q && (2.23) \end{aligned}$$

The computations of *sum* have the shapes of balanced binary trees. The function *sum* is an example of a *reduction*. In general, we define the reduction function $reduce : (\mathsf{Y} \times \mathsf{Y} \longrightarrow \mathsf{Y}) \times \mathsf{PowerList}.\mathsf{Y}.n \longrightarrow \mathsf{PowerList}.\mathsf{Y}.n$ by:

$$\begin{aligned} reduce.\oplus.\langle a \rangle &= a && (2.24) \\ reduce.\oplus.(p \mid q) &= reduce.\oplus.p \;\oplus\; reduce.\oplus.q && (2.25) \end{aligned}$$

It follows by instantiation that $sum = reduce.\oplus$.

The function *reduce* is an example of a higher order function, i.e., a function that takes a function as an argument. Another example of a higher order PowerList function is: $map : (\mathsf{X} \longrightarrow \mathsf{Z}) \times \mathsf{PowerList}.\mathsf{X}.n \longrightarrow \mathsf{PowerList}.\mathsf{Z}.n$ that takes a function and a PowerList and applies the function to each element of the PowerList. We define *map* by:

$$\begin{aligned} map.f.\langle a \rangle &= \langle f.a \rangle && (2.26) \\ map.f.(p \bowtie q) &= map.f.p \;\bowtie\; map.f.q && (2.27) \end{aligned}$$

as an example we apply the function *abs*, that returns the absolute value of an integer, to the PowerList ⟨17 -3 0 -2⟩:

$$map.abs.\langle 17 \;\; \text{-}3 \;\; 0 \;\; \text{-}2 \rangle = \langle 17 \;\; 3 \;\; 0 \;\; 2 \rangle$$

Note that for scalar functions, like *abs*, we could have lifted its definition to operate on PowerList like we did for scalar binary operators in (2.13), (2.14) and (2.15). Such a notation is simpler than using *map* and will be used in the following where applicable.

2.1.1 Induction Principle for $\mathsf{PowerLists}$

Functions over $\mathsf{PowerLists}$ are defined by structural induction. In proving properties of $\mathsf{PowerLists}$ and $\mathsf{PowerList}$ functions we exploit their structural definition. Let $\Pi : \mathsf{PowerList.X}.n \longrightarrow \mathsf{Bool}$ be a predicate whose truth is to be established for all $\mathsf{PowerList}$ over X. We can establish Π by the following induction principle:

$$\begin{array}{ll} & (\forall x : x \in \mathsf{X} : \Pi.\langle x \rangle) \\ \wedge\ (& (\forall p, q, n : p, q \in \mathsf{PowerList.X}.n \ \wedge\ n \in \mathsf{Nat} : \Pi.p \ \wedge\ \Pi.q \ \Rightarrow\ \Pi.(p \mid q)) \\ & \vee\ (\forall p, q, n : p, q \in \mathsf{PowerList.X}.n \ \wedge\ n \in \mathsf{Nat} : \Pi.p \ \wedge\ \Pi.q \ \Rightarrow\ \Pi.(p \bowtie q)) \ \ ) \\ \Rightarrow & \\ & (\forall p, n : p \in \mathsf{PowerList.X}.n \ \wedge\ n \in \mathsf{Nat} : \Pi.p) \end{array}$$

As an example of a proof by structural induction we prove the following commutative property between *map* and *rev*:

$$rev.(map.f.p) = map.f.(rev.p) \tag{2.28}$$

Proof of (2.28). Base case:

$$\begin{array}{ll} & rev.(map.f.\langle a \rangle) \\ = & \{\ map\ (2.26)\ \} \\ & rev.\langle f.a \rangle \\ = & \{\ rev\ (2.19)\ \} \\ & \langle f.a \rangle \\ = & \{\ map\ (2.26)\ \} \\ & map.f.\langle a \rangle \\ = & \{\ rev\ (2.19)\ \} \\ & map.f.(rev.\langle a \rangle) \end{array}$$

Inductive step:

$$\begin{array}{ll} & rev.(map.f.(p \bowtie q)) \\ = & \{\ map\ (2.27)\ \} \end{array}$$

$rev.(map.f.p \bowtie map.f.q)$

$= \{ \ rev \ (2.21) \ \}$

$rev.(map.f.q) \bowtie rev.(map.f.p)$

$= \{ \ \text{induction} \ (2.28) \ \}$

$map.f.(rev.q) \bowtie map.f.(rev.p)$

$= \{ \ map \ (2.27) \ \}$

$map.f.(rev.q \bowtie rev.p)$

$= \{ \ rev \ (2.21) \ \}$

$map.f.(rev.(p \bowtie q))$

End of Proof

2.1.2 Data Movement and Permutation Functions

In the following we define the functions *rr*, *rl* and *inv* that permute the elements of a PowerList, like *rev* defined above. We also define the operators $\rightarrow$ and $\leftarrow$ that perform data movements on PowerLists that are closely related to the permutation functions *rr* and *rl*. These functions are fundamental building blocks for PowerList functions.

The operators $\rightarrow$ ("right-shift") and $\leftarrow$ ("left-shift") have the type:

$$\rightarrow : \mathsf{X} \times \mathsf{PowerList.X}.n \longrightarrow \mathsf{PowerList.X}.n$$

$$\leftarrow : \mathsf{PowerList.X}.n \times \mathsf{X} \longrightarrow \mathsf{PowerList.X}.n$$

They can be defined as follows [Ada94][1]:

$$x{\rightarrow}\langle a\rangle = \langle x\rangle \tag{2.29}$$

$$x{\rightarrow}(p \bowtie q) = x{\rightarrow}q \bowtie p \tag{2.30}$$

[1]The operators $\rightarrow$ and $\leftarrow$ have a binding power that is greater than that of the other binary operators, with the exception of function application (infix dot). See Section (1.2.2) for the complete table of binding powers.

$$\langle a\rangle \leftarrow x = \langle x\rangle \tag{2.31}$$

$$(p \bowtie q)\leftarrow x = q \bowtie p\leftarrow x \tag{2.32}$$

The operator $\rightarrow$ takes a scalar and a PowerList as arguments, and returns the PowerList obtained by shifting all the elements of the supplied list one position to the right, and inserting the scalar as the leftmost element. Note that the rightmost element of the supplied PowerList is lost by this operation. The dual operator $\leftarrow$ performs a similar operation, except that the PowerList is shifted to the left and the scalar is inserted as the rightmost element. As examples of applying these operators we have:

$$0\rightarrow\langle 1\ 2\ 3\ 4\rangle = \langle 0\ 1\ 2\ 3\rangle \qquad \langle 0\ 1\ 2\ 3\rangle\leftarrow 4 = \langle 1\ 2\ 3\ 4\rangle$$

The functions $\mathit{first} : \mathsf{PowerList}.\mathsf{X}.n \longrightarrow \mathsf{X}$ and $\mathit{last} : \mathsf{PowerList}.\mathsf{X}.n \longrightarrow \mathsf{X}$ return the first and last element of a PowerList, respectively:

$$\mathit{first}.\langle a\rangle = a \tag{2.33}$$

$$\mathit{first}.(p \mid q) = \mathit{first}.p \tag{2.34}$$

$$\mathit{last}.\langle a\rangle = a \tag{2.35}$$

$$\mathit{last}.(p \mid q) = \mathit{last}.q \tag{2.36}$$

We use the following shorthand for *first* and *last* when convenient:

$$\overleftarrow{p} = \mathit{first}.p \ \text{ and } \ \overrightarrow{p} = \mathit{last}.p$$

Using *first* and *last* we can provide an alternative definition of $\rightarrow$ and $\leftarrow$ using $\mid$ as the constructor (a proof of (2.37) can be found in [Ada94]; the proof of (2.38) is dual):

$$a\rightarrow(p \mid q) = a\rightarrow p \mid \overrightarrow{p}\rightarrow q \tag{2.37}$$

$$(p \mid q)\leftarrow a = p\leftarrow\overleftarrow{q} \mid q\leftarrow a \tag{2.38}$$

The function $rr : \mathsf{PowerList}.\mathsf{X}.n \longrightarrow \mathsf{PowerList}.\mathsf{X}.n$ rotates the elements of a PowerList one position to the right, wrapping the rightmost element around; its inverse $rl : \mathsf{PowerList}.\mathsf{X}.n \longrightarrow \mathsf{PowerList}.\mathsf{X}.n$ rotates the elements of a PowerList one position to the left, wrapping the leftmost element around.

$$rr.\langle a \rangle = \langle a \rangle \tag{2.39}$$

$$rr.(p \bowtie q) = rr.q \bowtie p \tag{2.40}$$

$$rl.\langle a \rangle = \langle a \rangle \tag{2.41}$$

$$rl.(p \bowtie q) = q \bowtie rl.p \tag{2.42}$$

As examples we have

$$rr.\langle 0\ 1\ 2\ 3 \rangle = \langle 3\ 0\ 1\ 2 \rangle \quad \text{and} \quad rl.\langle 0\ 1\ 2\ 3 \rangle = \langle 1\ 2\ 3\ 0 \rangle$$

From the definitions of rr and rl it is possible to prove the following identities:

$$rr.(p \mid q) = \vec{q}\rightarrow p \mid \vec{p}\rightarrow q \tag{2.43}$$

$$rl.(p \mid q) = p\leftarrow\overleftarrow{q} \mid q\leftarrow\overleftarrow{p} \tag{2.44}$$

Finally, we define a permutation function $inv : \mathsf{PowerList}.\mathsf{X}.n \longrightarrow \mathsf{PowerList}.\mathsf{X}.n$, that we will use in Chapter 4 to prove isomorphisms between interconnection networks.

$$inv.\langle a \rangle = \langle a \rangle \tag{2.45}$$

$$inv.(p \mid q) = inv.p \bowtie inv.q \tag{2.46}$$

It is simple to show that

$$inv.(p \bowtie q) = inv.p \mid inv.q \tag{2.47}$$

The functions rev, rr, rl and inv are permutation functions, i.e., they rearrange the elements of a PowerList. Permutation functions have inverses and enjoy the property that they distribute over scalar operators, as stated by the following Lemma:

Lemma 3 Let X be a scalar type, $\sim : \mathsf{X} \longrightarrow \mathsf{X}$ be a unary, prefix operator, $\otimes : \mathsf{X} \times \mathsf{X} \longrightarrow \mathsf{X}$ be a binary operator, $\triangle : \mathsf{X} \times \mathsf{X} \longrightarrow \mathsf{Bool}$ be a binary relation, and $f : \mathsf{PowerList}.\mathsf{X}.n \longrightarrow \mathsf{PowerList}.\mathsf{X}.n$ be a permutation function, then:

$$f.(\sim p) \quad = \quad \sim f.p \tag{2.48}$$

$$f.(p \ \otimes \ q) \quad = \quad f.p \ \otimes \ f.q \tag{2.49}$$

$$p \triangle q \quad \equiv \quad f.p \triangle f.q \tag{2.50}$$

This lemma is difficult to prove within the PowerList theory, without introducing explicit indices[2]. Informally, the lemma holds since scalar operators and relations are applied to elements, regardless of their position in the PowerList; it does not matter whether this application takes place before or after the elements are permuted. We omit the proof of the Lemma, since it is not very interesting and involves defining a new notation for index-based reasoning.

2.2 Hypercubes

Like PowerLists, hypercubes only come in sizes that are powers of two. They also share the property that two hypercubes of the same size can be combined into a single hypercube of twice the size. Many commercial supercomputer architectures are based on the hypercube, e.g. NCube's Mediacube series.

An n-dimensional hypercube can be viewed as a graph with 2^n nodes, each uniquely labeled with an n-bit string. Two nodes are connected by an edge if their labels differ in exactly one position, so each node has n neighbors. We note that the diameter (maximum distance between any two nodes) is n.

The hypercube topology is very versatile, and many topologies can be embedded in the hypercube; Leighton [Lei92] shows a number of these embeddings.

[2]For a specific permutation function it is straightforward to establish that the the lemma applies. For instance, we proved (2.48) for *rev* by proving (2.28).

We will consider a mapping of a PowerList function to a hypercube to be *efficient* if each parallel step of a corresponding mapping to a CREW PRAM is equivalent to a constant number of basic operations and communications with neighbors on the hypercube.

Two hypercubes, each of size 2^n, can be combined and labeled in $n+1$ different ways, in an "orderly" fashion, to form a hypercube of size 2^{n+1}, one for each position: connect the nodes from the two cubes with the same label by an edge, and relabel each node to an $n+1$ bit index by shifting the bits from a fixed position one position to the left. The nodes from the first cube all obtain a zero bit in the fixed position, whereas the nodes from the second cube obtain a one bit.

There is a strong connection between PowerLists and hypercubes: if we label each element of a PowerList of length 2^n with a bit string (of length n), representing the position of the element in the PowerList, this element can be mapped to the node with the same label on a hypercube of size 2^n. We refer to this representation as the *standard* encoding. By the construction above, it follows by induction that the zip (tie) of the representation of two PowerLists can be implemented efficiently by combining the representing cubes in the low (high) order bit.

2.3 A Cost Calculus for PowerLists

In this section we will present a cost calculus for the PowerList algebra, building on the general framework that we developed in Chapter 1. This approach is somewhat naive, since we do not provide an operational model of the parallel architectures. This means that we can only state, but not prove the more involved results.

By cost we mean the time it takes to evaluate a function on a particular architecture. We do this by introducing a function for the architectures we study: P for PRAM, H for hypercube. Each of these functions is of the type

$$(\mathsf{PowerList.X} \longrightarrow \mathsf{PowerList.X}) \times \mathsf{Nat} \longrightarrow \mathsf{Real}$$

They return the time it takes to evaluate the given function on an argument PowerList whose *loglen* is equal to the second argument on the architecture. As an example $P.rev.n$ is the time it takes on a PRAM to reverse a PowerList whose *loglen* is n using the function *rev*.

To simplify the calculus we assume that each architecture has enough processors to evaluate the function on the given argument. This is done so we can focus on the idealized time it takes to evaluate the function. In a more realistic scenario where there are not enough processors, each processor acts as a PRAM simulation on the data elements assigned to it. Since the PRAM is the most powerful parallel model we consider this does not invalidate any claim we make.

A hypercube implementation can be simulated with a constant factor slowdown on a PRAM, stated formally by:

$$(\forall f :: P.f \;\; \mathcal{O} \;\; H.f)$$

2.3.1 Basic Functions

First, we present the basic functions that will be used in evaluating the running time of algorithms, along with the complexity we assume that they have on the two architectures:

$$swaptie.(p \mid q) = q \mid p \tag{2.51}$$

$$swapzip.(p \bowtie q) = q \bowtie p \tag{2.52}$$

The two swap functions correspond to very simple data movements. On a PRAM these operations can clearly be performed in a constant number of steps. On a hypercube this is not so obvious. Under the *standard encoding* each element of a PowerList is mapped to the processor with the same binary index as the element. *swaptie* correspond to exchanging values between neighbors in the highest dimension; similarly, *swapzip* corresponds to an exchange in the lowest dimension. Hence it is clear that this operation can be performed in constant time. In summary we have:

$$P.swaptie \; \mathcal{O} \; \mathbf{one} \tag{2.53}$$

$$H.swaptie \; \mathcal{O} \; \mathbf{one} \tag{2.54}$$

$$P.swapzip \; \mathcal{O} \; \mathbf{one} \tag{2.55}$$

$$H.swapzip \; \mathcal{O} \; \mathbf{one} \tag{2.56}$$

As an example we analyze the running time of the function *sum* as defined by (2.23):

$(\lambda n :: H.sum.(n+1))$

$=$ { Definition of *sum* (2.23) }

$(\lambda n :: H.sum.n + H.'+'.n)$

$\lesssim$ { By Lemma 2 (1.13) there exists a c }

$(\lambda n :: H.sum.n + c)$

$\mathcal{O}$ { Solve recurrence, Lemma 2 (1.11) }

$\mathbf{id}$

Since we established that $H.swaptie = P.swaptie$ we can conclude that $P.sum \; \mathcal{O} \; \mathbf{id}$. Since the second arguments of P and H are the logarithmic length of the PowerList, the above results corresponds to a logarithmic execution time.

Let us turn to analyzing the reverse function *rev*, defined by (2.19) and (2.20) in Section 2.1.2: we get:

$(\lambda n :: H.rev.(n+1))$

$=$ { Definition of *rev* }

$(\lambda n :: H.rev.n + H.swaptie.n)$

$\lesssim$ { By Lemma 2 (1.13) there exists a c }

$(\lambda n :: H.rev.n + c)$

$\mathcal{O}$ { Solve recurrence }

$\mathbf{id}$

The above result is "tight" for the standard encoding on the hypercube. An element moves to a processor whose index is the inverted bit string of the index of the processor where it was located. On a hypercube this means that the element has to traverse all dimensions, i.e., $\mathbf{id} \; \mathcal{O} \; H.rev$.

The recursive definition of *rev* corresponds closely to how the function behaves on a hypercube, but on a PRAM all but one of these exchanges are superfluous, since the *rev* operation can be achieved by performing the movement from the source to the the destination in one step. Note that in the case of *sum* the elements of the PowerList are changed (i.e., added together) in each step and the steps cannot be collapsed. The above derivation is still correct, i.e., $P.rev \; \mathcal{O} \; \mathbf{id}$, but the bound for *rev* is not tight. By doing a slightly different analysis for the PRAM, we get:

$$\begin{array}{ll} & (\lambda n :: P.rev.(n+1)) \\ = & \{ \text{ Definition of } rev \text{ and property of PRAM } \} \\ & (\lambda n :: P.rev.n) \\ \mathcal{O} & \{ \text{ Solve recurrence } \} \\ & \mathbf{one} \end{array}$$

A similar situation arises when we study the operator $\rightarrow$ (and dually $\leftarrow$) defined in Section 2.1.2 by (2.29) and (2.30). On a hypercube the operation corresponds to moving an element at node i to node $i+1$ if we interpret the identity of nodes as natural numbers. From the calculus of binary numbers it is well known that the Hamming distance between 2^n and $2^n - 1$ is $n+1$. This means some elements in the PowerList need to traverse all dimensions of the hypercube under the $\rightarrow$ operation. Through a similar derivation as given for *rev* we get

$$H.\!\rightarrow \; \mathcal{O} \; \mathbf{id}$$

As was the case for *rev*, we can prove that

$$P.\!\rightarrow \; \mathcal{O} \; \mathbf{one}$$

2.4 Prefix Sum

The prefix sum algorithm is one of the most fundamental parallel algorithms. It is often used as a building block for other parallel algorithms [Ble89, Ble90, Ble93]. We will see its use in the specification of the *Carry lookahead adder* in Chapter 3. Given a PowerList of scalars and an associative, binary operator $\oplus$ on these scalars, the prefix sum ps returns a PowerList of the same length where each element is the result of applying the operator on the elements up to and including the element in that position in the original PowerList. For example, if $\oplus$ is addition over the integers we have:

$$ps.\langle 3\ 2\ 0\ 5\rangle = \langle 3\ 5\ 5\ 10\rangle$$

More formally, the prefix sum of a PowerList p, where $p \in \mathsf{PowerList.Y}.n$ and the data type Y has the property that $(\mathsf{Y}, +, 0)$ is a monoid, can be defined [Mis94] as the (unique) solution to the equation (in u):

$$u = (0 \rightarrow u) \oplus p \tag{2.57}$$

A proof that (2.57) has a unique solution can be found in [Ada94].

We define the function $ps : \mathsf{PowerList.Y}.n \longrightarrow \mathsf{PowerList.Y}.n$ to realize a well known algorithm for computing the prefix sum due to Ladner and Fischer [LF80]. This algorithm has roots in an algorithm presented by Ofman [Ofm63] and later implemented on a perfect shuffle network by Stone [Sto71]. Misra [Mis94] derived the algorithm for PowerLists; we show a slightly different derivation below:

$u \bowtie v$

$=$ { define $u \bowtie v := ps.(p \bowtie q)$ }

$ps.(p \bowtie q)$

$=$ { defining equation for ps (2.57) }

$0 \rightarrow ps.(p \bowtie q) \oplus p \bowtie q$

$=$ { definition of u, v }

$0 \rightarrow (u \bowtie v) \oplus p \bowtie q$

$= \quad \{ \ \rightarrow (2.30) \ \}$

$0 \rightarrow v \bowtie u \oplus p \bowtie q$

$= \quad \{ \ \bowtie, \oplus \ (2.15) \ \}$

$(0 \rightarrow v \oplus p) \ \bowtie \ (u \oplus q)$

Summarizing:

$u \bowtie v \ = \ 0 \rightarrow v \oplus p \quad \bowtie \quad u \oplus q$

$\equiv \quad \{ \ \text{Axiom (2.10)} \ \}$

$u = 0 \rightarrow v \oplus p \quad \wedge \quad v = u \oplus q$

$\equiv \quad \{ \ \text{solving for } v \ \}$

$u = 0 \rightarrow v \oplus p \quad \wedge \quad v = (0 \rightarrow v \oplus p) \oplus q$

$\equiv \quad \{ \ \oplus \text{ is associative } \ \}$

$u = 0 \rightarrow v \oplus p \quad \wedge \quad v = 0 \rightarrow v \oplus (p \oplus q)$

$\equiv \quad \{ \ \text{defining equation for } ps \text{ (2.57)} \ \}$

$u = 0 \rightarrow v \oplus p \quad \wedge \quad v = ps.(p \oplus q)$

$\equiv \quad \{ \ \text{solving for } u \ \}$

$u = 0 \rightarrow ps.(p \oplus q) \oplus p \quad \wedge \quad v = ps.(p \oplus q)$

$\equiv \quad \{ \ \text{definition of } u, v \text{ and Axiom (2.10)} \ \}$

$ps.(p \bowtie q) \ = \ 0 \rightarrow ps.(p \oplus q) \oplus p \quad \bowtie \quad ps.(p \oplus q)$

Above we have derived the following algorithm for computing the prefix sum:

$$ps.\langle a \rangle \quad = \quad \langle a \rangle \tag{2.58}$$

$$ps.(p \bowtie q) \quad = \quad (0 \rightarrow t \oplus p) \ \bowtie \ t, \quad \text{where } t = ps.(p \oplus q) \tag{2.59}$$

2.4.1 A Hypercube Algorithm for Prefix Sum

Ladner and Fischer's algorithm is not efficient when mapped onto hypercubes using the standard encoding of PowerLists, since the $\rightarrow$ operation cannot be performed efficiently under this encoding. As we discovered in Section 2.3, adjacent elements of the PowerList can be as far apart on the hypercube as its diameter. We have

$$(\lambda n :: n^2) \;\; \mathcal{O} \;\; H.ps \tag{2.60}$$

We will address the general problem of mapping operators like $\rightarrow$ efficiently onto hypercubes in Section 2.5, aiming for a logarithmic execution time.

As noted in Section 2.3 both zip and tie can be performed efficiently on a hypercube under the standard encoding. Thus we can obtain an efficient algorithm by eliminating the $\rightarrow$ operation from (2.59).
We generalize the defining equation for prefix sum (2.57) to the function *cube_ps* in two arguments:

$$cube\_ps : \mathsf{PowerList.Y}.n \times \mathsf{PowerList.Y}.n \longrightarrow \mathsf{PowerList.Y}.n$$

defined by the equation:

$$cube\_ps.p.q = 0{\rightarrow}ps.p \oplus q \tag{2.61}$$

It follows from the defining equation for $ps.r$ (2.57) that:

$$ps.r = cube\_ps.r.r$$

We explore the definition of *cube_ps*:

$$\begin{array}{ll}
 & cube\_ps.(p \bowtie q).(u \bowtie v) \\
= & \{ \text{ defining equation for } cube\_ps \text{ (2.61) } \} \\
 & 0{\rightarrow}ps.(p \bowtie q) \;\; \oplus \;\; (u \bowtie v) \\
= & \{ \text{ Ladner \& Fischer (2.59), where } t = ps.(p \oplus q) \;\; \} \\
 & 0{\rightarrow}((0{\rightarrow}t \oplus p) \bowtie t) \;\; \oplus \;\; (u \bowtie v) \\
= & \{ \text{ definition of } \rightarrow \text{ (2.30) } \} \\
 & 0{\rightarrow}t \bowtie (0{\rightarrow}t \oplus p) \;\; \oplus \;\; (u \bowtie v) \\
= & \{ \text{ commutativity } \oplus, \bowtie \text{ (2.15) } \}
\end{array}$$

$$
\begin{array}{ll}
& 0{\rightarrow}t \oplus u \;\; \bowtie \;\; (0{\rightarrow}t \oplus p) \oplus v \\
= & \{ \text{ associativity of } \oplus \; \} \\
& 0{\rightarrow}t \oplus u \;\; \bowtie \;\; 0{\rightarrow}t \oplus (p \oplus v) \\
= & \{ \;\; t = ps.(p \oplus q) \; \} \\
& 0{\rightarrow}ps.(p \oplus q) \oplus u \;\; \bowtie \;\; 0{\rightarrow}ps.(p \oplus q) \oplus (p \oplus v) \\
= & \{ \text{ definition of } \mathit{cube\_ps} \; \} \\
& \mathit{cube\_ps}.(p \oplus q).u \;\; \bowtie \;\; \mathit{cube\_ps}.(p \oplus q).(p \oplus v)
\end{array}
$$

This gives the following recursive definition of *cube_ps*:

$$
\begin{array}{rcl}
\mathit{cube\_ps}.\langle a \rangle.\langle b \rangle & = & \langle b \rangle \\
\mathit{cube\_ps}.(p \bowtie q).(u \bowtie v) & = & \mathit{cube\_ps}.(p \oplus q).u \;\; \bowtie \;\; \mathit{cube\_ps}.(p \oplus q).(p \oplus v)
\end{array}
$$

By using two variables the $\rightarrow$ operator has disappeared; thus the algorithm can be implemented efficiently on the hypercube. We have

$$
H.\mathit{cube\_ps} \;\; \mathcal{O} \;\; \mathbf{id} \tag{2.62}
$$

through a similar derivation as was performed for *sum* in Section 2.3. The algorithm *cube_ps* is well known in the literature [MP89, JáJ92], and is considered as part of the folklore; its close connection to the algorithm by Ladner and Fischer is interesting.

2.5 Mapping PowerLists Onto Hypercubes

So far we have studied the standard encoding of PowerLists onto hypercubes. We saw that this encoding poses a problem with certain operators on the hypercube, such as the $\rightarrow$ operator and the reverse function *rev*. In this section we introduce the *reflected Gray coding*. We utilize this encoding as a domain transformation like the Fourier Transform, transforming the operands into a domain where operations like $\rightarrow$ can be performed efficiently. We then study how a class of functions using the fundamental operators can be implemented efficiently under this encoding.

The Gray coding was invented by Dr. Frank Gray to lower the data loss when transmitting signals across noisy wires. The coding was patented by his employer, Bell Labs, in 1953 [Gra53]. The reflected Gray coding of a PowerList permutes the elements in such a way that neighboring elements in the original PowerList are placed in positions of the coded PowerList whose indices written as a binary string only differ in one position. We define the permutation function $gray : \mathsf{PowerList.X}.n \longrightarrow \mathsf{PowerList.X}.n$ as a realization of the reflected Gray code

$$gray.\langle a \rangle = \langle a \rangle \tag{2.63}$$

$$gray.(u \mid v) = gray.u \mid gray.(rev.v) \tag{2.64}$$

As an example we have:

$$gray.\langle a\ b\ c\ d\ e\ f\ g\ h \rangle = \langle a\ b\ d\ c\ h\ g\ e\ f \rangle$$

The time complexity of *gray* on a hypercube is:

$$H.gray\ \mathcal{O}\ \mathbf{id} \tag{2.65}$$

Note that this property does not follow directly from (2.64); more properties of the hypercube are necessary to establish it[3]. From this point onward we will not be able to prove the stated complexity results within the PowerList model, they should only be taken as conjectures.

An interesting property of *gray* is

$$gray.((p \bowtie u)\ \bowtie\ (q \bowtie v)) = (gray.p \bowtie v)\ \bowtie\ (gray.q \bowtie u) \tag{2.66}$$

The inverse function of *gray* is *yarg*, defined by:

$$yarg.\langle a \rangle = \langle a \rangle \tag{2.67}$$

$$yarg.(u \mid v) = yarg.u \mid rev.(yarg.v) \tag{2.68}$$

[3]The proof of a more general complexity result, presented in a different formalism, can be found in [JH95].

2.5.1 The Gray Coded Operators

Next we study how operators in the PowerList notation can be implemented in the Gray coded domain. For a binary operator $\dagger$ and unary operator $\natural$, this amounts to defining the Gray coded counterparts: $\dagger^G$ and $\natural^G$ with the properties:

$$\begin{aligned} gray.u \dagger^G gray.v &= gray.(u \dagger v) && (2.69) \\ \natural^G(gray.u) &= gray.(\natural u) && (2.70) \end{aligned}$$

Scalar Operators

The simplest operator to study is a scalar operator $\oplus$. We define $\oplus^G$, the Gray coded version of $\oplus$ by:

$$gray.u \oplus^G gray.v = gray.(u \oplus v) \qquad (2.71)$$

Since *gray* is a permutation function, we have by Lemma 3

$$gray.(u \oplus v) = gray.u \oplus gray.v \qquad (2.72)$$

There is no point in introducing a special $\oplus^G$ operator since $\oplus^G = \oplus$ from (2.71) and (2.72).

