

Automatic Worst Case Complexity Analysis of Parallel Programs

Wolf Zimmermann

TR-90-066

December, 1990

Abstract

This paper introduces a first approach for the automatic worst case complexity analysis. It is an extension of previous work on the automatic complexity analysis of functional programs. The language is a first order parallel functional language which allows the definition of indexed data types and parallel execution of indexed terms. The machine model is a parallel reduction system based on eager evaluation. It is shown how parallel programs based on the basic design principles balanced binary tree technique, divide-and-conquer technique and pointer jumping technique can be analyzed automatically. The analysis techniques are demonstrated by various examples. Finally it is shown that an average case analysis of parallel programs is difficult.

Contents

1	Introduction	2
2	The Language PARSTYFL	3
2.1	Types and Indexed Types	3
2.2	Functions and their Semantics	6
3	Analysis of Algorithms based on the Balanced Binary Technique	14
3.1	Deriving the Time Functions	15
3.2	Normalisation	17
3.3	Symbolic Evaluation	20
3.4	Deriving Recurrences	21
3.5	Example: Prefix Sums	24
3.6	Polynomial Evaluation	26
4	Algorithms based on Pointer Jumping	32
4.1	Introduction to Pointer Jumping	32
4.2	Proper Pointer Jumping	35
4.3	Analysis of Pointer Jumping Algorithms	42
4.4	Example: Finding a Sublist of a List	47
5	Conclusions	61

Chapter 1

Introduction

This paper introduces a first approach for the automatic worst case complexity analysis. It is an extension of previous work on the automatic complexity analysis of functional programs. The language is a first order parallel functional language which allows the definition of indexed data types and parallel execution of indexed terms. The machine model is a parallel reduction system based on eager evaluation. It is shown how the basic design-principles of parallel programs can be analyzed automatically. The programs in this report are designed for EREW-PRAMS. However, the analysis technique for programs having concurrent reads is the same. The analysis techniques are demonstrated by various examples.

The design principles covered in this paper are the *balanced binary tree method*, the *divide-and-conquer technique*, the *compression technique*, and the *pointer jumping technique* [GR88, KR88]. In a functional language, the compression technique and the balanced binary tree method lead often to the same program. At least in our examples, it is in fact the same algorithm.

As far as known to the author the method introduced in this paper is the first method for the automatic complexity analysis of *parallel algorithms*. For sequential ones, there are well-known methods based on recurrences [Weg75, HC88, LeM88, Zim90b] and on generating functions [FSZ88, Fla88, FO88, FSZ91, Zim89, Zim90a]. An overview of the main results on both methods can be found in [ZZ89].

The second chapter introduces the language PARSTYFL (*parallel simple typed functional language*). The syntax and semantics are not defined in a complete formal way, but it can be done similar to [Zim90b]. In the third chapter we introduce the analysis method in general, and apply it to algorithms designed by the balanced binary tree method and divide-and-conquer technique. In the fourth chapter, the difficulties with the pointer jumping technique are discussed, and how they can be solved by a powerful heuristic. Finally, we give an outlook towards further work in the field.

The algorithms analyzed in this paper base on the algorithms of section 1 in [GR88] and section 2.1 in [KR88].

Chapter 2

The Language PARSTYFL

Each PARSTYFL-program Π consists of a set of type definitions T and a set of function definitions F . Sometime we denote this fact by $\Pi = (T, F)$. Therefore, this chapter is divided in two sections. In the first section we introduce the syntax and semantics of the types in an informal way. For the design of parallel programs it is useful to have *indexed types*. This is in fact the only extension of STYFL [Zim90b] on the type level. The second section introduces the program structures of PARSTYFL. The extensions to STYFL are the use of indices in indexed objects of indexed types and the parallel execution of terms containing indices. The semantics, the definition of time complexity, space complexity and processor complexity (i.e. the number or processors used by a program) is given in a formal way, while the syntax is defined in an informal way.

2.1 Types and Indexed Types

Types are described by an algebraic specification mechanism. They could be parametrized. Each type consists of a declaration of its name, of the types (i.e. *sorts*) used in the definition of a type, of a set of constructors together with their signature, of a set of operations together with their signature, and of a set of equations. We describe therefore a type syntactically by:

```
type   Name( $A_1, \dots, A_q$ )
sorts   $S_1, \dots, S_n$ 
constructors
     $c_1 : I_{1,1} \times \dots \times I_{1,r_1} \mapsto O_1$ 
     $\vdots$ 
     $c_k : I_{k,1} \times \dots \times I_{k,r_k} \mapsto O_k$ 
operations
     $o_1 : I_{1,1}^o \times \dots \times I_{1,s_1}^o \mapsto O_1^o$ 
     $\vdots$ 
     $o_l : I_{l,1}^o \times \dots \times I_{l,s_l}^o \mapsto O_l^o$ 
```

variables $x_1 : T_1, \dots, x_m : T_m$

equations

$$LHS_1 = RHS_1$$

\vdots

$$LHS_p = RHS_p$$

where:

- $l, m, n, p, q \geq 0$, $r_i \geq 0$ for $1 \leq i \leq k$, $s_j \geq 0$ for $1 \leq j \leq l$, and $k \geq 1$.
- All $I_{i,j}$, $I_{i,j}^o$, O_i , O_i^o and T_i are in $\{S_1, \dots, S_m\}$.
- The terms LHS_i and RHS_i are terms of the same type and contain only the symbols c_1, \dots, c_k , o_1, \dots, o_l , and x_1, \dots, x_m .

We call *Name* the *name* of the type, the A_i the *parameters* of the type, the set $S = \{S_1, \dots, S_m\}$ the *sorts* of the type, the set $\Sigma = \{c_1, \dots, c_k, o_1, \dots, o_l\}$ together with their functionality the *operations* of the type, the set $C = \{c_1, \dots, c_n\}$ the *constructors* of the type, and the set $E(\{x_1 : T_1, \dots, x_m : T_m\}) = \{LHS_1 = RHS_1, \dots, LHS_p = RHS_p\}$ the *equations* of the type. In proofs and comments we denote these facts by $T = (Name(A_1, \dots, A_q), \Sigma, C, E(X))$.

The semantics of a type is the classical quotient term algebra [EM85]. We require that the quotient term algebra is isomorphic to the term algebra for the constructors C . This can for example be achieved if the set of equations satisfies the definition principle of Huet and Hullot [HH80].

Example 2.1 (Lists) *The following type defines lists similar to the lists in LISP¹:*

type $List(A)$

sorts $List, A, Bool$

constructors

$$nil \mapsto List(A)$$

$$cons : A \times List(A) \mapsto List(A)$$

operations

$$empty : List(A) \mapsto Bool$$

$$car : List(A) \mapsto A$$

$$cdr : List(A) \mapsto List(A)$$

variables $l : List(A), a : A$

equations

$$empty(nil) = true$$

$$empty(cons(a, l)) = false$$

$$car(cons(a, l)) = a$$

$$cdr(cons(a, l)) = l$$

¹The type *Bool* contains the boolean constants *true* and *false*, as well as the basic logical junctors

For parallel programs it is useful to have indexed data types (i.e. arrays) and to perform operations in parallel on the objects referred to by the indices. An indexed data type is a parameterized type (with *one* parameter) together with an operation defining the *size* of the terms (i.e. the number of subterms of the parameter type) and a random access function to the *i*-th object in a term. In particular it is expressed as²:

```

indexed type  $Name(A)$ 
sorts  $S_1, \dots, S_n$ 
constructors
     $c_1 : I_{1,1} \times \dots \times I_{1,r_1} \mapsto O_1$ 
     $\vdots$ 
     $c_k : I_{k,1} \times \dots \times I_{k,r_k} \mapsto O_k$ 
operations
     $o_1 : I_{1,1}^o \times \dots \times I_{1,s_1}^o \mapsto O_1^o$ 
     $\vdots$ 
     $o_l : I_{l,1}^o \times \dots \times I_{l,s_l}^o \mapsto O_l^o$ 
variables  $x_1 : T_1, \dots, x_m : T_m$ 
equations
     $LHS_1 = RHS_1$ 
     $\vdots$ 
     $LHS_p = RHS_p$ 
indexing
     $length : Name(A) \mapsto \mathbf{N}$ 
# Equations defining  $length$ 
     $LHS_1^l = RHS_1^l$ 
     $\vdots$ 
     $LHS_k^l = RHS_k^l$ 
     $access : Name(A) \times \mathbf{N} \mapsto A$ 
# Equations defining  $access$ 
     $LHS_1^a = RHS_1^a$ 
     $\vdots$ 
     $LHS_k^a = RHS_k^a$ 

```

The semantics is just the same as if *length* and *access* would be added to the operations and the equations defining *length* and *access* to the equation part. We abbreviate $access(t, i)$ by $t[i]$.

Keeping this in mind, we sometimes mean by Σ_Π the set of all constructors and operations defined in any type of a program Π , by C_Π the set of all constructors defined in any type of Π , and by $E_\Pi(X_\Pi)$ the set of all equations of the types of Π .

² \mathbf{N} denotes the type of natural numbers, it contains the numbers (expressed by the 0 and successor function) and the usual arithmetical operations

Example 2.2 (Indexed Lists) *The following type defines indexed lists. Remember that indexed lists and arrays is nearly equivalent (in arrays the borders are explicitly specified while in a indexed list l the borders are $0..\text{length}(l) - 1$).*

```

indexed type List( $A$ )
sorts List,  $A$ , Bool
constructors
  nil  $\mapsto$  List( $A$ )
  cons  $: A \times \text{List}(A) \mapsto \text{List}(A)$ 
operations
  empty  $: \text{List}(A) \mapsto \text{Bool}$ 
  car  $: \text{List}(A) \mapsto A$ 
  cdr  $: \text{List}(A) \mapsto \text{List}(A)$ 
variables  $l : \text{List}(A)$ ,  $a : A$ 
equations
  empty(nil) = true
  empty(cons( $a, l$ )) = false
  car(cons( $a, l$ )) =  $a$ 
  cdr(cons( $a, l$ )) =  $l$ 
indexing
  length  $: \text{List}(A) \mapsto \mathbf{N}$ 
  length(nil) = 0
  length(cons( $a, l$ )) = 1 + length( $l$ )
  ( $\cdot$ )[( $\cdot$ )]  $: \text{List}(A) \times \mathbf{N} \mapsto A$ 
  cons( $a, l$ )[0] =  $a$ 
  cons( $a, l$ )[ $i + 1$ ] =  $l[i]$ 

```

Remark 2.3 (Indexed Types with More than One Parameter) *The restriction to one parameter in the definition of indexed types is easy to extend to more than one parameter. Just define an index-tuple where the i -th tuple entry refers to the i -th object of type of the i -th parameter. It makes therefore sense to define for each parameter its own length and its own access function. This way of the extension is a kind of currying, i.e. $T(A_1, \dots, A_m) = T(A_1)(A_2) \cdots (A_m)$.*

Even if we use in this paper just indexed lists, it should be clear that indexing is not restricted to lists. It could for example also be applied to binary trees: if the index of a node is i then the index of the left son to $2 \cdot i$ and the index of the right son to $2 \cdot i + 1$.