The $\bowtie$ operator

In order to implement $\bowtie$ under the Gray coded mapping, we define the operator $\bowtie^G$ satisfying:

$$gray.u \bowtie^G gray.v = gray.(u \bowtie v) \qquad (2.73)$$

By defining a permutation function *cube_even*, that is efficient to implement on a hypercube, with the property

$$gray.(u \bowtie v) = cube\_even.(gray.u \bowtie gray.v) \qquad (2.74)$$

the complexity of $\bowtie^G$ is the same as that of $\bowtie$. Note that it is a simple consequence of (2.73), (2.74) and the existence of *yarg* that:

$$u \bowtie^G v = \mathit{cube\_even}.(u \bowtie v) \tag{2.75}$$

We proceed by exploring what properties are needed of *cube_even* in order to prove the inductive step for (2.73).

$$\begin{aligned}
&\mathit{cube\_even}.(\mathit{gray}.(u \mid v) \bowtie \mathit{gray}.(p \mid q)) \\
=\ &\{ \text{ definition of } \mathit{gray} \text{ (2.74) } \} \\
&\mathit{cube\_even}.((\mathit{gray}.u \mid \mathit{gray}.(\mathit{rev}.v)) \bowtie (\mathit{gray}.p \mid \mathit{gray}.(\mathit{rev}.q))) \\
=\ &\{ \text{ commutativity } \bowtie, \mid \text{ (2.12) } \} \\
&\mathit{cube\_even}.((\mathit{gray}.u \bowtie \mathit{gray}.p) \mid (\mathit{gray}.(\mathit{rev}.v) \bowtie \mathit{gray}.(\mathit{rev}.q))) \\
=\ &\{ \text{ define } \mathit{cube\_even}.(u \mid v) = \mathit{cube\_even}.u \mid \mathit{cube\_odd}.v \text{, see below } \} \\
&\mathit{cube\_even}.(\mathit{gray}.u \bowtie \mathit{gray}.p) \mid \mathit{cube\_odd}.(\mathit{gray}.(\mathit{rev}.v) \bowtie \mathit{gray}.(\mathit{rev}.q)) \\
=\ &\{ \text{ induction, see (2.76) and (2.77) below } \} \\
&\mathit{gray}.(u \bowtie p) \mid \mathit{gray}.(\mathit{rev}.q \bowtie \mathit{rev}.v) \\
=\ &\{ \text{ property of } \mathit{rev}\text{(2.21) } \} \\
&\mathit{gray}.(u \bowtie p) \mid \mathit{gray}.(\mathit{rev}.(v \bowtie q)) \\
=\ &\{ \text{ definition of } \mathit{gray} \text{ (2.64) } \} \\
&\mathit{gray}.((u \bowtie p) \mid (v \bowtie q)) \\
=\ &\{ \text{ commutativity Axiom (2.12) } \bowtie, \mid \ \} \\
&\mathit{gray}.((u \mid v) \bowtie (p \mid q))
\end{aligned}$$

Two equations were left unproven in the above:

$$\mathit{cube\_even}.(u \mid v) \quad = \quad \mathit{cube\_even}.u \mid \mathit{cube\_odd}.v \tag{2.76}$$

$$\mathit{cube\_odd}.(\mathit{gray}.u \bowtie \mathit{gray}.v) \quad = \quad \mathit{gray}.(v \bowtie u) \tag{2.77}$$

We use (2.76) as the definition of *cube_even*, along with the two base cases:

$$\mathit{cube\_even}.\langle a \rangle \quad = \quad \langle a \rangle \tag{2.78}$$

$$\mathit{cube\_even}.(\langle a \rangle \mid \langle b \rangle) \quad = \quad \langle a \rangle \mid \langle b \rangle \tag{2.79}$$

The proof of (2.77) is similar to the inductive proof given for (2.73), yielding the following definition of *cube_odd*:

$$
\begin{aligned}
cube\_odd.\langle a \rangle &= \langle a \rangle & (2.80)\\
cube\_odd.(\langle a \rangle \mid \langle b \rangle) &= \langle b \rangle \mid \langle a \rangle & (2.81)\\
cube\_odd.((u \mid v) \mid (p \mid q)) &= cube\_odd.(u \mid v) \ \mid \ cube\_even.(p \mid q) & (2.82)
\end{aligned}
$$

$cube\_even.(u \bowtie v)$ is the permutation on $u \bowtie v$ that swaps each element of u with index (in u) of odd parity with the element in v with the same index. The two PowerLists are then zipped back together. If the list $u \bowtie v$ is encoded directly on the hypercube, this operation can be performed efficiently by swapping elements among the nodes with this property, i.e.

$$H.cube\_even \ \mathcal{O} \ \textbf{one} \ \wedge \ H.cube\_odd \ \mathcal{O} \ \textbf{one}$$

It is a simple exercise to show that both *cube_even* and *cube_odd* are their own inverses (involutions).

The $\mid$ operator

Next we explore an efficient implementation of $\mid$ under the Gray coding. Just as we introduced $\bowtie^G$ to satisfy commuting property (2.69), we introduce $\mid^G$:

$$gray.p \mid^G gray.q = gray.(p \mid q) \qquad (2.83)$$

and continue by exploring this definition

$$
\begin{aligned}
& p \mid^G q\\
= & \quad \{ \ gray \text{ and } yarg \text{ are inverses } \}\\
& gray.(yarg.p) \mid^G gray.(yarg.q)\\
= & \quad \{ \ \mid^G \ (2.83) \ \}\\
& gray.(yarg.p \mid yarg.q)
\end{aligned}
$$

$= \{ \; \mathit{gray} \; (2.64) \; \}$

$\mathit{gray}.(\mathit{yarg}.p) \mid \mathit{gray}.(\mathit{rev}.(\mathit{yarg}.q))$

$= \{ \; \mathit{gray}$ and yarg are inverses $\}$

$p \mid \mathit{gray}.(\mathit{rev}.(\mathit{yarg}.q))$

We continue by exploring the right hand side

$\mathit{gray}.(\mathit{rev}.(\mathit{yarg}.(u \mid v)))$

$= \{ \; \mathit{yarg} \; (2.68) \; \}$

$\mathit{gray}.(\mathit{rev}.(\mathit{yarg}.u \mid \mathit{rev}.\mathit{yarg}.v))$

$= \{ \; \mathit{rev}$ (2.20) is an involution $\}$

$\mathit{gray}.(\mathit{yarg}.v \mid \mathit{rev}.(\mathit{yarg}.u))$

$= \{ \; \mathit{gray} \; (2.64) \; \}$

$\mathit{gray}.(\mathit{yarg}.v) \mid \mathit{gray}.(\mathit{rev}.(\mathit{rev}.(\mathit{yarg}.u)))$

$= \{ \; \mathit{gray}, \mathit{yarg}$ are inverses; rev is an involution $\}$

$v \mid \mathit{gray}.(\mathit{yarg}.u)$

$= \{ \; \mathit{gray}$ and yarg are inverses $\}$

$u \mid v$

Putting the above together we define the permutation function *flip*:

$$\mathit{flip}.\langle a \rangle = \langle a \rangle \tag{2.84}$$

$$\mathit{flip}.(\langle a \rangle \mid \langle b \rangle) = \langle a \rangle \mid \langle b \rangle \tag{2.85}$$

$$\mathit{flip}.((p \mid q) \mid (u \mid v)) = (p \mid q) \mid (v \mid u) \tag{2.86}$$

with the property

$$\mathit{flip}.(p \mid q) = p \mid^{G} q \tag{2.87}$$

flip has an efficient implementation on a hypercube: nodes with a one in the highest bit of the label exchange their value with their neighbor in the next to highest dimension:

$$H.\mathit{flip} \; \mathcal{O} \; \textbf{one}$$

The $\rightarrow$ operator

The $\rightarrow$ operator is defined in terms of the fundamental operator $\bowtie$. We define the Gray coded equivalent in terms of the Gray coded $\bowtie^G$ operator:

$$a \overset{G}{\rightarrow} \langle b \rangle \quad = \quad \langle a \rangle \tag{2.88}$$

$$a \overset{G}{\rightarrow} (u \bowtie^G v) \quad = \quad a \overset{G}{\rightarrow} v \bowtie^G u \tag{2.89}$$

This operator can be implemented in constant time on the hypercube, since neighboring elements of the PowerList are neighbors on the hypercube under the Gray coded mapping:

$$H.\overset{G}{\rightarrow} \quad \mathcal{O} \ \textbf{one}$$

Note that this property does not follow directly from (2.89), since a proof that utilizes adjacency is needed; such a proof seems to lie outside of the PowerList theory.

The operator $\overset{G}{\rightarrow}$ satisfies the commuting property in (2.70), i.e.

$$a \overset{G}{\rightarrow} gray.u = gray.(a{\rightarrow}u) \tag{2.90}$$

Proof of (2.90), base case omitted. Inductive step:

$a \overset{G}{\rightarrow} gray.(p \bowtie q)$

$=$ { $\bowtie^G$ (2.73) }

$a \overset{G}{\rightarrow} (gray.p \bowtie^G gray.q)$

$=$ { $\overset{G}{\rightarrow}$ (2.89) }

$a \overset{G}{\rightarrow} gray.q \bowtie^G gray.p$

$=$ { induction (2.90) }

$gray.(a{\rightarrow}q) \bowtie^G gray.p$

$=$ { $\bowtie^G$ (2.73) }

$gray.(a{\rightarrow}q \bowtie p)$

$=$ { $\rightarrow$ (2.30) }

$gray.(a{\rightarrow}(p \bowtie q))$

End of Proof

We have shown that under Gray coding the fundamental operators and some derived operators have efficient implementations on the hypercube. From the above it does not follow that all PowerList functions can be implemented as efficiently on a hypercube as on a PRAM. The Gray coding satisfies (2.65) $H.gray\,\mathcal{O}\,\mathbf{id}$, but a PowerList function that takes less time on a CREW PRAM, like the function $\rightarrow$, is not implemented efficiently due to the overhead introduced by the Gray coding. However, as shown below, when $\rightarrow$ is used in Ladner and Fischer's prefix sum algorithm, the Gray coded implementation on a hypercube is efficient.

2.5.2 Ladner and Fischer's Algorithm Revisited

As we observed, properties from the original theory carry over into the Gray coded domain. As an example we revisit the Ladner and Fischer algorithm for prefix sum. Using the Gray coded operators we can define the Gray coded version of Ladner and Fischer's algorithm:

$$psg.\langle a\rangle \;=\; \langle a\rangle \tag{2.91}$$

$$psg.(p \bowtie^G q) \;=\; 0{\rightarrow}r \oplus p \;\bowtie^G\; r \quad \text{where} \;\; r = psg.(p \oplus q) \tag{2.92}$$

psg satisfies the commuting property:

$$psg.(gray.p) = gray.(ps.p) \tag{2.93}$$

Proof Induction, base case is omitted:

$psg.(gray.(p \bowtie q))$

$= \;\; \{ \;\; \bowtie^G \; (2.73) \;\; \}$

$psg.(gray.p \bowtie^G gray.q)$

$= \;\; \{ \;\; psg \; (2.92) \;\; \}$

$0 \xrightarrow{G} psg.(gray.p \oplus gray.q) \oplus gray.p \;\; \bowtie^G \;\; psg.(gray.p \oplus gray.q)$

$= \quad \{ \ \oplus \text{ is scalar } (2.15) \ \}$

$0 \overset{G}{\rightarrow} psg.(gray.(p \oplus q)) \oplus gray.p \quad \bowtie^{G} \quad psg.(gray.(p \oplus q))$

$= \quad \{ \ \text{Induction hypothesis } (2.93) \ \}$

$0 \overset{G}{\rightarrow} gray.(ps.(p \oplus q)) \oplus gray.p \quad \bowtie^{G} \quad gray.(ps.(p \oplus q))$

$= \quad \{ \ \overset{G}{\rightarrow} \ (2.89) \ \}$

$gray.(0 \rightarrow ps.(p \oplus q)) \oplus gray.p \quad \bowtie^{G} \quad gray.(ps.(p \oplus q))$

$= \quad \{ \ \oplus \text{ is scalar, } gray \text{ is a permutation function, Lemma 3 } (2.49) \ \}$

$gray.(0 \rightarrow ps.(p \oplus q) \oplus p) \quad \bowtie^{G} \quad gray.(ps.(p \oplus q))$

$= \quad \{ \ \bowtie^{G} \text{ and } gray \ (2.73) \ \}$

$gray.(0 \rightarrow ps.(p \oplus q) \oplus p \quad \bowtie \quad ps.(p \oplus q))$

$= \quad \{ \ ps \ (2.59) \ \}$

$gray.(ps.(p \bowtie q))$

End of Proof

Since each of the Gray coded operators have efficient implementations, we have obtained an efficient implementation of Ladner and Fischer's algorithm for hypercubic architectures.

2.6 Fast Fourier Transform

In this section we present the Discrete Fast Fourier Transform algorithm as it was derived for PowerList by Misra [Mis94]. The succinctness of the PowerList description illustrates the expressive power of having both $\bowtie$ and $|$ in the PowerList notation.

The *Discrete Fourier Transform* is an important tool used in many scientific applications, especially in digital signal processing. It can be used for time series analysis, convolutions and to solve partial differential equations. The transform maps a sample from a cycle of data points of a periodic signal onto a frequency spectrum representation containing the same number of points.

The *Fast Fourier Transform* is a method to compute the Discrete Fourier Transform made popular by Cooley & Tukey [CT65]. Misra [Mis94] derived this algorithm from its definition. The function $\mathit{fft} : \mathsf{PowerList}.n.\mathsf{Com} \longrightarrow \mathsf{PowerList}.n.\mathsf{Com}$ can be written as:

$$\mathit{fft}.\langle a\rangle = \langle a\rangle \tag{2.94}$$

$$\mathit{fft}.(p \bowtie q) = (r + u * s) \mid (r - u * s) \tag{2.95}$$

$$\begin{aligned} \text{where} \quad r &= \mathit{fft}.p \\ s &= \mathit{fft}.q \\ u &= \mathit{powers}.z.p \\ z &= \mathit{root}.(\mathit{length}.(p \bowtie q)) \end{aligned}$$

where $\mathit{root} : \mathsf{Nat} \longrightarrow \mathsf{Com}$ applied to n returns the nth root of unity:

$$\mathit{root}.n = e^{\frac{2*\pi*\sqrt{-1}}{n}} \tag{2.96}$$

and the function $\mathit{powers} : \mathsf{Com} \times \mathsf{PowerList}.\mathsf{X}.n \longrightarrow \mathsf{PowerList}.\mathsf{Com}.n$ is defined by

$$\mathit{powers}.x.\langle a\rangle = \langle x^0\rangle \tag{2.97}$$

$$\mathit{powers}.x.(p \bowtie q) = \mathit{powers}.x^2.p \bowtie \mathit{map}.[x*].(\mathit{powers}.x^2.q)) \tag{2.98}$$

where $[x*] : \mathsf{Com} \longrightarrow \mathsf{Com}$ is the scalar function that multiplies its argument by x:

$$[x*].y = x * y \tag{2.99}$$

The function $\mathit{powers}.x.p$ returns a $\mathsf{PowerList}$ of the same length as p containing the powers of x from 0 up to the length of p, for example:

$$\mathit{powers}.3.\langle a \quad b \quad c \quad d\rangle = \langle 1 \quad 3 \quad 9 \quad 27\rangle$$

As an example of applying *fft* we have:

$$\mathit{fft}.\langle \sqrt{-1} \quad -\sqrt{-1} \quad 2 \quad 1{+}\sqrt{-1}\,\rangle = \langle\, 3{+}\sqrt{-1} \quad 0 \quad 1{+}\sqrt{-1} \quad -4{+}2{*}\sqrt{-1}\,\rangle$$

2.7 Sorting

This section focuses on the derivation of the odd-even transposition sort as a $\mathsf{PowerList}$ function. It turns out that this algorithm is difficult to express in $\mathsf{PowerList}$ since it does not have a simple recursive description. We start the section by presenting two sorting networks, *batcher* and *bitonic*, due to Batcher [Bat68] that Misra [Mis94] gave elegant $\mathsf{PowerList}$ descriptions of. These descriptions are included to show that the $\mathsf{PowerList}$ can be used effectively and elegantly to specify sorting algorithms.

We study sorting over a totally ordered domain $(\mathsf{M}, \leq)$. For specification purposes we assume that M contains a minimum element $-$ and a maximum element $\top$ and that the symmetric and associative operators $\uparrow$ (for maximum) and $\downarrow$ (for minimum) are defined by:

$$(\forall x, y : x, y \in \mathsf{M} : x \uparrow y = y \quad \equiv \quad x \leq y) \tag{2.100}$$

$$(\forall x, y : x, y \in \mathsf{M} : x \downarrow y = y \quad \equiv \quad y \leq x) \tag{2.101}$$

$$(\forall x : x \in \mathsf{M} : - \leq x \;\wedge\; x \leq \top\,) \tag{2.102}$$

In [Mis94] Misra presented two sorting networks due to Batcher [Bat68], the *Bitonic sort* and the *Batcher merge*. We present these networks below, using a slightly modified syntax. First, we present the Batcher sort

$$\mathit{batcher} : \mathsf{PowerList}.\mathsf{M}.n \longrightarrow \mathsf{PowerList}.\mathsf{M}.n$$

defined in terms of the auxiliary operators $\wr$ and $\updownarrow$:

$$\mathit{batcher}.\langle x \rangle \quad = \quad \langle x \rangle \tag{2.103}$$

$$\mathit{batcher}.(p \bowtie q) \quad = \quad \mathit{batcher}.p \;\wr\; \mathit{batcher}.q \tag{2.104}$$

$$\langle x \rangle \wr \langle y \rangle \quad = \quad \langle x \rangle \updownarrow \langle y \rangle \tag{2.105}$$

$$(p \bowtie q) \wr (u \bowtie v) \quad = \quad (p \wr v) \updownarrow (q \wr u) \tag{2.106}$$

$$p \updownarrow q \quad = \quad (p \downarrow q) \;\bowtie\; (p \uparrow q) \tag{2.107}$$

The Bitonic sort

$$\mathit{bitonic} : \mathsf{PowerList.M}.n \longrightarrow \mathsf{PowerList.M}.n$$

is defined by:

$$\begin{aligned} \mathit{bitonic}.\langle x\rangle &= \langle x\rangle && (2.108)\\ \mathit{bitonic}.(p \bowtie q) &= \mathit{bitonic\_merge}.(\mathit{bitonic}.p \mid \mathit{rev}.(\mathit{bitonic}.q)) && (2.109)\\ \mathit{bitonic\_merge}.\langle x\rangle &= \langle x\rangle && (2.110)\\ \mathit{bitonic\_merge}.(p \bowtie q) &= \mathit{bitonic\_merge}.p \updownarrow \mathit{bitonic\_merge}.q && (2.111) \end{aligned}$$

To prove the correctness of these networks, Misra used the *0-1 principle*, which states that a compare-and-swap sorting algorithm is correct iff it sorts all inputs consisting of 0s and 1s. The 0-1 principle is often attributed to Knuth [Knu73], where Batcher's networks are also presented.

2.7.1 Odd-even Sort in PowerLists

We will study the *odd-even sort* which can be considered a parallel version of *bubble sort*; it is simple to implement and to explain operationally, yet it is inefficient and somewhat tedious to prove correct[4]. The algorithm consists of a sequence of phases, where each phase consists of an "even" step followed by an "odd" step. It is often described operationally as follows [Lei92]:

> "At odd steps, we compare the contents of cells 1 and 2, 3 and 4, etc., exchanging values if necessary so that the smaller value ends up in the leftmost cell. At even steps, we perform the same operation for cells 2 and 3, 4 and 5, etc."

[4]As a parallel sorting technique the odd-even sort is well established in the literature [Sew54, Dem56]. Knuth [Knu73, exercise 5.3.4.37] poses its proof of correctness as an exercise.

In contrast to Batcher's networks, the odd-even sort is iterative in nature and does not have a simple definition in the PowerList notation. Our derivation is somewhat surprising: from a simple characterization of what it means for a PowerList to be sorted, the algorithm emerges through a series of transformations. The remaining proof of correctness consists of proving that after a finite number of phases of the computation, odd-even sort reaches a fixpoint, and that each phase produces a permutation of the input.

In order to derive the algorithm we use the operators $\rightarrow$ and $\leftarrow$, defined by (2.29) – (2.32). They are monotonic in both arguments:

$$x \le y \ \wedge \ p \le q \ \Rightarrow \ x{\rightarrow}p \le y{\rightarrow}q \tag{2.112}$$

$$x \le y \ \wedge \ p \le q \ \Rightarrow \ p{\leftarrow}x \le q{\leftarrow}y \tag{2.113}$$

and they distribute over scalar operators (like $\uparrow$ and $\downarrow$):

$$x{\rightarrow}(p \uparrow q) \quad = \quad x{\rightarrow}p \uparrow x{\rightarrow}q \tag{2.114}$$

$$(p \uparrow q){\leftarrow}x \quad = \quad p{\leftarrow}x \downarrow q{\leftarrow}x \tag{2.115}$$

These properties are simple to prove by structural induction.

A PowerList is ascending when *the value of every element in the* PowerList *is at most the value of its right neighbor*. In the PowerList notation this can be written using the $\rightarrow$ operator:

$$\mathit{ascending}.p \quad \equiv \quad -{\rightarrow}p \le p \tag{2.116}$$

The dual way to express this, using the $\leftarrow$ operator, is

$$\mathit{ascending}.p \quad \equiv \quad p \le p{\leftarrow}\top \tag{2.117}$$

We will use both (2.116) and (2.117) in our derivation of the odd-even sort. They can be generalized into a Galois-connection between (the Curried functions) $-{\rightarrow}$ and ${\leftarrow}\top$

$$-{\rightarrow}q \le p \quad \equiv \quad q \le p{\leftarrow}\top \tag{2.118}$$

legitimizing the use of the word "dual"; (2.118) can be proven by structural induction.

In the rest of this section we assume that the elements of the PowerLists are distinct; this implies that for a PowerList $p \bowtie q$ we have:

$$p \neq q \ \wedge \ p \neq {-\!\!\rightarrow} p \ \wedge \ p \neq p {\leftarrow}\top \ \wedge \ q \neq {-\!\!\rightarrow} q \ \wedge \ q \neq q {\leftarrow}\top$$

This property can be established by extending the order on the elements of a PowerList to a lexical order in the standard way; that is, by using the position of an element in the PowerList as the second component of the lexical order.

We state the following equalities that generalize properties of $\uparrow$ and $\downarrow$ on scalars from M to PowerLists over M. For similar PowerLists u, v, r with the property $u \neq v \ \wedge \ u \neq r$ we have:

$$(u \downarrow v) \uparrow r = u \quad \equiv \quad u \downarrow v = u \ \wedge \ u \uparrow r = u \tag{2.119}$$

$$(u \uparrow v) \downarrow r = u \quad \equiv \quad u \uparrow v = u \ \wedge \ u \downarrow r = u \tag{2.120}$$

$$u \uparrow v \uparrow r = u \quad \equiv \quad u \uparrow v = u \ \wedge \ u \uparrow r = u \tag{2.121}$$

$$u \downarrow v \downarrow r = u \quad \equiv \quad u \downarrow v = u \ \wedge \ u \downarrow r = u \tag{2.122}$$

We only prove (2.119) as the remaining proofs are similar. The only property of $\uparrow$ and $\downarrow$ that is used, stated below, follows from $(\mathsf{M}, \leq)$ being total:

$$(\forall x, y :: (x \uparrow y = x \ \vee \ x \uparrow y = y) \ \wedge \ (x \downarrow y = x \ \vee \ x \downarrow y = y)) \tag{2.123}$$

Proof Base case

$(\langle a \rangle \downarrow \langle b \rangle) \uparrow \langle c \rangle = \langle a \rangle$

$\equiv$ { $c \neq a \ \wedge \ b \neq a$, $(\mathsf{M}, \leq)$ is total (2.123) }

$\langle a \rangle \downarrow \langle b \rangle = \langle a \rangle \ \wedge \ \langle a \rangle \uparrow \langle c \rangle = \langle a \rangle$

Inductive step

$((p \bowtie q) \downarrow (u \bowtie v)) \uparrow (r \bowtie s) = p \bowtie q$

$\equiv$ { commutativity (2.12) }

$(p{\downarrow}u \bowtie q{\downarrow}v){\uparrow}(r \bowtie s) = p \bowtie q$

$\equiv$ { commutativity (2.15) }

$((p{\downarrow}u){\uparrow}r) \bowtie ((q{\downarrow}v){\uparrow}s) = p \bowtie q$

$\equiv$ { unique decomposition (2.10) }

$(p{\downarrow}u){\uparrow}r = p \ \wedge\ (q{\downarrow}v){\uparrow}s = q$

$\equiv$ { induction (2.119) }

$p{\downarrow}u = p \ \wedge\ p{\uparrow}r = p \ \wedge\ q{\downarrow}v = q \ \wedge\ q{\uparrow}s = q$

$\equiv$ { unique decomposition (2.10) }

$p{\downarrow}u \bowtie q{\downarrow}v = p \bowtie q \ \wedge\ p{\uparrow}r \bowtie q{\uparrow}s = p \bowtie q$

$\equiv$ { commutativity (2.15) }

$(p \bowtie q){\downarrow}(u \bowtie v) = p \bowtie q \ \wedge\ (p \bowtie q){\uparrow}(r \bowtie s) = p \bowtie q$

End of Proof

We proceed in the derivation of the odd-even sort by deriving two recursive definitions of *ascending* from (2.116):

$ascending.(p \bowtie q)$

$\equiv$ { *ascending* (2.116) }

$-\!\!\rightarrow(p \bowtie q) \ \leq\ p \bowtie q$

$\equiv$ { $\rightarrow$ (2.30) }

$-\!\!\rightarrow q \bowtie p \ \leq\ p \bowtie q$

$\equiv$ { $\leq$ (2.17) }

$-\!\!\rightarrow q \leq p \ \wedge\ p \leq q$

$\equiv$ { transitivity of $\leq$ }

$-\!\!\rightarrow q \leq p \ \wedge\ p \leq q \ \wedge\ -\!\!\rightarrow q \leq q$

$\equiv$ { monotonicity of $\rightarrow$ }

$-\!\!\rightarrow q \leq p \ \wedge\ p \leq q \ \wedge\ -\!\!\rightarrow q \leq q \ \wedge\ -\!\!\rightarrow p \leq -\!\!\rightarrow q$

$\equiv$ { transitivity of $\leq$ }

$\longrightarrow q \le p \;\wedge\; p \le q \;\wedge\; \longrightarrow q \le q \;\wedge\; \longrightarrow p \le p$

$\equiv \quad \{ \;$ *ascending* (2.116) twice $\}$

$$\longrightarrow q \le p \;\wedge\; p \le q \;\wedge\; ascending.q \;\wedge\; ascending.p \tag{2.124}$$

$\equiv \quad \{ \;$ Galois-connection (2.118) $\}$

$$q \le p \leftarrow \top \;\wedge\; p \le q \;\wedge\; ascending.q \;\wedge\; ascending.p \tag{2.125}$$

We continue by exploring the conjunction of the definitions of *ascending* given by (2.124) and (2.125) above:

$ascending.(p \bowtie q)$

$\equiv \quad \{$ expanding (2.124),(2.125) and the definitions of *ascending* (2.116),(2.117) $\}$

$p \le q \;\wedge\; \longrightarrow p \le p \;\wedge\; \longrightarrow q \le p \;\wedge\; q \le p \leftarrow \top \;\wedge\; q \le q \leftarrow \top$

$\equiv \quad \{ \; \uparrow\downarrow$–calculus (2.101) (2.100) $\}$

$\quad p \downarrow q = p \;\wedge\; \longrightarrow p \uparrow p = p \;\wedge\; \longrightarrow q \uparrow p = p$

$\wedge\; p \uparrow q = q \;\wedge\; q \downarrow p \leftarrow \top = q \;\wedge\; q \downarrow q \leftarrow \top = q$

$\equiv \quad \{ \; \uparrow\downarrow$–calculus; (2.119) $u,v,r := p,q,\longrightarrow p$ (2.120) $u,v,r := q,p,q \leftarrow \top \}$

$\quad p \downarrow q = p \;\wedge\; (p \downarrow q) \uparrow \longrightarrow p = p \;\wedge\; \longrightarrow q \uparrow p = p$

$\wedge\; p \uparrow q = q \;\wedge\; (p \uparrow q) \downarrow q \leftarrow \top = q \;\wedge\; q \downarrow p \leftarrow \top = q$

$\equiv \quad \{ \; \uparrow, \downarrow$ idempotent $\}$

$\quad p \downarrow q = p \;\wedge\; (p \downarrow q) \uparrow (p \downarrow q) \uparrow \longrightarrow p = p \;\wedge\; \longrightarrow q \uparrow p = p$

$\wedge\; p \uparrow q = q \;\wedge\; (p \uparrow q) \downarrow (p \uparrow q) \downarrow q \leftarrow \top = q \;\wedge\; q \downarrow p \leftarrow \top = q$

$\equiv \quad \{ \uparrow, \downarrow$ idempotent, (2.121) $u,v,r := p, p \downarrow q, (p \downarrow q) \uparrow \longrightarrow p$

$\qquad$ (2.122) $u,v,r := q, p \uparrow q, (p \uparrow q) \downarrow q \leftarrow \top \;\}$

$\quad p \uparrow (p \downarrow q) \uparrow \longrightarrow p = p \;\wedge\; \longrightarrow q \uparrow p = p$

$\wedge\; q \downarrow (p \uparrow q) \downarrow q \leftarrow \top = q \;\wedge\; q \downarrow p \leftarrow \top = q$

$\equiv \quad \{ \uparrow\downarrow$–calculus, (2.121) $u,v,r := p, (p \downarrow q) \uparrow \longrightarrow p, \longrightarrow q$

$\qquad$ (2.122) $u,v,r := q, (p \uparrow q) \downarrow q \leftarrow \top, p \leftarrow \top \;\}$

$\longrightarrow p \uparrow \longrightarrow q \uparrow (p \downarrow q) = p \;\wedge\; (p \uparrow q) \downarrow p \leftarrow \top \downarrow q \leftarrow \top = q$

$\equiv \quad \{ \; \rightarrow$ (2.114), $\leftarrow$ (2.115) $\}$

$$
\begin{aligned}
& \longrightarrow (p\uparrow q)\uparrow(p\downarrow q) = p \ \wedge\ (p\uparrow q)\downarrow(p\downarrow q)\leftarrow\top = q \\
\equiv\ & \{\ \text{Axiom (2.10)}\ \} \\
& p \bowtie q \ =\ \longrightarrow(p\uparrow q)\uparrow(p\downarrow q) \ \bowtie\ (p\uparrow q)\downarrow(p\downarrow q)\leftarrow\top
\end{aligned}
$$

From the equality derived above we can conclude that *ascending* characterizes the fixpoints of the function *oddeven*, defined by:

$$oddeven.\langle x\rangle \ =\ \langle x\rangle \tag{2.126}$$

$$oddeven.(p\bowtie q) \ =\ \longrightarrow(p\uparrow q)\uparrow(p\downarrow q) \ \bowtie\ (p\uparrow q)\downarrow(p\downarrow q)\leftarrow\top \tag{2.127}$$

Note that p and q only appear as $p\downarrow q$ and $p\uparrow q$ in (2.127). We can thus split the definition of *oddeven* into its even phase (*even*) and its odd phase (*odd*):

$$even.(p\bowtie q) \ =\ p\downarrow q \ \bowtie\ p\uparrow q \tag{2.128}$$

$$even.\langle x\rangle \ =\ \langle x\rangle \tag{2.129}$$

$$odd.(u\bowtie v) \ =\ \longrightarrow v\uparrow u \ \bowtie\ v\downarrow u\leftarrow\top \tag{2.130}$$

$$odd.\langle x\rangle \ =\ \langle x\rangle \tag{2.131}$$

$$oddeven.p \ =\ odd.(even.p) \tag{2.132}$$

2.7.2 Proving that *oddeven* Terminates

We proceed by showing that a finite number of applications of *oddeven* will converge to a fixpoint, i.e., termination.

$$(\forall p :: (\exists n : n \geq 0 : oddeven^{(n+1)}.p = oddeven^{n}.p)) \tag{2.133}$$

If we can establish termination (2.133), we have by the derivation above for a sufficiently large n:

$$ascending.(oddeven^{n}.p) \tag{2.134}$$

In order to prove termination (2.133) we introduce the *lexical ordering* ($\prec$) over PowerLists:

$$(p \mid q) \ \prec\ (u \mid v) \ \equiv\ (p \prec u) \ \vee\ (p = u \ \wedge\ q \prec v) \tag{2.135}$$

$$\langle x\rangle \prec \langle y\rangle \quad\equiv\quad x < y \tag{2.136}$$

Since *oddeven* returns a permutation of its argument PowerList (this will be proven in Section 2.7.3) it follows that the elements of both PowerLists come from the same, finite subset L of M. By the finiteness of L we have that $(\mathsf{L}, <)$ is well-founded. By construction it follows that $\prec$ is also well-founded on PowerLists whose elements are in L; i.e., a sequence consisting of permutations of a PowerList p, where neighboring elements are related by $\prec$ is finite. The PowerList that is the result of sorting p is as small as any element of such a sequence.

We proceed to prove separately that the result of applying *even* and *odd* to a PowerList either returns the PowerList itself or a PowerList that is lexically "smaller":

Lemma 4

$$even.p = p \quad\vee\quad even.p \prec p \tag{2.137}$$

$$odd.p = p \quad\vee\quad odd.p \prec p \tag{2.138}$$

(2.133) now follows from Lemma 4 and (2.132), by the well-foundedness established above. We proceed by proving (2.137) and (2.138) separately.

The following equality (2.139) is a simple consequence of the definition of *even*; because $\prec$ is defined with the $|$ operator, it is needed in the proof of Lemma 5:

$$length.p \geq 2 \ \wedge\ length.q \geq 2 \ \Rightarrow\ even.(p \mid q) = even.p \mid even.q \tag{2.139}$$

(2.137) follows, by predicate calculus, from the following Lemma.

Lemma 5

$$even.p \ \prec\ p \quad\equiv\quad \neg(even.p = p) \tag{2.140}$$

Proof Base cases: $\langle a\rangle$ by inspection, $\langle a\rangle \mid \langle b\rangle$

$even.(\langle a\rangle \mid \langle b\rangle) \ \prec\ \langle a\rangle \mid \langle b\rangle$

$\equiv \quad \{ \ \text{Axiom (2.11)}, \ \textit{even} \ (2.129), \ \text{Axiom (2.11)} \ \}$

$\langle a{\downarrow}b\rangle \mid \langle a{\uparrow}b\rangle \prec \langle a\rangle \mid \langle b\rangle$

$\equiv \quad \{ \ \prec (2.135) \ \}$

$a{\downarrow}b < a \ \vee \ (a{\downarrow}b = a \ \wedge \ a{\uparrow}b < b)$

$\equiv \quad \{ \ \neg(a{\uparrow}b < b) \ \}$

$b < a$

$\equiv \quad \{ \ \uparrow\downarrow\text{-calculus} \ \}$

$\neg(a{\downarrow}b = a \ \wedge \ a{\uparrow}b = b)$

$\equiv \quad \{ \ \text{Axiom (2.11)}, \ \textit{even} \ (2.129), \ \text{Axiom (2.11)} \ \}$

$\neg(\mathit{even}.(\langle a\rangle \mid \langle b\rangle) = \langle a\rangle \mid \langle b\rangle)$

inductive case ($\mathit{length}.p \geq 2$):

$\mathit{even}.(p \mid q) \ \prec \ p \mid q$

$\equiv \quad \{ \ \textit{even} \text{ and } \mid (2.139) \ \}$

$\mathit{even}.p \mid \mathit{even}.q \ \prec \ p \mid q$

$\equiv \quad \{ \ \prec (2.135) \ \}$

$\mathit{even}.p \prec p \ \vee \ (\mathit{even}.p = p \ \wedge \ \mathit{even}.q \prec q)$

$\equiv \quad \{ \ \text{induction (2.140)} \ \}$

$\neg(\mathit{even}.p = p) \ \vee \ (\mathit{even}.p = p \ \wedge \ \neg(\mathit{even}.q = q))$

$\equiv \quad \{ \ \text{predicate calculus} \ \}$

$\neg(\mathit{even}.p = p \ \wedge \ \mathit{even}.q = q)$

$\equiv \quad \{ \ \text{Axiom (2.9)} \ \}$

$\neg(\mathit{even}.p \mid \mathit{even}.q \ = \ p \mid q)$

$\equiv \quad \{ \ \textit{even} \text{ and } \mid (2.139) \ \}$

$\neg(\mathit{even}.(p \mid q) \ = \ p \mid q)$

End of Proof

In order to prove (2.138), we need a little more machinery. First we generalize the definition of *odd* (2.131)

$$genodd.x.(p \bowtie q).y \;=\; x{\rightarrow}q{\uparrow}p \;\bowtie\; q{\downarrow}p{\leftarrow}y \tag{2.141}$$

$$odd.(p \bowtie q) \;=\; genodd.-.(p \bowtie q).\top \tag{2.142}$$

Note that the function *genodd* is only defined for PowerLists of length at least 2.

The following equality (a consequence of the definitions of $\rightarrow$ and $\leftarrow$) provides an alternative to (2.141) with $|$ as the constructor:

$$length.p \geq 2 \;\wedge\; length.q \geq 2 \;\Rightarrow\; genodd.x.(p \mid q).y = genodd.x.p.\overleftarrow{q} \mid genodd.\overrightarrow{p}.q.y \tag{2.143}$$

(2.138) follows by the instantiation $x, y := -, \top$ in lemma 6 (2.144), below.

Lemma 6

$$genodd.x.(p \mid q).y \;\prec\; p \mid q \;\;\equiv\;\; x \leq \overleftarrow{p} \;\wedge\; \neg(genodd.x.(p \mid q).y = (p \mid q)) \tag{2.144}$$

Proof Base case omitted. Inductive case, $length.p \geq 2 \;\wedge\; length.q \geq 2$:

$$\begin{aligned}
& genodd.x.(p \mid q).y \;\prec\; p \mid q \\
\equiv\;& \{\; genodd \text{ and } \mid \text{ (2.143) } \} \\
& genodd.x.p.\overleftarrow{q} \;\mid\; genodd.\overrightarrow{p}.q.y \;\prec\; p \mid q \\
\equiv\;& \{\; \prec \text{ (2.135) } \} \\
& genodd.x.p.\overleftarrow{q} \prec p \;\vee\; (genodd.x.p.\overleftarrow{q} = p \;\wedge\; genodd.\overrightarrow{p}.q.y \prec q) \\
\equiv\;& \{\; \text{induction (2.144) } \} \\
& \quad (x \leq \overleftarrow{p} \;\wedge\; \neg(genodd.x.p.\overleftarrow{q} = p)) \\
& \;\vee\; (genodd.x.p.\overleftarrow{q} = p \;\wedge\; \overrightarrow{p} \leq \overleftarrow{q} \;\wedge\; \neg(genodd.\overrightarrow{p}.q.y = q)) \\
\equiv\;& \{\; \text{see (2.145) below, } genodd.x.p.\overleftarrow{q} = p \;\Rightarrow\; \overrightarrow{p} \leq \overleftarrow{q} \;\} \\
& \quad (x \leq \overleftarrow{p} \;\wedge\; \neg(genodd.x.p.\overleftarrow{q} = p)) \\
& \;\vee\; (genodd.x.p.\overleftarrow{q} = p \;\wedge\; \neg(genodd.\overrightarrow{p}.q.y = q)) \\
\equiv\;& \{\; \text{see (2.145) below, } genodd.x.p.\overleftarrow{q} = p \;\Rightarrow\; x \leq \overleftarrow{p} \;\} \\
& \quad (x \leq \overleftarrow{p} \;\wedge\; \neg(genodd.x.p.\overleftarrow{q} = p))
\end{aligned}$$

$$\lor \ (x \le \overleftarrow{p} \ \land \ genodd.x.p.\overleftarrow{q} = p \ \land \ \neg(genodd.\overrightarrow{p}.q.y = q))$$

$$\equiv \quad \{ \ \text{abbreviate } u, v := genodd.x.p.\overleftarrow{q}, \ genodd.\overrightarrow{p}.q.y \ \}$$

$$(x \le \overleftarrow{p} \ \land \ \neg(u = p)) \ \lor \ (x \le \overleftarrow{p} \ \land \ u = p \ \land \ \neg(v = q))$$

$$\equiv \quad \{ \ \text{predicate calculus} \ \}$$

$$x \le \overleftarrow{p} \ \land \ (\neg(u = p) \ \lor \ (u = p \ \land \ \neg(v = q)))$$

$$\equiv \quad \{ \ \text{predicate calculus} \ \}$$

$$x \le \overleftarrow{p} \ \land \ \neg(u = p \ \land \ v = q)$$

$$\equiv \quad \{ \ \text{Axiom (2.9)} \ \}$$

$$x \le \overleftarrow{p} \ \land \ \neg(u \mid v = p \mid q)$$

$$\equiv \quad \{ \ \text{abbreviations } u, v := genodd.x.p.\overleftarrow{q}, \ genodd.\overrightarrow{p}.q.y \ \}$$

$$x \le \overleftarrow{p} \ \land \ \neg(genodd.x.p.\overleftarrow{q} \mid genodd.\overrightarrow{p}.q.y = p \mid q)$$

$$\equiv \quad \{ \ genodd \text{ and } \mid \text{ (2.143)} \ \}$$

$$x \le \overleftarrow{p} \ \land \ \neg(genodd.x.(p \mid q).y = p \mid q)$$

End of Proof

The proof of Lemma 6 left us with the proof obligation:

$$genodd.a.r.b = r \ \Rightarrow \ (a \le \overleftarrow{r} \ \land \ \overrightarrow{r} \le b) \tag{2.145}$$

this is a simple consequence of the definition of *genodd*.

2.7.3 Proving that *oddeven* Permutes its Inputs

In the section above we proved that *oddeven* reaches a fixpoint and that this fixpoint is indeed a sorted PowerList. The only remaining obligation is to prove that *oddeven* permutes[5] its input. The key observation is that *oddeven* only exchanges neighboring elements, and only if they are out of order. Our approach is to divide PowerLists into appropriate sets of neighboring pairs and prove that the functions *even* and *odd*

[5]Note that saying that a function permutes its inputs does not imply that the function is a permutation function.

either act as the identity on a pair or swap the elements of the pair. To this end we define a permutation relation $\sim$ as follows, for PowerLists p, q, u and v of length at least 2:

$$\langle a\rangle \mid \langle b\rangle \;\sim\; \langle c\rangle \mid \langle d\rangle \quad\equiv\quad (a = c \;\wedge\; b = d) \;\vee\; (a = d \;\wedge\; b = c) \tag{2.146}$$

$$(p \mid q) \;\sim\; (u \mid v) \quad\equiv\quad p \sim u \;\wedge\; q \sim v \tag{2.147}$$

When two pairs are related by $\sim$ they are permutations of each other. Note that the above definition does not relate singleton PowerLists, as it is a trivial exercise to show that *oddeven* acts as a permutation function on singletons.

Lemma 7

$$even.(p \mid q) \;\sim\; (p \mid q) \tag{2.148}$$

Proof Base case:

$even.(\langle a\rangle \mid \langle b\rangle) \;\sim\; \langle a\rangle \mid \langle b\rangle$

$\equiv$ { *even*, (2.129) }

$\langle a{\downarrow}b\rangle \mid \langle a{\uparrow}b\rangle \;\sim\; \langle a\rangle \mid \langle b\rangle$

$\equiv$ { $\sim$ (2.146) }

$(a = a{\downarrow}b \;\wedge\; b = a{\uparrow}b) \;\vee\; (a = a{\uparrow}b \;\vee\; b = a{\downarrow}b)$

$\equiv$ { $\uparrow\downarrow$ calculus }

true

inductive step for $length.p \geq 2$

$even.(p \mid q) \;\sim\; p \mid q$

$\equiv$ { *even*, (2.139) }

$even.p \mid even.q \;\sim\; p \mid q$

$\equiv$ { $\sim$ (2.147) }

$even.p \sim p \;\wedge\; even.q \sim q$

$\equiv$ { induction }

true

End of Proof

From (2.148) it follows that *even* permutes its inputs.

Proving the same result for *odd* is more complicated. Using the same neighboring pairs as $\sim$ does will not do the job, since *odd* may move elements between such pairs. However, by applying the function *rr* (defined by (2.39) and (2.40)) to the PowerList, the result is then related to the PowerList by $\sim$.

Lemma 8

$$rr.(odd.(p \mid q)) \;\sim\; rr.(p \mid q) \tag{2.149}$$

In order to prove Lemma 8 we start by exploring when $\sim$ relates the right shifted argument and the right shifted result of *genodd*.

$$
\begin{array}{ll}
 & z{\to}genodd.x.(\langle a\rangle \bowtie \langle b\rangle).y \;\sim\; w{\to}(\langle a\rangle \bowtie \langle b\rangle) \\
\equiv & \{\;\; genodd\ (2.141)\ \} \\
 & z{\to}(\langle x{\uparrow}\, a\rangle \bowtie \langle b{\downarrow}\, y\rangle) \;\sim\; w{\to}(\langle a\rangle \bowtie \langle b\rangle) \\
\equiv & \{\;\; \to (2.30)\ \} \\
 & \langle z\rangle \bowtie \langle x \uparrow a\rangle \;\sim\; \langle w\rangle \bowtie \langle a\rangle \\
\equiv & \{\;\; \text{Axiom } (2.11)\ \} \\
 & \langle z\rangle \mid \langle x \uparrow a\rangle \;\sim\; \langle w\rangle \mid \langle a\rangle \\
\equiv & \{\;\; \sim (2.146)\ \} \\
 & (z = w \;\wedge\; x{\uparrow} a = a) \;\vee\; (z = a \;\vee\; x{\uparrow} a = w)
\end{array}
$$

We take the above derivation as the proof of the base case of the following Lemma:

Lemma 9

$$z{\to}genodd.x.(p \mid q).y \;\sim\; w{\to}(p \mid q) \;\;\equiv\;\; (z = w \wedge x{\uparrow}\overleftarrow{p} = \overleftarrow{p}) \vee (z = \overleftarrow{p} \wedge x{\uparrow}\overleftarrow{p} = w) \tag{2.150}$$

Proof base case proven above, assume $\mathit{length}.p \geq 2 \ \wedge \ \mathit{length}.q \geq 2$

$$
\begin{aligned}
& z{\rightarrow}\mathit{genodd}.x.(p \mid q).y \ \sim \ w{\rightarrow}(p \mid q) \\
\equiv & \quad \{ \ \mathit{genodd} \ (2.141) \ \} \\
& z{\rightarrow}(\mathit{genodd}.x.p.\overleftarrow{q} \mid \mathit{genodd}.\vec{p}.q.y) \ \sim \ w{\rightarrow}(p \mid q) \\
\equiv & \quad \{ \ \rightarrow, \ \ \mathit{last}.(\mathit{genodd}.x.p.\overleftarrow{q}) = \vec{p}\downarrow\overleftarrow{q} \ \} \\
& z{\rightarrow}\mathit{genodd}.x.p.\overleftarrow{q} \ \mid \ (\vec{p}\downarrow\overleftarrow{q}){\rightarrow}\mathit{genodd}.\vec{p}.q.y \ \sim \ w{\rightarrow}p \ \mid \ \vec{p}{\rightarrow}q \\
\equiv & \quad \{ \ \sim (2.147) \ \} \\
& z{\rightarrow}\mathit{genodd}.x.p.\overleftarrow{q} \sim w{\rightarrow}p \ \wedge \ (\vec{p}\downarrow\overleftarrow{q}){\rightarrow}\mathit{genodd}.\vec{p}.q.y \sim \vec{p}{\rightarrow}q \\
\equiv & \quad \{ \ \text{induction } (2.150) \ \} \\
& \qquad ((z = w \ \wedge \ x\uparrow\overleftarrow{p} = \overleftarrow{p}) \ \vee \ (z = \overleftarrow{p} \ \wedge \ x\uparrow\overleftarrow{p} = w)) \\
& \quad \wedge \ ((\vec{p}\downarrow\overleftarrow{q} = \vec{p} \ \wedge \ \vec{p}\uparrow\overleftarrow{q} = \overleftarrow{q}) \ \vee \ (\vec{p}\downarrow\overleftarrow{q} = \overleftarrow{q} \ \wedge \ \vec{p}\uparrow\overleftarrow{q} = \vec{p})) \\
\equiv & \quad \{ \ \uparrow\downarrow\text{–calculus} \ \} \\
& (z = w \ \wedge \ x\uparrow\overleftarrow{p} = \overleftarrow{p}) \ \vee \ (z = \overleftarrow{p} \ \wedge \ x\uparrow\overleftarrow{p} = w)
\end{aligned}
$$

End of Proof

Lemma 8 follows from Lemma 9 with the instantiation $x, y, z, w := -, \top, \vec{q}, \vec{q}$ and the definitions of the functions *genodd*, $\rightarrow$ and *rr*. This concludes our presentation of the odd-even sort in PowerLists.

2.8 Summary

The PowerList is a versatile data structure that can be used to describe a range of different algorithms, including the Fast Fourier Transform, Ladner and Fischer's prefix sum algorithm, and Batcher's sorting networks[6]. The PowerList theory is simple, it can be described on a single page. Properties of PowerLists can be proven using a simple induction principle that closely mimics how PowerList functions are defined. We derived Ladner and Fischer's algorithm and an efficient hypercube

[6]We take no credit for the PowerList description of these algorithms, they were originally presented in [Mis94].

algorithm for prefix sum from their specifications using equational reasoning over PowerList.

We established the close connection between the PowerList notation and hypercubic architectures. Using the Gray coded mapping, we obtained efficient implementations of PowerList functions on hypercubic architectures. The Gray coded operators were obtained by formal derivations from their counterparts under the standard encoding.

The derivation of the odd-even sorting algorithm was surprisingly elegant. From a simple characterization of the goal of the algorithm, the algorithm was derived using properties of PowerLists and total orders. It is encouraging that the derivation did not use any operational considerations, instead it was a case of letting the "symbols" do the work. No special consideration was given to the fact that the odd-even sort is a parallel algorithm; this is one of the strengths of the PowerList notation.

Chapter 3

Parlists

As we saw in Chapter 2 it is possible to specify algorithms such as the Discrete Fast Fourier Transform and Batcher's sorting networks elegantly in the PowerList notation, without resorting to "index gymnastics". Restricting the lengths of the inputs to powers of two is reasonable for these algorithms, as they are most often presented this way in the literature. However, for most algorithms the restriction is unnatural.

In this chapter we present an extension of the PowerList notation to lists of arbitrary positive lengths and work through a number of examples. This new data structure is called "ParList", which is short for *parallel list*. Functions over ParLists are defined using structural induction over the data structure, by a base case for singleton ParLists and two inductive cases: one for even length and one for odd length ParLists.

An earlier version of this work was presented in [Kor97b] and [Kor97c] based on ideas from my advisor, Jayadev Misra [Mis96]. This presentation simplifies the axioms presented in the earlier versions and presents results that were not provable in the earlier theory.

3.1 ParList Theory

A ParList is a non-empty list whose elements are all of the same type, either scalars from the same base type or (recursively) ParLists that enjoy the same property. Two ParLists are *similar* if they have the same length and their elements are similar; two scalars are similar when they are from the same base type. We categorize ParLists according to their length. The shortest ParList has length 1; it is called a *singleton*. We denote the singleton containing the scalar x by $\langle x \rangle$.

A non-singleton ParList v can be *deconstructed* into a single element and a ParList whose length is one less than that of v, using the $\triangleright$ ("cons") and the $\triangleleft$ ("snoc") operator:

$$v = a \triangleright p \;\; \wedge \;\; v = q \triangleleft b \tag{3.1}$$

where a, b and the elements of p and q are similar to the elements of v, and p and q are similar ParLists. In (3.1) a is the first element of v and b is the last element of v. This definition corresponds to linear list theory, which is well-known from sequential, functional languages such as Miranda^{TM} [Tur86], ML [MTH90] and Haskell [HJW$^+$92], and from the Bird-Meertens theory of lists [Bir89, BW88, Ski94].

A ParList, p, of even length has the property that it can be deconstructed using the $\bowtie$ ("zip") and the $|$ ("tie") operator:

$$p = u \bowtie v \;\; \wedge \;\; p = r \mid s \tag{3.2}$$

where u, v, r and s are similar ParLists with the properties:

u contains the elements at the even positions[1] of p,

v contains the elements at the odd positions of p,

r is the first half of p, and

[1]Counting starts at zero in this dissertation.

s is the second half of p.

Note the similarity to how the operators $\bowtie$ and $|$ were defined for PowerLists in Chapter 2.

We formalize the involved types and lengths by introducing the type function ParList that takes two arguments, a type and a positive integer, and returns the type of all ParLists with elements of the given type and length equal to the given length.

$$\mathsf{ParList} : \mathsf{Type} \times \mathsf{Pos} \longrightarrow \mathsf{Type}$$

Using ParList we can give the signature for the ParList operators (X is a type and n is in Pos, the positive natural numbers)

$$\begin{aligned}
\langle\ \_\ \rangle &: \mathsf{X} \longrightarrow \mathsf{ParList.X.1} \\
\_ \triangleright \_ &: \mathsf{X} \times \mathsf{ParList.X.}n \longrightarrow \mathsf{ParList.X.}(n+1) \\
\_ \triangleleft \_ &: \mathsf{ParList.X.}n \times \mathsf{X} \longrightarrow \mathsf{ParList.X.}(n+1) \\
\_ \,|\, \_ &: \mathsf{ParList.X.}n \times \mathsf{ParList.X.}n \longrightarrow \mathsf{ParList.X.}(2*n) \\
\_ \bowtie \_ &: \mathsf{ParList.X.}n \times \mathsf{ParList.X.}n \longrightarrow \mathsf{ParList.X.}(2*n)
\end{aligned}$$

We overload the name ParList, by having it denote the type of all ParLists (corresponding to ParList.X.n for all X and n) and naming the algebra we define below. We further refine the type ParList, by introducing the subtype ParList.X that corresponds to all ParLists whose elements are taken from X. Finally, we partition the type ParList.X into the subtypes:

$$\begin{aligned}
\mathsf{Singleton.X} &= \mathsf{ParList.X.1} \\
\mathsf{EvenParList.X} &= (\cup k : k \in \mathsf{Pos} : \mathsf{ParList.X.}(2*k)) \\
\mathsf{OddParList.X} &= (\cup k : k \in \mathsf{Pos} : \mathsf{ParList.X.}(2*k+1))
\end{aligned}$$

Note that PowerLists is a subtype of ParList, corresponding to the lists whose length is a power of two (ParList.X.(2^n) for $n \in$ Nat).

The length function *length* : $\mathsf{ParList.X}.n \longrightarrow \mathsf{Pos}$ is defined by

$$(\forall p : p \in \mathsf{ParList.X}.n : \mathit{length}.p = n) \tag{3.3}$$

Remark The constructors for ParList are defined similarly to how one might define the function *power* : $\mathsf{Real} \times \mathsf{Pos} \longrightarrow \mathsf{Real}$, that computes the value of its first argument raised to the power of its second argument, i.e., $\mathit{power}.x.n = x^n$. We can define *power* recursively as follows:

$$\mathit{power}.x.1 = x \tag{3.4}$$

$$\mathit{power}.x.(2{*}n+1) = x{*}\mathit{power}.x.(2{*}n) \tag{3.5}$$

$$\mathit{power}.x.(2{*}n) = (\mathit{power}.x.n)^2 \tag{3.6}$$

The choices for inductive cases were rather arbitrary, as we could equally well have chosen:

$$\mathit{power}.x.(2{*}n+1) = \mathit{power}.x.(2{*}n) * x \tag{3.7}$$

$$\mathit{power}.x.(2{*}n) = \mathit{power}.x^2.n \tag{3.8}$$

Note how (3.2) corresponds to (3.6) and (3.8), and (3.1) corresponds to (3.5) and (3.7). **End Remark**

3.1.1 Axioms

In the following, we extend the axioms of the PowerList theory [Mis94] to an axiomatization of the ParList algebra. The five constructors for the ParList algebra: $\langle\ \rangle, |, \bowtie, \triangleright$ and $\triangleleft$ are all *isomorphisms* on their respective domains, with the following laws as consequence, where $p, q, u, v \in \mathsf{ParList.X}.n \ \wedge \ a, b, c \in \mathsf{X} \ \wedge \ n \in \mathsf{Pos}$:

$$\langle a\rangle = \langle b\rangle \equiv a = b \tag{3.9}$$

$$p \mid q = u \mid v \equiv p = u \ \wedge \ q = v \tag{3.10}$$

$$p \bowtie q = u \bowtie v \quad \equiv \quad p = u \;\wedge\; q = v \tag{3.11}$$

$$a \triangleright p = b \triangleright q \quad \equiv \quad a = b \;\wedge\; p = q \tag{3.12}$$

$$p \triangleleft a = q \triangleleft b \quad \equiv \quad a = b \;\wedge\; p = q \tag{3.13}$$

$$t \in \mathsf{ParList.X.1} \quad \Rightarrow \quad (\exists\, a :: t = \langle a \rangle\,) \tag{3.14}$$

$$t \in \mathsf{ParList.X.}(2{*}n) \quad \Rightarrow \quad (\exists\, u, v :: t = u \mid v\,) \tag{3.15}$$

$$t \in \mathsf{ParList.X.}(2{*}n) \quad \Rightarrow \quad (\exists\, u, v :: t = u \bowtie v\,) \tag{3.16}$$

$$t \in \mathsf{ParList.X.}(n+1) \quad \Rightarrow \quad (\exists\, a, p :: t = a \triangleright p\,) \tag{3.17}$$

$$t \in \mathsf{ParList.X.}(n+1) \quad \Rightarrow \quad (\exists\, b, q :: t = q \triangleleft b\,) \tag{3.18}$$

The following axioms are retained from the PowerList theory:

$$\langle a \rangle \bowtie \langle b \rangle \;=\; \langle a \rangle \mid \langle b \rangle \tag{3.19}$$

$$(p \mid q) \;\bowtie\; (u \mid v) \;=\; (p \bowtie u) \;\mid\; (q \bowtie v) \tag{3.20}$$

The remaining axioms extend the PowerList algebra to define the full ParList algebra. Note that is not necessary to parenthesize axioms (3.24) and (3.25) since axiom (3.23) allows two equally valid bracketings.

$$a \triangleright \langle b \rangle \;=\; \langle a \rangle \mid \langle b \rangle \tag{3.21}$$

$$\langle a \rangle \triangleleft b \;=\; \langle a \rangle \mid \langle b \rangle \tag{3.22}$$

$$a \triangleright (p \triangleleft b) \;=\; (a \triangleright p) \triangleleft b \tag{3.23}$$

$$a \triangleright p \;\mid\; q \triangleleft b \;=\; a \triangleright (p \mid q) \triangleleft b \tag{3.24}$$

$$a \triangleright p \;\bowtie\; q \triangleleft b \;=\; a \triangleright (q \bowtie p) \triangleleft b \tag{3.25}$$

$$a \;\triangleright (b \triangleright (p \bowtie q)) \;=\; a \triangleright p \;\bowtie\; b \triangleright q \tag{3.26}$$

Note the symmetry between $\bowtie$ and $\mid$ in Axiom (3.20). The roles of $\bowtie$ and $\mid$ can be interchanged in the PowerList algebra, if we do not provide an operational model for it. This is not the case when we consider the ParList algebra. If we interpret $\triangleright$ and

$\triangleleft$ as prepending and appending an element to a ParList then the contrast between (3.24) and (3.25) captures the operational difference between $\bowtie$ and $|$. Note the symmetry between the constructors used in the axioms; the only "asymmetric" axiom is (3.26), which does not have a $|$ counterpart.

From the ParList axioms we can prove the following laws that are useful in manipulating ParList expressions[2]

Lemma 10

$$a \triangleright (p \mid q) \;\bowtie\; (u \mid v) \triangleleft b \;=\; a \triangleright (u \bowtie p) \;\mid\; (v \bowtie q) \triangleleft b \tag{3.27}$$

$$((p \bowtie q) \triangleleft a) \triangleleft b \;=\; p \triangleleft a \;\bowtie\; q \triangleleft b \tag{3.28}$$

$$a \triangleright (p \bowtie q) = (u \bowtie v) \triangleleft b \;\equiv\; a \triangleright q = u \triangleleft b \;\wedge\; p = v \tag{3.29}$$

$$a \triangleright (p \mid q) = (u \mid v) \triangleleft c \;\equiv\; (\exists\, b :: a \triangleright p = u \triangleleft b \;\wedge\; b \triangleright q = v \triangleleft c) \tag{3.30}$$

Proof of (3.27)

$a \triangleright (p \mid q) \;\bowtie\; (u \mid v) \triangleleft b$

$=\;\{$ Axiom (3.25) $\}$

$a \triangleright ((u \mid v) \bowtie (p \mid q)) \triangleleft b$

$=\;\{$ Axiom (3.20) $\}$

$a \triangleright ((u \bowtie p) \mid (v \bowtie q)) \triangleleft b$

$=\;\{$ Axiom (3.24) $\}$

$a \triangleright (u \bowtie p) \;\mid\; (v \bowtie q) \triangleleft b$

Proof of (3.29)

$a \triangleright (p \bowtie q) = (u \bowtie v) \triangleleft b$

$\equiv\;\{$ introduce symmetry on right-hand side with Axiom (3.12) $\}$

$c \triangleright (a \triangleright (p \bowtie q)) = c \triangleright (u \bowtie v) \triangleleft b$

$\equiv\;\{$ Axiom (3.26) and Axiom (3.25) $\}$

[2]The axiom set above is simpler and more expressive than the one found in [Kor97b, Kor97c] where (3.24), (3.25) and (3.26) were replaced by (3.27), (3.30) and (3.31).

$c \triangleright p \bowtie a \triangleright q = c \triangleright v \bowtie u \triangleleft b$

$\equiv$ { Axioms (3.11) and (3.12) }

$p = v \ \wedge \ a \triangleright q = u \triangleleft b$

Proof of (3.28)

$p \triangleleft a \bowtie q \triangleleft b = ((p \bowtie q) \triangleleft a) \triangleleft b$

$\equiv$ { introduce symmetry to right-hand side with Axioms (3.12) and (3.23) }

$c \triangleright (p \triangleleft a \bowtie q \triangleleft b) = (c \triangleright (p \bowtie q) \triangleleft a) \triangleleft b$

$\equiv$ { Axiom (3.25) }

$c \triangleright (p \triangleleft a \bowtie q \triangleleft b) = (c \triangleright q \ \bowtie \ p \triangleleft a) \triangleleft b$

$\equiv$ { (3.29) }

$c \triangleright (q \triangleleft b) = (c \triangleright q) \triangleleft b$

$\equiv$ { Axiom (3.23) }

true

End of Proof

The proof of (3.30) can be found in Section 3.1.2 below. From (3.29) and (3.18) we can derive the following laws, which are useful in proofs of properties of ParLists.