2.2 Functions and their Semantics

Syntactically, functions are equations of the form

$$\text{fun } f(x_1 : I_1, \dots, x_k : I_k) : O = B$$

where the I_j are the *input types* of the function f , the O is the *output type* of f , and B is the body of f . Each input type and the output type must be declared in the type part of the program. The body of the program must be an expression of type O . Expressions can be constructed by the program structures which are defined below. The semantics is an operational call by value semantics. This semantics tells how to evaluate an expression under a given program Π and an environment ENV . Environments give for each variable occurring in an expression its value. Thus, environments are finite mappings from variables to constructor terms (remember that the types are interpreted by the set of their constructor terms). They are denoted by a sequence of pairs $[v \leftarrow t]$ where v is a variable and t is the constructor term. We denote environment by ρ and the value of a variable v under an environment ρ by $\rho(v)$. The semantics of expressions is therefore a function

$$EVAL : EXPR \times PROG \times ENV \mapsto T_C$$

where $PROG$ is the (syntactic) set of all PARSTYFL-programs, ENV is the set of all environments, $EXPR$ is the set of all expressions, and T_C the set of all constructor terms of the types defined in the program. We give below the syntactical program structures together with their semantics.

Definition 2.4 (Variables) *A variable v is an expression. Let Π be a program and ρ be an environment. Then*

$$EVAL \llbracket v \rrbracket \Pi \rho = \rho(v)$$

Definition 2.5 (Constructor Call) *Let Π be a program and $c : I_1 \times \dots \times I_r \mapsto O$ a constructor defined in the types of Π . Furthermore, let t_1, \dots, t_k be terms of type I_1, \dots, I_k , respectively. Then, the constructor call*

$$c(t_1, \dots, t_k)$$

is also an expression. Its semantics is defined by:

$$EVAL \llbracket c(t_1, \dots, t_k) \rrbracket \Pi \rho = c(EVAL \llbracket t_1 \rrbracket \Pi \rho, \dots, EVAL \llbracket t_k \rrbracket \Pi \rho)$$

Definition 2.6 (Operation Call) *Let Π be a program and $o : I_1 \times \dots \times I_r \mapsto O$ be an operation defined in the types of Π . Furthermore, let t_1, \dots, t_k be terms of type I_1, \dots, I_k , respectively. Then, the operation call*

$$o(t_1, \dots, t_k)$$

is also an expression.

Let now $LHS = RHS$ be an equation, where LHS matches the term

$$o(EVAL \llbracket t_1 \rrbracket \Pi \rho, \dots, EVAL \llbracket t_k \rrbracket \Pi \rho)$$

under an environment ρ , i.e. there is a substitution σ such that $\sigma(LHS)$ yields the above term. Then the semantics of an operation call is defined by:

$$EVAL \llbracket o(t_1, \dots, t_k) \rrbracket \Pi \rho = \sigma(RHS)$$

Definition 2.7 (Function Call) Let

$$\text{fun } g(x_1 : I_1, \dots, x_k : I_k) : O = B$$

be a function definition in a program Π , and t_1, \dots, t_k expressions of types I_1, \dots, I_k , respectively. Then, the function call

$$g(t_1, \dots, t_k)$$

is also an expression. Its semantics is defined by:

$$\begin{aligned} EVAL \llbracket g(t_1, \dots, t_k) \rrbracket \Pi \rho = \\ EVAL \llbracket B \rrbracket \Pi \rho [x_1 \leftarrow EVAL \llbracket t_1 \rrbracket \Pi \rho] \dots [x_k \leftarrow EVAL \llbracket t_k \rrbracket \Pi \rho] \end{aligned}$$

Remark 2.8 (Strict and Lazy Semantics) The strict semantics arises from the fact that the environment contains only constructor terms and that in the operation calls and function calls the arguments must be evaluated before performing the body. In a lazy semantics it would also be allowed to have arbitrary expressions in the environment. In the operation calls and function calls, the environment would just be extended by the terms as they appear (not evaluated). The semantics of a variable is then the evaluation of the corresponding expression in the environment.

Definition 2.9 (Conditional Statement) Let b be an expression of type `Bool` or an equation $s_1 = s_2$ where s_1 and s_2 are terms of the same type. Furthermore let t_1 and t_2 be two expressions of the same type. Then the conditional statement

$$\text{if } b \text{ then } t_1 \text{ else } t_2$$

is also an expression. Let Π be a program and ρ be an environment³:

$$EVAL \llbracket \text{if } b \text{ then } t_1 \text{ else } t_2 \rrbracket \Pi \rho = \begin{cases} EVAL \llbracket t_1 \rrbracket \Pi \rho & \text{if } EVAL \llbracket b \rrbracket \Pi \rho = \text{true} \\ EVAL \llbracket t_2 \rrbracket \Pi \rho & \text{if } EVAL \llbracket b \rrbracket \Pi \rho = \text{false} \\ \perp & \text{otherwise} \end{cases}$$

³ \perp defines the bottom element ("undefined")

$$EVAL \llbracket s_1 = s_2 \rrbracket \Pi \rho = \begin{cases} \perp & \text{if } EVAL \llbracket s_1 \rrbracket \Pi \rho = \perp \text{ or } EVAL \llbracket s_2 \rrbracket \Pi \rho = \perp \\ true & \text{if } EVAL \llbracket s_1 \rrbracket \Pi \rho = EVAL \llbracket s_2 \rrbracket \Pi \rho \neq \perp \\ false & \text{otherwise} \end{cases}$$

Definition 2.10 (Local Variables) Let x be a new variable (not occurring somewhere in an environment ρ) and t_1 and t_2 two expressions of not necessarily the same type. Then the local use of x denoted by

let $x = t_1$ **in** t_2

is also an expression. Its semantics under a program Π is:

$$EVAL \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket \Pi \rho = EVAL \llbracket t_2 \rrbracket \Pi \rho[x \leftarrow EVAL \llbracket t_1 \rrbracket \Pi \rho]$$

Definition 2.11 (Parallel Statement) Let Π be a program and $t(i)$ an expression containing i and ub be a natural number. Then the parallel statement

forall $i < ub$ **do in parallel** $t(i)$ **of type** $N(T)$

is also an expression. If $t(i)$ is type T , then the parallel statement is of an indexed type $N(T)$. If $N(T)$ is clear from the context or plays no role, then its declaration is omitted. Its semantics is defined by:

$$EVAL \llbracket \text{forall } i < ub \text{ do in parallel } t(i) \text{ of type } N(T) \rrbracket \Pi \rho = x$$

where $x[i] = EVAL \llbracket t(i) \rrbracket \Pi \rho$ for all $0 \leq i < ub$.

We have not yet defined that the execution of this statement is parallel! This is the task of the definition of the complexities. The semantics of the parallel statement is complete because indexed types explain how to build up structures, when they are defined via indices. In trees for example, balanced binary trees will be built up (according to the remarks at the end of the last section).

The definition of time is based on the basic complexities in figure 2.1. These basic complexities can be defined by the user of an automatic complexity analysis system. We will assume for simplicity that each of these complexities needs 1 time unit. That represents the number of (parallel) *EVALs* needed to be evaluate an expression. The execution time of an expression is defined by a function:

$$TIME : EXPR \times PROG \times ENV \mapsto \mathbb{N}$$

Its defining equations are given in the following definition:

τ_f	for each $f \in \Sigma$
τ_{var}	for a reference to a variable
τ_{call}	for function calls, defined in the function part
τ_{if}	for evaluation of conditionals
τ_{eq}	for checking terms on equality
τ_{let}	for introducing local variables
τ_{par}	for starting the parallel statement

Figure 2.1: Basic Complexities

Definition 2.12 (Time Complexity) *The execution time of an expression t under a program Π and an environment ρ is defined by the following equations:*

(i) *For a variable v :*

$$TIME \llbracket v \rrbracket \Pi \rho = \tau_{var}$$

(ii) *For a constructor call or operation call $o(t_1, \dots, t_k)$:*

$$TIME \llbracket o(t_1, \dots, t_k) \rrbracket \Pi \rho = \tau_o + TIME \llbracket t_1 \rrbracket \Pi \rho + \dots + TIME \llbracket t_k \rrbracket \Pi \rho$$

(iii) *For function call $g(t_1, \dots, t_k)$:*

$$TIME \llbracket g(t_1, \dots, t_k) \rrbracket \Pi \rho = \tau_{call} + TIME \llbracket t_1 \rrbracket \Pi \rho + \dots + TIME \llbracket t_k \rrbracket \Pi \rho + TIME \llbracket B \rrbracket \Pi \rho'$$

where

- **fun** $g(x_1 : I_1, \dots, x_k : I_k) : O = B$ is a function definition in Π , and
- $\rho' = \rho[x_1 \leftarrow EVAL \llbracket t_1 \rrbracket \Pi \rho] \dots [x_k \leftarrow EVAL \llbracket t_k \rrbracket \Pi \rho]$

(iv) *For a conditional statement:*

$$TIME \llbracket \text{if } c \text{ then } t_1 \text{ else } t_2 \rrbracket \Pi \rho =$$

$$\begin{cases} \tau_{if} + TIME \llbracket c \rrbracket \Pi \rho + TIME \llbracket t_1 \rrbracket \Pi \rho & \text{if } EVAL \llbracket c \rrbracket \Pi \rho = \text{true} \\ \tau_{if} + TIME \llbracket c \rrbracket \Pi \rho + TIME \llbracket t_2 \rrbracket \Pi \rho & \text{if } EVAL \llbracket c \rrbracket \Pi \rho = \text{false} \\ \infty & \text{otherwise} \end{cases}$$

$$TIME \llbracket s_1 = s_2 \rrbracket \Pi \rho =$$

$$\begin{cases} \tau_{eq} + TIME \llbracket t_1 \rrbracket \Pi \rho + TIME \llbracket t_2 \rrbracket \Pi \rho & \text{if } EVAL \llbracket t_1 \rrbracket \Pi \rho \neq \perp \text{ and } EVAL \llbracket t_2 \rrbracket \Pi \rho \neq \perp \\ \infty & \text{otherwise} \end{cases}$$

(v) *For the introduction of local variables:*

$$TIME \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket \Pi \rho = \tau_{let} + TIME \llbracket t_1 \rrbracket \Pi \rho + TIME \llbracket t_2 \rrbracket \Pi \rho'$$

where $\rho' = \rho[x \leftarrow EVAL \llbracket t_1 \rrbracket \Pi \rho]$

(vi) For the parallel statement:

$$TIME \llbracket \text{forall } i < ub \text{ do in parallel } t(i) \rrbracket \Pi \rho = \tau_{par} + TIME \llbracket ub \rrbracket \Pi \rho + \max_{0 \leq i < ub} TIME \llbracket t(i) \rrbracket \Pi \rho$$

Now it is clear why the parallel statement is in fact a parallel execution. In part (vi) of the above definition, the evaluation of the parallel statement takes a time determined by the *maximum* execution time of the execution of the body. This means that the body statements are executed in parallel, and the whole parallel statement is completed when the last process terminates.

The following two definitions confirms also the parallel execution. In the space complexity the number of symbols in a term is counted. The space complexity counts the maximum number of symbols needed to evaluate an expression. Hence, the parallel evaluation of terms needs the sum of all space complexities needed by the parallel evaluation processes. The space complexity is a function

$$SPACE : EXPR \times PROG \times ENV \mapsto N$$

Its meaning is defined in definition 2.13. Finally we have to define the processor complexity. That is the maximal number of processors used by the evaluation of an expression. It is obvious that the only statement which needs more than one processor is the parallel statement. All the other statements are intended to be sequential and need therefore only one processor. The *processor complexity* is a function

$$PROC : EXPR \times PROG \times ENV \mapsto N$$

which will be defined in definition 2.14.