Lemma 11

$$a \triangleright (p \bowtie q) = (u \bowtie p) \triangleleft b \ \equiv \ a \triangleright q = u \triangleleft b \tag{3.31}$$

$$(\forall a, p, q :: (\exists\, b, u \ :: \ a \triangleright (p \bowtie q) = (u \bowtie v) \triangleleft b \ \wedge \ a \triangleright q = u \triangleleft b)) \tag{3.32}$$

$$(\forall b, u, v :: (\exists\, a, q \ :: \ a \triangleright (v \bowtie q) = (u \bowtie v) \triangleleft b \ \wedge \ a \triangleright q = u \triangleleft b)) \tag{3.33}$$

Proof of (3.32) ((3.33) is similar); (3.31) follows from (3.29) by instantiation.

true

$\equiv$ { Axiom (3.18) }

$(\forall a, q :: (\exists\, b, u :: a \triangleright q = u \triangleleft b))$

$\equiv$ { Lemma 11 (3.31) }

$(\forall a, p, q :: (\exists\, b, u :: a \triangleright (p \bowtie q) = (u \bowtie p) \triangleleft b \ \wedge \ a \triangleright q = u \triangleleft b))$

End of Proof

Scalar Operators

Let $\otimes : \mathsf{X} \times \mathsf{X} \longrightarrow \mathsf{X}$ be a binary operator, defined on the scalar type X. We lift $\otimes$ to operate on $\mathsf{ParList.X}$, i.e., $\otimes : \mathsf{ParList.}X.n \times \mathsf{ParListX.}n \longrightarrow \mathsf{ParList.X.}n$ in an "element-wise" fashion, with the following laws

$$\langle a \rangle \otimes \langle b \rangle \quad = \quad \langle a \otimes b \rangle \tag{3.34}$$

$$(a \triangleright p) \otimes (b \triangleright q) \quad = \quad (a \otimes b) \triangleright (p \otimes q) \tag{3.35}$$

$$(p \bowtie q) \otimes (u \bowtie v) \quad = \quad (p \otimes u) \ \bowtie \ (q \otimes v) \tag{3.36}$$

As alternatives to (3.35) and (3.36) we could have chosen (3.37) and (3.38) as they are interchangeable:

$$(p \triangleleft a) \otimes (q \triangleleft b) \quad = \quad (p \otimes q) \triangleleft (a \otimes b) \tag{3.37}$$

$$(p \mid q) \otimes (u \mid v) \quad = \quad (p \otimes u) \ \mid \ (q \otimes v) \tag{3.38}$$

The proofs that (3.37) and (3.38) follows from (3.34), (3.35) and (3.36) can be found in [Kor97c].

3.1.2 Induction Principle for ParList

A $\mathsf{ParList}$ p with elements from the type X (i.e., $p \in \mathsf{ParLists.X}$) can be deconstructed uniquely into an ordered sequence of its elements; this can be achieved by building a *constructor tree* for p. We use constructor trees as the formal basis for the induction principle for $\mathsf{ParLists}$ and in the definition of functions over $\mathsf{ParLists}$. We build the constructor tree as follows. First, restrict the use of $\triangleright$ and $\triangleleft$ so they only construct elements in $\mathsf{OddParList.X}$. Pick one of $\bowtie$ and $\mid$, and one of $\triangleright$ and $\triangleleft$; without loss of generality we choose $\bowtie$ and $\triangleright$ below. Construct a tree from p by labeling the root with p; for each leaf labeled with an element of $\mathsf{ParList.X}$, say q, perform the following operations:

- If $q \in \mathsf{Singleton.X}$ then by Axiom (3.14) there exists a in X such that $q = \langle a \rangle$. Make the leaf node an interior node by creating a child leaf labeled with a.
- If $q \in \mathsf{EvenParList.X}$ then by Axiom (3.16) there exists similar $\mathsf{ParLists}$ u and v such that $q = u \bowtie v$. Make the leaf node labeled q interior by creating its two child leaves labeled u and v from left to right.
- If $q \in \mathsf{OddParList.X}$ then by Axiom (3.16) there exists a $\mathsf{ParList}$ u and an element a in X such that $q = a \triangleright u$. Make the leaf node labeled q interior by creating its two child leaves labeled a and u from left to right.

These operations are performed until all the leaves in the tree are labeled by elements in X. Each operation produces children whose labels have lengths that are shorter than the label of their parent[3]; hence, the tree construction terminates. The elements of p appear in order in the leaves of the resulting tree. The construction of the tree is deterministic since the types $\mathsf{Singleton.X}$, $\mathsf{EvenParList.X}$, and $\mathsf{OddParList.X}$ are disjoint, and Axioms (3.11) and (3.12) assert that the choices of u, v and a are unique. Thus, by picking a pair of constructors the constructor tree for a $\mathsf{ParList}$ is unique.

We use constructor trees as the structure that defines the inductive principle for $\mathsf{ParLists}$. Let $\Pi : \mathsf{ParList.X}.n \longrightarrow \mathsf{Bool}$ be a predicate whose truth is to be established for all $\mathsf{ParLists}$ over X. To establish the property $\Pi.p$ for a $\mathsf{ParList}$ p, build the constructor tree for p using a constructor pair. If we can prove that Π holds at a node when Π holds at each of its non-leaf children, then we can conclude that Π holds at the root of the tree. These observations are captured in the induction

[3]Consider the lengths of elements of X as 0.

principle for $\mathsf{ParLists}$:

$$\begin{array}{ll}
& \quad (\forall x : x \in \mathsf{X} : \Pi.\langle x \rangle) \\
& \wedge \ (\ (\forall p,q,n : p,q \in \mathsf{ParList.X}.n \ \wedge \ n \in \mathsf{Pos} : \Pi.p \ \wedge \ \Pi.q \ \Rightarrow \ \Pi.(p \mid q)) \\
& \quad \vee(\forall p,q,n : p,q \in \mathsf{ParList.X}.n \ \wedge \ n \in \mathsf{Pos} : \Pi.p \ \wedge \ \Pi.q \ \Rightarrow \ \Pi.(p \bowtie q)) \ \ ) \\
& \wedge \ (\ (\forall p,x : p \in \mathsf{EvenParList.X} \ \wedge \ x \in \mathsf{X} : \Pi.p \ \Rightarrow \ \Pi.(x \triangleright p)) \\
& \quad \vee(\forall p,x : p \in \mathsf{EvenParList.X} \ \wedge \ x \in \mathsf{X} : \Pi.p \ \Rightarrow \ \Pi.(p \triangleleft x)) \ ) \\
\Rightarrow & \\
& \quad (\forall p,n : p \in \mathsf{ParList.X}.n \ \wedge \ n \in \mathsf{Pos} : \Pi.p)
\end{array}$$

In the induction principle the choice of the constructor pair is captured by the four disjuncts. A proof that follows the induction principle consists of three parts: a base case, an even inductive step and an odd inductive step. This is illustrated in the proof of (3.30)

$$a \triangleright (p \mid q) = (u \mid v) \triangleleft c \quad \equiv \quad (\exists\, b :: a \triangleright p = u \triangleleft b \ \wedge \ b \triangleright q = v \triangleleft c)$$

given below.

Proof of (3.30). Base case:

$a \triangleright (\langle x \rangle \mid \langle y \rangle) = (\langle z \rangle \mid \langle d \rangle) \triangleleft c$

$\equiv$ { Axiom (3.19) }

$a \triangleright (\langle x \rangle \bowtie \langle y \rangle) = (\langle z \rangle \bowtie \langle d \rangle) \triangleleft c$

$\equiv$ { Lemma 11 (3.31) }

$a \triangleright \langle y \rangle = \langle z \rangle \triangleleft c \ \wedge \ \langle x \rangle = \langle d \rangle$

$\equiv$ { Axioms (3.21) (3.22) and (3.14) }

$a = z \ \wedge \ y = c \ \wedge \ x = d$

$\equiv$ { Axioms (3.21) (3.22) and (3.14) }

$a \triangleright \langle x \rangle = \langle z \rangle \triangleleft d \ \wedge \ d \triangleright \langle y \rangle = \langle d \rangle \triangleleft c$

$\equiv$ { one-point rule $b := d$ }

$(\exists b :: a \triangleright \langle x \rangle = \langle z \rangle \triangleleft b \;\wedge\; b \triangleright \langle y \rangle = \langle d \rangle \triangleleft c)$

Odd inductive step

$a \triangleright (d \triangleright p \mid q \triangleleft x) = (y \triangleright u \mid v \triangleleft z) \triangleleft c$

$\equiv$ { Axiom (3.24) and Axiom (3.23) }

$a \triangleright (d \triangleright (p \mid q)) \triangleleft x = y \triangleright ((u \mid v) \triangleleft z) \triangleleft c$

$\equiv$ { Axiom (3.12) and Axiom (3.13) }

$d \triangleright (p \mid q) = (u \mid v) \triangleleft z \;\wedge\; a = y \;\wedge\; x = c$

$\equiv$ { induction (3.30) }

$(\exists b :: d \triangleright p = u \triangleleft b \;\wedge\; b \triangleright q = v \triangleleft z) \;\wedge\; a = y \;\wedge\; x = c$

$\equiv$ { predicate calculus and Axioms (3.12) and (3.13) }

$(\exists b :: a \triangleright (d \triangleright p) = y \triangleright (u \triangleleft b) \;\wedge\; (b \triangleright q) \triangleleft x = (v \triangleleft z) \triangleleft c)$

$\equiv$ { Axiom (3.23) }

$(\exists b :: a \triangleright (d \triangleright p) = (y \triangleright u) \triangleleft b \;\wedge\; b \triangleright (q \triangleleft x) = (v \triangleleft z) \triangleleft c)$

Even inductive step:

$a \triangleright ((p \bowtie q) \mid (u \bowtie v)) = ((r \bowtie s) \mid (t \bowtie w)) \triangleleft c$

$\equiv$ { Axiom (3.20) }

$a \triangleright ((p \mid u) \bowtie (q \mid v)) = ((r \mid t) \bowtie (s \mid w)) \triangleleft c$

$\equiv$ { (3.29) }

$a \triangleright (q \mid v) = (r \mid t) \triangleleft c \;\wedge\; p \mid u = s \mid w$

$\equiv$ { induction (3.30) }

$(\exists b :: a \triangleright q = r \triangleleft b \;\wedge\; b \triangleright v = t \triangleleft c \;\wedge\; p = s \;\wedge\; u = w)$

$\equiv$ { (3.29) }

$(\exists b :: a \triangleright (p \bowtie q) = (r \bowtie s) \triangleleft b \;\wedge\; b \triangleright (u \bowtie v) = (t \bowtie w) \triangleleft c)$

End of Proof

3.1.3 Functions in ParList

A function over ParLists is defined by picking a constructor pair and giving three different defining cases based on the length of the argument ParList: singleton, even length and odd length. Functions are defined unambiguously this way, since the constructor tree for a ParList is unique given a constructor pair. Each defining case is specified using pattern-matching on the argument ParList:

Subtype	Allowed Constructors
Singleton.X	$\langle\ \rangle$
EvenParList.X	$\bowtie \quad \mid$
OddParList.X	$\triangleright \quad \triangleleft$

We exploit parallelism as much as possible by requiring that $\triangleright$ and $\triangleleft$ only be used in function definitions for ParLists of odd lengths. When the argument is of even length, the computation should be expressed using a balanced divide-and-conquer strategy. Arguments of odd lengths are handled by an alignment step, introduced by necessity.

As an example of a function definition over ParLists, we define the function $rev : \mathsf{ParList.X}.n \longrightarrow \mathsf{ParList.X}.n$ that reverses its argument.

$$rev.\langle a\rangle = \langle a\rangle \tag{3.39}$$

$$rev.(p \bowtie q) = rev.q \bowtie rev.p \tag{3.40}$$

$$rev.(a \triangleright p) = rev.p \triangleleft a \tag{3.41}$$

Note that the choice of $\bowtie$ and $\triangleright$ as the constructor pair was arbitrary: (3.40) can be replaced by (3.42), and (3.41) can be replaced by (3.43) defined below without changing the value of *rev*.

$$rev.(p \mid q) = rev.q \mid rev.p \tag{3.42}$$

$$rev.(p \triangleleft a) = a \triangleright rev.p \tag{3.43}$$

In the definition of *rev*, (3.40) expresses that each recursive case is independent and can be evaluated in parallel. The step described by (3.41) corresponds to a sequential "alignment" step, necessary before a balanced recursive step can be performed. The "alignment" step does not have to be sequential, depending on the parallel architecture and the concrete implementation of ParList, *rev* can be evaluated in constant time. This would be the case on a CREW PRAM with the straightforward implementation of ParList.

A familiar property of *rev* is that it is its own inverse (an involution):

$$rev.(rev.p) = p \tag{3.44}$$

We use the proof of (3.44) as an illustration of applying the inductive principle for ParLists to a function definition:

Proof of (3.44), base case:

$rev.(rev.\langle a \rangle)$

$= \quad \{ \ \ rev \ (3.39) \ \}$

$rev.\langle a \rangle$

$= \quad \{ \ \ rev \ (3.39) \ \}$

$\langle a \rangle$

Inductive even case:

$rev.(rev.(p \bowtie q))$

$= \quad \{ \ \ rev \ (3.40) \ \}$

$rev.(rev.q \bowtie rev.p)$

$= \quad \{ \ \ rev \ (3.40) \ \}$

$rev.(rev.p) \bowtie rev.(rev.q)$

$= \quad \{$ induction (3.44) twice $\}$

$p \bowtie q$

Inductive odd case:

$$
\begin{aligned}
& rev.(rev.(a \triangleright (p \bowtie q))) \\
= & \quad \{ \ rev\ (3.41) \ \} \\
& rev.(rev.(p \bowtie q) \triangleleft a) \\
= & \quad \{ \ rev\ (3.43) \ \} \\
& a \triangleright rev.(rev.(p \bowtie q)) \\
= & \quad \{ \ \text{induction } (3.44) \ \} \\
& a \triangleright (p \bowtie q)
\end{aligned}
$$

End of Proof

The odd inductive case in the proof used (3.43). A longer proof that does not use (3.43) can be found in [Kor97c].

3.1.4 Data Movement Functions

In this section we define operators and functions that move elements within a ParList. The operators $\rightarrow$ and $\leftarrow$ are used in defining the odd-even sort in Section 3.3, and in defining the prefix sum. In Lemma 12 the operators $\rightarrow$ and $\leftarrow$ provide a way to rewrite $\triangleright$ expressions into $\triangleleft$ expressions and vice-versa.

The operator $\rightarrow : \mathsf{X} \times \mathsf{ParList.X}.n \longrightarrow \mathsf{ParList.X}.n$ takes an element and a ParList, and "pushes" a scalar into the list from the left. The rightmost element of the list is lost under this operation. The operator $\rightarrow$ is defined as follows:

$$a \rightarrow \langle b \rangle = \langle a \rangle \tag{3.45}$$

$$a \rightarrow (p \triangleleft b) = a \triangleright p \tag{3.46}$$

$$a \rightarrow (p \bowtie q) = a \rightarrow q \ \bowtie \ p \tag{3.47}$$

The dual operator $\leftarrow : \mathsf{ParList.X}.n \times \mathsf{X} \longrightarrow \mathsf{ParList.X}.n$ "pushes" a scalar into the list from the right and the leftmost element of the list is lost:

$$\langle b \rangle \leftarrow a = \langle a \rangle \tag{3.48}$$

$$(b \triangleright p) \leftarrow a \quad = \quad p \triangleleft a \tag{3.49}$$

$$(p \bowtie q) \leftarrow a \quad = \quad q \bowtie p \leftarrow a \tag{3.50}$$

Next we define the functions *first* and *last* that return the first and last elements, respectively, of a ParList. Their types are $\mathit{first} : \mathsf{ParList.X}.n \longrightarrow \mathsf{X}$ and $\mathit{last} : \mathsf{ParList.X}.n \longrightarrow \mathsf{X}$; they are defined by:

$$\mathit{first}.\langle a \rangle \quad = \quad a \tag{3.51}$$

$$\mathit{first}.(a \triangleright p) \quad = \quad a \tag{3.52}$$

$$\mathit{first}.(p \mid q) \quad = \quad \mathit{first}.p \tag{3.53}$$

$$\mathit{last}.\langle a \rangle \quad = \quad a \tag{3.54}$$

$$\mathit{last}.(p \triangleleft b) \quad = \quad b \tag{3.55}$$

$$\mathit{last}.(p \mid q) \quad = \quad \mathit{last}.q \tag{3.56}$$

We could equally well have chosen the following definitions for the even case:

$$\mathit{first}.(p \bowtie q) \quad = \quad \mathit{first}.p \tag{3.57}$$

$$\mathit{last}.(p \bowtie q) \quad = \quad \mathit{last}.q \tag{3.58}$$

Where convenient we use the following abbreviations for *first* and *last*:

$$\overleftarrow{p} = \mathit{first}.p \quad \text{and} \quad \overrightarrow{p} = \mathit{last}.p$$

Using the definitions above we can state the following pairwise dual properties. We prove (3.59), the proof of (3.60) is dual. The proofs of (3.61) through (3.64) are not very interesting and are omitted.

Lemma 12

$$a \triangleright p \quad = \quad a \rightarrow p \triangleleft \overrightarrow{p} \tag{3.59}$$

$$p \triangleleft a \quad = \quad \overleftarrow{p} \triangleright p \leftarrow a \tag{3.60}$$

$$p = \overleftarrow{p} \rightarrow (p \leftarrow a) \tag{3.61}$$

$$p = (a \rightarrow p) \leftarrow \overrightarrow{p} \tag{3.62}$$

$$a \triangleright (b \rightarrow p) = a \rightarrow (b \triangleright p) \tag{3.63}$$

$$(p \triangleleft b) \leftarrow a = (p \leftarrow b) \triangleleft a \tag{3.64}$$

Proof of (3.59). Base Case:

$a \rightarrow \langle x \rangle \triangleleft \textit{last}.\langle x \rangle$

$=$ { $\rightarrow$ (3.45) and *last* (2.35) }

$\langle a \rangle \triangleleft x$

$=$ { (3.21) and (3.22) }

$a \triangleright \langle x \rangle$

Even inductive case:

$a \triangleright (p \bowtie q) = a \rightarrow (p \bowtie q) \triangleleft \textit{last}.(p \bowtie q)$

$\equiv$ { $\rightarrow$ (3.47), *last* (3.56) }

$a \triangleright (p \bowtie q) = (a \rightarrow q \bowtie p) \triangleleft \overrightarrow{q}$

$\equiv$ { Lemma 11 (3.31) }

$a \triangleright q = a \rightarrow q \triangleleft \overrightarrow{q}$

$\equiv$ { induction (3.59) }

true

Odd inductive case:

$a \rightarrow (p \triangleleft b) \triangleleft \textit{last}.(p \triangleleft b)$

$=$ { $\rightarrow$ (3.47) }

$(a \triangleright p) \triangleleft \textit{last}.(p \triangleleft b)$

$=$ { *last* (3.56) }

$(a \triangleright p) \triangleleft b$

$=$ { Axiom (3.23) }

$a \triangleright (p \triangleleft b)$

End of Proof

3.1.5 Broadcast Sum

We turn to the definition of the function $b\_sum : \mathsf{ParList.Y}.n \longrightarrow \mathsf{ParList.Y}.n$, that returns a list where each element is the sum of all the elements of the argument list (a broadcast sum). Here Y is a type with the property that $(\mathsf{Y}, +)$ is a semigroup (i.e., $+$ is associative). It is necessary to define the function $[a+] : \mathsf{ParList.Y}.n \longrightarrow \mathsf{ParList.Y}.n$, that returns the ParList where a has been added to each element of the argument ParList.

$$b\_sum.\langle a \rangle = a \tag{3.65}$$

$$b\_sum.(a \triangleright p) = (a + \vec{t}\,) \triangleright [a+].t, \quad \text{where } t = b\_sum.p \tag{3.66}$$

$$b\_sum.(p \bowtie q) = t \bowtie t, \quad \text{where } t = b\_sum.(p+q) \tag{3.67}$$

$$[a+].\langle b \rangle = \langle a + b \rangle \tag{3.68}$$

$$[a+].(b \triangleright p) = (a+b) \triangleright [a+].p \tag{3.69}$$

$$[a+].(p \mid q) = [a+].p \mid [a+].q \tag{3.70}$$

When b_sum is evaluated with an argument of length $2^n - 1,\ n \geq 1$ there are $n-1$ deconstructions using $\triangleright$ and $n-1$ deconstructions using $\bowtie$. Each deconstruction takes one parallel time step in order to perform the sum. The total number of parallel steps thus becomes $2*n-2$. In contrast, if the argument is of length 2^n, only n parallel steps are needed. Adding a sufficient number of dummy elements (i.e., identity elements of $+$ if they exist) to a list makes it into a PowerList. Thus, functions like b_sum can be evaluated in parallel in fewer steps than with the original list.

3.1.6 Reusing PowerList Proofs in the ParList Algebra

One of the advantages of the ParList algebra is that it is an extension of the PowerList algebra. Assume that we have proved a property of a function defined in the

PowerList algebra. When we extend the definition of the function to a ParList function by adding an odd defining case, the theorem still holds for those ParLists that are also PowerLists, i.e., whose length is a power of two. Moreover, inductive proofs of properties done in the PowerList algebra can be reused in the proof of the same property for the extended function in the ParList algebra. Depending on the structure of the PowerList proof, the only remaining proof obligation may be to prove the odd inductive step.

Take as an example the function *rev* defined in the PowerList algebra by (3.39) and (3.40). A proof of (3.44), (i.e., $rev.(rev.p) = p$) consisting of the base and even cases is sufficient to prove the property in the PowerList algebra. When (3.41) is added to make *rev* a ParList function, the odd case is the only missing part of the proof; the two others can be reused.

In general, the base case can always be reused from the PowerList proof; the inductive case from the PowerList proof can be reused as part of the proof of the even case. If there are no assumptions made about the structure or lengths of the sub-terms in the proof and the proof does not use other equalities, then the entire even case can be reused. In the even inductive step in the proof of (3.44), the "shape" of the sub-terms were left unspecified; thus the entire case can be reused.

When the inductive case in the PowerList proof assumes that the sub-terms are constructed in two levels, e.g., the proof obligation is written as $\Pi.((p \bowtie q) \mid (u \bowtie v))$, then the even step needs to be completed with a proof of $\Pi.((a \triangleright p) \mid (b \triangleright q))$.

When a PowerList proof uses other equalities proven in the PowerList algebra that have not yet been extended to ParList, then these proofs need to be extended to ParLists as well.

The following is an example of a property that can be proven in the PowerList algebra:

$$length.p \text{ is even} \quad \Rightarrow \quad length.p \text{ is a power of } 2$$

However, this property is specific to PowerLists. An attempt at an inductive ParList proof breaks down in the even case, if the sub-terms are of odd length. Note that the implication is vacuously true in the odd case, so a naive reuse of the PowerList proof could have dire consequences.

3.1.7 Concatenation

A very useful operation on lists is to append one list onto another, regardless of the length of the lists. We define the concatenation operator

$$\Diamond : \mathsf{ParList.X}.n \times \mathsf{ParList.X}.m \longrightarrow \mathsf{ParList.X}.(n+m)$$

by the following nine equations[4]; note that $\Diamond$ has a lower binding power than that of $\bowtie$, $|$, $\triangleright$ and $\triangleleft$:

$$\begin{aligned}
\langle a\rangle \Diamond \langle b\rangle &= \langle a\rangle \bowtie \langle b\rangle && (3.71)\\
\langle a\rangle \Diamond p \triangleleft b &= a \triangleright p \triangleleft b && (3.72)\\
\langle a\rangle \Diamond (p \bowtie q) &= a \triangleright (p \bowtie q) && (3.73)\\
a \triangleright p \Diamond \langle b\rangle &= a \triangleright p \triangleleft b && (3.74)\\
a \triangleright p \Diamond q \triangleleft b &= a \triangleright (p \Diamond q) \triangleleft b && (3.75)\\
a \triangleright (p \bowtie q) \Diamond u \bowtie v &= a \triangleright ((p \Diamond u) \bowtie (q \Diamond v)) && (3.76)\\
p \bowtie q \Diamond \langle a\rangle &= (p \bowtie q) \triangleleft a && (3.77)\\
p \bowtie q \Diamond (u \bowtie v) \triangleleft a &= ((p \Diamond u) \bowtie (q \Diamond v)) \triangleleft a && (3.78)\\
p \bowtie q \Diamond u \bowtie v &= (p \Diamond u) \bowtie (q \Diamond v) && (3.79)
\end{aligned}$$

By its nature $\Diamond$ is a generalization of $|$, so it is no surprise that $\Diamond$ is defined using $\bowtie$ as the constructor. It does not appear that $|$ can be used as the defining constructor for $\Diamond$. Note the similarity between (3.20) and (3.79); in fact, by remove the equations above where the arguments to $\Diamond$ have different length (i.e., (3.72), (3.73), (3.74),

[4]There are nine defining cases to account for all combinations of the three cases for each operand.

(3.76), (3.77) and (3.78)) we are left with axioms ((3.71), (3.75) and (3.79)) that define an operator isomorphic to $|$. Restricting the type of arguments of $\diamondsuit$ to lists of equal length and only keeping those equations that make sense under this restriction (i.e., (3.71), (3.75) and (3.79)) we have defined an operator that is isomorphic to $|$.

Many properties that hold for $|$ hold for $\diamondsuit$ as well; however, they are more tedious to prove since there are 9 defining cases to consider. We list a few properties of $\diamondsuit$ below:

$$\begin{aligned}
\mathit{first}.(p \diamondsuit q) &= \mathit{first}.p && (3.80)\\
\mathit{last}.(p \diamondsuit q) &= \mathit{last}.q && (3.81)\\
a{\rightarrow}(p \diamondsuit q) &= a{\rightarrow}p \diamondsuit \vec{p}{\rightarrow}q && (3.82)\\
(p \diamondsuit q){\leftarrow}a &= p{\leftarrow}\overleftarrow{q} \diamondsuit q{\leftarrow}a && (3.83)\\
[a+].(p \diamondsuit q) &= [a+].p \diamondsuit [a+].q && (3.84)\\
\mathit{sum}.(p \diamondsuit q) &= \mathit{sum}.p \diamondsuit \mathit{sum}.q && (3.85)
\end{aligned}$$

One important law that holds for $|$ but not for $\diamondsuit$ is (3.38), due to the ambiguity that arises when deconstructing the arguments using $\diamondsuit$.

Since $\diamondsuit$ is a generalization of $|$, one could ask why $\diamondsuit$ was not chosen as one of the fundamental constructors for ParList. The arguments of $|$ and $\bowtie$ are of equal length, enforcing a balanced construction, which is essential to obtaining efficient parallel implementations. The Bird-Meertens theory of lists is based on a concatenation operator similar to $\diamondsuit$. We discuss this theory in Chapter 5 along with other related work.

3.2 Prefix Sum

In Chapter 2 we saw that the prefix sum computation can be specified by a PowerList function (ps), as the unique solution to the equation (in u):

$$u = (0 \rightarrow u) + p \tag{3.86}$$

In Chapter 2 we derived a solution to (3.86) for the even case. Here we explore the odd case:

$$\begin{array}{ll}
 & ps.(p \triangleleft a) \\
= & \{ \text{ introduce } q \triangleleft b = ps.(p \triangleleft a) \ \} \\
 & q \triangleleft b \\
= & \{ \text{ defining equation for } ps \text{ (3.86) } \} \\
 & 0 \rightarrow (q \triangleleft b) + p \triangleleft a \\
= & \{ \ \rightarrow \text{(3.46) } \} \\
 & 0 \triangleright q + p \triangleleft a \\
= & \{ \text{ Lemma 12 (3.59) } \} \\
 & 0 \rightarrow q \triangleleft \vec{q} + p \triangleleft a \\
= & \{ \text{ Axiom (3.35) } \} \\
 & (0 \rightarrow q + p) \triangleleft (\vec{q} + a)
\end{array}$$

Summarizing:

$$\begin{array}{ll}
 & q \triangleleft b \ = \ (0 \rightarrow q + p) \triangleleft (\vec{q} + a) \\
\equiv & \{ \text{ Axiom (3.13) } \} \\
 & q = 0 \rightarrow q + p \ \wedge \ b = \vec{q} + a \\
\equiv & \{ \text{ defining equation for } ps \text{ (3.86), Leibnitz Rule } \} \\
 & q = ps.p \ \wedge \ b = last.(ps.p) + a
\end{array}$$

From the above, along with the PowerList definition from Chapter 2, we get the following definition of Ladner and Fischer's algorithm:

$$ps.\langle a \rangle \ = \ \langle a \rangle \tag{3.87}$$

$$ps.(p \bowtie q) \quad = \quad (0 \rightarrow t + p) \ \bowtie \ t, \quad \text{where } t = ps.(p+q) \tag{3.88}$$

$$ps.(p \triangleleft a) \quad = \quad ps.p \ \triangleleft \ (last.(ps.p) + a) \tag{3.89}$$

Just as in the case of the broadcast sum, a sequential alignment step was introduced for the odd case.

3.3 Odd-Even Sort

In this section we revisit the odd-even sort that we derived for PowerLists in Chapter 2. As in the case of the prefix sum discussed in Section 3.2 above, the restriction to inputs whose lengths are a power of two is unnatural for the odd-even sort. We will extend the PowerList algorithm presented in Chapter 2 to a ParList algorithm, and extend the results on sorting from the PowerList Chapter to ParLists. We present the derivation of the algorithm from its specification and prove that the algorithm terminates. We only exhibit the top level of the termination proof, and omit the extensions of the PowerList results needed to complete the proof.

3.3.1 Sorting

We start by extending relational operators to ParLists. Let $\triangle$ be a relation defined on the data type X, i.e., $\triangle : \mathsf{X} \times \mathsf{X} \longrightarrow \mathsf{Bool}$ and let $p, q, u, v \in \mathsf{ParList.X}.n$, and $x, y \in \mathsf{X}$; we define:

$$\langle x \rangle \ \triangle \ \langle y \rangle \quad \equiv \quad x \ \triangle \ y \tag{3.90}$$

$$(p \bowtie q) \ \triangle \ (u \bowtie v) \quad \equiv \quad (p \ \triangle \ u) \ \wedge \ (q \ \triangle \ v) \tag{3.91}$$

$$(a \triangleright p) \ \triangle \ (b \triangleright q) \quad \equiv \quad (a \triangle b) \ \wedge \ (p \ \triangle \ q) \tag{3.92}$$

As in the case of PowerLists, the laws for the other constructors are a consequence of this definition:

$$(p \mid q) \ \triangle \ (u \mid v) \quad \equiv \quad (p \ \triangle \ u) \ \wedge \ (q \ \triangle \ v) \tag{3.93}$$

$$(p \triangleleft a) \bigtriangleup (q \triangleleft b) \quad \equiv \quad (p \bigtriangleup q) \ \wedge \ (a \bigtriangleup b) \tag{3.94}$$

We extend the definition of *ascending* given in Chapter 2 to OddParList.M by

$$ascending.(a \triangleright p) \quad \equiv \quad a \rightarrow p \leq p \tag{3.95}$$

$$ascending.(q \triangleleft b) \quad \equiv \quad q \leq q \leftarrow b \tag{3.96}$$

As is most often the case, only one of the above equations is needed, since one can be proven from the other. It is worth noting that (3.95) and (3.96) do not use $\top$ and $-$.

We recall the following identities of the $\uparrow$-$\downarrow$ calculus of Chapter 2:

$$(u \downarrow v) \uparrow r = u \quad \equiv \quad u \downarrow v = u \ \wedge \ u \uparrow r = u \tag{3.97}$$

$$(u \uparrow v) \downarrow r = u \quad \equiv \quad u \uparrow v = u \ \wedge \ u \downarrow r = u \tag{3.98}$$

$$u \uparrow v \uparrow r = u \quad \equiv \quad u \uparrow v = u \ \wedge \ u \uparrow r = u \tag{3.99}$$

$$u \downarrow v \downarrow r = u \quad \equiv \quad u \downarrow v = u \ \wedge \ u \downarrow r = u \tag{3.100}$$

We prove (3.97) by reusing the proof of (2.119) in Chapter 2 for the base case and the even case. Since the even case did not make any assumptions about the lengths of sub-terms in the inductive step, the only remaining proof obligation is the odd inductive case.

Proof of (3.97). Odd inductive case:

$((a \triangleright p) \downarrow (b \triangleright q)) \uparrow (c \triangleright v) = a \triangleright p$

$\equiv \quad \{ \ \downarrow$ over $\triangleright$ (3.35) $\}$

$(a \downarrow b \ \triangleright \ p \downarrow q) \uparrow (c \triangleright v) = a \triangleright p$

$\equiv \quad \{ \ \uparrow$ over $\triangleright$ (3.35) $\}$

$(a \downarrow b) \uparrow c \ \triangleright \ (p \downarrow q) \uparrow v = a \triangleright p$

$\equiv \quad \{$ Axiom (3.13) $\}$

$(a \downarrow b) \uparrow c = a \ \wedge \ (p \downarrow q) \uparrow v = p$

$\equiv$ { inductive and base case }

$a{\downarrow}b = a \ \wedge\ a{\uparrow}c = a \ \wedge\ p{\downarrow}q = p \ \wedge\ p{\uparrow}v = p$

$\equiv$ { Axiom (3.12) }

$a{\downarrow}b \ \triangleright\ p{\downarrow}q = a \triangleright p \ \wedge\ a{\uparrow}c \ \triangleright\ p{\uparrow}v = a \triangleright p$

$\equiv$ { $\uparrow\ \downarrow$ over $\triangleright$ (3.35) }

$(a \triangleright p){\downarrow}(b \triangleright q) = a \triangleright p \ \wedge\ (a \triangleright p){\uparrow}(c \triangleright v) = a \triangleright p$

End of Proof

We start our derivation of odd-even sort in ParLists by exploring the first definition (3.95) of *ascending*:

$ascending.(a \triangleright (p \bowtie q))$

$\equiv$ { *ascending* (3.95) }

$a{\rightarrow}(p \bowtie q) \ \leq\ p \bowtie q$

$\equiv$ { $\rightarrow$ (3.46) }

$a{\rightarrow}q \bowtie p \leq p \bowtie q$

$\equiv$ { $\leq$ (3.91) }

$a{\rightarrow}q \leq p \ \wedge\ p \leq q$

$\equiv$ { monotonicity of $\leftarrow$ }

$a{\rightarrow}q \leq p \ \wedge\ p \leq q \ \wedge\ (a{\rightarrow}q){\leftarrow}\vec{q} \ \leq\ p{\leftarrow}\vec{q}$

$\equiv$ { Lemma 12 (3.62) }

$a{\rightarrow}q \leq p \ \wedge\ p \leq q \ \wedge\ q \leq p{\leftarrow}\vec{q}$

$\equiv$ { transitivity of $\leq$, twice }

$a{\rightarrow}q \leq p \ \wedge\ p \leq q \ \wedge\ q \leq p{\leftarrow}\vec{q} \ \wedge\ p \leq p{\leftarrow}\vec{q} \ \wedge\ a{\rightarrow}q \leq q$

$\equiv$ { *first* is monotonic, so $a{\rightarrow}q \leq p \ \Rightarrow\ a \leq \overleftarrow{p}$ }

$a{\rightarrow}q \leq p \ \wedge\ p \leq q \ \wedge\ q \leq p{\leftarrow}\vec{q} \ \wedge\ p \leq p{\leftarrow}\vec{q} \ \wedge\ a{\rightarrow}q \leq q \ \wedge\ a \leq \overleftarrow{p}$

$\equiv$ { $\uparrow\ \downarrow$ calculus (2.100) and (2.101) }

$p = a{\rightarrow}q{\uparrow}p \ \wedge\ p = p{\downarrow}q \ \wedge\ q = p{\uparrow}q \ \wedge\ q = p{\leftarrow}\vec{q}{\downarrow}q$

$$\wedge\;\; p = p{\downarrow}p{\leftarrow}\vec{q} \;\;\wedge\;\; q = a{\to}q{\uparrow}q \;\;\wedge\;\; a = a{\downarrow}\overleftarrow{p}$$

$\equiv$ { reorder terms }

$$a = a{\downarrow}\overleftarrow{p} \;\;\wedge\;\; p = a{\to}q{\uparrow}p \;\;\wedge\;\; p = p{\downarrow}q \;\;\wedge\;\; p = p{\downarrow}p{\leftarrow}\vec{q}$$
$$\wedge\;\; q = p{\leftarrow}\vec{q}{\downarrow}q \;\;\wedge\;\; q = a{\to}q{\uparrow}q \;\;\wedge\;\; q = p{\uparrow}q$$

$\equiv$ { (3.100) with $u, v, r := p, q, p{\leftarrow}\vec{q}$ and (3.99) with $u, v, r := q, a{\to}q, p$ }

$$a = a{\downarrow}\overleftarrow{p} \;\;\wedge\;\; p = a{\to}q{\uparrow}p \;\;\wedge\;\; p = p{\downarrow}q{\downarrow}p{\leftarrow}\vec{q} \;\;\wedge\;\; q = p{\leftarrow}\vec{q}{\downarrow}q \;\;\wedge\;\; q = q{\uparrow}a{\to}q{\uparrow}p$$

$\equiv$ { (3.97) $u, v, r := p, a{\to}q, p{\downarrow}q{\downarrow}p{\leftarrow}\vec{q}$; (3.98) $u, v, r := q, p{\leftarrow}\vec{q}, q{\uparrow}a{\to}q{\uparrow}p$ }

$$a = a{\downarrow}\overleftarrow{p} \;\;\wedge\;\; p = (a{\to}q{\uparrow}p){\downarrow}p{\downarrow}q{\downarrow}p{\leftarrow}\vec{q} \;\;\wedge\;\; q = (p{\leftarrow}\vec{q}{\downarrow}q){\uparrow}q{\uparrow}a{\to}q{\uparrow}p$$

$\equiv$ { Axiom (3.11) and Axiom (3.12) }

$$a \triangleright (p \bowtie q) = (a{\downarrow}\overleftarrow{p}) \triangleright ((a{\to}q{\uparrow}p){\downarrow}p{\downarrow}q{\downarrow}p{\leftarrow}\vec{q} \;\bowtie\; (p{\leftarrow}\vec{q}{\downarrow}q){\uparrow}q{\uparrow}a{\to}q{\uparrow}p) \quad (3.101)$$

In Chapter 2 we defined the odd-even sort *oddeven* for PowerLists by:

$$\begin{aligned}
even.(p \bowtie q) &= p{\downarrow}q \;\bowtie\; p{\uparrow}q && (3.102)\\
even.\langle x\rangle &= \langle x\rangle && (3.103)\\
odd.(u \bowtie v) &= -{\to}v{\uparrow}u \;\bowtie\; v{\downarrow}u{\leftarrow}\top && (3.104)\\
odd.\langle x\rangle &= \langle x\rangle && (3.105)\\
oddeven.p &= odd.(even.p) && (3.106)
\end{aligned}$$

We will use the above as the definition of *oddeven* for the Singleton and EvenParList cases, and continue by exploring (3.101) for a definition for the OddParList.M case:

$$a{\downarrow}\overleftarrow{p} \;\triangleright\; ((a{\to}q{\uparrow}p){\downarrow}p{\downarrow}q{\downarrow}p{\leftarrow}\vec{q} \;\bowtie\; (p{\leftarrow}\vec{q}{\downarrow}q){\uparrow}q{\uparrow}a{\to}q{\uparrow}p)$$

$=$ { property of $\leftarrow$, $\to$: $(a{\to}q){\leftarrow}\vec{q} = q$ }

$$a{\downarrow}\overleftarrow{p} \;\triangleright\; ((a{\to}q{\uparrow}p){\downarrow}p{\downarrow}(a{\to}q){\leftarrow}\vec{q}{\downarrow}p{\leftarrow}\vec{q} \;\bowtie\; (p{\leftarrow}\vec{q}{\downarrow}(a{\to}q){\leftarrow}\vec{q}){\uparrow}q{\uparrow}a{\to}q{\uparrow}p)$$

$=$ { $\to$ over $\uparrow$ and $\downarrow$ }

$$a{\downarrow}\overleftarrow{p} \;\triangleright\; ((a{\to}q{\uparrow}p){\downarrow}p{\downarrow}(a{\to}q{\downarrow}p){\leftarrow}\vec{q} \;\bowtie\; (p{\downarrow}a{\to}q){\leftarrow}\vec{q}{\uparrow}q{\uparrow}a{\to}q{\uparrow}p)$$

$=$ { *even* (3.102) for EvenParList }

$$a{\downarrow}\overleftarrow{p} \;\triangleright\; even.(a{\to}q{\uparrow}p \;\bowtie\; p{\downarrow}(a{\to}q{\downarrow}p){\leftarrow}\vec{q})$$

$=$ { *first* distributes over $\downarrow$; $\leftarrow$(3.50) }

$$
\begin{aligned}
& \mathit{first}.(a{\rightarrow}q{\downarrow}p)\ \triangleright\ \mathit{even}.(a{\rightarrow}q{\uparrow}p\ \bowtie\ p{\downarrow}(a{\rightarrow}q{\downarrow}p){\leftarrow}\overrightarrow{q}) \\
=\ & \{\ \text{define } \mathit{odd} \text{ for } \mathsf{OddParList.M} \text{ by } (3.108)\ \} \\
& \mathit{odd}.((a{\rightarrow}q{\downarrow}p\ \bowtie\ a{\rightarrow}q{\uparrow}p){\triangleleft}\,\overrightarrow{q}) \\
=\ & \{\ \mathit{even}\ (3.102)\ \} \\
& \mathit{odd}.(\mathit{even}.(a{\rightarrow}q \bowtie p){\triangleleft}\,\overrightarrow{q}) \\
=\ & \{\ \rightarrow (3.47)\ \} \\
& \mathit{odd}.(\mathit{even}.(a{\rightarrow}(p \bowtie q)){\triangleleft}\,\overrightarrow{q}) \\
=\ & \{\ \text{define } \mathit{even} \text{ for } \mathsf{OddParList.M} \text{ by } (3.107)\ \} \\
& \mathit{odd}.(\mathit{even}.(a \triangleright (p \bowtie q))) \\
=\ & \{\ \text{define } \mathit{oddeven} \text{ for } \mathsf{OddParList.M} \text{ by } (3.109)\ \} \\
& \mathit{oddeven}.(a \triangleright (p \bowtie q))
\end{aligned}
$$

We have derived the following definition of the odd-even sort for OddParList.M:

$$\mathit{even}.(a \triangleright q) \quad = \quad \mathit{even}.(a{\rightarrow}q){\triangleleft}\,\overrightarrow{q} \tag{3.107}$$

$$\mathit{odd}.(p{\triangleleft}\, b) \quad = \quad \overleftarrow{p} \triangleright \mathit{even}.(p{\leftarrow}b) \tag{3.108}$$

$$\mathit{oddeven}.(a \triangleright p) \quad = \quad \mathit{odd}.(\mathit{even}.(a \triangleright p)) \tag{3.109}$$

Note that we use the PowerList function *even* in the above derivation and definition. Since *even* is defined without the use of $\top$ and $-$, the above definition does not depend on their existence. It is also worth noting the duality between (3.107) and (3.108); had we started with (3.96) as a definition of *ascending*, we would have derived an algorithm where the roles of *odd* and *even* were reversed:

$$\mathit{oddeven}'.(q{\triangleleft}\, b) \ = \ \mathit{even}.(\mathit{odd}.(q{\triangleleft}\, b)) \tag{3.110}$$

We proceed by proving that iterating *oddeven* converges towards a (sorted) fixpoint. The lexical ordering ($\prec$) used for PowerLists is extended to ParLists as follows:

$$a \triangleright u \prec b \triangleright v \quad = \quad a < b\ \vee\ (a = b\ \wedge\ u \prec v) \tag{3.111}$$

$$u{\triangleleft}\, a \prec v{\triangleleft}\, b \quad = \quad u \prec v\ \vee\ (u = v\ \wedge\ a < b) \tag{3.112}$$

The convergence property follows from

$$even.p \prec p \quad \equiv \quad \neg(even.p = p) \tag{3.113}$$

We start by proving the odd inductive case:

$$even.(a \triangleright q) \prec a \triangleright q \quad \equiv \quad \neg(even.(a \triangleright q) = a \triangleright q) \tag{3.114}$$

Proof of (3.114)

$even.(a \triangleright q) \prec a \triangleright q$

$\equiv$ { *even* (3.107) }

$even.(a \to q) \triangleleft \vec{q} \prec a \triangleright q$

$\equiv$ { Lemma 12 (3.59) }

$even.(a \to q) \triangleleft \vec{q} \prec a \to q \triangleleft \vec{q}$

$\equiv$ { $\prec$ (3.112) }

$even.(a \to q) \prec a \to q \;\vee\; (even.(a \to q) = a \to q \;\wedge\; \vec{q} < \vec{q})$

$\equiv$ { induction (3.114), see (3.113) below }

$\neg(even.(a \to q) = a \to q) \;\vee\; (even.(a \to q) = a \to q \;\wedge\; \vec{q} < \vec{q})$

$\equiv$ { predicate calculus }

$\neg(even.(a \to q) = a \to q \;\wedge\; \vec{q} = \vec{q})$

$\equiv$ { Axiom (3.12) }

$\neg(even.(a \to q) \triangleleft \vec{q} = a \to q \triangleleft \vec{q})$

$\equiv$ { *even* (3.107); Lemma 12 (3.59) }

$\neg(even.(a \triangleright q) = a \triangleright q)$

End of Proof

To reuse the proof of (3.113) from Section 2.7.2, we need to establish

$$even.(a \triangleright p \mid q \triangleleft b) \prec a \triangleright p \mid q \triangleleft b \quad \equiv \quad \neg(even.(a \triangleright p \mid q \triangleleft b) = a \triangleright p \mid q \triangleleft b) \tag{3.115}$$

This proof is omitted, since it follows closely the proof given in Chapter 2.

The proof of the dual result

$$odd.(p \triangleleft b) \prec p \triangleleft b \quad \equiv \quad \neg(odd.(p \triangleleft b) = p \triangleleft b) \tag{3.116}$$

is similar to the above proof and is omitted.

3.4 Adder Circuits

In [Ada94] Will Adams presented PowerList descriptions for two arithmetic circuits that perform addition on natural numbers: the *ripple carry adder* and the *carry lookahead adder*. The ripple carry adder performs addition as it is first taught in grade school; it is an inheritly sequential method, yielding a running time that is linear in the number of bits to be added. The carry lookahead adder uses a prefix sum calculation to propagate carries, yielding a method that is logarithmic in the number of bits to be added in a setting where sufficient parallelism is available.

Adams proved that the ripple carry circuit correctly implements addition, and that the carry lookahead and the ripple carry circuits implement the same function. This result was established in the PowerList algebra. Since the PowerList algebra only contains lists whose length are a power of two, and there are no a priori restrictions on the length of either addition circuit, these circuits should be specified as ParList functions.

In the following we extend the definition of the addition circuits and the equivalence result to the ParList algebra. We start by defining the data types

$$\mathsf{Bit} = \{\mathsf{0},\mathsf{1}\}$$
$$\mathsf{Trit} = \{\mathsf{0},\mathsf{1},\pi\}$$

where 0 and 1 are the binary digits, and π corresponds to a "propagate" action for the carry-in value to a position, in the carry lookahead adder.

The ripple carry adder takes three arguments:

$$rc : \mathsf{Bit} \times \mathsf{ParList.Bit}.n \times \mathsf{ParList.Bit}.n \longrightarrow \mathsf{ParList.Bit}.n \times \mathsf{Bit}$$

The first argument is the carry-in bit and the second and third arguments are the two ParLists of bits that are to be added. The result is a pair; the first component of the pair is a ParList containing the result of the addition, and the second component is the carry-out bit from the addition. The following equations defines rc, where (3.117) and (3.118) are taken from [Ada94]:

$$\begin{aligned}
rc.b.\langle x\rangle.\langle y\rangle &= (\langle (x+y+b) \bmod 2\rangle, (x+y+b) \div 2) && (3.117)\\
rc.b.(p \mid q).(r \mid s) &= (t, d) && (3.118)\\
&\quad \text{where} \quad \begin{aligned} t &= u \mid v\\ (u,c) &= rc.b.p.r\\ (v,d) &= rc.c.q.s \end{aligned}\\
& && (3.119)\\
rc.c.(p \triangleleft a).(q \triangleleft b) &= (u \triangleleft y, x) && (3.120)\\
&\quad \text{where} \quad \begin{aligned} x &= (a+b+d) \div 2\\ y &= (a+b+d) \bmod 2\\ (u,d) &= rc.c.p.q \end{aligned}
\end{aligned}$$

The carry lookahead adder has the following type

$$cl : \mathsf{Trit} \times \mathsf{ParList.Trit}.n \times \mathsf{ParList.Trit}.n \longrightarrow \mathsf{ParList.Trit}.n \times \mathsf{Trit}$$

To specify the carry lookahead adder, Adams introduced the associative scalar operators $\bullet$, $\star$ and $\odot$ defined by:

$$\bullet : \mathsf{Trit} \times \mathsf{Trit} \longrightarrow \mathsf{Trit} \qquad x \bullet y = \begin{cases} x & \text{if } x = y \\ \pi & \text{if } x \neq y \end{cases} \qquad (3.121)$$

$$\star : \mathsf{Trit} \times \mathsf{Trit} \longrightarrow \mathsf{Trit} \qquad x \star y = \begin{cases} y & \text{if } y \neq \pi \\ x & \text{if } y = \pi \end{cases} \qquad (3.122)$$

$$\odot : \mathsf{Trit} \times \mathsf{Trit} \longrightarrow \mathsf{Trit} \qquad x \odot y = \begin{cases} x & \text{if } y \neq \pi \\ \neg y & \text{if } y = \pi \end{cases} \tag{3.123}$$

$$\begin{array}{rrcl} & \neg 0 & = & 1 \\ \text{where} & \neg 1 & = & 0 \\ & \neg \pi & = & \pi \end{array}$$

Adams [Ada94] defined the carry lookahead adder by

$$cl.b.p.q \;=\; (t,d) \tag{3.124}$$

$$\begin{array}{rrcl} \text{where} & t & = & s \odot r \\ & d & = & \vec{s} \star \vec{r} \\ & r & = & p \bullet q \\ & s & = & ps.(b \rightarrow r) \end{array}$$

and ps is computed using the associative operator $\star$ (that has π as its neutral element). He proceeded by deriving the following recursive description of cl for the even case:

$$cl.b.(p \mid q).(u \mid v) \;=\; (t,d) \tag{3.125}$$

$$\begin{array}{rrcl} \text{where} & t & = & r \mid s \\ & (r,a) & = & cl.b.p.u \\ & (s,c) & = & cl.a.q.v \end{array}$$

By expanding the odd case of the definition of cl we get:

$$cl.c.(p \triangleleft x).(q \triangleleft y) \;=\; (a,w) \tag{3.126}$$

$$\begin{array}{rrcl} \text{where} & w & = & u \odot v \\ & a & = & \vec{u} \star \vec{v} \\ & v & = & (p \triangleleft x) \bullet (q \triangleleft y) \\ & u & = & ps.(b \rightarrow v) \end{array}$$

Comparing this with the quantities defined by $cl.b.p.q$ (3.124), we get

u

$= \{ \ (3.126) \ \}$

$ps.(b{\to}v)$

$= \{ \ \text{above} \ \}$

$ps.(b{\to}(r\triangleleft(x \bullet y)))$

$= \{ \ \to (3.46) \ \}$

$ps.(b \triangleright r)$

$= \{ \ \text{Lemma 12 (3.60)} \ \}$

$ps.((b{\to}r)\triangleleft \vec{r})$

$= \{ \ ps \ (3.89) \ \}$

$ps.(b{\to}r)\triangleleft(last.(ps.(b{\to}r)) \star \vec{r})$

$= \{ \ (3.124) \ \}$

$s\triangleleft(\vec{s} \star \vec{r})$

w

$= \{ \ (3.126) \ \}$

$u \odot v$

$= \{ \ \text{above} \ \}$

$(s\triangleleft(\vec{s} \star \vec{r})) \odot (r\triangleleft(x \bullet y))$

$= \{ \ \text{Axiom (3.37)} \ \}$

$(s \odot r)\triangleleft((\vec{s} \star \vec{r}) \odot (x \bullet y))$

$= \{ \ (3.124) \ \}$

$t\triangleleft(d \odot (x \bullet y))$

In summary, we have

$$cl.c.(p\triangleleft x).(q\triangleleft y) \quad = \quad (t\triangleleft(d \odot (x \bullet y)), d \star (x \bullet y)) \qquad (3.127)$$
$$\text{where} \quad cl.b.p.q = (t, d)$$

We can now prove the missing case in the proof of the equivalence of the ripple carry and carry lookahead adders.

Proof

$rc.c.(p\triangleleft a).(q\triangleleft b) = cl.c.(p\triangleleft a).(q\triangleleft b)$

$\equiv$ { rc (3.120) and cl (3.127) }

$(s\triangleleft((a+b+d) \bmod 2), (a+b+d) \div 2) = (t\triangleleft(e\odot(x\bullet y)), e\star(x\bullet y))$

$\wedge\ (s,d) = rc.c.p.q\ \wedge\ (t,e) = cl.c.p.q$

$\equiv$ { by induction $(s,d) = (t,e)$ [Ada94] }

$(s\triangleleft((a+b+d) \bmod 2), (a+b+d) \div 2) = (s\triangleleft(d\odot(x\bullet y)), d\star(x\bullet y))$

$\equiv$ { equality on pairs }

$(a+b+d) \div 2 = d\star(x\bullet y)\ \wedge\ s\triangleleft((a+b+d) \bmod 2) = s\triangleleft(d\odot(x\bullet y))$

$\equiv$ { Axiom (3.13) }

$(a+b+d) \div 2 = d\star(x\bullet y)\ \wedge\ s=s\ \wedge\ (a+b+d) \bmod 2 = d\odot(x\bullet y)$

$\equiv$ { (3.128) and (3.129) see below }

true

End of Proof

In the last hint we used the following identities established in [Ada94]:

$$d\star(x\bullet y) = (x+y+d) \div 2 \tag{3.128}$$

$$d\odot(x\bullet y) = (x+y+d) \bmod 2 \tag{3.129}$$

Note that in the inductive step we need to establish that the lemmas that were used in proving the equivalence [Ada94] generalize to ParLists. These proofs are omitted in this presentation, since they add little insight into the problem or the ParList theory.

3.5 Summary

The ParList notation is an appropriate generalization of the PowerList notation. For certain PowerList functions, such as the prefix sum, it is unnatural to require that the length of its input is a power of two. With ParLists it is possible to express parallel computations over inputs of arbitrary lengths. For some ParList functions this approach has the drawback that for each odd length encountered during deconstruction of the argument, a sequential alignment step is introduced.

The ParList theory is an extension of the PowerList theory, obtained by adding the constructors $\triangleright$ and $\triangleleft$ from linear list theory. The ParList theory is more complicated; it has 18 axioms in comparison with the 7 axioms in the PowerList theory. However, the additional axioms are simple and have reasonable interpretations in standard models of linear lists. The induction principle for ParLists is simple; it closely follows the way that functions are defined over ParLists.

Many of the PowerList functions that we studied in Chapter 2 have simple extensions in the ParList notation; this was done by providing the inductive case for ParLists of odd length. We derived the odd cases for Ladner and Fischer's prefix sum algorithm and the odd-even sort, and extended Adams' definitions of the ripple carry and the carry lookahead addition circuits to ParLists.

The set of shared axioms makes it possible to reuse proofs of properties of the corresponding PowerList functions when proving the same properties of ParList functions. Combining this observation with the induction principle for ParLists we presented a strategy for reusing PowerList proofs in the ParList theory.

Chapter 4

Plists

In this chapter we generalize the PowerList data structure to PLists. PLists are constructed with the n-way $\bowtie$ and $|$ operators; e.g., for positive n the n-way $|$ takes n similar PLists and returns their concatenation. While the PowerList notation is intimately tied to radix 2, the PList notation enables us to state properties and algorithms in the radix that is most suited for the problem. The PList notation is even more general; it allows the use of mixed radices in specifications, and facilitates algebraic reasoning about such specifications.

We illustrate the PList notation by specifying three generalized connection networks and proving that these network are isomorphic. This work is joint work with my advisor Jayadev Misra [MK97].

Note on Notation

We use square brackets to denote ordered quantification in the PList algebra. The expression $[\bowtie i : i \in \overline{n} : p.i]$ is a closed form for the application of the n-ary operator $\bowtie$ applied to the PLists $p.i$ in order. The range $i \in \overline{n}$ means that the terms of the expression are written from 0 through $n-1$ in their numeric order. We assume that these ranges are non-empty. The same convention applies to the expression

$[| i : i \in \overline{n} : p.i]$; it will also be used for other non-commutative operators, such as string concatenation.

4.1 PList - An Extension of the PowerList Algebra

A PList is a non-empty linear data structure, whose elements are all of the same type, either scalars from the same base type, or (recursively) PLists that enjoy the same property. We define the length of a PList by $\mathit{length} : \mathsf{PList.X}.n \longrightarrow \mathsf{Pos}$:

$$(\forall p : p \in \mathsf{PList.X}.n : \mathit{length}.p = n) \tag{4.1}$$

Two PLists are *similar* if they have the same length and their elements are similar; two scalars are similar when they belong to the same base type. The simplest PList is called a *singleton* and consists of a single element; the singleton containing x is written as $\langle x \rangle$. Let $p.i$, where $0 \leq i < n$ and $n \in \mathsf{Pos}$, be n pairwise similar PLists, each of length m. Define the PList u as

$$u = [| i : i \in \overline{n} : p.i]$$

u is obtained by concatenating the contents of the lists $p.i$ in order, i.e., the jth element of $p.i$ appears as element $i * m + j$ of u. Similarly, the PList v defined by

$$v = [\bowtie i : i \in \overline{n} : p.i]$$

contains the interleaving of the contents of the lists $p.i$ in order, i.e., the jth element of $p.i$ appears as element $i + j * m$ of v.
Formally, the constructors have the following types:

$$\langle \_ \rangle : \mathsf{X} \longrightarrow \mathsf{PList.X.1}$$
$$[| i : i \in \overline{n} : \_] : (\mathsf{PList.X}.m)^n \longrightarrow \mathsf{PList.X}.(n * m)$$
$$[\bowtie i : i \in \overline{n} : \_] : (\mathsf{PList.X}.m)^n \longrightarrow \mathsf{PList.X}.(n * m)$$

PList Axioms

For $0 \le i < n$ and $0 \le j < m$ (where $n, m \in \mathsf{Pos}$) let $p.i.j \in \mathsf{PList.X}.k$, i.e., p ranges over $n * m$ similar PLists; let $u.i, v.i \in \mathsf{PList.X}.k$, and let $x.i, a, b \in \mathsf{X}$. The following axioms define the PList algebra:

$$(\forall t : t \in \mathsf{PList.X}.1 : (\exists\, a :: t = \langle a \rangle)) \tag{4.2}$$

$$(\forall t : t \in \mathsf{PList.X}.(k{*}n) : (\exists\, u :: t = [\bowtie i : i \in \overline{n} : u.i])) \tag{4.3}$$

$$(\forall t : t \in \mathsf{PList.X}.(k{*}n) : (\exists\, u :: t = [\,|\, i : i \in \overline{n} : u.i])) \tag{4.4}$$

$$\begin{aligned}
\langle a \rangle = \langle b \rangle \quad &\equiv \quad a = b && (4.5)\\
[\bowtie i : i \in \overline{n} : u.i] = [\bowtie i : i \in \overline{n} : v.i] \quad &\equiv \quad (\forall i : 0 \le i < n : u.i = v.i) && (4.6)\\
[\,|\, i : i \in \overline{n} : u.i] = [\,|\, i : i \in \overline{n} : v.i] \quad &\equiv \quad (\forall i : 0 \le i < n : u.i = v.i) && (4.7)\\
[\bowtie i : i \in \overline{n} : \langle x.i \rangle] \quad &= \quad [\,|\, i : i \in \overline{n} : \langle x.i \rangle] && (4.8)\\
[\bowtie i : i \in \overline{n} : [\,|\, j : j \in \overline{m} : p.i.j]] \quad &= \quad [\,|\, j : j \in \overline{m} : [\bowtie i : i \in \overline{n} : p.i.j]] && (4.9)
\end{aligned}$$

Let the n-ary operator $[\natural i : i \in \overline{n} : \_\,] : \mathsf{X}^n \longrightarrow \mathsf{X}$ and the unary operator $\sim : \mathsf{X} \longrightarrow \mathsf{X}$ be defined on the scalars of $p.i.j$ and $x.i$. We lift these operators to PLists in the following way:

$$\begin{aligned}
\sim \langle a \rangle \quad &= \quad \langle \sim a \rangle && (4.10)\\
\sim [\,|\, i : i \in \overline{n} : u.i] \quad &= \quad [\,|\, i : i \in \overline{n} : \sim u.i] && (4.11)\\
\sim [\bowtie i : i \in \overline{n} : u.i] \quad &= \quad [\bowtie i : i \in \overline{n} : \sim u.i] && (4.12)\\
[\natural i : i \in \overline{n} : \langle x.i \rangle] \quad &= \quad \langle [\natural i : i \in \overline{n} : x.i] \rangle && (4.13)\\
[\natural i : i \in \overline{n} : [\,|\, j : j \in \overline{m} : p.i.j]] \quad &= \quad [\,|\, j : j \in \overline{m} : [\natural i : i \in \overline{n} : p.i.j]] && (4.14)\\
[\natural i : i \in \overline{n} : [\bowtie j : j \in \overline{m} : p.i.j]] \quad &= \quad [\bowtie j : j \in \overline{m} : [\natural i : i \in \overline{n} : p.i.j]] && (4.15)
\end{aligned}$$

Note that only one of (4.11) and (4.12) and one of (4.14) and (4.15) are needed; for each pair one equation follows from the other.