Definition 2.13 (Space Complexity) *The space needed for the evaluation of an expression t under a program Π and an environment ρ is defined by the following equations:*

(i) For a variable v :

$$SPACE \llbracket v \rrbracket \Pi \rho = size(\rho(v))$$

where *size* counts all the non-constant symbols in a constructor term.

(ii) For a constructor call $c(t_1, \dots, t_k)$:

$$\begin{aligned} SPACE \llbracket c(t_1, \dots, t_k) \rrbracket \Pi \rho = \\ 1 + \max\{SPACE \llbracket t_1 \rrbracket \Pi \rho, \dots, SPACE \llbracket t_k \rrbracket \Pi \rho, \\ 1 + size(EVAL \llbracket t_1 \rrbracket \Pi \rho) + \dots + size(EVAL \llbracket t_k \rrbracket \Pi \rho)\} \end{aligned}$$

(iii) For an operation call $o(t_1, \dots, t_k)$:

$$\begin{aligned} SPACE \llbracket c(o_1, \dots, t_k) \rrbracket \Pi \rho = 1 + \max\{SPACE \llbracket t_1 \rrbracket \Pi \rho, \dots, SPACE \llbracket t_k \rrbracket \Pi \rho, \\ 1 + size(EVAL \llbracket o(t_1, \dots, t_k) \rrbracket \Pi \rho)\} \end{aligned}$$

(iv) For function call $g(t_1, \dots, t_k)$:

$$SPACE \llbracket g(t_1, \dots, t_k) \rrbracket \Pi \rho = \max\{SPACE \llbracket t_1 \rrbracket \Pi \rho, \dots, SPACE \llbracket t_k \rrbracket \Pi \rho, SPACE \llbracket B \rrbracket \Pi \rho'\}$$

where

- **fun** $g(x_1 : I_1, \dots, x_k : I_k) : O = B$ is a function definiton in Π , and
- $\rho' = \rho[x_1 \leftarrow EVAL \llbracket t_1 \rrbracket \Pi \rho] \dots [x_k \leftarrow EVAL \llbracket t_k \rrbracket \Pi \rho]$

(v) For a conditional statement:

$$SPACE \llbracket \text{if } c \text{ then } t_1 \text{ else } t_2 \rrbracket \Pi \rho = \begin{cases} \max\{SPACE \llbracket c \rrbracket \Pi \rho, SPACE \llbracket t_1 \rrbracket \Pi \rho\} & \text{if } EVAL \llbracket c \rrbracket \Pi \rho = \text{true} \\ \max\{SPACE \llbracket c \rrbracket \Pi \rho, SPACE \llbracket t_2 \rrbracket \Pi \rho\} & \text{if } EVAL \llbracket c \rrbracket \Pi \rho = \text{false} \\ \infty & \text{otherwise} \end{cases}$$

$$SPACE \llbracket s_1 = s_2 \rrbracket \Pi \rho = \begin{cases} 1 + \max\{SPACE \llbracket t_1 \rrbracket \Pi \rho, SPACE \llbracket t_2 \rrbracket \Pi \rho\} & \text{if } EVAL \llbracket t_1 \rrbracket \Pi \rho \neq \perp \\ & \text{and } EVAL \llbracket t_2 \rrbracket \Pi \rho \neq \perp \\ \infty & \text{otherwise} \end{cases}$$

(vi) For the introduction of local variables:

$$SPACE \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket \Pi \rho = \max\{SPACE \llbracket t_1 \rrbracket \Pi \rho, SPACE \llbracket t_2 \rrbracket \Pi \rho'\}$$

where $\rho' = \rho[x \leftarrow EVAL \llbracket t_1 \rrbracket \Pi \rho]$

(vii) For the parallel statement:

$$TIME \llbracket \text{forall } i < ub \text{ do in parallel } t(i) \rrbracket \Pi \rho = 1 + \sum_{i=0}^{ub-1} SPACE \llbracket t(i) \rrbracket \Pi \rho$$

Definition 2.14 (Processor Complexity) The number of processors needed for the evaluation of an expression t under a program Π and an environment ρ is defined by the following equations:

(i) For a variable v :

$$PROC \llbracket v \rrbracket \Pi \rho = 1$$

(ii) For a constructor call, or operation call $o(t_1, \dots, t_k)$:

$$PROC \llbracket c(t_1, \dots, t_k) \rrbracket \Pi \rho = \max\{PROC \llbracket t_1 \rrbracket \Pi \rho, \dots, PROC \llbracket t_k \rrbracket \Pi \rho\}$$

(iii) For function call $g(t_1, \dots, t_k)$:

$$PROC \llbracket g(t_1, \dots, t_k) \rrbracket \Pi \rho = \max\{PROC \llbracket t_1 \rrbracket \Pi \rho, \dots, PROC \llbracket t_k \rrbracket \Pi \rho, PROC \llbracket B \rrbracket \Pi \rho'\}$$

where

- **fun** $g(x_1 : I_1, \dots, x_k : I_k) : O = B$ is a function definiton in Π , and

$$- \rho' = \rho[x_1 \leftarrow EVAL \llbracket t_1 \rrbracket \Pi \rho] \cdots [x_k \leftarrow EVAL \llbracket t_k \rrbracket \Pi \rho]$$

(iv) For a conditional statement:

$$PROC \llbracket \text{if } c \text{ then } t_1 \text{ else } t_2 \rrbracket \Pi \rho = \begin{cases} \max\{PROC \llbracket c \rrbracket \Pi \rho, PROC \llbracket t_1 \rrbracket \Pi \rho\} & \text{if } EVAL \llbracket c \rrbracket \Pi \rho = \text{true} \\ \max\{PROC \llbracket c \rrbracket \Pi \rho, PROC \llbracket t_2 \rrbracket \Pi \rho\} & \text{if } EVAL \llbracket c \rrbracket \Pi \rho = \text{false} \\ \infty & \text{otherwise} \end{cases}$$

$$PROC \llbracket s_1 = s_2 \rrbracket \Pi \rho = \begin{cases} \max\{PROC \llbracket t_1 \rrbracket \Pi \rho, PROC \llbracket t_2 \rrbracket \Pi \rho\} & \text{if } EVAL \llbracket t_1 \rrbracket \Pi \rho \neq \perp \\ & \text{and } EVAL \llbracket t_2 \rrbracket \Pi \rho \neq \perp \\ \infty & \text{otherwise} \end{cases}$$

(v) For the introduction of local variables:

$$PROC \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket \Pi \rho = \max\{PROC \llbracket t_1 \rrbracket \Pi \rho, PROC \llbracket t_2 \rrbracket \Pi \rho'\}$$

$$\text{where } \rho' = \rho[x \leftarrow EVAL \llbracket t_1 \rrbracket \Pi \rho]$$

(vii) For the parallel statement:

$$PROC \llbracket \text{forall } i < ub \text{ do in parallel } t(i) \rrbracket \Pi \rho = \sum_{i=0}^{ub-1} PROC \llbracket t(i) \rrbracket \Pi \rho$$

Sometimes the *size* or the *length* of the evaluated expression is needed. This can be defined in a similar way as above by defining function *SIZE* and *LENGTH* by similar equations. It is left to the reader to do this exercise. In the following chapter we demonstrate how the complexity analysis of parallel programs can be mechanized, yielding a correct result w.r.t. the definitions in this chapter. It cannot be expected that this method is complete, because computing the time complexity is in general a non-computable function (the halting problem would become decidable). We demonstrate the method for the time complexity, but it should be clear that the method works also for the other complexities of this chapter.

Chapter 3

Analysis of Algorithms based on the Balanced Binary Technique

This chapter deals with the general analysis method. With this method it is in principle possible to analyze programs designed by the balanced binary tree technique and by the divide-and-conquer technique. In a functional parallel language these two techniques leads often to the same program. This is no surprise, because the parallel execution of a divide-and-conquer program can be described by a balanced binary tree. The only difference between these two techniques is that the balanced binary tree technique starts computation from the leafs propagating information just to the parent nodes, while in the divide-and-conquer technique information control is also from the root to the sons. Hence, the balanced-binary-tree method results in bottom-up computation, while the divide-and-conquer method results in top-down computation. It is therefore obvious, that the same complexity analysis technique can be applied to programs designed by both methods.

We consider here just the time complexity analysis (and if it is necessary also the output length and output size of a function). The analysis is divided into 4 substeps. In fact these substeps are the same as for sequential algorithms [Zim90b, Weg75]:

1. Derive a set of functions, which compute the time complexity of the original functions, i.e. if $f(x)$ is a function in a program Π then $time_f(x)$ is the function yielding for each argument t the time needed for the evaluation of $f(t)$.
2. Perform some normalization steps on the functions derived in step 1. These are transforming the body of the functions $time_f$ into a special form (no nested conditionals), eliminating irrelevant argument positions.
3. Perform symbolic evaluation, i.e. try to find substitutions for the argument, such that conditions in the body of a function $time_f$ becomes true and false, respectively, and evaluate this function with the above arguments.
4. Map types to naturals by the two mappings *length* and *size*. This step is exactly the same as in the case of sequential algorithms. We describe it therefore just shortly. The result is now a set of recurrences to be solved. This step requires sometimes the analysis of output length or output size of certain functions.

5. Solve the recurrences.

It should be clear that the analysis of the other complexity measures differs only in the first step. We describe in the first sections the particular substeps in the automatic analysis method. We demonstrate the ability by the example finding the minimum of a list of elements. Later sections are dealing with more complicated examples. Note that the pointer jumping technique is not discussed in this chapter. We will discuss this in the next chapter.

Example 3.1 (List Minimum) *The problem to be solved is stated as follows:*

Input: A list $l = [a_0, \dots, a_{n-1}]$ of natural numbers.

Output: A $x \in l$ such that $x \leq a_i$ for all i .

We assume that n is an integral power of 2 (otherwise extend l by a suitable number of ∞ s). The balanced binary tree technique works as follows: Compare always for $i \leq n/2$ in parallel a_{2i} and a_{2i+1} . Proceed recursively with the minimas obtained by these comparisons. At the end remains just one element which is the minimum. The program is therefore:

```
type nat    \* as usual including a binary min-operation *\

indexed type list(A) \* as in chapter 2 *\

fun list_min(l: list(nat)):nat =
  if empty(cdr(l)) then car(l)
  else
    let l' = forall i < length(l)/2 do in parallel min(l[2*i],l[2*i+1])
    in list_min(l')
end
```

3.1 Deriving the Time Functions

This section deals with the first step in the complexity analysis. It defines a transformation $TE \llbracket \cdot \rrbracket$ on expressions and a transformation $TF \llbracket \cdot \rrbracket$ such that the program Π' defined through Π and the by TF transformed functions compute the time complexity of expressions, i.e. in terms of the semantic functions in chapter 2:

$$EVAL \llbracket TE \llbracket t \rrbracket \rrbracket \Pi' \rho = TIME \llbracket t \rrbracket \Pi \rho$$

for all expressions t and environments ρ . The transformations are given in figure 3.1. For other complexities it is possible to define similar transformations. It is easy but tedious to prove the correctness (i.e. the above formula) of this transformation by structural induction. If we apply the time transformations to example 3.1 we obtain

(0) A program $\Pi = (T, F)$ is transformed into $\Pi' = (T, F \cup F')$ where

$$F' = \{\mathbf{TF} \llbracket \text{def} \rrbracket \mid \text{def} \in F\}$$

(1) For function definitions

$$\mathbf{TF} \llbracket \text{fun } f(x_1 : I_1, \dots, x_n : I_n) : O = B \rrbracket = \text{fun } \text{time_f}(x_1 : I_1, \dots, x_n : I_n) : \text{nat} = \mathbf{TE} \llbracket B \rrbracket$$

(2) For constructor and operation calls:

$$\mathbf{TE} \llbracket c(t_1, \dots, t_n) \rrbracket = \tau_c + \mathbf{TE} \llbracket t_1 \rrbracket + \dots + \mathbf{TE} \llbracket t_n \rrbracket$$

(3) For function calls:

$$\mathbf{TE} \llbracket f(t_1, \dots, t_n) \rrbracket = \tau_{\text{call}} + \mathbf{TE} \llbracket t_1 \rrbracket + \dots + \mathbf{TE} \llbracket t_n \rrbracket + \text{time_f}(t_1, \dots, t_n)$$

(4) For conditionals:

$$\mathbf{TE} \llbracket \text{if } b \text{ then } t_1 \text{ else } t_2 \rrbracket = \text{if } b \text{ then } \tau_{\text{if}} + \mathbf{TE} \llbracket b \rrbracket + \mathbf{TE} \llbracket t_1 \rrbracket \text{ else } \tau_{\text{if}} + \mathbf{TE} \llbracket b \rrbracket + \mathbf{TE} \llbracket t_2 \rrbracket$$

$$\mathbf{TE} \llbracket s_1 = s_2 \rrbracket = \tau_{\text{eq}} + \mathbf{TE} \llbracket s_1 \rrbracket + \mathbf{TE} \llbracket s_2 \rrbracket$$

(5) For local variables:

$$\mathbf{TE} \llbracket \text{let } x = t_1 \text{ in } t_2 \rrbracket = \tau_{\text{let}} + \mathbf{TE} \llbracket t_1 \rrbracket + \mathbf{TE} \llbracket t_2 \rrbracket [t_1 \leftarrow x]$$

(6) For parallel evaluation:

$$\mathbf{TE} \llbracket \text{for all } i < ub \text{ do in parallel } t(i) \rrbracket = \tau_{\text{par}} + \mathbf{TE} \llbracket ub \rrbracket + \max_{i=0}^{ub-1} (\mathbf{TE} \llbracket t(i) \rrbracket)$$

Figure 3.1: Time Transformations

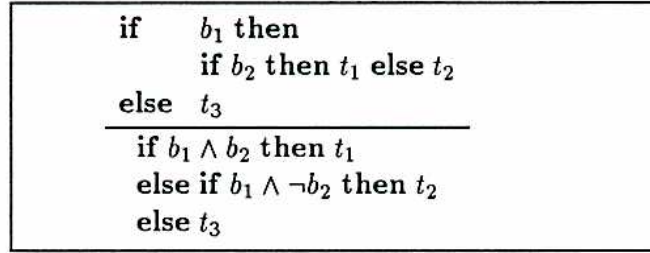


Figure 3.2: Elimination of Nested Conditionals

Example 3.2 (List Minimum, Time Function) *We assume that each basic complexity has the value 1. This results in counting (parallel) reductions for the evaluation.*

```

fun time_list_min(l:list(nat)): nat =
  if empty(cdr(l)) then 6
  else 13 + max(0<=i<length(l)/2,11)
    + time_list_min(for all i < length(l)/2 do in parallel min(l[2*i],l[2*i+1]))

```

This simplifies to

```

fun time_list_min(l:list(nat)) : nat =
  if empty(cdr(l)) then 6
  else 24 + time_list_min(for all ... do in parallel ...)

```

3.2 Normalisation

The normalisation step is an extension of the one described in [Zim90b]. This extension arises from the parallel statement. If no parallel statement is used, it leads to the same result as in [Zim90b]. It consists of the two transformations described in figure 3.2 and figure 3.4, and the new transformation described in figure 3.3.

The elimination of conditionals as an argument of the maximum operation is not covered by the transformation of figure 3.2.

The removal cannot be done by the above transformation, and a simplification is at this stage not yet possible (even if there are some special cases where it is possible). Assume that there is a maximum with a conditional argument:

$$\max_{i=0}^{ub-1} \text{if } b(i) \text{ then } t_1(i) \text{ else } t_2(i)$$

Then the time complexity of the overall statement is the maximum of the complexities $t_1(i)$ in the then-part and the time complexities $t_2(i)$ in the else-part (remember that we consider time

$\frac{\max_{i=0}^{ub-1} \text{if } b(i) \text{ then } t_1(i) \text{ else } t_2(i)}{\max_{i=0}^{ub-1} f'(i, x_1, \dots, x_k)}$ <p style="text-align: center;"> fun $f'(i, x_1, \dots, x_k) =$ if $b(i)$ then $t_1(i)$ else $t_2(i)$ </p> <p>where x_1, \dots, x_k are free variables in $b(i)$, $t_1(i)$, and $t_2(i)$, and f' is a new function symbol.</p>
--

Figure 3.3: Eliminating Conditionals in Parallel Statements

computing function here). This simplification is always possible for computing an upper bound, but it cannot be done before the functions used in $t_1(i)$ and $t_2(i)$ have been analyzed. For the average case the analysis is even more complicated. If the complexity of the conditional is not constant, then there is one term $t_1(i)$ or $t_2(i)$ dominating for large input sizes. Then it is necessary to compute the probability that $b(i)$ becomes true and false, respectively, for all i .

But if a conditional argument in a maximum is left as it is, the preconditions of subsequent analysis steps would not be satisfied anymore (symbolic evaluation). We therefore introduce a new function definition for the body of such parallel constructs (figure 3.3)

In the example, no normalisation as described in figures 3.2 and 3.3 is necessary.

The final substep is the elimination of irrelevant argument positions. These are defined as in [Zim90b]:

Definition 3.3 (Irrelevant Argument Positions) *Let $f(x_1, \dots, x_k)$ be a function in a program Π .*

- (a) *The argument position f/i is called irrelevant, iff for all arguments $t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_k$ for the formal parameters $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_k$ and arguments t, t' for the argument x_i holds:*

$$f(t_1, \dots, t_{i-1}, t, t_{i+1}, \dots, t_k) = f(t_1, \dots, t_{i-1}, t', t_{i+1}, \dots, t_k)$$

Otherwise, the argument position f/i is called relevant.

- (b) *An argument position f/i occurs in a term t , if the formal parameter x_i occurs in t .*

If an argument position is irrelevant it can be removed safely from function definitions and function calls. These transformations are described in figure 3.4 and are exactly the same as in [Zim90b].

It is a well-known result from computability theory that it is undecidable whether an argument position is irrelevant or not. There is a closure algorithm [Zim90b] which determines relevant positions. If an argument position is not marked as relevant, then it is an irrelevant argument position, but not every argument position relevant as marked need necessarily be relevant. It could be in fact irrelevant. The algorithm is described in figure 3.5. It holds:

If f/i is an irrelevant argument position, then:

1. elimination in function definitions:

$$\frac{\text{fun } f(x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_k) = B}{\text{fun } f(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_k) = B}$$

2. elimination in function applications:

$$\frac{f(t_1, \dots, t_{i-1}, t_i, t_{i+1}, \dots, t_k)}{f(t_1, \dots, t_{i-1}, t_{i+1}, \dots, t_k)}$$

Figure 3.4: Elimination of Irrelevant Argument Positions

Input: A program $\Pi = (T, F)$.

Output: A set of irrelevant argument positions

Algorithm:

1. [Initialisierung]: $R := \emptyset$

2. [Closure]

Repeat

$$R := R \cup \{f/i \mid f/i \text{ occurs in a non-recursive expression}\} \\ \cup \{f/i \mid f/i \text{ occurs in a condition}\} \\ \cup \{f/i \mid f/i \text{ occurs in an argument on a position } g/j \in R\}$$

until there is no change in R

3. [Output]

$$\{f/i \mid \text{fun } f(x_1, \dots, x_k) = B \in F, 1 \leq i \leq k, f \notin R\}$$

Figure 3.5: Finding Irrelevant Argument Positions

Theorem 3.4 (Soundness of Algorithm 3.5) *Each argument position found by the algorithm in figure 3.5 is irrelevant.*

Proof: [Zim90b]

In our example, the argument position `time_list_min/1` is relevant, because it occurs in the condition `empty(cdr(1))`. Hence no transformation is applicable, and the program is not changed by the normalisation step.

3.3 Symbolic Evaluation

In this step the time functions are transformed into a set of equations specifying the time complexity. This transformation is based on symbolic substitutions for the formal parameter (yielding an LHS of an equation), and symbolic evaluation of the body with this argument (yielding the corresponding RHS of the equation). The transformation is described in figure 3.6. This mainly the same as the symbolic evaluation step as in [Weg75] or [Zim90b]. The substitutions must be *complete*, i.e. together they must describe the whole type (assuming that variables can be substituted arbitrarily). As shown in [Zim90b] the suitable substitutions can be found by a narrowing procedure described in [Ech88]. If we perform this step in our example we would obtain $l = \text{cons}(a, \text{nil})$ and $l = \text{cons}(a, \text{cons}(b, l))$ and:

$$\begin{aligned} \text{time_list_min}(\text{cons}(a, \text{nil})) &= 6 \\ \text{time_list_min}(\text{cons}(a, \text{cons}(b, l))) &= 24 + \text{time_list_min}(\text{for all } i < \text{length}(l)/2 + 1 \\ &\quad \text{do in parallel} \\ &\quad \min(\text{cons}(a, \text{cons}(b, l))[2\ i], \\ &\quad \text{cons}(a, \text{cons}(b, l))[2\ i + 1])) \end{aligned}$$

If the expressions $\text{cons}(a, \text{cons}(b, l))[2\ i]$ would be simplified, this would lead to a new case distinction ($i = 1$ and $i > 1$). On the other hand if we just use l as the symbolic argument, it would lead to a simpler form, and as we will see later, it is possible to proceed with this simpler form. Hence, if we have a parallel statement in an argument, the symbolic argument will be just the parameter used together with indices. In other words, we don't consider the non-indexed structure of an indexed type, if indices are used in an argument. We therefore obtain in our example:

Example 3.5 (Symbolic Evaluation)

$$\begin{aligned} \text{time_list_min}(\text{cons}(a, \text{nil})) &= 6 \\ \text{time_list_min}(l) &= 24 + \text{time_list_min}(\text{for all } i < \text{length}(l)/2 \\ &\quad \text{do in parallel } \min(l[2\ i], l[2\ i + 1])) \end{aligned}$$

If there remain conditional equations we write

$$LHS = \begin{cases} RHS_1 & \text{if } cond_1 \\ \vdots & \vdots \\ RHS_{k-1} & \text{if } cond_{k-1} \\ RHS_k & \text{otherwise} \end{cases}$$

1. Transforming a function definition to a simple equation:

$$\frac{\text{fun } f(x_1 : T_1, \dots, x_n : T_n) : T = B}{f(x_1, \dots, x_n) = B}$$

2. Transforming equations by symbolic evaluation

$$\frac{f(t_1, \dots, t_n) = \text{if } b(x_1, \dots, x_k) \text{ then } s_1 \text{ else } s_2}{\begin{aligned} f(t_1\sigma_1, \dots, t_n\sigma_1) &= s_1\sigma_1 \\ f(t_1\sigma_2, \dots, t_n\sigma_2) &= s_2\sigma_2 \end{aligned}}$$

where σ_1 is a substitution, such that $b(x_1, \dots, x_k)\sigma_1 = \text{true}$ and σ_2 is a substitution, such that $b(x_1, \dots, x_k)\sigma_2 = \text{false}$, and $(\sigma_1(x_1), \dots, \sigma_1(x_n)), (\sigma_2(x_1), \dots, \sigma_2(x_n))$ define a partition of $T_1 \times \dots \times T_n$.