Let *permute* be a permutation function on PLists. For n ($n \in \mathsf{Pos}$) similar PLists $q.i$ where $0 \leq i < n$ we have:

$$permute.(\sim p) \quad = \quad \sim permute.p \qquad (4.16)$$

$$permute.[\natural i : i \in \overline{n} : q.i] \quad = \quad [\natural i : i \in \overline{n} : permute.(q.i)] \qquad (4.17)$$

4.1.1 Scalar Data Structures

We use linear lists to describe the arities that apply in definitions of functions over PLists. In this section we define linear lists, as well as strings and sets; these data structures are used in defining the connection networks of Section 4.3.

Linear Lists

We use the type function

$$\mathsf{List} : \mathsf{Type} \longrightarrow \mathsf{Type}$$

to construct the type of linear lists over a data type. The empty list is denoted by $[\,]$. For an element $x \in \mathsf{X}$ and a list $l \in \mathsf{List.X}$ we write $x \triangleright l$ for the list that has x as its head and l as its tail, and we write $l \triangleleft x$ for the list that has x as its last element and l as its beginning[1]. When convenient, we use the notation $[x]$ for the list that contains the single element x.

Since we will primarily use linear lists over positive natural numbers, we introduce the name PosList as an abbreviation for List.Pos. We define the function *prod* that computes the product of the elements of a linear list in PosList, i.e., $prod : \mathsf{PosList} \longrightarrow \mathsf{Pos}$:

$$prod.[\,] \quad = \quad 1 \qquad (4.18)$$

$$prod.(x \triangleright l) \quad = \quad x * prod.l \qquad (4.19)$$

[1]We have overloaded the operators $\triangleright$ and $\triangleleft$ from the ParList notation. This is intentional, since the ParList theory can be viewed as a unification of the PowerList and linear list theories.

We also define the function $linrev : \mathsf{List.X} \longrightarrow \mathsf{List.X}$ that reverses a linear list:

$$linrev.[\,] = [\,] \tag{4.20}$$

$$linrev.(x \triangleright l) = linrev.l \triangleleft x \tag{4.21}$$

Strings

We use strings to label the nodes of the connection networks in Section 4.3. The type of strings from an unspecified alphabet is called String. We write the concatenation of the n strings $s.i$, for $0 \leq i < n$, by $[+\!\!+\, i : i \in \overline{n} : s.i]$, using the generalized notation described above. In the special case of $n = 2$ we use the infix version of the operator, i.e., $s +\!\!+\, t$ is the concatenation of the strings s and t; we have $[+\!\!+\, i : i \in \overline{n} : \_\,] : \mathsf{String}^n \longrightarrow \mathsf{String}$.

If s is a string and i is a natural number, then $s{\diamond}i$ (read s "tag" i) is the string obtained by concatenating s with a string representation of the number i. We have $\_{\diamond}\_ : \mathsf{String} \times \mathsf{Nat} \longrightarrow \mathsf{String}$. We recall that $\diamond$ has a higher binding power than $+\!\!+$.

Sets

The elements of a PList can be regarded as a set; this is useful when we prove isomorphisms between network topologies. We use the type $\mathsf{Set.X}$ to denote the type of sets whose elements are in X. We define the "setify" operator $\{\_\} : \mathsf{PList.X}.n \longrightarrow \mathsf{Set.X}$ as follows:

$$\{\langle x\rangle\} = \{x\} \tag{4.22}$$

$$\{[\bowtie i : i \in \overline{n} : u.i]\} = (\cup i : 0 \leq i < n : \{u.i\}) \tag{4.23}$$

$$\{[|\, i : i \in \overline{n} : u.i]\} = (\cup i : 0 \leq i < n : \{u.i\}) \tag{4.24}$$

Note that only one of (4.23) and (4.24) is needed, as one can be proven from the other. Letting *permute* be a permutation function on PLists, we have

$$\{permute.p\} = \{p\} \tag{4.25}$$

4.2 Functions over PLists

Functions over PLists are defined using two arguments. The first argument is a list of arities, and the second is the argument PList. Functions over PLists are only defined for certain pairs of these input values; to express the valid pairs we require that the specification of the function defines the predicate

$$defined : ((\mathsf{List} \times \mathsf{PList}) \longrightarrow \mathsf{X}) \times \mathsf{List} \times \mathsf{PList} \longrightarrow \mathsf{Bool}$$

to characterize where the function is defined. We only write properties of functions where they are defined and it becomes a proof obligation to ensure that the introduced terms are well defined.

We illustrate this convention by defining the function *sum*, which computes the sum of all elements of a PList over a type where $+$ is defined:

$$defined.sum.l.p \equiv prod.l = length.p \tag{4.26}$$

$$sum.[\,].\langle a\rangle = a \tag{4.27}$$

$$sum.(x \triangleright l).[|\,i : i \in \overline{x} : p.i] = (+i : 0 \le i < x : sum.l.(p.i)) \tag{4.28}$$

An example of applying *sum* is:

$sum.[5].[|\,i : i \in \overline{5} : \langle i\rangle]$

$=\quad\{\ \ sum\ (4.28)\ \}$

$(+i : 0 \le i < 5 : sum.[\,].\langle i\rangle)$

$=\quad\{\ \ sum\ (4.27)\ \}$

$(+i : 0 \le i < 5 : i)$

$$
\begin{array}{ll}
= & \{ \text{ arithmetic } \} \\
& 0+1+2+3+4 \\
= & \{ \text{ arithmetic } \} \\
& 10
\end{array}
$$

Note that if we instantiate *sum* with a PowerList p and a linear list consisting of *loglen.p* 2's, we have a function that is the same as the function *sum* defined for PowerList in Chapter 2. This observation will hold for most PList functions, although the predicate *defined* can in principle be written in such a way that the function is undefined for some or all PowerLists. All of the PList functions defined in this chapter can be specialized to PowerList functions.

4.2.1 An Induction Principle for PLists

Functions over PLists are defined by structural induction over two structures, PosList and PList, where the valid pairs are determined by *defined*. We can define two equally valid induction principles for PLists, one based on each of these structures. We present the inductive principle over PosList below. Let

$$\Pi : \mathsf{PosList} \times \mathsf{PList.X}.n \longrightarrow \mathsf{Bool}$$

be a predicate whose truth is to be established for all pairs of PosList and PLists over X where the function applications are defined. We define the predicate

$$\mathit{valid} : (\mathsf{PosList} \times \mathsf{PList.X}.n \longrightarrow \mathsf{Bool}) \times \mathsf{PosList} \times \mathsf{PList.X}.n \longrightarrow \mathsf{Bool}$$

to characterize these pairs. We can establish the predicate Π by the following induction principle:

$$\begin{array}{ll}
& (\forall p : p \in \mathsf{PList.X}.n \ \wedge \ \mathit{valid}.\Pi.[\,].p : \Pi.[\,].p) \\
\wedge\ (& (\forall p, q, l : p \in \mathsf{PList.X}.n \ \wedge \ q \in \mathsf{PList.X}.m \ \wedge \ l \in \mathsf{PosList} : \\
& \quad (\mathit{valid}.\Pi.l.p \ \Rightarrow \ \Pi.l.p) \ \Rightarrow \ (\mathit{valid}.\Pi.(x \triangleright l).q \ \Rightarrow \ \Pi.(x \triangleright l).q) \ ) \\
& \vee\ (\forall p, q, l : p \in \mathsf{PList.X}.n \ \wedge \ q \in \mathsf{PList.X}.m \ \wedge \ l \in \mathsf{PosList} : \\
& \quad (\mathit{valid}.\Pi.l.p \ \Rightarrow \ \Pi.l.p) \ \Rightarrow \ (\mathit{valid}.\Pi.(l \triangleleft x).q \ \Rightarrow \ \Pi.(l \triangleleft x).q) \ ) \ ) \\
\Rightarrow & (\forall p, l : p \in \mathsf{PList.X}.n \ \wedge \ l \in \mathsf{PosList} : \mathit{valid}.\Pi.l.p \ \Rightarrow \ \Pi.l.p)
\end{array}$$

We do not give explicit formulations of *valid* in proofs of properties, but we do state relevant consequences of *valid*. All formulas we write in proofs satisfy *valid*, given the assumptions made in the context.

4.2.2 Permutation Functions in PLists

We continue by defining four permutation functions over PLists. We will only use one of them (*inv*) in our treatment of the permutation networks. The others are included since they illustrate how PLists can be used to reason algebraically about mixed-radix representations.

The permutation function $\mathit{inv} : \mathsf{PosList} \times \mathsf{PList}.X.n \longrightarrow \mathsf{PList}.X.n$ generalizes the PowerList function *inv* to PLists. Operationally, *inv* maps an element of a PList whose position can be written as a string of digits in a mixed-radix notation to a position that can be written as the reverse of the string.

$$\mathit{defined}.\mathit{inv}.l.p \ \equiv \ \mathit{prod}.l = \mathit{length}.p \tag{4.29}$$

$$\mathit{inv}.[\,].\langle a \rangle \ = \ \langle a \rangle \tag{4.30}$$

$$\mathit{inv}.(x \triangleright l).[|\, i : i \in \overline{x} : p.i] \ = \ [\bowtie i : i \in \overline{x} : \mathit{inv}.l.(p.i)] \tag{4.31}$$

Two interesting properties of *inv* are:

$$inv.(l \triangleleft x).[\bowtie i : i \in \overline{x} : p.i] \;=\; [|\, i : i \in \overline{x} : inv.l.(p.i)] \qquad (4.32)$$

$$inv.l.(inv.(linrev.l).p) \;=\; p \qquad (4.33)$$

Proof of (4.32), base case

$inv.[x].[\bowtie i : i \in \overline{x} : \langle a.i\rangle]$

$=$ { Axiom (4.8) }

$inv.[x].[|\, i : i \in \overline{x} : \langle a.i\rangle]$

$=$ { *inv* (4.31) }

$[\bowtie i : i \in \overline{x} : inv.[\,].\langle a.i\rangle]$

$=$ { *inv* (4.30) }

$[\bowtie i : i \in \overline{x} : \langle a.i\rangle]$

$=$ { Axiom (4.8) }

$[|\, i : i \in \overline{x} : \langle a.i\rangle]$

$=$ { *inv* (4.30) }

$[|\, i : i \in \overline{x} : inv.[\,].\langle a.i\rangle]$

Inductive step:

$inv.((y \triangleright l) \triangleleft x).[\bowtie i : i \in \overline{x} : [|\, j : j \in \overline{y} : p.i.j]]$

$=$ { Axiom (4.9) }

$inv.((y \triangleright l) \triangleleft x).[|\, j : j \in \overline{y} : [\bowtie i : i \in \overline{x} : p.i.j]]$

$=$ { *inv* (4.31) }

$[\bowtie j : j \in \overline{y} : inv.(l \triangleleft x).[\bowtie i : i \in \overline{x} : p.i.j]]$

$=$ { induction hypothesis (4.32) }

$[\bowtie j : j \in \overline{y} : [|\, i : i \in \overline{x} : inv.l.(p.i.j)]]$

$=$ { Axiom (4.9) }

$[|\, i : i \in \overline{x} : [\bowtie j : j \in \overline{y} : inv.l.(p.i.j)]]$

$=$ { *inv* (4.31) }

$[|\, i : i \in \overline{x} : inv.(y \triangleright l).[\bowtie j : j \in \overline{y} : p.i.j]]$

End of Proof

Proof of (4.33), base case omitted. Inductive step:

$$\begin{aligned}
& inv.(x \triangleright l).(inv.(linrev.(x \triangleright l)).[\bowtie i : i \in \overline{x} : p.i]) \\
= & \quad \{ \ \ linrev \ (4.21) \ \ \} \\
& inv.(x \triangleright l).(inv.(linrev.l \triangleleft x).[\bowtie i : i \in \overline{x} : p.i]) \\
= & \quad \{ \ \ \text{result above } (4.32) \ \ \} \\
& inv.(x \triangleright l).[| \, i : i \in \overline{x} : inv.(linrev.l).(p.i)] \\
= & \quad \{ \ \ inv \ (4.31) \ \} \\
& [\bowtie i : i \in \overline{x} : inv.l.(inv.(linrev.l).(p.i))] \\
= & \quad \{ \ \ \text{induction } (4.33) \ \} \\
& [\bowtie i : i \in \overline{x} : p.i]
\end{aligned}$$

End of Proof

Note that we omitted any reference to the definedness of expressions in the proof above, since this property is simple to check. In Section 4.3.4 we will see proofs where these proof obligations are non-trivial, and hence are not omitted.

Next, we define the function $rev : \mathsf{PList.X}.n \longrightarrow \mathsf{PList.X}.n$ that reverses the order of the elements of a PList:

$$defined.rev.l.p \quad \equiv \quad prod.l = length.p \tag{4.34}$$

$$rev.[\ ].\langle a \rangle \quad = \quad \langle a \rangle \tag{4.35}$$

$$rev.(y \triangleright l).[| \, i : i \in \overline{y} : p.i] \quad = \quad [| \, i : i \in \overline{y} : rev.l.(p.(y-(i+1)))] \tag{4.36}$$

As in the case of PowerList; (4.36) can be replaced by:

$$rev.(y \triangleright l).[\bowtie i : i \in \overline{y} : p.i] = [\bowtie i : i \in \overline{y} : rev.l.(p.(y-(i+1)))] \tag{4.37}$$

There is an interesting relationship between *rev* and *inv*:

$$rev.l.(inv.(linrev.l).p) = inv.(linrev.l).(rev.l.p) \tag{4.38}$$

Proof By induction over the length of l. Base case:

$$rev.[\,].(inv.(linrev.[\,]).\langle a\rangle) \;=\; inv.(linrev.[\,]).(rev.[\,].\langle a\rangle)$$

$$\equiv \quad \{\ linrev\ (4.20)\ \}$$

$$rev.[\,].(inv.[\,].\langle a\rangle) \;=\; inv.[\,].(rev.[\,].\langle a\rangle)$$

$$\equiv \quad \{\ inv\ (4.30);\ rev\ (4.35)\ \}$$

$$rev.[\,].\langle a\rangle \;=\; inv.[\,].\langle a\rangle$$

$$\equiv \quad \{\ rev\ (4.35);\ inv\ (4.30)\ \}$$

$$\langle a\rangle \;=\; \langle a\rangle$$

Inductive step:

$$rev.(y \triangleright l).(inv.(linrev.(y \triangleright l).[\bowtie i : i \in \overline{y} : p.i]))$$

$$= \quad \{\ linrev\ (4.21)\ \}$$

$$rev.(y \triangleright l).(inv.(linrev.l \triangleleft y).[\bowtie i : i \in \overline{y} : p.i])$$

$$= \quad \{\ inv\ (4.32)\ \}$$

$$rev.(y \triangleright l).[|\, i : i \in \overline{y} : inv.(linrev.l).(p.i)]$$

$$= \quad \{\ rev\ (4.36)\ \}$$

$$[|\, i : i \in \overline{y} : rev.l.(inv.(linrev.l).(p.(y-(i+1))))]$$

$$= \quad \{\ \text{induction}\ (4.38)\ \}$$

$$[|\, i : i \in \overline{y} : inv.(linrev.l).(rev.l.(p.(y-(i+1))))]$$

$$= \quad \{\ inv\ (4.32)\ \}$$

$$inv.(linrev.l \triangleleft y).[\bowtie i : i \in \overline{y} : (rev.l.(p.(y-(i+1))))]$$

$$= \quad \{\ rev\ (4.37)\ \}$$

$$inv.(linrev.l \triangleleft y).rev.(y \triangleright l).[\bowtie i : i \in \overline{y} : p.i]$$

$$= \quad \{\ linrev\ (4.21)\ \}$$

$$inv.(linrev.(y \triangleright l)).rev.(y \triangleright l).[\bowtie i : i \in \overline{y} : p.i]$$

End of Proof

Next, we define two permutation functions that are inverses of one another: *rir* and *ril*. These functions are are similar to *inv* since they permute the elements of a PList according to their mixed-radix representation as specified by the given list of arities. The function $rir : \mathsf{PosList} \times \mathsf{PList}.X.n \longrightarrow \mathsf{PList}.X.n$ is defined by:

$$defined.rir.l.p \quad \equiv \quad prod.l = length.p \tag{4.39}$$

$$rir.[\,].\langle a\rangle \quad = \quad \langle a\rangle \tag{4.40}$$

$$rir.(l \triangleleft y).[\bowtie i : i \in \overline{y} : p.i] \quad = \quad [| \, i : i \in \overline{y} : p.i] \tag{4.41}$$

Operationally, $rir.l.p$ permutes an element of p whose position can be written in a mix-radix representation where the radices are specified by l, to a position that is obtained by rotating the representation one position to the right. The inverse to rir is $ril : \mathsf{PList}.X.n \longrightarrow \mathsf{PList}.X.n$, which is specified by:

$$defined.ril.l.p \quad \equiv \quad prod.l = length.p \tag{4.42}$$

$$ril.[\,].\langle a\rangle \quad = \quad \langle a\rangle \tag{4.43}$$

$$ril.(y \triangleright l).[| \, i : i \in \overline{y} : p.i] \quad = \quad [\bowtie i : i \in \overline{y} : p.i] \tag{4.44}$$

Operationally, ril is similar to rir except that it rotates the representation to the left.

The fact that ril and rir are inverses is simple to prove:

$$ril.l.(rir.(linrev.l).p) = p \tag{4.45}$$

Proof Over the structure of l. Case $[\,]$

$ril.[\,].(rir.(linrev.[\,]).\langle a\rangle)$

$=$ { $linrev$ (4.20) }

$ril.[\,].(rir.[\,].\langle a\rangle)$

$=$ { rir (4.40) }

$ril.[\,].\langle a\rangle$

$=$ { ril (4.43) }

$\langle a\rangle$

Case $y \triangleright l$:

$ril.(y \triangleright l).(rir.(linrev.(y \triangleright l).[\bowtie i : i \in \overline{y} : p.i]))$

$= \quad \{ \ linrev \ (4.21) \ \}$

$ril.(y \triangleright l).(rir.(linrev.l \triangleleft y).[\bowtie i : i \in \overline{y} : p.i])$

$= \quad \{ \ rir \ (4.41) \ \}$

$ril.(y \triangleright l).[| \, i : i \in \overline{y} : p.i]$

$= \quad \{ \ ril \ (4.44) \ \}$

$[\bowtie i : i \in \overline{y} : p.i]$

End of Proof

It is possible to generalize the PowerList functions rr and rl and the PowerList operators $\rightarrow$ and $\leftarrow$ to PLists. We omit their definitions since they require manipulations of the indices of the generalized notations.

4.3 Interconnection Networks

In this section we describe four interconnection networks. These networks can be configured to realize the routing from input nodes to output nodes specified by any permutation. First, we describe binary networks where the nodes are 2×2 switches (i.e., they have *arity* 2) as PowerList functions, and prove that the four networks are isomorphic. The PowerList functions are then generalized to PList functions describing generalized networks, where the nodes in each column have the same, positive arity, but nodes in different columns may have different arities. Based on these specifications we prove that the generalized networks are isomorphic, by lifting the PowerList proofs to corresponding PList proofs.

4.3.1 Binary Interconnection Networks

An interconnection network of size 2^n, where $n \in \mathsf{Pos}$, has $n + 1$ stages numbered 0 through n (the stages will appear in increasing order from left to right in the figures). Each stage has 2^n nodes; nodes in stage 0 are *initial* nodes and those in stage n are *final* nodes. Each non-initial node has two input ports, known as *top* and *bottom*. Each non-final node has two output ports known as *top* and *bottom*. A node has the property that the values on the input ports either pass through to the same output ports, or they are exchanged[2]. This node behavior can be controlled by an external routing protocol. The output ports of nodes in stage i are connected to the input ports of nodes in stage $i + 1$, $0 \leq i < n$. Examples of interconnection networks are the *butterfly*, *iterative* and *recursive* networks[3]. Each of these, e.g., the butterfly network, actually denotes a family of networks where each member of the family has a number of input ports equal to a different value of 2^n.

We describe the structure of a family of networks by a PowerList function. These functional descriptions can be used to prove that the different network families are isomorphic.

Examples of Networks

First, we consider an interconnection network that we call the *iterative network* because the connections are identical from stage to stage. The network for $2^n = 4$ is shown in Figure 4.1. In stage 1, the top lines of the stage come in order from the upper half of the previous stage and all the bottom lines come from the lower half in order. The connections for the remaining stages are the same as in the first stage.

Next, we consider a *recursive network*, an interconnection network created in a recursive fashion. For $2^n = 1$, the network is a single node. For $2^n = 2$, the

[2]A node with this property is often called a *switch* in the literature.

[3]We use the names iterative and recursive, since there does not seem to be a consistent usage of the names Benes, Clos, Waxman, Omega and Baseline for these networks in the literature.

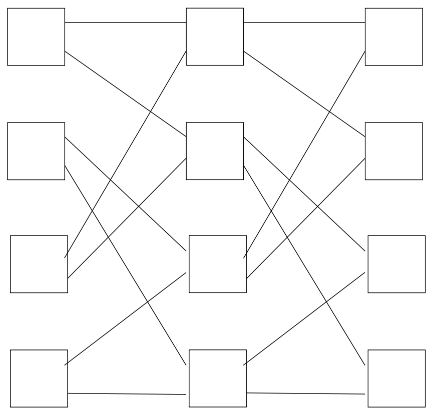

Figure 4.1: An Iteratively Constructed Interconnection Network, $2^n = 4$

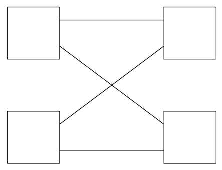

Figure 4.2: The Recursive Network for $2^n = 2$

network is a butterfly network with 2 stages, as shown in Figure 4.2. We show the general construction scheme in Figure 4.3, for $2^n = 4$. In stage 1, all the lines in the top half are the top output lines of the previous stage and all the lines in the bottom half come from the bottom lines in the previous stage, in order. Next, two copies of the same network of the next smaller size, for $2^n = 2$, are appended to the upper and lower halves.

A *butterfly network* of size n, where $2^n = 2$ is shown in Figure 4.2, the butterfly network of size $2^n = 4$ is shown in Figure 4.4 and the butterfly network of size $2^n = 8$ is shown in Figure 4.5. The interconnection structure can be described as follows. The initial nodes in the upper half have their top lines connected to the top lines in the upper half of the next stage and their bottom lines connected to

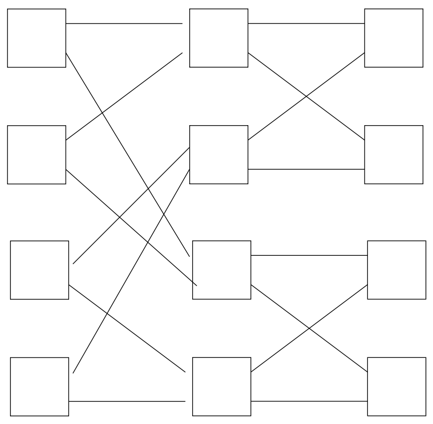

Figure 4.3: The Recursive Network, $2^n = 4$

the top lines of the bottom half of the next stage, in order. The connections for the bottom half of the initial nodes are analogous.

The mirror image of the butterfly network for $2^n = 8$ is shown in Figure 4.6.

4.3.2 Describing the Binary Networks in PowerList

We adopt the following scheme to describe the structure of a network. Name each node in stage 0 by a distinct character from some alphabet. For a node named b, name its top outgoing edge $b \diamond 0$ and its bottom outgoing edge $b \diamond 1$. A node whose top (respectively, bottom) incoming edge is named b (respectively, c), is assigned $b +\!\!+\, c$ as its name. Thus, in Figure 4.7, given that the nodes in stage 0 are named a, b, c, d from top to bottom, the other nodes are named as shown. It is clear that given the PowerList of names for the nodes in stage 0, all the node and edge names are determined. Further, given the PowerList of node names at the last stage, it is possible to reconstruct the names assigned to all the nodes and edges and their interconnections.

We describe the structure of a network by a function whose argument is a

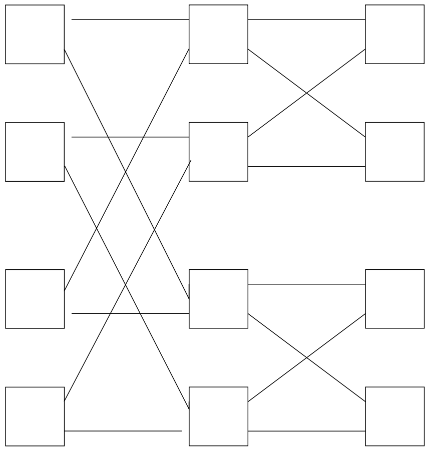

Figure 4.4: Butterfly Network for $2^n = 4$

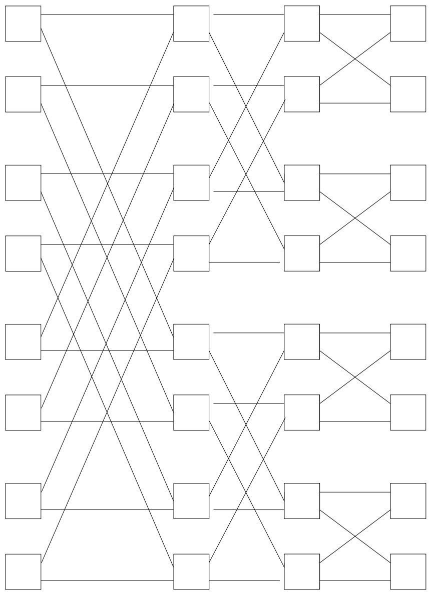

Figure 4.5: Butterfly Network for $2^n = 8$

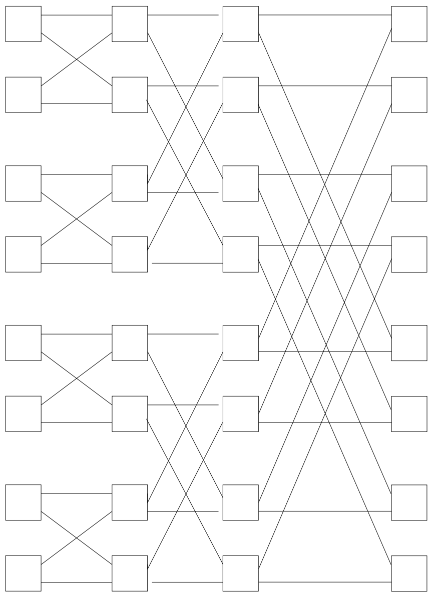

Figure 4.6: Mirror Image of the Butterfly Network

PowerList of names, to be assigned in sequence to the nodes in stage 0, and whose result is a PowerList of the names assigned to the nodes in the last stage of the network. The networks are described by the functions, *iter*, *rec*, *but* and *tub*, with the following types:

$$iter : \mathsf{Nat} \times \mathsf{PowerList.String}.n \longrightarrow \mathsf{PowerList.String}.n$$

$$rec : \mathsf{PowerList.String}.n \longrightarrow \mathsf{PowerList.String}.n$$

$$but : \mathsf{PowerList.String}.n \longrightarrow \mathsf{PowerList.String}.n$$

$$tub : \mathsf{PowerList.String}.n \longrightarrow \mathsf{PowerList.String}.n$$

Note that the iterative nature of *iter* is described by its first argument, which denotes the number of stages in the network. The term $iter.(loglen.p).p$ describes the labels in the last stage of the iterative network, with p describing the labels in stage 0. The functions are defined as follows:

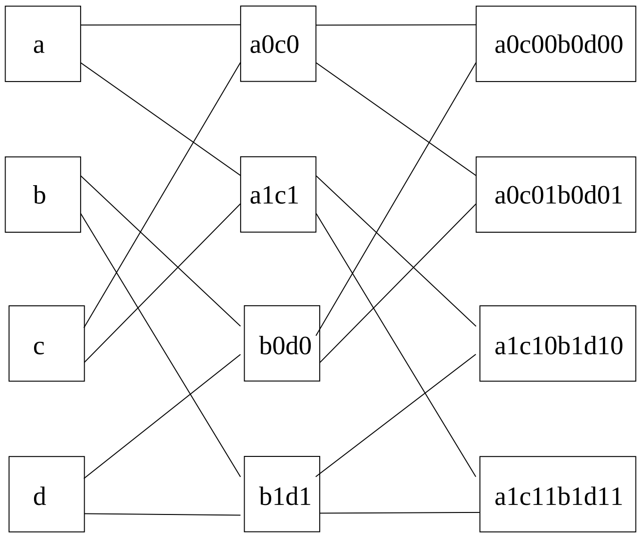

Figure 4.7: Naming the Nodes in a Network

Iterative Network

$$iter.0.p \quad = \quad p \tag{4.46}$$

$$iter.(k+1).(p \mid q) \quad = \quad iter.k.(p{\diamond}0 +\!\!+\, q{\diamond}0 \;\bowtie\; p{\diamond}1 +\!\!+\, q{\diamond}1) \tag{4.47}$$

Recursive Network

$$rec.\langle a \rangle \quad = \quad \langle a \rangle \tag{4.48}$$

$$rec.(p \bowtie q) \quad = \quad rec.(p{\diamond}0 +\!\!+\, q{\diamond}0) \mid rec.(p{\diamond}1 +\!\!+\, q{\diamond}1) \tag{4.49}$$

Butterfly Network

$$but.\langle a \rangle \quad = \quad \langle a \rangle \tag{4.50}$$

$$but.(p \mid q) \quad = \quad but.(p{\diamond}0 +\!\!+\, q{\diamond}0) \mid but.(p{\diamond}1 +\!\!+\, q{\diamond}1) \tag{4.51}$$

Mirror Butterfly Network

$$tub.\langle a \rangle \quad = \quad \langle a \rangle \tag{4.52}$$

$$tub.(p \bowtie q) \quad = \quad tub.(p{\diamond}0 +\!\!+\, q{\diamond}0) \;\bowtie\; tub.(p{\diamond}1 +\!\!+\, q{\diamond}1) \tag{4.53}$$

The PowerList representation is useful when we wish to concatenate different networks; functional composition corresponds to network concatenation. The labeling in Figure 4.7 is obtained by

$iter.2.\langle a \quad b \quad c \quad d\rangle$

$= \quad \{ \; iter \; (4.47) \; \}$

$iter.1.\langle \mathrm{a0c0} \quad \mathrm{a1c1} \quad \mathrm{b0d0} \quad \mathrm{b1d1}\rangle$

$= \quad \{ \; iter \; (4.47) \; \}$

$iter.0.\langle \mathrm{a0c00b0d00} \quad \mathrm{a0c01b0d01} \quad \mathrm{a1c10b1d10} \quad \mathrm{a1c11b1d11}\rangle$

$= \quad \{ \; iter \; (4.46) \; \}$

$\langle \mathrm{a0c00b0d00} \quad \mathrm{a0c01b0d01} \quad \mathrm{a1c10b1d10} \quad \mathrm{a1c11b1d11}\rangle$