Figure 3.6: Symbolic Evaluation

instead of

$$\begin{aligned} LHS = & \text{if } cond_1 \text{ then } RHS_1 \\ & \text{else if } cond_2 \text{ then } RHS_2 \\ & \text{else if } \dots \\ & \text{else if } cond_{k-1} \text{ then } RHS_{k-1} \\ & \text{else } RHS_k \end{aligned}$$

At the end of this step is always a set of (possibly conditional) equations describing the time complexity of the program.

3.4 Deriving Recurrences

By applying *length* or *size* onto the arguments of the equations, recurrences can be obtained (figure 3.7). This is always the case by applying *size*, but not always by applying *length*. However there are necessary and sufficient criteria when *length* can be applied [Zim90b]. Another easy way is just to apply *length* to an argument and see whether a recurrence is obtained or not. If a recurrence is obtained by applying *length*, then the time complexity depends on the *length* of the corresponding argument, otherwise it depends on the *size*. Compared to the sequential case just two enhancements are necessary:

$$\text{length}(\text{for all } i < ub \text{ do in parallel } t(i)) = ub$$

$$\text{size}(\text{for all } i < ub \text{ do in parallel } t(i)) = \sum_{i=0}^{ub-1} \text{size}(t(i))$$

The following three transformations are applied in the corresponding order. Each of these transformations is applied as long as possible.

1. Applying one $M \in \{length, size\}$ to argument position f/i on the LHS of equations:

$$\frac{f(t_1, \dots, t_{i-1}, t_i, t_{i+1}, \dots, t_n) = RHS}{f(t_1, \dots, t_{i-1}, M(t_i), t_{i+1}, \dots, t_n) = RHS}$$

2. Applying this $M \in \{length, size\}$ to the same argument position f/i on subterms of the RHS of equations:

$$\frac{f(t_1, \dots, t_{i-1}, t_i, t_{i+1}, \dots, t_n)}{f(t_1, \dots, t_{i-1}, M(t_i), t_{i+1}, \dots, t_n)}$$

3. Introducing new variables (over naturals):

$$\frac{LHS = \begin{cases} RHS_1 & \text{if } cond_1 \\ \vdots & \vdots \\ RHS_{k-1} & \text{if } cond_{k-1} \\ RHS_k & \text{otherwise} \end{cases}}{LHS \sigma = \begin{cases} RHS_1 \sigma & \text{if } cond_1 \\ \vdots & \vdots \\ RHS_{k-1} \sigma & \text{if } cond_{k-1} \\ RHS_k \sigma & \text{otherwise} \end{cases}}$$

where $\sigma = [n_i \leftarrow M(x_i) | x_i \text{ is a variable}]$. $k = 1$ is possible, in this case the equation is unconditional.

Figure 3.7: Transformation to Recurrences

Example 3.6 (Creating Recurrences) *We apply the length mapping to the equation and with the substitution $[n = \text{length}(l)]$ get the recurrence:*

$$\begin{aligned} \text{time_list_min}(1) &= 6 \\ \text{time_list_min}(n) &= 24 + \text{time_list_min}(n/2) \end{aligned}$$

which has the solution:

$$\text{time_list_min}(n) = 6 + 24 \log_2 n$$

This is a linear geometric recurrence with constant coefficients. Such kinds of recurrences are very often obtained in the analysis of parallel algorithms. The reason is the division of the input data and their assignment to processors. Let the input size be n . Then, by applying the divide-and-conquer technique, the complexity of the program is computed from the complexity of the merge step and the complexity of the p subproblems of size n/p . By applying the balanced binary tree technique or the compression technique, the complexity depends mainly on solving a problem of size $n/2$.

Sometimes it is necessary to analyze the output length or output size of certain functions. This is the case if the second transformation creates terms $M(f(t_1, \dots, t_k))$ where f is function in the program to be analyzed. In this case the output M of f has to be analyzed. It is just necessary to change the first step, in order to obtain M computing functions. All the other steps are the same. We can therefore substitute $M_f(t_1, \dots, t_n)$ for $M(f(t_1, \dots, t_k))$. It is sometimes possible that such new analyses must be done in analyzing the output M of f . If already considered, then it can be substituted as above. Otherwise, the entire analysis process has to be invoked again. Nevertheless, the analysis terminates at this step, because $M \in \{\text{length}, \text{size}\}$, and there are finitely many functions in a program.

At this step, a complete system of recurrences is obtained. These recurrences could be conditional recurrences (derived from conditional equations) or recurrence families. In [Zim90b] it is described how to solve such a recurrence system, and how for conditional recurrences and recurrence families lower and upper bounds on the solutions as well as the average solution can be determined. We do investigate solution methods of recurrences in particular examples. The power of automatic complexity analysis methods depends on the power of the recurrence solver used there. It makes therefore sense to consider classes of recurrences, obtained by the analysis of program examples.

It should be clear, that the process of creating recurrences always terminates. The quality of the recurrences depends on the symbolic evaluation step (how many conditions can be solved by symbolic evaluation), but this does not affect the termination.

3.5 Example: Prefix Sums

Many parallel algorithms are based on this example [KR88, GR88]. The automatic analysis of this method proves therefore the usefulness of the method. The problem is stated as follows:

Input: A list $[a_0, \dots, a_{n-1}]$ of naturals.

Output: A list $[s_0, \dots, s_{n-1}]$ where $s_j = \sum_{i=0}^j a_i$

This problem can be easily generalized to lists over arbitrary types T and associative operators \oplus over T :

Input: A list $[a_0, \dots, a_{n-1}]$ over a type T .

Output: A list $[s_0, \dots, s_{n-1}]$ where $s_j = \bigoplus_{i=0}^j a_i$

In this section, however, we consider just the list over naturals. It should be clear that the analysis for the generalized prefix sums is the same as the analysis over naturals.

The parallel algorithm is based on the balanced binary tree technique. We assume that n is a power of 2. Otherwise extend the list by a suitable number of zeros. First, the pairwise sums $a_{2i} + a_{2i+1}$ are computed in parallel for all $0 \leq i < n/2$. Second, the partial sums of $[a_{2i} + a_{2i+1} | 0 \leq i < n/2]$ are computed. Let the result of this step be $[s_i | 0 \leq i < n/2]$. Observe that $s_i = a_0 + \dots + a_{2i+1}$. Hence, for odd j we have the partial sum already computed: it is $s_{\lfloor j/2 \rfloor}$. For the even numbers, we have just to subtract a_{j+1} from $s_{j/2}$. This adjustment is therefore the final step in the computation of the prefix sums. It can be done in parallel for all $0 \leq j < n$. The PARSTYFL-program looks as follows:

```
indexed type list(A)  \* compare example 2.2 *\

fun partial_sums(l:list(nat)):list(nat) =
  if empty(cdr(l)) then car(l)
  else let n = length(l) in
    let l1 = for all j < n/2 do in parallel l[2*j] + l[2*j+1] in
    let l2 = partial_sums(l1) in
    for all j < n do in parallel
      if odd(j) then l2[(j-1)/2] else l2[j/2] - l[j+1]
```

In the analysis of the time complexity, we assume that each basic complexity has the value 1.

Step 1: Derivation of the time functions.

Applying the translations of figure 3.1 yields after subsequent arithmetic simplifications:

```
fun time_partial_sums(l:list(nat)):nat =
  if empty(cdr(l)) then 6
  else time_partial_sums(for all j < length(l)/2 do in parallel l[2*j]+l[2*j+1])
    + max(0<=j<length(l),if odd(j) then 10 else 14) + 32
```

Step 2: Normalization

The transformations of figures 3.2 and 3.4 are not applicable (`time_partial_sums/1` is according to the algorithm in figure 3.5 a relevant argument position). The transformation in figure 3.3, however, is applicable, because a conditional statement is an argument of a max-operation. Hence, the following program is obtained:

```
fun time_partial_sums(l:list(nat)):nat =  
  if empty(cdr(l)) then 6  
  else time_partial_sums(for all j < length(l)/2 do in parallel l[2*j]+l[2*j+1])  
    + max(0<=j<length(l),f1(j)) + 32  
  
fun f1(j:nat):nat = if odd(j) then 10 else 14
```

Step 3: Symbolic Evaluation.

In this step, two equations for `time_partial_sums` and `f1` are created, respectively. The first one, considering the then parts are obtained by the substitutions $[l \leftarrow \text{cons}(a, \text{nil})]$ and $[j \leftarrow 2j+1]$. In the second equation for `time_partial_sums` the parameter l is left as it is, because the argument in the recursive call is a parallel statement. The second equation for `f1` is obtained by the substitution $[j \leftarrow 2j]$:

$$\begin{aligned} \text{time_partial_sums}(\text{cons}(a, \text{nil})) &= 6 \\ \text{time_partial_sums}(l) &= \text{time_partial_sums}(\text{for all } \dots \text{do in parallel } \dots) \\ &\quad + \max_{j=0}^{\text{length}(l)-1} f_1(j) + 32 \\ &\quad \text{where } \text{length}(l) > 1 \\ f_1(2j+1) &= 10 \\ f_1(2j) &= 14 \end{aligned}$$

Step 4: Mapping onto Naturals.

This is only necessary for `time_partial_sums`, because `f1` has as its argument already a natural. The function `time_partial_sums` has as its argument in the recursive call a parallel statement. Hence, the mapping `length` is chosen. By applying the transformations in figure 3.7, the following recurrence is obtained:

$$\begin{aligned} \text{time_partial_sums}(1) &= 6 \\ \text{time_partial_sums}(n) &= \text{time_partial_sums}(n/2) + \max_{j=0}^{n-1} f_1(j) + 32 \\ &\quad \text{where } n > 1 \\ f_1(2j+1) &= 10 \\ f_1(2j) &= 14 \end{aligned}$$

Step 5: Solving the Recurrence.

In a first step, the maximum term has to be evaluated. In this example, the evaluation is easy, although in general the evaluation of maximums within recurrences could arise serious problems. We know here that each natural is either odd or even. Therefore the $\max_{j=0}^{n-1} f_1(j)$ evaluates to 14.

More precisely, it has to be proven that the proposition $\forall 0 \leq k < n \forall j : k \neq 2j$ is not satisfiable. The contradiction is obtained by the special value $j = 0$. In general we make use of theorem proving at this step. We will see in further examples what to do if such a kind of propositions can be satisfied. We get after these considerations the recurrence:

$$\begin{aligned} \text{time_partial_sums}(1) &= 6 \\ \text{time_partial_sums}(n) &= 46 + \text{time_partial_sums}(n/2) \\ &\text{where } n > 1 \end{aligned}$$

which has the solution (obtained by standard techniques for geometric recurrences):

$$\text{time_partial_sums}(n) = 6 + 46 \log_2 n$$

In fact, this is the time complexity we expected (and compatible to the well-known $O(\log n)$ results in algorithm books [GR88, KR88]).

3.6 Polynomial Evaluation

This is an example of an algorithm designed according to the divide-and-conquer technique. While in the balanced-binary-tree technique the computation tree is processed from the bottom to the top, in the divide-and-conquer technique it is processed from top to the bottom. It is therefore no surprise that nearly the same analysis technique can be applied to the divide-and-conquer technique. As we will see in this section, only the normalisation step need to be enhanced by a transformation. The problem of polynomial evaluation is stated as follows:

Input: A polynom $p(x) = a_d x^d + \dots + a_1 x + a_0$ and a value x_0
Output: The value $p(x_0)$

The algorithm is based on the fact that the polynomial can be evaluated by $p(x) = r(x) + x^{(d+1)/2} q(x)$, where $r(x)$ and $q(x)$ are polynomials of degree $(d+1)/2 - 1$. We assume without loss of generality, that $d = 2^k - 1$ for some natural k (otherwise enhance the polynomial by the suitable number of zero coefficients). The values $r(x_0)$ and $q(x_0)$ are computed recursively (in parallel). The computation of $p(x_0)$ can be done in constant time, because the $x_0^{(d+1)/2}$ can be computed from the previous powers of x_0 just by squaring. In figure 3.8 a computation tree is shown, making the computations clear. The polynomial is represented by the list of its coefficients: $[a_0, \dots, a_d]$. For computing the intermediate powers of x_0 and the polynomial evaluation of lower degree, it is necessary to introduce a function `poly1` performing both computations at the same node. The output of this function is a pair of values. The first component is the value of a polynomial of degree d , evaluated at x_0 , the second component is the value x_0^d . Thus the algorithm is performed by the following program:

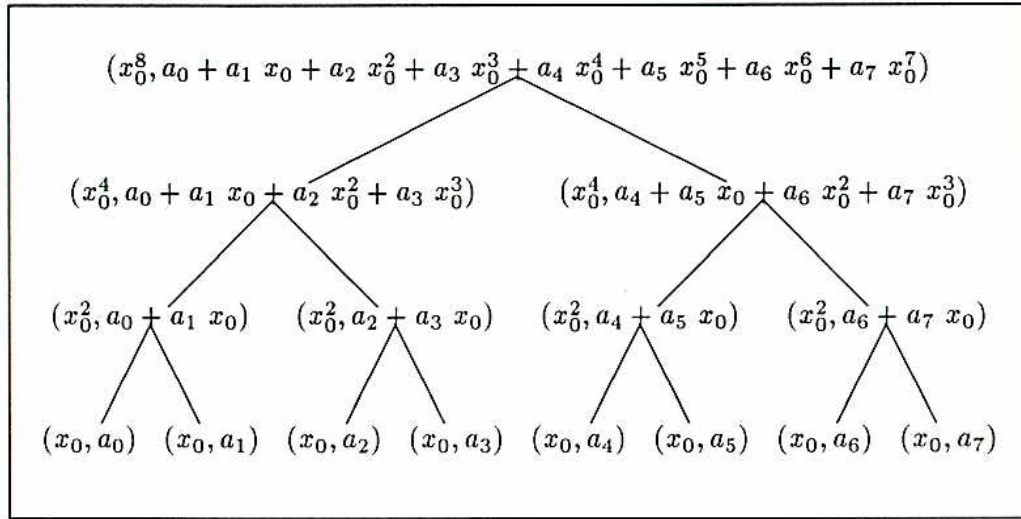


Figure 3.8: Computation Tree for Polynomial Evaluation

```

indexed type list(A)  \* see example 2.2 *\

indexed type pair(A,B)
sorts A,B
constructors
  (.,.): A x B -> pair(A,B)
operations
  first: pair(A,B) -> A;
  sec: pair(A,B) -> B;
variables a:A, b:B
equations
  first((a,b)) = a;
  second((a,b)) = b
indexing
  length: pair(A,B) -> nat;
  length((a,b)) = 2;
  ((a,b))[0] = a;
  ((a,b))[1] = b

fun poly(p:list(nat),x:nat):nat = first(poly1(p,x))

fun poly1(p:list(nat),x:nat):pair(nat,nat) =
  if empty(cdr(p)) then (car(p),x)
  else let l = split(p) in
    let y = for all i<2 do in parallel poly1(l[i],x) in
      (first(y[0])+second(y[1])*first(y[1]),second(y[1])*second(y[1]))

```

```

fun split(p:list(nat)):pair(list(nat),list(nat)) =
  for all i<2 do in parallel
    for all j < length(p)/2 do in parallel p[j+i*length(p)/2]

```

1st Step: Translation to Time Functions

The transformation of figure 3.1 compiles the above program into:

```

fun time_poly(p:list(nat),x:nat):nat = 4 + time_poly1(p,x)

fun time_poly1(p:list(nat),x:nat):nat =
  if empty(cdr(p)) then 8
  else 41 + max(0<=i<2,time_poly1(split(p)[i],x) + time_split(p)

fun time_split(p:list(nat)):nat = 20

```

2nd Step: Normalisation

There are no nested conditionals in the program. Hence the transformation of figure 3.2 is not applicable. It turns out – by the algorithm of figure 3.5 – that `time_split/1`, `time_poly1/2`, and `time_poly/2` are irrelevant argument positions. Thus their elimination by the transformations in figure 3.4 yields:

```

fun time_poly(p:list(nat)):nat = 4 + time_poly1(p)

fun time_poly1(p:list(nat)):nat =
  if empty(cdr(p)) then 8
  else 41 + max(0<=i<2,time_poly1(split(p)[i])) + time_split()

fun time_split():nat = 20

```

No conditional statement is an argument of the maximum operation. Therefore the transformation of figure 3.3 is not applicable.

In the fourth step, a suitable mapping $M : list(N) \mapsto N$ for `time_poly1/1` has to be found. For this purpose the term `split(p)[i]` is not useful in this form, because $M(split(p)[i])$ has to be evaluated, and there is no definition for this evaluation. In fact, a function computing $M(split(p)[i])$ must be obtained in that step. This becomes easier if `split(p)[i]` is folded to a function `access_split(i,p)`, because then the same techniques as for $M(f(x))$ where f is a function can be applied. The function `access_split(i,p)` is obtained from `split(p)` by performing $(\cdot)[i]$ on each possible output. The whole transformation is described in figure 3.9. Because from a logical point of view, this transformation fits better into normalisation than into mapping onto naturals, it is performed here.

Let **fun** $f(x_1 : I_1, \dots, x_n : I_n) : O = B$ be a function definition in Π (and therefore in Π'), and $o : T_1 \times \dots \times O \times \dots \times T_k \mapsto T$ a type operation (*not* a constructor!). Then the following transformation is performed:

$$\frac{o(t_1, \dots, f(s_1, \dots, s_n), \dots, t_k)}{o_f(y_1, \dots, y_m, s_1, \dots, s_n)}$$

where y_1, \dots, y_m are variables of type S_1, \dots, S_m , respectively, and used in the terms t_1, \dots, t_k . The following function is added to Π' :

fun $o_f(y_1 : S_1, \dots, y_m : S_m, x_1 : I_1, \dots, x_n : I_n) : T = o(t_1, \dots, B, \dots, t_k)$

where the following simplifications are applied to evaluate the body of the new function:

$$\frac{o(t_1, \dots, \text{if } b \text{ then } u_1 \text{ else } u_2, \dots, t_k)}{\text{if } b \text{ then } o(t_1, \dots, u_1, \dots, t_k) \text{ else } o(t_1, \dots, u_2, \dots, t_k)}$$

$$\frac{o(t_1, \dots, f(u_1, \dots, u_n), \dots, t_k)}{o_f(y_1, \dots, y_m, u_1, \dots, u_n)}$$

$$\frac{o(t_1, \dots, \text{let } x = u_1 \text{ in } u_2, \dots, t_k)}{\text{let } x = o(t_1, \dots, u_1, \dots, t_k) \text{ in } x = o(t_1, \dots, u_2, \dots, t_k)}$$

$$\frac{\text{for all } i < ub \text{ do in parallel } t(i))[j]}{t(j)}$$

Figure 3.9: Elimination of Operation Calls

The elimination of operation calls yields:

```

fun time_poly(p:list(nat)):nat = 4 + time_poly1(p)

fun time_poly1(p:list(nat)):nat =
  if empty(cdr(p)) then 8
  else 41 + max(0<=i<2,time_poly1(access_split(i,p)) + time_split())

fun time_split():nat = 20

fun access_split(i:nat,p:list(nat)):list(nat) =
  for all j < length(p)/2 do in parallel p[j+i*length(p)/2]

```

3rd Step: Symbolic Evaluation

The only time function where symbolic evaluation has to be performed is `time_poly1` (with $p \leftarrow \text{cons}(a, \text{nil})$ and $p \leftarrow \text{cons}(b, \text{cons}(a, p))$). Then the following equations are obtained:

$$\begin{aligned}
 \text{time_poly}(p) &= 4 + \text{time_poly1}(p) \\
 \text{time_poly1}(\text{cons}(a, \text{nil})) &= 8 \\
 \text{time_poly1}(\text{cons}(b, \text{cons}(a, p))) &= 41 + \text{time_split} + \\
 &\quad + \max_{i=0}^1 \text{time_poly1}(\text{access_split}(i, \text{cons}(b, \text{cons}(a, p)))) \\
 \text{time_split} &= 20
 \end{aligned}$$

4th Step: Mapping onto Naturals

The mapping `length` is chosen for `time_poly1/1`, because the body of `access_split` can be evaluated under `length` (i.e. it depends on the `length` of its second parameter). The choice of `length` for `time_poly1/1` implies the choice of `length` for `time_poly/1`. Hence, the following equations are obtained ($n = \text{length}(p)$):

$$\begin{aligned}
 \text{time_poly}(n) &= 4 + \text{time_poly1}(n) \\
 \text{time_poly1}(1) &= 8 \\
 \text{time_poly1}(n + 2) &= 41 + \text{time_split} + \\
 &\quad + \max_{i=0}^1 \text{time_poly1}(\text{length}(\text{access_split}(i, \text{cons}(b, \text{cons}(a, p))))) \\
 \text{time_split} &= 20
 \end{aligned}$$

Thus the `length` of `access_split(i, p)` has to be analyzed:

Step 4.1: Translation into `length` Functions

By the application of a similar transformation as in figure 3.1 on `access_split` yields:

```

fun length_access_split(i:nat,p:list(nat)):nat = length(p)/2

```

Step 4.2: Normalization

The algorithm in figure 3.5 detects `length_access_split/1` as irrelevant. Other transformations are not applicable:

```
fun length_access_split(p:list(nat)):nat = length(p)/2
```

Step 4.3: Symbolic Evaluation

There is no condition in the body of `length_access_split`. Hence the following equation is obtained:

$$\text{length_access_split}(p) = \text{length}(p)/2$$

Step 4.4: Mapping onto Naturals

It is obvious to choose `length` for `length_access_split/1`, because the RHS already contains `length(p)`. Thus, the following equation is obtained ($n = \text{length}(p)$):

$$\text{length_access_split}(n) = n/2$$

Using this result, we get the following recurrence system:

$$\begin{aligned} \text{time_poly}(n) &= 4 + \text{time_poly1}(n) \\ \text{time_poly1}(1) &= 8 \\ \text{time_poly1}(n+2) &= 41 + \text{time_split} + \max_{i=0}^1 \text{time_poly1}(\text{length_access_split}(n+2)) \\ \text{time_split} &= 20 \\ \text{length_access_split}(n) &= n/2 \end{aligned}$$

5th Step: Solving the Recurrence System

First the third equation is simplified to:

$$\begin{aligned} \text{time_poly1}(n) &= 41 + \text{time_split} + \text{time_poly1}(\text{length_access_split}(n)) \\ &\text{where } n \geq 2 \end{aligned}$$

Second, the non-recursive equations are unfolded and eliminated:

$$\begin{aligned} \text{time_poly}(n) &= 4 + \text{time_poly1}(n) \\ \text{time_poly1}(1) &= 8 \\ \text{time_poly1}(n) &= 61 + \text{time_poly1}(n/2) \end{aligned}$$

This recurrence can be solved by standard solution method for linear geometric recurrences with constant coefficients:

$$\begin{aligned} \text{time_poly1}(n) &= 8 + 61 \log_2 n \\ \text{time_poly}(n) &= 12 + 61 \log_2 n \end{aligned}$$

Chapter 4

Algorithms based on Pointer Jumping

The pointer jumping technique is a often used design principle for parallel algorithms. Especially for algorithms in asynchronous machine models [CZ90], it is necessary to use this technique.