4.3.3 Equivalence Between the Binary Networks

We first prove that the recursive network is isomorphic to the butterfly:

$$rec \circ inv = but \tag{4.54}$$

Proof of (4.54). Base case omitted. Inductive step:

$rec.(inv.(p \mid q))$

$= \quad \{ \; inv \; (4.31)$ as defined in Chapter 2 $\}$

$rec.(inv.p \bowtie inv.q)$

$= \quad \{ \; rec \; (4.49) \; \}$

$rec.(inv.p{\diamond}0 +\!\!+ inv.q{\diamond}0) \mid rec.(inv.p{\diamond}1 +\!\!+ inv.q{\diamond}1)$

$= \quad \{ \; inv$ is a permutation function $(4.17) \; \}$

$rec.(inv.(p{\diamond}0) +\!\!+ inv.(q{\diamond}0)) \mid rec.(inv.(p{\diamond}1) +\!\!+ inv.(q{\diamond}1))$

$= \quad \{ \; inv$ is a permutation function $(4.17) \; \}$

$rec.(inv.(p{\diamond}0 +\!\!+ q{\diamond}0)) \mid rec.(inv.(p{\diamond}1 +\!\!+ q{\diamond}1))$

$= \quad \{$ induction $(4.54) \; \}$

$but.(p{\diamond}0 +\!\!+ q{\diamond}0) \mid but.(p{\diamond}1 +\!\!+ q{\diamond}1)$

$= \quad \{ \; but \; (4.51) \; \}$

$but.(p \mid q)$

End of Proof

Next, we prove that the recursive network is isomorphic to the mirror butterfly:

$$rec = inv \circ tub \tag{4.55}$$

Proof of (4.55). Base case omitted. Inductive step:

$inv.(tub.(p \bowtie q))$

$= \quad \{ \quad tub \ (4.53) \quad \}$

$inv.(tub.(p{\diamond}0 + q{\diamond}0) \ \bowtie \ tub.(p{\diamond}1 + q{\diamond}1))$

$= \quad \{ \quad inv$ (4.31) as defined in Chapter 2 $\}$

$inv.(tub.(p{\diamond}0 + q{\diamond}0)) \ \mid \ inv.(tub.(p{\diamond}1 + q{\diamond}1))$

$= \quad \{$ induction (4.55) $\}$

$rec.(p{\diamond}0 + q{\diamond}0) \ \mid \ rec.(p{\diamond}1 + q{\diamond}1)$

$= \quad \{ \quad rec \ (4.49) \ \}$

$rec.(p \bowtie q)$

End of Proof

It is an immediate consequence of (4.54) and (4.55) that

$$but \circ inv = inv \circ tub \tag{4.56}$$

To enable us to prove that the iterative network is isomorphic to the other networks we need the following lemma that gives a recursive structure definition of the labels in an iterative network:

Lemma 13

$$k \leq loglen.p \ \Rightarrow \ \{\, iter.k.(p \bowtie q) \,\} = \{\, iter.k.p \,\} \bigcup \{\, iter.k.q \,\} \tag{4.57}$$

Proof of (4.57) by induction on k. Base case $k = 0$

$\{\, \mathit{iter}.0.(p \bowtie q) \,\} = \{\, \mathit{iter}.0.p \,\} \bigcup \{\, \mathit{iter}.0.q \,\}$

$\equiv$ { *iter* (4.46) }

$\{ p \bowtie q \} = \{ p \} \bigcup \{ q \}$

$\equiv$ { $\{\ \}$ (4.23) }

true

inductive step: (we have (4.60) below)

$\{\, \mathit{iter}.(k+1).((p \mid q) \ \bowtie\ (u \mid v)) \,\}$

$=$ { Axiom (3.20) }

$\{\, \mathit{iter}.(k+1).((p \bowtie u) \ \mid\ (q \bowtie v)) \,\}$

$=$ { *iter* (4.47) }

$\{\, \mathit{iter}.k.(((p \bowtie u){\diamond}0 +\!\!+ (q \bowtie v){\diamond}0) \ \bowtie\ ((p \bowtie u){\diamond}1 +\!\!+ (q \bowtie v){\diamond}1)) \,\}$

$=$ { $\diamond$ is a scalar operator (2.15) }

$\{\, \mathit{iter}.k.((p{\diamond}0 \bowtie u{\diamond}0) +\!\!+ (q{\diamond}0 \bowtie v{\diamond}0) \ \bowtie\ (p{\diamond}1 \bowtie u{\diamond}1) +\!\!+ (q{\diamond}1 \bowtie v{\diamond}1)) \,\}$

$=$ { induction (4.57) see (4.58) below }

$\{\, \mathit{iter}.k.((p{\diamond}0 \bowtie u{\diamond}0) +\!\!+ (q{\diamond}0 \bowtie v{\diamond}0)) \,\}$

$\bigcup \{\, \mathit{iter}.k.((p{\diamond}1 \bowtie u{\diamond}1) +\!\!+ (q{\diamond}1 \bowtie v{\diamond}1)) \,\}$

$=$ { $+\!\!+$ is scalar (2.15) }

$\{\, \mathit{iter}.k.((p{\diamond}0 +\!\!+ q{\diamond}0) \ \bowtie\ (u{\diamond}0 +\!\!+ v{\diamond}0)) \,\}$

$\bigcup \{\, \mathit{iter}.k.((p{\diamond}1 +\!\!+ q{\diamond}1) \ \bowtie\ (u{\diamond}1 +\!\!+ v{\diamond}1)) \,\}$

$=$ { induction (4.57) see (4.59) below }

$\{\, \mathit{iter}.k.(p{\diamond}0 +\!\!+ q{\diamond}0) \,\} \bigcup \{\, \mathit{iter}.k.(u{\diamond}0 +\!\!+ v{\diamond}0) \,\}$

$\bigcup \{\, \mathit{iter}.k.(p{\diamond}1 +\!\!+ q{\diamond}1) \,\} \bigcup \{\, \mathit{iter}.k.(u{\diamond}1 +\!\!+ v{\diamond}1) \,\}$

$=$ { set union is symmetric }

$\{\, \mathit{iter}.k.(p{\diamond}0 +\!\!+ q{\diamond}0) \,\} \bigcup \{\, \mathit{iter}.k.(p{\diamond}1 +\!\!+ q{\diamond}1) \,\}$

$\bigcup \{\, \mathit{iter}.k.(u{\diamond}0 +\!\!+ v{\diamond}0) \,\} \bigcup \{\, \mathit{iter}.k.(u{\diamond}1 +\!\!+ v{\diamond}1) \,\}$

$=$ { induction (4.57) see (4.59) below }

$\{\, iter.k.((p{\diamond}0 +\!\!+\, q{\diamond}0) \;\bowtie\; (p{\diamond}1 +\!\!+\, q{\diamond}1))\,\}$

$\bigcup \{\, iter.k.((u{\diamond}0 +\!\!+\, v{\diamond}0) \;\bowtie\; (u{\diamond}1 +\!\!+\, v{\diamond}1))\,\}$

$=\quad \{\ \ iter\ (4.47)\ \}$

$\{\, iter.(k+1).(p \mid q)\,\} \bigcup \{\, iter.(k+1).(u \mid v)\,\}$

End of Proof

In the proof above we need to establish that the inductive hypothesis was applied correctly:

$$k \le loglen.((p{\diamond}0 \bowtie u{\diamond}0) +\!\!+ (q{\diamond}0 \bowtie v{\diamond}0)) \tag{4.58}$$

$$k \le loglen.(p{\diamond}b +\!\!+ q{\diamond}b) \quad \text{where } b = 0 \;\vee\; b = 1 \tag{4.59}$$

It is simple to show that these inequalities follow from the assumption

$$(k+1) \le loglen.(p \mid q) \tag{4.60}$$

We can now prove the isomorphism between the iterative and the butterfly networks:

$$\{\, iter.k.p\,\} = \{\, but.p\,\} \quad \text{where } k = loglen.p \tag{4.61}$$

Proof of (4.61). Base case omitted. Inductive step: assume $k+1 = loglen.(p \mid q)$

$\{\, iter.(k+1).(p \mid q)\,\}$

$=\quad \{\ \ iter\ (4.47)\ \}$

$\{\, iter.k.(p{\diamond}0 +\!\!+\, q{\diamond}0 \;\bowtie\; p{\diamond}1 +\!\!+\, q{\diamond}1)\,\}$

$=\quad \{$ Lemma 13 (4.57) is applicable, see (4.62) below $\}$

$\{\, iter.k.(p{\diamond}0 +\!\!+\, q{\diamond}0)\,\} \bigcup \{\, iter.k.(p{\diamond}1 +\!\!+\, q{\diamond}1)\,\}$

$=\quad \{$ induction see (4.63) below $\}$

$\{\, but.(p{\diamond}0 +\!\!+\, q{\diamond}0)\,\} \bigcup \{\, but.(p{\diamond}1 +\!\!+\, q{\diamond}1)\,\}$

$=\quad \{\ \ \{\ \}\ (4.24)\ \ \}$

$\{\, but.(p{\diamond}0 +\!\!+\, q{\diamond}0) \;\mid\; but.(p{\diamond}1 +\!\!+\, q{\diamond}1)\,\}$

$=\quad \{\ \ but\ (4.51)\ \}$

$\{\, but.(p \mid q)\,\}$

End of Proof

In the proof above the following properties were left unproven:

$$k = \mathit{loglen}.p \tag{4.62}$$

$$k \leq \mathit{loglen}.(p\diamond 0 \mathbin{+\!\!+} q\diamond 0) \tag{4.63}$$

both follow from the assumption: $k+1 = \mathit{loglen}.(p \mid q)$.

In summary we have proven that the iterative, recursive, butterfly and mirror-butterfly are isomorphic networks:

Theorem 1

$$\begin{aligned} \{\, \mathit{rec}.p \,\} &= \{\, \mathit{tub}.p \,\} \\ \{\, \mathit{but}.p \,\} &= \{\, \mathit{rec}.(\mathit{inv}.p) \,\} \\ \{\, \mathit{but}.p \,\} &= \{\, \mathit{iter}.k.p \,\} \ \text{ where } k = \mathit{loglen}.p \end{aligned}$$

Theorem 1 follows from (4.54), (4.56), (4.61) and (4.25).

4.3.4 Generalized Networks

The generalized networks consists of stages, where a stage is constructed using nodes of the same arity. A node of arity m has the property that it can be configured to realize any permutation of the values on its m input wires to its m output wires. The networks may utilize different arities in different stages. It is the pattern used in connecting the output wires from one stage of a network to the input wires of the next stage that define a particular network topology.

4.3.5 Describing the Generalized Networks in PLists

In this section we describe the interconnection networks as PList functions of two arguments. The first argument is a non-empty list (e.g., $y \triangleright l$) that describes the arity

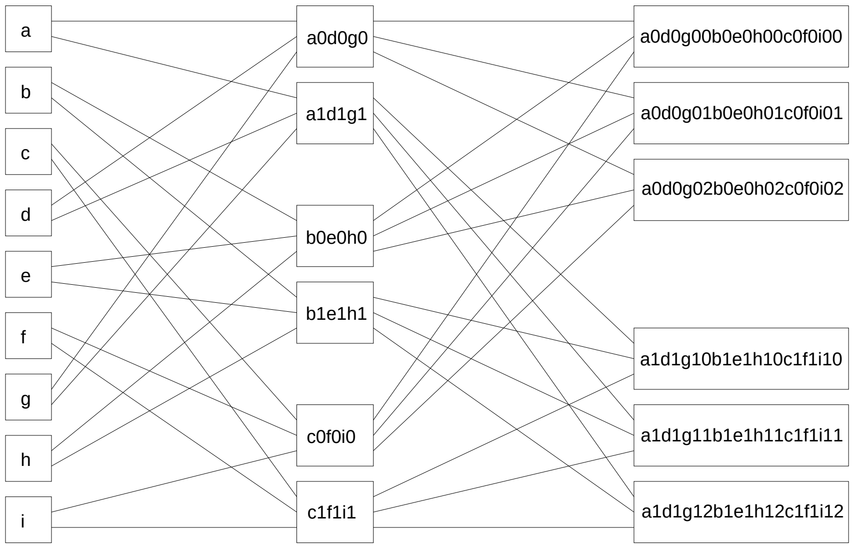

Figure 4.8: Iterative Network with the Arities 2, 3 and 3

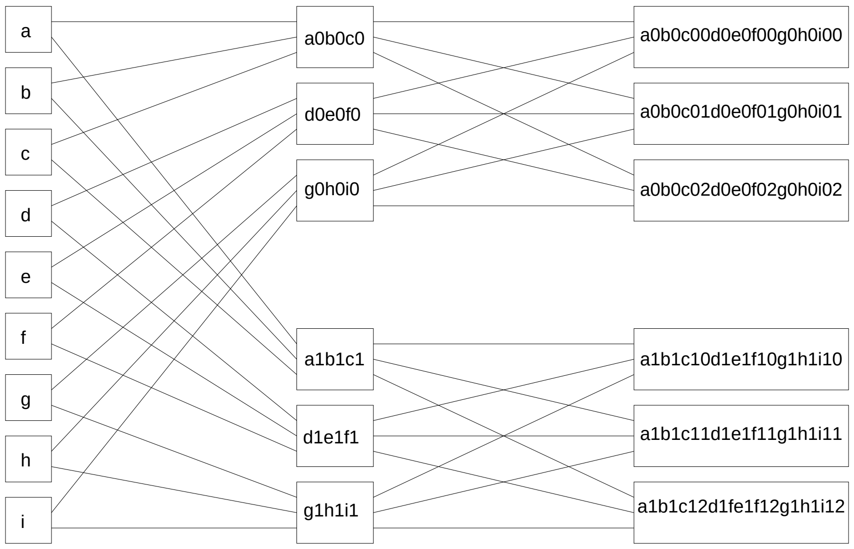

Figure 4.9: Recursive Network with the Arities 2, 3 and 3

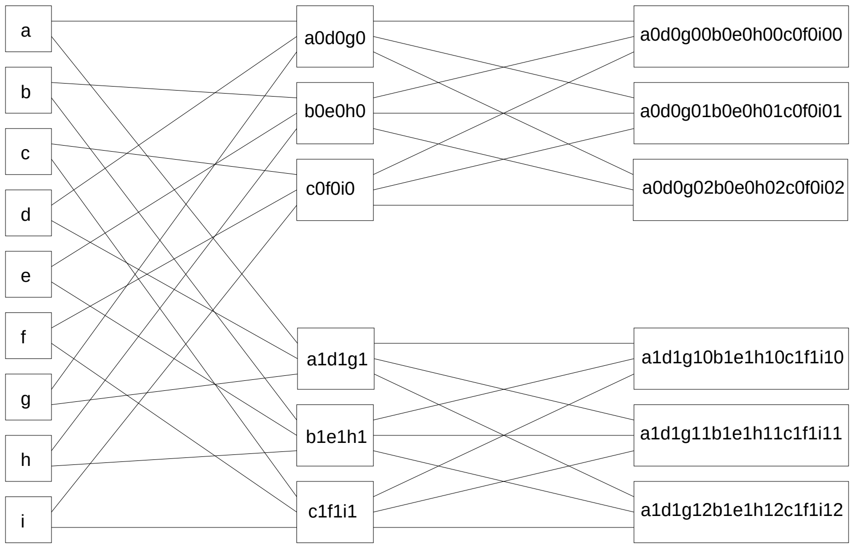

Figure 4.10: Butterfly Network with the Arities 2, 3 and 3

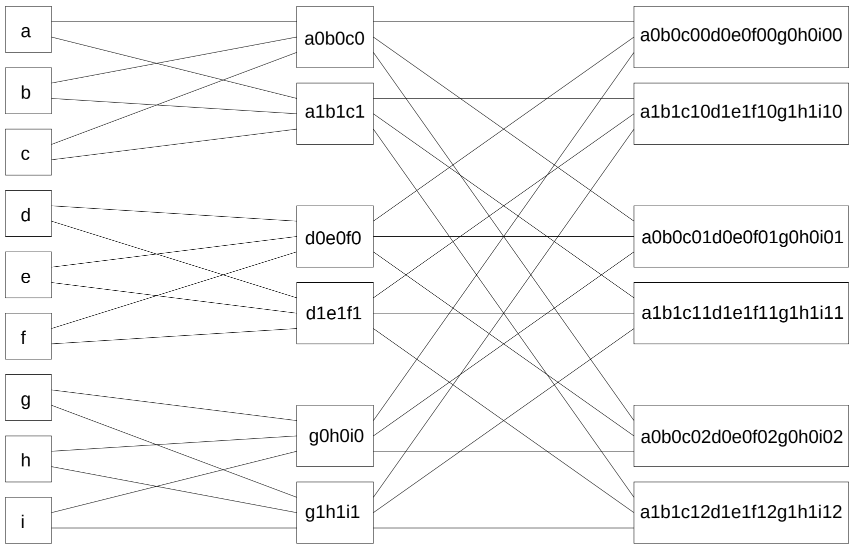

Figure 4.11: Mirror Butterfly Network with the Arities 2, 3 and 3

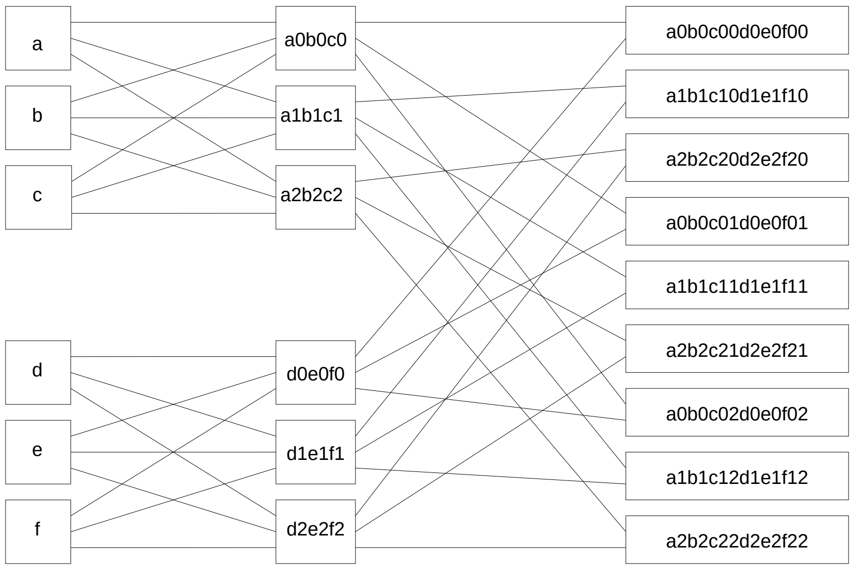

Figure 4.12: Mirror Butterfly Network with the Arities 3, 3 and 2

of the nodes in each column (e.g., $y \times y$ in the first row). The second argument is a PList of length *prod.l* consisting of distinct strings; this argument corresponds to a labeling of the boxes in the first column. The return value from such a function is a PList that corresponds to a labeling of the nodes in the last column. The scheme for labeling nodes is a generalization of the scheme used in the binary case. The main difference is that the outputs from a node of arity n are labeled 0 through $n-1$. Figures 4.8 through 4.12 illustrate the generalized networks and the labeling convention. The PList functions that define the generalized networks have the following types:

$$iter : \mathsf{PosList} \times \mathsf{PList.String}.n \longrightarrow \mathsf{PList.String}.n$$

$$rec : \mathsf{PosList} \times \mathsf{PList.String}.n \longrightarrow \mathsf{PList.String}.n$$

$$but : \mathsf{PosList} \times \mathsf{PList.String}.n \longrightarrow \mathsf{PList.String}.n$$

$$tub : \mathsf{PosList} \times \mathsf{PList.String}.n \longrightarrow \mathsf{PList.String}.n$$

These functions are defined by:

Iterative Network

$$defined.iter.(y \triangleright l).p \equiv length.p = prod.l \tag{4.64}$$

$$iter.[x].\langle a \rangle = \langle a \rangle \tag{4.65}$$

$$iter.(x \triangleright (y \triangleright l)).[| \, i : i \in \overline{y} : p.i] = iter.(y \triangleright l).[\bowtie j : j \in \overline{x} : [+\!\!+ \, i : i \in \overline{y} : (p.i) \diamond j]] \tag{4.66}$$

Recursive Network

$$defined.rec.(y \triangleright l).p \equiv length.p = prod.l \tag{4.67}$$

$$rec.[x].\langle a \rangle = \langle a \rangle \tag{4.68}$$

$$rec.(x \triangleright (y \triangleright l)).[\bowtie i : i \in \overline{y} : p.i] = [| \, j : j \in \overline{x} : rec.(y \triangleright l).[+\!\!+ \, i : i \in \overline{y} : (p.i) \diamond j]] \tag{4.69}$$

Butterfly Network

$$defined.but.(y \triangleright l).p \equiv length.p = prod.l \tag{4.70}$$

$$but.[x].\langle a \rangle = \langle a \rangle \tag{4.71}$$

$$but.(x \triangleright (y \triangleright l)).[| \, i : i \in \overline{y} : p.i] = [| \, j : j \in \overline{x} : but.(y \triangleright l).[+\!\!+ \, i : i \in \overline{y} : (p.i) \diamond j]] \tag{4.72}$$

Mirror Butterfly Network

$$defined.tub.(y \triangleright l).p \equiv length.p = prod.l \tag{4.73}$$

$$tub.[x].\langle a \rangle = \langle a \rangle \tag{4.74}$$

$$tub.(x \triangleright (y \triangleright l)).[\bowtie i : i \in \overline{y} : p.i] = [\bowtie j : j \in \overline{x} : tub.(y \triangleright l).[+\!\!+ \, i : i \in \overline{y} : (p.i) \diamond j]] \tag{4.75}$$

4.3.6 Equivalence Between the Generalized Networks

We start by proving the isomorphism between the butterfly and the recursive network. We prove this by showing that when the inputs to the recursive network are permuted (using *inv*), then the resulting network is the same as the butterfly network.

Lemma 14

$$rec.(x \triangleright l).(inv.l.p) = but.(x \triangleright l).p \qquad \text{where } length.p = prod.l \tag{4.76}$$

Proof By induction over the length of l. Base case:

$rec.[x].(inv.[\,].\langle a\rangle)$

$= \{\ inv\ (4.30)\ \}$

$rec.[x].\langle a\rangle$

$= \{\ rec\ (4.68)\ \}$

$\langle a\rangle$

$= \{\ but\ (4.71)\ \}$

$but.[x].\langle a\rangle$

Inductive step, $length.[|\, i : i \in \overline{y} : p.i] = prod.(y \triangleright l)$:

$rec.(x \triangleright (y \triangleright l)).(inv.(y \triangleright l).[|\, i : i \in \overline{y} : p.i])$

$= \{\ inv\ (4.31)\ \}$

$rec.(x \triangleright (y \triangleright l)).[\bowtie i : i \in \overline{y} : inv.l.(p.i)]$

$= \{\ rec\ (4.69)\ \}$

$[|\, j : j \in \overline{x} : rec.(y \triangleright l).[+\!\!+\, i : i \in \overline{y} : (inv.l.(p.i))\diamond j]]$

$= \{\ inv$ over scalar operators $(4.16, 4.17)\ \}$

$[|\, j : j \in \overline{x} : rec.(y \triangleright l).(inv.l.[+\!\!+\, i : i \in \overline{y} : (p.i)\diamond j])]$

$= \{$ induction $(4.76),\ length.(inv.l.[+\!\!+\, i : i \in \overline{y} : (p.i)\diamond j]) = length.p.0 = prod.l\ \}$

$[|\, j : j \in \overline{x} : but.(y \triangleright l).[+\!\!+\, i : i \in \overline{y} : (p.i)\diamond j]]$

$= \{\ but\ (4.72)\ \}$

$$but.(x \triangleright (y \triangleright l)).[| i : i \in \overline{y} : p.i]$$

End of Proof

Next, we prove the isomorphism between the mirror butterfly and the recursive network:

Lemma 15

$$tub.(x \triangleright l).p = inv.l.(rec.(x \triangleright l).p) \qquad \text{where } length.p = prod.l \tag{4.77}$$

Proof By induction over the length of l. Base case:

$inv.[\,].(rec.[x].\langle a \rangle)$

$= \{ \; rec \; (4.68) \; \}$

$inv.[\,].\langle a \rangle$

$= \{ \; inv \; (4.30) \; \}$

$\langle a \rangle$

$= \{ \; tub \; (4.74) \; \}$

$tub.[x].\langle a \rangle$

Inductive step $length.[| i : i \in \overline{y} : p.i] = prod.(y \triangleright l)$:

$inv.(x \triangleright l).(rec.(x \triangleright (y \triangleright l)).[\bowtie i : i \in \overline{y} : p.i])$

$= \{ \; rec \; (4.69) \; \}$

$inv.(x \triangleright l).[| j : j \in \overline{x} : rec.(y \triangleright l).[+\!\!+\, i : i \in \overline{y} : (p.i) \diamond j]]$

$= \{ \; inv \; (4.31) \; \}$

$[\bowtie j : j \in \overline{x} : inv.l.(rec.(y \triangleright l).[+\!\!+\, i : i \in \overline{y} : (p.i) \diamond j])]$

$= \{ \;$ induction (4.77), $length.([+\!\!+\, i : i \in \overline{y} : (p.i) \diamond j]) = length.p.0 = prod.l \; \}$

$[\bowtie j : j \in \overline{x} : tub.(y \triangleright l).[+\!\!+\, i : i \in \overline{y} : (p.i) \diamond j]]$

$= \{ \; tub \; (4.75) \; \}$

$tub.(x \triangleright (y \triangleright l)).[\bowtie i : i \in \overline{y} : p.i]$

End of Proof

Combining Lemmas 14 and 15 we can prove the equivalence of the butterfly and the mirror-butterfly networks:

Lemma 16

$$inv.(linrev.l).(tub.(x \triangleright l).p) = but.(x \triangleright l).(inv.(linrev.l).p) \quad \text{where } length.p = prod.l \tag{4.78}$$

Proof

$but.(x \triangleright l).(inv.(linrev.l).p)$

$=$ { Lemma 14 (4.76) }

$rec.(x \triangleright l).(inv.l.(inv.(linrev.l).p))$

$=$ { property of *inv* (4.33) }

$rec.(x \triangleright l).p$

$=$ { property of *inv* (4.33) }

$inv.(linrev.l).(inv.l.(rec.(x \triangleright l).p))$

$=$ { Lemma 15 (4.77) }

$inv.(linrev.l).(tub.(x \triangleright l).p)$

End of Proof

We proceed by proving the equivalence between the iterative and the butterfly networks. For each network we construct the set of labels at its final stage; the two networks are isomorphic when the two sets are equal. Before we can prove this result we need a lemma that establishes a property of the iterative network.

Lemma 17

$$\begin{array}{l}\{\, iter.(y \triangleright l).[\bowtie j : j \in \overline{x} : p.j]\,\} = (\cup j : 0 \le j < x : \{\, iter.(y \triangleright l).(p.j)\,\}) \\ \quad \text{where } length.(p.0) \ge prod.l\end{array} \tag{4.79}$$

Proof Base Case $length.\langle a.0\rangle \ge prod.[\,]$

$\boldsymbol{\{}\, iter.[y].[\bowtie j : j \in \overline{x} : \langle a.j \rangle]\, \boldsymbol{\}}$

$= \quad \{ \quad iter \ (4.65) \quad \}$

$\boldsymbol{\{}\, [\bowtie j : j \in \overline{x} : \langle a.j \rangle]\, \boldsymbol{\}}$

$= \quad \{ \quad \text{definition } \boldsymbol{\{} \ \boldsymbol{\}} \ (4.23) \ \}$

$(\cup j : 0 \leq j < x : \boldsymbol{\{}\, \langle a.j \rangle\, \boldsymbol{\}})$

$= \quad \{ \quad iter \ (4.65) \quad \}$

$(\cup j : 0 \leq j < x : \boldsymbol{\{}\, iter.[y].\langle a.j \rangle\, \boldsymbol{\}})$

Inductive step, where $y * length.(p.0.j) \geq prod.(y \triangleright l)$

$\boldsymbol{\{}\, iter.(z \triangleright (y \triangleright l)).[\bowtie j : j \in \overline{x} : [\| i : i \in \overline{y} : p.i.j]]\, \boldsymbol{\}}$

$= \quad \{ \quad \text{Axiom } (4.9) \ \}$

$\boldsymbol{\{}\, iter.(z \triangleright (y \triangleright l)).[\| i : i \in \overline{y} : [\bowtie j : j \in \overline{x} : p.i.j]]\, \boldsymbol{\}}$

$= \quad \{ \quad iter \ (4.66) \ \}$

$\boldsymbol{\{}\, iter.(y \triangleright l).[\bowtie k : k \in \overline{z} : [+\!\!+ i : i \in \overline{y} : [\bowtie j : j \in \overline{x} : p.i.j] \diamond k]]\, \boldsymbol{\}}$

$= \quad \{ \quad \text{commutativity of } \diamond k \text{ and } [\bowtie i : i \in \overline{y} : \ ] \ (4.12) \ \}$

$\boldsymbol{\{}\, iter.(y \triangleright l).[\bowtie k : k \in \overline{z} : [+\!\!+ i : i \in \overline{y} : [\bowtie j : j \in \overline{x} : (p.i.j) \diamond k]]]\, \boldsymbol{\}}$

$= \quad \{ \quad \text{induction, see } (4.81) \text{ below} \quad \}$

$(\cup k : 0 \leq k < z : \boldsymbol{\{}\, iter.(y \triangleright l).[+\!\!+ i : i \in \overline{y} : [\bowtie j : j \in \overline{x} : (p.i.j) \diamond k]]\, \boldsymbol{\}})$

$= \quad \{ \quad \text{commutativity } [+\!\!+ i : i \in \overline{y} : \ ] \text{ and } [\bowtie j : j \in \overline{x} : \ ] \ (4.15) \ \}$

$(\cup k : 0 \leq k < z : \boldsymbol{\{}\, iter.(y \triangleright l).[\bowtie j : j \in \overline{x} : [+\!\!+ i : i \in \overline{y} : (p.i.j) \diamond k]]\, \boldsymbol{\}})$

$= \quad \{ \quad \text{induction, see } (4.82) \text{ below } \}$

$(\cup k : 0 \leq k < z : (\cup j : 0 \leq j < x : \boldsymbol{\{}\, iter.(y \triangleright l).[+\!\!+ i : i \in \overline{y} : (p.i.j) \diamond k]\, \boldsymbol{\}}))$

$= \quad \{ \quad \text{set union commutes} \quad \}$

$(\cup j : 0 \leq j < x : (\cup k : 0 \leq k < z : \boldsymbol{\{}\, iter.(y \triangleright l).[+\!\!+ i : i \in \overline{y} : (p.i.j) \diamond k]\, \boldsymbol{\}}))$

$= \quad \{ \quad \text{induction see } (4.83) \text{ below} \quad \}$

$(\cup j : 0 \leq j < x : \boldsymbol{\{}\, iter.(y \triangleright l).[\bowtie k : k \in \overline{z} : [+\!\!+ i : i \in \overline{y} : (p.i.j) \diamond k]]\, \boldsymbol{\}})$

$= \quad \{ \quad iter \ (4.66) \ \}$

$(\cup j : 0 \leq j < x : \boldsymbol{\{}\, iter.(z \triangleright (y \triangleright l)).[\| i : i \in \overline{y} : p.i.j]\, \boldsymbol{\}})$

End of Proof

The inductive assumption in the proof above is equivalent to

$$\mathit{length}.(p.0.j) \geq \mathit{prod}.l \tag{4.80}$$

from which the following inequalities used in the proof above can be proven

$$\mathit{length}.[+\!\!+\, i : i \in \overline{y} : [\bowtie j : j \in \overline{x} : (p.i.j){\diamond}0]] \;\geq\; \mathit{prod}.l \tag{4.81}$$

$$\mathit{length}.[+\!\!+\, i : i \in \overline{y} : (p.i.0){\diamond}l] \;\geq\; \mathit{prod}.l \tag{4.82}$$

$$\mathit{length}.[+\!\!+\, i : i \in \overline{y} : (p.i.j){\diamond}0] \;\geq\; \mathit{prod}.l \tag{4.83}$$

We only prove (4.81), as the proofs of the others are similar

Proof

$\mathit{length}.[+\!\!+\, i : i \in \overline{y} : [\bowtie j : j \in \overline{x} : (p.i.j){\diamond}0]]$

$=$ { Property of $[+\!\!+\, i : i \in \overline{y} :\ ]$ }

$\mathit{length}.[\bowtie j : j \in \overline{x} : (p.i.j){\diamond}0]$

$=$ { Property of $[\bowtie j : j \in \overline{x} :\ ]$ and $\diamond$ }

$x * \mathit{length}.(p.0.0)$

$\geq$ { $x > 0$ }

$\mathit{length}.p.0.0$

$\geq$ { inductive hypothesis (4.80) }

$\mathit{prod}.l$

End of Proof

We are now ready to prove the equivalence between the iterative and the butterfly networks:

Lemma 18

$$\{\mathit{iter}.(x \triangleright l).p\} = \{\mathit{but}.(x \triangleright l).p\} \quad \text{where } \mathit{length}.p = \mathit{prod}.l \tag{4.84}$$

Proof Induction over the length of l. Base case:

$\{\, \mathit{iter}.[x].\langle a\rangle \,\}$

$=$ { iter (4.65) }

$\{\, \langle a\rangle \,\}$

$=$ { but (4.71) }

$\{\, \mathit{but}.[x].\langle a\rangle \,\}$

Inductive step, where $\mathit{length}.[|\, i : i \in \overline{y} : p.i] = \mathit{prod}.(y \triangleright l)$

$\{\, \mathit{iter}.(x \triangleright (y \triangleright l)).[|\, i : i \in \overline{y} : p.i] \,\}$

$=$ { iter (4.66) }

$\{\, \mathit{iter}.(y \triangleright l).[\bowtie j : j \in \overline{x} : [+\!\!+\, i : i \in \overline{y} : (p.i)\diamond j]] \,\}$

$=$ { Lemma 17, see 4.85 below }

$(\cup j : 0 \le j < x : \{\, \mathit{iter}.(y \triangleright l).[+\!\!+\, i : i \in \overline{y} : (p.i)\diamond j] \,\})$

$=$ { induction $\mathit{length}.(p.i)\diamond j = \mathit{prod}.(y \triangleright l)$ }

$(\cup j : 0 \le j < x : \{\, \mathit{but}.(y \triangleright l).[+\!\!+\, i : i \in \overline{y} : (p.i)\diamond j] \,\})$

$=$ { definition of $\{\ \}$ (4.24) }

$\{\, [|\, j : j \in \overline{x} : \mathit{but}.(y \triangleright l).[+\!\!+\, i : i \in \overline{y} : (p.i)\diamond j]] \,\}$

$=$ { but (4.72) }

$\{\, [|\, j : j \in \overline{x} : \mathit{but}.(x \triangleright (y \triangleright l)).[|\, i : i \in \overline{y} : p.i]] \,\}$

End of Proof

In the proof above we used

$$\mathit{length}.[+\!\!+\, i : i \in \overline{y} : (p.i)\diamond 0] \ge \mathit{prod}.l \tag{4.85}$$

which follows from $\mathit{length}.[|\, i : i \in \overline{y} : p.i] = \mathit{prod}.(y \triangleright l)$.