4.1 Introduction to Pointer Jumping

Because arrays (indexed types) are used, pointers are introduced explicitly (they are also implicitly present in non-indexed types, but we need not to care about them, because of the term representation). Assume there is an object l of an indexed type T . Then an indexed list p of length $\text{length}(l)$ can be used to introduce additional pointers in l , i.e. if $p[i] = j$ then a pointer from $l[i]$ to $l[j]$ is introduced. We therefore call such arrays *pointer arrays*. The range of the array must be $\{0, \dots, \text{length}(l)\}$. If graphs based on p are constructed ((i, j) is an edge iff $p[i] = j$) then each node of this graph will have outdegree 1. Linear lists can be represented by letting point the last element to itself. The main operation of pointer jumping is :

```
fun double( $p : \text{List}(\text{nat})$ ) :  $\text{List}(\text{nat})$  =  
  for all  $i < \text{length}(p)$  do in parallel  
    if  $p[i] \neq p[p[i]]$  then  $p[p[i]]$  else  $p[i]$ 
```

If EREW-PRAMs are used to compute this operation, then the indegree of nodes i , where $p[i] \neq p[p[i]]$ is at most 1, otherwise read conflicts can occur. The graph defined by p consists therefore only of components forming a single loop or linear lists pointing to a fixpoint (i.e. $i = p[i]$). It is possible that more than one list points to a fixpoint. The technique is called pointer jumping, because if the end of a list is not yet reached, then it jumps to the position following the successor of i . In figure 4.1 the effect of successive applications of *double* on a list is shown.

After three operations each element has as its successor the last element. The pointer jumping technique is based on the fact that $\log \text{length}(p)$ repeated calls create this situation. Unfortunately

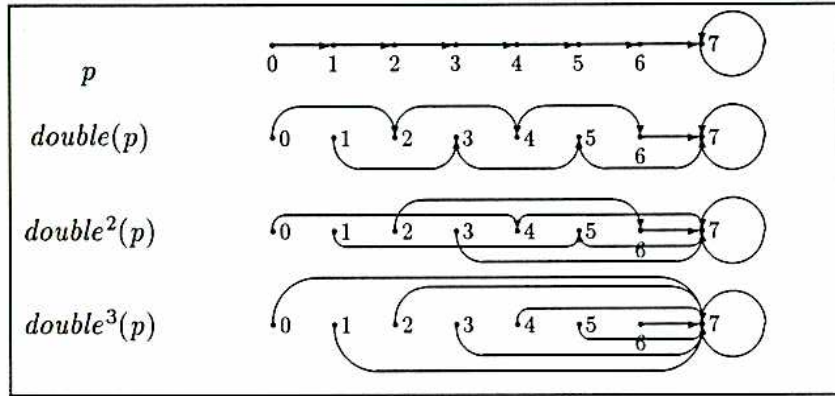


Figure 4.1: Pointer Jumping on Lists

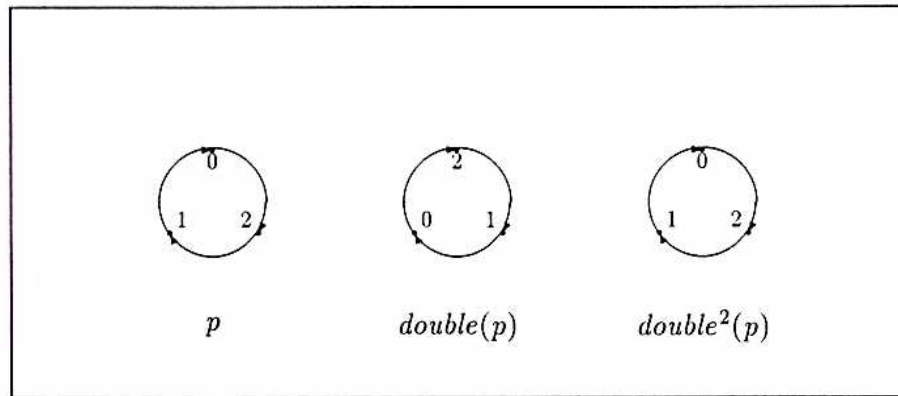


Figure 4.2: Pointer Jumping on Odd Circles

this is not always the case, consider for example figure 4.2. In general, a circle containing an odd number of nodes jumps into a circle containing an odd number of nodes. On the other hand the pointer jumping technique works well for circles with 2^k nodes (see in figure 4.3 for a circle with four nodes).

Situations like odd cycles must be excluded from pointer jumping. If pointer jumping is applied to a proper pointer array p and the application program uses the condition $p[i] \neq p[p[i]]$, then it must be examined how long it must be iterated until for all i holds $p[i] = p[p[i]]$. In programs based on pointer jumping such situations usually never occur, because unnecessary applications of *double* would be done. In an automatic complexity analysis system, however, we have to take into account this possibility. It is therefore necessary to compute the number of necessary applications of *double*. If applied to a proper pointer array p , after $k = \log \text{length}(p)$ calls of *double* it always holds $p[i] = p[p[i]]$ for all i . On the other hand, if pointer jumping is applied to an improper pointer array p , there is always a j where $p[j] \neq p[p[j]]$. Unfortunately, in the proper case, it could hold $p[i] = p[p[i]]$ for all i even earlier, see for example figure 4.4. In general only $\log k$ iterations are needed to reach the final situation, where k is the length of the longest chain (or longest circle) in p .

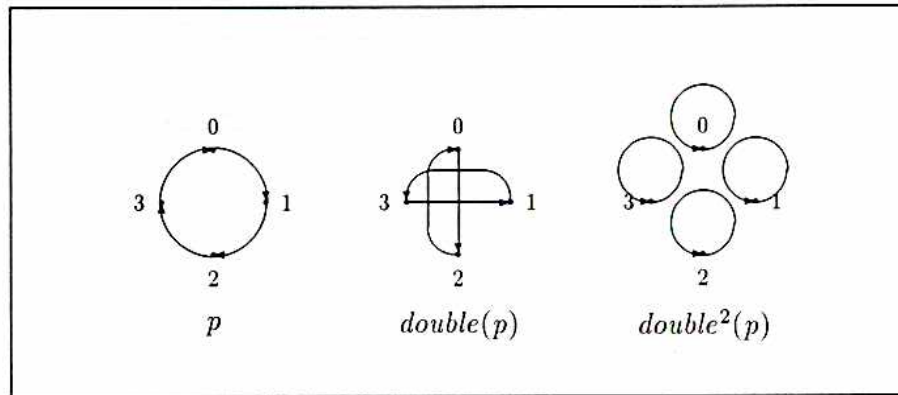


Figure 4.3: Pointer Jumping on Even Circles

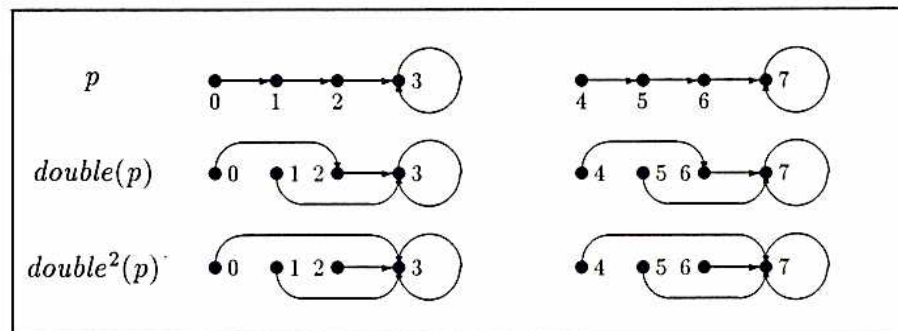


Figure 4.4: Pointer Jumping on Disconnected Lists

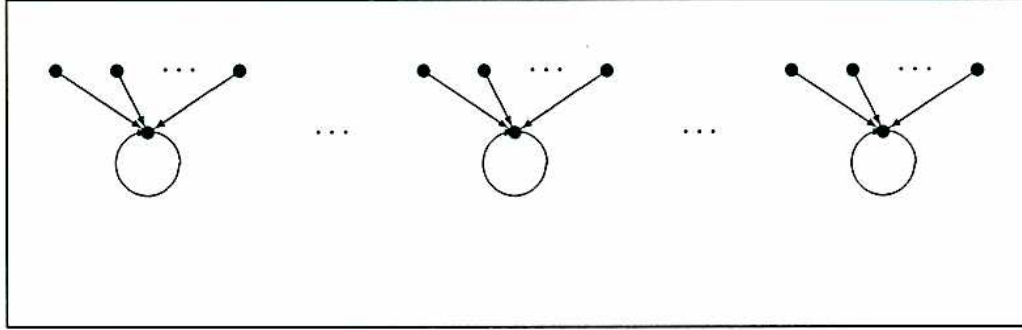


Figure 4.5: The Final Situation in Pointer Jumping

The outline of this chapter is as follows: First, a necessary and sufficient condition on the pointer array p is defined, such that pointer jumping can be applied properly and this condition is maintained by $double(p)$. It is therefore just necessary to check this condition on the initial pointer array. In the second section, it is shown how the information derived by the first section can be used to analyze the complexity of algorithms based on pointer jumping. In the third section, we show the analysis of an algorithm, where pointer jumping is not based on the condition $p[i] \neq p[p[i]]$ and there is in fact one case where no jumps are made.

4.2 Proper Pointer Jumping

Recalling the remarks in the introduction of this chapter, proper pointer jumping is based on the fact that the pointer array p defines a set of lists and circles of length 2^k for naturals k . These lists can have the same final element, but other elements are not shared. If these elements would be shared, then concurrent reads would be necessary for performing $double(p)$. Thus, it is easy to generalize this section for CREW-PRAMs or CRCW-PRAMs. The results are very similar to the results in this section. After each application of $double$, the length of the lists and circles are halved, their number is doubled. The end of a list is given by an index satisfying $i = p[i]$. These indices are called *fixpoints*. The pointer jumping is ready, if for all i holds $p[i] = p[p[i]]$, i.e. each i is either a fixpoint or points to a fixpoint. The final situation is shown in figure 4.5.

The goal is now to find a condition $C(p)$ ensuring on the one hand that $double(p)$ halves the length of each list and cycle, and on the other hand ensures that $C(double(p))$ holds. Then it is easy to see that $\lceil \log_2 \max(l, m) \rceil$ (l is the maximal length of a list, and m the maximal length of cycle) iterations on a initial pointer array p are needed for reaching the final situation. This property and the above notions are now defined formally. The required properties will be proven.

Definition 4.1 (Final Situation) A pointer array p is in the final situation iff for all $0 \leq i < \text{length}(p)$:

- (i) $i = p[i]$, or
- (ii) $p[i] = p[p[i]]$

The i in condition (i) is called a fixpoint, and denotes the end of a list.

Corollary 4.2 (Pointer Jumping on the Final Situation) A pointer array p is in the final situation iff $p = \text{double}(p)$.