Theorem 2

$$\mathit{defined}.\mathit{but}.l.p \quad \equiv \quad \mathit{defined}.\mathit{iter}.l.p \tag{4.86}$$

$$\mathit{defined}.\mathit{but}.l.p \quad \equiv \quad \mathit{defined}.\mathit{rec}.l.p \tag{4.87}$$

$$defined.but.l.p \equiv defined.tub.l.p \tag{4.88}$$

$$\{but.l.p\} = \{iter.l.p\} \tag{4.89}$$

$$\{but.l.p\} = \{rec.l.(inv.l.p)\} \tag{4.90}$$

$$\{rec.l.p\} = \{tub.l.p\} \tag{4.91}$$

Equations (4.86), (4.87) and (4.88) follow by inspection, (4.89) follows from Lemma 18, (4.90) follows from Lemma 14 and (4.91) follows from Lemma 16.

4.4 Summary

Most of the permutation functions that we defined for PowerLists in Chapter 2 have the property that they correspond to simple manipulations on the binary representation of the position of elements of a PowerList. One of the advantages of the PowerList notation is that it provides a layer of abstraction that is higher than that of the indices of elements in a PowerList. The PList theory extends this correspondence to number systems in radix n for any $n \in$ Pos, allowing divide and conquer solutions that break large problems into more than two subproblems.

In this chapter we described four binary interconnection networks by PowerList functions, and their generalized counterparts were described by PList functions. We were able to prove that the four network are isomorphic using algebraic techniques. As far as the author knows this has not been achieved in the literature. None of the proofs used indexing notations[4] and although the labeling we use can be thought of as a "coding" of indices, we have confined ourselves to algebraic reasoning about these labels.

The PList notation is very rich. It includes the PowerList theory as a special case. While this generality is not always needed in order to describe parallel computations, it may prove useful when the problem is stated in a different radix than

[4]This work was inspired by a paper by McIlroy and Savicki [MS97] where similar results were proven using index-based notations.

2, or in a mixed radix as in the case of the generalized networks discussed in this chapter.

Chapter 5

Conclusion

We start this chapter by surveying related research and comparing it to the work presented in this dissertation. Then, we present future directions for research building on top of this work. We conclude the chapter by commenting on the lessons learned from developing and studying the three data structures.

5.1 Related Work

In this section we start by surveying the PowerList literature and proceed by surveying the related work done on other functional approaches to parallel programming. Finally, we present work that is related to PowerLists, but use a different approach.

5.1.1 PowerLists

The work presented in this dissertation is an extension of the work done on PowerLists developed by Misra. Misra [Mis94] presented the PowerList theory along with a number of fundamental parallel algorithms, such as Batcher's two sorting networks, the Fast Fourier Transform and two algorithms for the Prefix Sum; Misra also presented a theory for generalizing the notation to multidimensional PowerLists. We address the issue of multidimensional extensions of the three structures in Section 5.2.

Adams [Ada94] derived and verified two addition circuits in PowerLists in a rigorous manner. We extended his main results to ParLists in Section 3.4, but did not include the proofs of the lemmas that Adams introduced in order to prove the equivalence of the circuits. Using a similar framework, Adams [Ada95] presented a PowerList description of a multiplication circuit.

Many basic results of the PowerList theory, as presented in [Mis94], and many of Adam's results have been mechanically verified by Kapur and Subramaniam [KS95, KS96a, KS96b] using the inductive theorem prover *Rewrite Rule Laboratory*. Gamboa [Gam97] has verified many fundamental results about PowerLists using the ACL2 theorem prover. His work focuses on the verification of Batcher's sorting networks as found in [Mis94].

The use of mechanical verification has been valuable for the PowerList research. Many of the basic properties of PowerLists have been mechanically verified in a more rigorous way than in the original proofs generated by humans. This rigor is achieved by the use of theorem provers which require that all data structures and applied methods be axiomatized before a proof that utilizes them can be completed.

The PowerList data structure has been an interesting challenge for the theorem-proving community [Kap94]. The constructors are partial, since they require that their arguments have the same length, and a non-singleton PowerList can be constructed using either $\bowtie$ and $|$. Another complication for automated theorem provers posed by the PowerList theory is the use of two constructors. The current research has dealt with this issue by regarding one constructor as fundamental and the other as a derived operator. However, as seen in this dissertation, the equal treatment of the two constructors is one of the strengths of the PowerList theory. It is unfortunate that this symmetry does not carry over to automated approaches.

In [AS96] Aschatz and Schulte present rules to transform PowerList functions to programs with sequential work flows, aiming at an efficient implementation on a *Wavetracer* machine[1]. Their approach is somewhat unusual: they transform PowerList descriptions into an intermediate language based on skeletons [Col89], thereby eliminating most of the structure that is present in the PowerList descriptions. These skeleton descriptions are then transformed into the programming language *multiC* [Wav92] that can be compiled for the *Wavetracer* architecture.

5.1.2 Functional Parallel Programming

Iverson developed the programming language APL [Ive62] based on the idea of applying a single operation to each element of a data structure. APL is a rich language with many operators that allow complex algorithms to be expressed succinctly. The theories presented in this dissertation have been developed with this goal in mind, while keeping the number of built-in operators to a minimum.

Some very convincing arguments for functional parallel programming were made by Backus in his Turing Award lecture [Bac78]. Backus proposed the parallel functional language FP, based on the use of second-order functions (functionals, like *reduce* and *map* defined for PowerLists in Section 2.1) that manipulate basic functions over linear lists. Basic functions are either data movement functions similar to the ones we defined for PowerLists in Section 2.1.2 or scalar functions. FP is equiped with an algebra that enables equality preserving transformations of FP functions. With FP Backus provided the insight that it is just as important to facilitate expressing what a parallel program should do, as expressing its execution on a (parallel) architecture.

A main difference between FP and the structures that we have presented in this dissertation is that FP lists are linear, i.e., they are accessed using $\triangleleft$ and $\triangleright$-like

[1]A *Single Instruction Multiple Data* (SIMD) 3-dimensional mesh architecture.

operators. Parallelism in FP is introduced by evaluating the second order functions in parallel.

Mou and Hudak [MH88, Mou90] presented *Divacon*, a very general functional notation for describing divide-and-conquer programs. The Divacon notation is meant to capture the entire class of divide-and-conquer algorithms. The emphasis of their work is to implement divide-and-conquer descriptions efficiently on parallel architectures, and to demonstrate that these implementations are efficient. By restricting the Divacon notation to certain patterns, Mou and Hudak presented implementation strategies for certain architectures, such as hypercubes and mesh oriented architectures. No algebra or formal tools were provided to prove the correctness of Divacon programs or to transform one description into another. For these reasons it would be difficult to prove the kind of properties we have proven in this dissertation.

Guy Blelloch developed and implemented the functional programming language NESL [Ble95]. The language is based on *nested parallelism* over linear lists: NESL functions can be applied to lists that may in turn contain lists as elements. PowerLists as presented in [Mis94] do support this notion of nested parallelism. NESL lists are dynamic in length, and in contrast with PowerLists there are no restrictions on the lengths of lists that are returned from functions. This enables a NESL description of a parallel version of Quicksort, where the recursive calls can be of unequal lengths as determined by the values of the input list and the chosen pivot element. It does not appear that this algorithm can be expressed elegantly in the structures presented in this dissertation. NESL was designed to produce efficient implementations of parallel algorithms on actual architectures and to reason about their theoretical complexity measures. Little consideration was given to providing a framework to prove NESL programs correct.

The programming language Sisal [MSA^{+}85] is a functional, parallel programming language designed for efficient compilations to existing parallel architectures. The goal behind the language is to provide programmers with an alternative to Fortran yielding more efficient implementations than optimizing Fortran compilers can provide [Can92]. Sisal is based on the idea of *single assignment variables*, i.e., a "variable" that is either undefined or, if it attains a value, then its value does not change in the remainder of the computation. Computations that expect a variable to have a value are suspended until the variable gets a value; thus it is possible to avoid many *race conditions*[2]. Sisal was designed to produce efficient implementations; as a result, a number of "features" of other parallel functional language are missing, such as higher order functions and built-in permutations. These features were included in the proposal for a new version of the language [BCFO91] that has not yet been implemented. In comparison to PowerList it is interesting to observe that the fundamental data structure in Sisal is non-empty arrays. While Sisal programs are more concise than corresponding Fortran programs, they are not easily amenable to formal proofs of correctness due to extensive use of indexing notations.

5.1.3 Bird-Meertens Formalism

The Bird-Meertens formalism [Bir89, Mee86, Ski94] has its roots in FP, but is more general since it applies to a number of different *categorical data types* [Mal90], including linear lists. In the following we present a simple version of the formalism based on linear lists (constructed with the concatenation operator $\Diamond$), basic functions and higher order functions. The key concept in the formalism is a *list homomorphism*, an algebraic property that a function (say h) over lists has if it "respects" list construction, i.e.:

$$h.(p \Diamond q) = h.p \ \otimes \ h.q \tag{5.1}$$

[2]The concept of single assignment is also used in PCN [CT89, CT90], a notation for parallel composition of sequential programs.

for some associative operator $\otimes$. The functions *sum*, *reduce* and *map* are all examples of homomorphisms. A homomorphism like h above satisfies the following law[3] [Bir89]:

$$h = reduce.\otimes \;\circ\; map.f \quad \text{where } f.a = h.[a] \tag{5.2}$$

and can be implemented in time proportional to the logarithm of the length of the list on most parallel architectures [Ski94, Gor96].

The Bird-Meertens formalism is very rich, providing many interesting results about functions over linear lists. Most of these results can be reused in the theories we presented in this dissertation, since the data structures can be viewed as linear lists by ignoring the way they were constructed. The Bird-Meertens formalism is more abstract than the theories we presented, allowing the programmer to work at a very high level of abstraction. However, such a high level of abstraction may also deter the programmer from coming up with efficient solutions since most reasoning is done with higher order functions.

Gorlatch [Gor96] adapted the Bird-Meertens formalism towards PowerLists by restricting list concatenation to lists of the same length. He categorized a class of functions called *distributable homomorphisms* that includes the prefix sum. He then showed that this class has an efficient implementation on hypercubic architectures, using a technique similar to the one we derived for the prefix sum algorithm on hypercubes in Section 2.4.1.

5.1.4 Other Models for Parallel Programming

Ascend and Descend Algorithms

Preparata and Vuillemin [PV81] presented the *cube-connected-cycles* (CCC) a network that has many of the topological properties of a hypercube, with only a constant number of neighbors for each node. The n-dimensional CCC can be con-

[3]Where *reduce* and *map* are similar to the PowerList functions defined in Section 2.1.

structed from an n-dimensional hypercube by replacing each hypercube node with a ring of n nodes. The n incident edges to a hypercube node are assigned in their dimensional order to the nodes in the corresponding ring on the CCC. Thus, each node on the CCC has degree 3. The CCC and the butterfly networks have very similar topologies [Lei92] and both can simulate hypercube algorithms efficiently. We recall from Section 1.3 that $H.f.n$ is the time that it takes to evaluate function f on inputs of length 2^n on a hypercube; similarly, we define $B.f.n$ as a measure for simulating the function on a butterfly (or CCC), this can be done with a polylogarithmic slowdown:

$$B.f \;\mathcal{O}\; (\lambda n :: n^2) * H.f$$

An even more important result in [PV81] is the classification of the group of divide-and-conquer algorithms called *Ascend* and *Descend*, that in PowerList correspond closely to deconstruction arguments with $\bowtie$ and $|$ respectively[4]. Preparata and Vuillemin presented Ascend and Descend algorithms for Batcher's merge and the Fast Fourier Transform.

The class of Ascend and Descend algorithms are contained in the class of *normal algorithms*, consisting of the hypercube algorithms that utilize adjacent dimensions in adjacent steps. It can be shown [Sch90] that a normal hypercube algorithm g can be simulated with a constant slowdown on a butterfly (and thus on a CCC):

$$g \text{ is normal } \Rightarrow B.g \,\mathcal{O}\, H.g$$

Ruby

Ruby [JS90] is a relational algebra, developed by Jones and Sheeran for designing integrated circuits at a high level. The goal of Ruby is to algebraically specify the layout of the wires that connect computational elements. The advantage of this

[4]See for instance the definitions of *rev* given by (2.20) and (2.21).

approach is that it is possible to reason formally about a circuit, while still being able to draw its physical representation. The PowerList constructors can be found in Ruby as predefined relations. They are not given any special treatment; instead they are considered part of a “tool box” available to the circuit designer.

In [JS91] Jones and Sheeran present recursive descriptions in Ruby of the Butterfly network, the Fast Fourier Transform algorithm, and Batcher’s sorting networks. These descriptions were derived using geometrical considerations and are more complex than the corresponding PowerList descriptions given by Misra [Mis94].

5.2 Future Work

PowerList

In Chapter 2 we established that many PowerList functions can be implemented efficiently on hypercubic architectures. However, most parallel architectures are not hypercubic or hypercube-like. More work is needed to present efficient implementations on common architectures, like the different mesh-based architectures that prevail in the marketplace. One approach, suggested by Cole [Col89] in the context of divide and conquer algorithms, is to lay out the computation using *H-trees* on a 2-dimensional mesh. This layout does not utilize all the processors of the mesh, and thus other strategies need to be pursued in the search for an optimal solution.

Another approach to implementing PowerList, as well as ParLists and PLists, is to map the functions to Sisal programs [MSA^{+}85]. This approach has the advantage that Sisal has efficient implementations on many parallel architectures.

ParList

The issue of efficient implementations becomes even more interesting when we turn to the ParList structure. The structure is obtained by adding sequential “alignment” steps to functions when they are applied to inputs of odd length. Using a mapping strategy like the one we propose for PowerList onto hypercubes, these alignment steps can be mapped onto the same nodes as the sub-results they operate on.

We presented a strategy for extending an inductive proof of a property of PowerList function, to a proof of the same property of the ParList function that is obtained by adding an odd defining case. We did not formalize the process by which this reuse can be achieved. It would be interesting to study in general how induction proofs of properties over an inductively defined structure can be reused when the structure is extended with new constructors.

PList

The presentation of the PList notation in this dissertation lays the foundation for future work. Since the PList notation generalizes the PowerList notation, it follows that all PowerList functions have PList descriptions. With PLists we can describe algorithms where the number of sub cases identified in the divide phase of a divide and conquer description may vary as a function of the input.

Since any number in Pos can be represented uniquely as the ascending sequence of its prime factors, a PList function like *sum* (defined in Section 4.2) can be defined on PLists of any positive length, by changing the predicate *defined* accordingly[5]. This gives an alternative to the ParList theory for specifying functions over inputs of arbitrary lengths.

This observation is interesting, but our goal has been to present abstractions that have a close relationship to parallel architectures. As far as the author knows, no one has built a practical parallel architectures based on the properties of prime factorizations.

[5]For *sum* this is not necessary, since all possible ways to break down a PList to singletons yields the same result.

Higher Dimensions

The three data structures can be extended to more than one dimension by replicating the constructors for each dimension. This enables us to describe matrix computations, using a similar approach to the one presented in this dissertation. Misra [Mis94] presented an outline of this idea for PowerList. Preliminary results using these extensions appear promising for the other data structures as well. Some simple matrix algorithms have elegant descriptions in the higher dimension extensions of PowerList; for example, we have descriptions of different versions of matrix multiplication: the standard divide and conquer technique, the Strassen algorithm [Str69] and the hypercube algorithm by Dekel, Nassimi and Sahni [DNS81]. The latter algorithm is described using an extended version of PowerLists in [Kor94].

5.3 Final Comments

The three data structures we presented were useful in expressing parallel computations. Equally important was the use of formal techniques to derive many of these descriptions from their specifications. This was possible because the structures were designed for equational reasoning and the functional setting provided referential transparency. The successful application of mechanical verification techniques to the PowerList structure further validates the design of that structure.

We recognize that these three structures are not the final word in parallel programming. There are classes of computations that only have awkward descriptions in our structures. An example of such a computation is the parallel version of Quicksort [Ble95], where subproblems have different sizes depending on the values in the list to be sorted. However, we hope that we have demonstrated that for computations with regular communication patterns, these structures allow elegant and efficient solutions to be constructed and verified in a rigorous manner.

Bibliography

[Ada94] Will E. Adams. Verifying adder circuits using powerlists. Technical Report CS-TR-94-02, University of Texas at Austin, Department of Computer Sciences, March 1994. Available by ftp as ftp://ftp.cs.utexas.edu/pub/techreports/tr94-02.ps.Z.

[Ada95] Will E. Adams. Multiplication circuits in powerlists. Unpublished manuscript, 1995.

[ANSI90] American National Standard Institute. *American National Standard for Information Systems Programming Language Fortran (Fortran 90).* ANSI, X3.198 1991 edition, 1990.

[AS96] Klaus Achatz and Wolfram Schulte. Massive parallelization of divide-and-conquer algorithms over powerlists. *Science of Computer Programming*, 26(1–3):59–78, May 1996.

[Bac78] John W. Backus. Can programming be liberated from the von Neumann style? A functional programming style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bat68] K. Batcher. Sorting networks and their application. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, Reston, Va, 1968. AFIPS Press.

[BCFO91] A. P. W. Böhm, D. C. Cahn, J. T. Feo, and R. R. Oldehoft. SISAL 2.0 reference manual. Technical Report UCRL-MA-109098, Lawrence Livermore National Laboratory, December 1991.

[Bir89] Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computer Science*, NATO ASI Series, pages 151–216. Springer Verlag, 1989.

[Ble89] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, November 1989.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[Ble93] Guy E. Blelloch. Prefix sums and their applications. In John H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 1, pages 33–60. Morgan Kaufmann, San Mateo, California, 1993.

[Ble95] Guy E. Blelloch. NESL: A nested data-parallel language (version 3.1). Technical Report CMU//CS-95-170, Carnegie Mellon University, School of Computer Science, September 1995.

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall International Series in Computer Science. Prentice Hall, 1988.

[Can92] David Cann. Retire Fortran? *Communications of the ACM*, 35(8):81–89, August 1992.

[CKP$^+$93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and van Eicken Thorsten. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–12, May 1993.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald E. Rivest. *Introduction to Algorithms.* McGraw Hill, 1990.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monograms in Computer Science. MIT press, 1989.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, April 1965.

[CT89] K. M. Chandy and S. Taylor. The composition of parallel programs. In *Supercomputing 89*, pages 557–561, 1989.

[CT90] K Mani Chandy and Stephen Taylor. Primer for program composition notation. Technical Report CS-TR-90-10, California Institute of Technology, 1990.

[Dem56] Howard B. Demuth. *Electronic Data Sorting.* PhD thesis, Stanford University, 1956.

[DNS81] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal on Computing*, 10(4):657–675, 1981.

[DS90] Edsger W. Dijkstra and Carel S. Sholten. *Predicate Calculus and Program Semantics.* Springer Verlag, 1990.

[Eck46] J. P. Eckert, Jr. A parallel channel computing machine. In H. H. Goldstine, H. A. Aitken, A. W. Burks, Jr. J.P. Eckert, J B. Mauchly, and J. von Neumann, editors, *The Moore School Lectures*. The Moore School, University of Pennsylvania, 1946. Reprinted as The Moore School Lectures, lecture 45, in volume 9 of the Charles Babbage Institute Reprint Series for the History of Computing, by MIT Press and Tomash Publishers, 1986.

[Gam97] Ruben A. Gamboa. Defthms about zip and tie: Reasoning about powerlists in ACL2. Technical Report CS-TR-97-02, The University of Texas at Austin, Department of Computer Sciences, January 23 1997.

[Gor96] Sergei Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In Luc Bougé et al., editor, *Proceedings of Euro-Par'96*, number 1124 in LNCS, pages 401–408. Springer Verlag, 1996.

[Gra53] Frank Gray. Pulse code communication. U.S. Patent 2,632,058, 1953.

[HJW+92] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, et al. A report on the functional language Haskell. *SIGPLAN Notices*, 1992.

[Ive62] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, New York, 1962.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley, Reading, MA, 1992.

[JH95] S. Lennart Johnsson and Ching-Tien Ho. On the conversion between binary code and binary-reflected gray code on binary cubes. *IEEE Transactions on Computers*, 44(1):47–53, January 1995.

[JS90] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design*, IFIP WG 10.5 Lecture Notes, chapter 1, pages 13–70. North-Holland, 1990.

[JS91] Geraint Jones and Mary Sheeran. Collecting butterflies. Technical Monograph PRG-91, Oxford University, February 1991.

[Kap94] D. Kapur. Constructors can be partial too. Technical Report 94-16, SUNY Buffalo, 1994.

[Kel89] Paul Kelly. *Functional Programming for Loosely-Coupled Multiprocessors*. Research Monograms in Computer Science. MIT press, 1989.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Vol. 3 : Sorting and Searching*. Series in Computer Science and Information Processing. Addison-Wesley, Reading, 1973.

[Kor94] Jacob Kornerup. Mapping powerlists onto hypercubes. Technical Report CS-TR-94-05, University of Texas at Austin, Department of Computer Sciences, August 1994. Available for download as ftp://ftp.cs.utexas.edu/pub/techreports/tr94-05.ps.Z.

[Kor95] Jacob Kornerup. Mapping a functional notation for parallel programs onto hypercubes. *Information Processing Letters*, 53:153–158, 1995.

[Kor97a] Jacob Kornerup. Odd-even sort in powerlists. *Information Processing Letters*, 61:15–24, 1997.

[Kor97b] Jacob Kornerup. Parlists – a generalization of powerlists. In Christian Lengauer, editor, *Proceedings of Euro-Par'97*, LNCS. Springer Verlag, 1997. To appear.

[Kor97c] Jacob Kornerup. Parlists - a generalization of powerlists (extended version). Technical Report CS-TR-97-15, University of Texas at Austin, Department of Computer Sciences, June 1997. Available for download as ftp://ftp.cs.utexas.edu/pub/techreports/tr97-15.ps.Z.

[KR90] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A. Elsevier North-Holland, 1990.

[KS95] D. Kapur and M. Subramaniam. Automated reasoning about parallel algorithms using powerlists. In Vangalur S. Alagar and M. Nivat, editors, *AMAST '95*, volume 936 of *LNCS*, page 416. Springer-Verlag, 1995.

[KS96a] D. Kapur and M. Subramaniam. Automating induction over mutually recursive functions. *Lecture Notes in Computer Science*, 1101:117, 1996.

[KS96b] D. Kapur and M. Subramaniam. Mechanically verifying a family of multiplier circuits. *Lecture Notes in Computer Science*, 1102:135, 1996.

[Las86] Clifford Lasser. The essential *lisp manual. Technical report, Thinking Machines Corporation, Cambridge, MA, July 1986.

[Lei92] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes.* Morgan Kaufmann, San Mateo, CA 94403, 1992.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.

[Mal90] Grant Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, Rijkuniversiteit Groeningen, September 1990.

[McC91] William F. McColl. General purpose parallel computing. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, Spring School on Parallel Computation, pages 337–391. ALCOM, Cambridge University Press, 1991.

[Mee86] Lambert Meertens. Algorithmics – towards programming as a mathematical activity. In *CWI Symposium on mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[MH88] Zhijing G. Mou and Paul Hudak. An algebraic model for divide-and-conquer and its parallelism. *The Journal of Supercomputing*, 2(3):257–278, November 1988.

[Mis94] Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, November 1994.

[Mis96] Jayadev Misra. Generalized powerlists. Unpublished manuscript, May 1996.

[MK97] Jayadev Misra and Jacob Kornerup. Describing structures of interconnection networks. In preparation, 1997.

[Mou90] Zhijing G. Mou. *A Formal Model for Divide-and-Conquer and Its Parallel Realization.* PhD thesis, Department of Computer Science, Yale University, May 1990.

[MP89] Ernst W. Mayr and Greg Plaxton. Pipelined parallel computations, and sorting on a pipelined hypercube. Technical Report STAN–CS-89-1261, Department of Computer Science, Stanford University, 1989.

[MS97] M. Douglas McIlroy and Joseph P. Savicki. Routing and complexity of rearrangeable networks. Personal Communication, 1997.

[MSA+85] James McGraw, Stephen Skezielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, and Robert Thomas. *SISAL: Streams and Iterations in a Single Assignment Language*. Lawrence Livermore National Laboratory, Reference manual version 1.2. manual M-146, Rev. 1 edition, March 1985.

[MTH90] Robin Milner, Mads Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Ofm63] Yu. Ofman. On the algorithmic complexity of discrete function. *Soviet Physics Doklady*, 7(7):289–591, 1963.

[PV81] Franco P. Preparata and Jean Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[RS87] John R. Rose and Guy L. Steele Jr. C^*: An extended language for data parallel programming. In *Proceedings Second International Conference on Supercomputing*, volume 2, pages 2–16, San Francisco, CA, May 1987.

[Sch90] E. Schwabe. On the computational equivalence of hypercube-derived networks. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 388–397. ACM, July 1990.

[Sew54] Harold E. Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, Stanford University, 1954.

[Ski94] David B. Skillicorn. *Foundations of Parallel Programming*. Series in Parallel Computation. Cambridge University Press, 1994.

[Sto71] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, 1971.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[Tur86] David Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21:156–166, 1986.

[Wav92] Wavetracer, Inc. *MultiC Basics for DTC Systems*, 1992.

Vita

Jacob Kornerup was born on September 14, 1963 in Århus, Denmark. He attended Kochs Skole in Århus for the first through fifth and seventh through ninth grades. He traveled with his family to Lafayette, Louisiana in 1975, where he attended Prairie Elementary School for the sixth grade. He attended gymnasium (high school) at Århus Katedralskole from 1979 to 1982, receiving his studentereksamen with the highest grade point average in his graduating class.

In September of 1982 he enrolled at the University of Århus, studying mathematics and computer science. In 1984 he was appointed as a teaching assistant in the computer science department, a position he held until he graduated. He finished his minor in mathematics in 1985 and completed his Cand. Scient. (Masters degree) in computer science in February 1988. In March 1988 he was awarded a two and a half year postgraduate scholarship from the University of Århus for Ph.D. studies at the University of Texas at Austin.

In August of 1988 he moved to Austin, Texas, enrolling in the Ph.D. program in Computer Sciences at the University of Texas at Austin. In 1990 and 1991 he worked as a teaching assistant in the department. From 1991 to 1995 he was employed as a research assistant by his dissertation advisor, Professor Jayadev Misra, Ph.D., under support from grants from: Texas Advanced Research Program, the National Science Foundation, and The Office for Naval Research. In 1995, he worked as an Assistant Instructor in the department, teaching an undergraduate programming class.

Permanent Address: 6735 Old Quarry Lane

Austin, TX 78731

e-mail: `jkornerup@acm.org`

This dissertation was typeset with LaTeX 2_ε[6] by the author.

[6]LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.