Proof: Follows directly from definition 4.1 and the definition of *double*. ■

A slight generalization is the following lemma:

Lemma 4.3 (Pointer Jumping on Predecessors of Fixpoints) Let p be a pointer array, and $0 \leq i < \text{length}(p)$.

- (a) If $p[i] = p[p[i]]$ then for $p' = \text{double}(p)$ holds also $p'[i] = p'[p'[i]]$.
- (b) If i is a fixpoint in p , then i is also fixpoint in $\text{double}(p)$.

Proof: Follows directly from the definition of *double*. ■

As already discussed pointer arrays define functional graphs:

Definition 4.4 (Graph of a Pointer Array) The graph of a pointer array p is a direct graph $G(p) = (V, E)$ defined by:

$$\begin{aligned} V &:= \{0, \dots, \text{length}(p)\} \\ E &:= \{(i, p[i]) \mid i \in V\} \end{aligned}$$

Corollary 4.5 The graph $G(p)$ of a pointer array p is functional, i.e.

- (i) If $(i, j) \in E$ and $(i, k) \in E$ then $j = k$, and
- (ii) For each $i \in V$ there is a $j \in V$, such that $(i, j) \in E$

The following definition characterizes the required properties for pointer jumping on EREW-PRAMs:

Definition 4.6 (Condition on Proper Pointer Jumping) A pointer array p satisfies the condition on proper pointer jumping iff each connected component of $G(p)$ satisfies the following properties:

- (i) It is a cycle of length 2^k for a natural k , or
- (ii) it contains a cycle of length 1, and for all i, j holds:

$$i \neq j \wedge p[i] = p[j] \Rightarrow p[i] = p[p[i]] \quad (*)$$

Remark 4.7 (Pointer Jumping with Concurrent Reads) The condition $(*)$ is needed because concurrent reads have to be avoided. If concurrent reads are allowed, then this condition is not necessary. It is also allowed that arbitrary non-cyclic graphs are connected with cycles of length 2^k , if the graph is still functional. However, other cycles are not allowed, because the behaviour for odd cycles (figure 4.2) doesn't change. Hence, if concurrent reads are allowed the requirements on the graph are:

- (i) It contains a cycle, and
- (ii) each cycle has the length 2^k for a natural k

The following lemma shows the behaviour of pointer jumping on circles:

Lemma 4.8 (Pointer Jumping on Circles) Let p be a pointer array.

- (a) If $G(p)$ contains a cycle $c = (i_0, i_1, \dots, i_{2m-1})$ of length $2m$, then $G(\text{double}(p))$ contains two cycles $c_1 = (i_0, i_2, \dots, i_{2m-2})$ and $c_2 = (i_1, i_3, \dots, i_{2m-1})$, both of length m .
- (b) If $G(p)$ contains a cycle $c = (i_0, i_1, \dots, i_{2m})$ of length $2m + 1$, then $G(\text{double}(p))$ contains the cycle $c = (i_0, i_2, \dots, i_{2m}, i_1, i_3, \dots, i_{2m-1})$ also of length $2m + 1$.

Proof: Let p be a pointer array.

- (a) Follows directly from the definition of *double* and the fact that always $p[i_j] \neq p[p[i_j]]$ for all $i_j \in c$.
- (b) If $m = 0$ the claim follows from lemma 4.3, otherwise for all $i_j \in c$ holds $p[i_j] \neq p[p[i_j]]$ and then the claim follows directly from the definition of *double*. ■

The next lemma states the behaviour of pointer jumping on lists. This lemma holds for EREW-PRAMs. For PRAMs allowing concurrent reads, a more general lemma is possible.

Lemma 4.9 (Pointer Jumping on Lists) *Let p be a pointer array satisfying the condition on proper pointer jumping. Let¹ $\pi = (i = i_0, \dots, i_{|\pi|-1} = j)$ be a path in $G(p)$ from a vertex i to a fixpoint j . Then $G(\text{double}(p))$ contains the following two paths π_1 and π_2 :*

(i) *If $|\pi| = 2m$ then*

$$\begin{aligned}\pi_1 &= (i_0, i_2, \dots, i_{2m-2}, j) \\ \pi_2 &= (i_1, i_3, \dots, i_{2m-1} = j)\end{aligned}$$

Thus, $|\pi_1| = m + 1$ and $|\pi_2| = m$.

(ii) *If $|\pi| = 2m + 1$ then*

$$\begin{aligned}\pi_1 &= (i_0, i_2, \dots, i_{2m} = j) \\ \pi_2 &= (i_1, i_3, \dots, i_{2m-1}, j)\end{aligned}$$

Thus, $|\pi_1| = m + 1$ and $|\pi_2| = m + 1$.

The lemma is visualized in figure 4.6.

Proof: The claim follows directly from the definition of *double* and the observation that for all $j < |\pi| - 1$ holds $p[i_j] \neq p[p[i_j]]$, and $p[i_{|\pi|-1}] = p[p[i_{|\pi|-1}]]$. ■

Corollary 4.10 *If p is a pointer array satisfying the condition on proper pointer jumping, then each path π in $G(p)$ containing no vertex in a cycle is divided into two paths π_1 and π_2 in $G(\text{double}(p))$ which contain no vertex in cycles and satisfy $|\pi_1| = \lceil |\pi|/2 \rceil$ and $|\pi_2| = \lfloor |\pi|/2 \rfloor$.*

Proof: Delete in lemma 4.9 the last vertex in the paths. ■

Remark 4.11 (Concurrent Reads) *If concurrent reads are allowed, then it is possible to have circles of length > 1 at the end of lists. Thus two further cases occur:*

(iii) *If $|\pi| = 2m$ and $p[j] \neq j$ then:*

$$\begin{aligned}\pi_1 &= (i_0, i_2, \dots, i_{2m-2}, p[j]) \\ \pi_2 &= (i_1, i_3, \dots, i_{2m-1} = j)\end{aligned}$$

(iv) *If $|\pi| = 2m + 1$ and $p[j] \neq j$ then*

$$\begin{aligned}\pi_1 &= (i_0, i_2, \dots, i_{2m} = j) \\ \pi_2 &= (i_1, i_3, \dots, i_{2m-1}, p[j])\end{aligned}$$

The proof of lemma 4.9 makes no use of the fact that π is a part of list. Thus the lemma holds also if concurrent reads are allowed.

¹ $|\pi|$ denotes the length of a path π

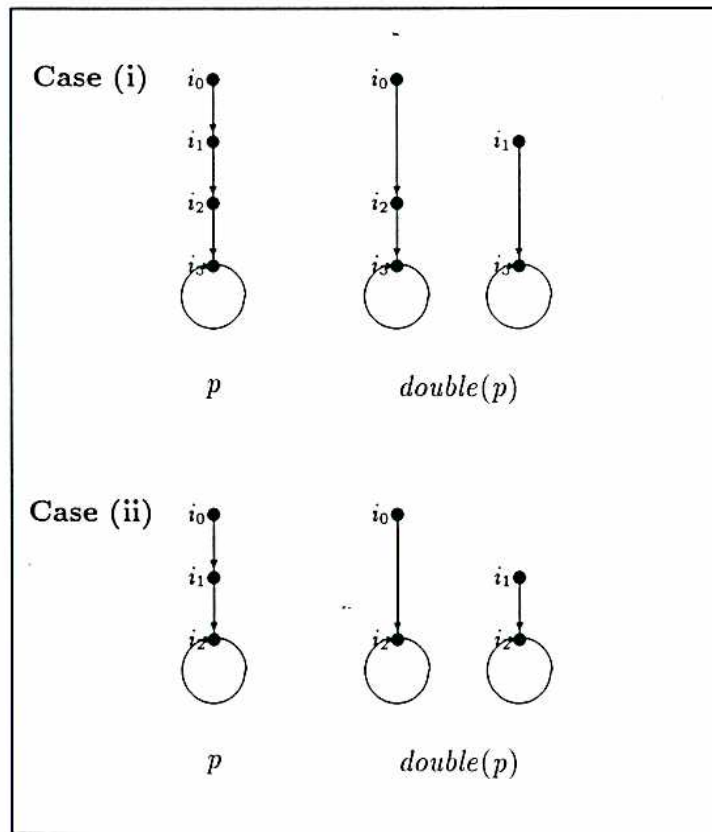


Figure 4.6: The Cases of Lemma 4.9

Now the tools are provided to prove the requirements for proper pointer jumping:

Lemma 4.12 (Maintainance of the Condition on Proper Pointer Jumping) *If p satisfies the condition on proper pointer jumping then $\text{double}(p)$ does.*

Proof: Consider a connected component of $G(p)$. If i is not weakly connected to j in $G(p)$ then it isn't in $G(\text{double}(p))$. Thus it is according to definition 4.6 sufficient to prove:

- (i) Assuming $G(\text{double}(p))$ contains a cycle c not of length 2^k for a natural k . Then, if the length of the cycle is odd by lemma 4.8, $G(p)$ contains either a cycle of the same length, or a cycle of the double length than that in $G(\text{double}(p))$. Both cases are not possible, because in none of them, the cycle has the length 2^k for $k > 0$. This is not possible because p satisfies the condition on proper pointer jumping.

If the length of the cycle is even, then by lemma 4.8 $G(p)$ contains a cycle of the double length. This is also not possible, because this cycle has not a length which is a natural power of 2 and p satisfies the condition on proper pointer jumping.

Thus, the only possible cycles in $G(\text{double}(p))$ have length 2^k for a natural k .

- (ii) Assuming in $p' = \text{double}(p)$ are indices $i \neq j$ where $p'[i] = p'[j]$. Thus, $p[p[i]] = p[p[j]]$. If $p[i] = p[j]$ then $p[i]$ must be a fixpoint and hence $p'[i]$ is a fixpoint by lemma 4.3.

If $p[i] \neq p[j]$ then $p[p[i]]$ must be a fixpoint, because p satisfies the condition on proper pointer jumping. Thus $p'[i]$ is also a fixpoint. Therefore (*) holds. ■

Remark 4.13 (Concurrent Reads) *Lemma 4.12 holds also if concurrent reads are allowed. The proof is the nearly the same, showing that no cycles which are not of length 2^k can be introduced, and that $G(\text{double}(p))$ contains at least one cycle (because the graph is always functional).*

Now we are able to prove the termination property:

Theorem 4.14 (Termination of Pointer Jumping) *Let p be a pointer array satisfying the condition on proper pointer jumping. Let m the length of the longest path in $G(p)$ containing no cycle vertices, and l be the length of the longest cycle in $G(p)$. Then after $n = \max(\lceil \log_2 m \rceil, \log_2 l)$ iterations of double , the resulting pointer array is in the final situation, and with less iterations it isn't. In other words $\text{double}^n(p)$ is in the final situation and for $n \geq 1$ $\text{double}^{n-1}(p)$ is not in the final situation.*

Proof: by induction on n .

BASE CASE: $n = 1$. Then, by the definition of n , $m \leq 2$ and $l \leq 2$ and at least one of these inequalities is also an equality. Thus $p = \text{double}^0(p)$ is not in the final situation.

By lemma 4.8 each cycle of length 2 is divided into two cycles of length 1. Thus every cycle in $G(\text{double}(p))$ has length 1. By lemma 4.9 each path of length 2 containing no cycle vertex is divided by double into two paths of length 1. Thus, $G(\text{double}(p))$ contains only paths of length 1 containing

no cycle vertex. This means that $\text{double}(p)$ is in the final situation.

INDUCTION STEP: $n > 1$. Let $p' = \text{double}(p)$, l the length of the longest cycle in $G(p)$, m the length of the longest path in $G(p)$ containing no cycle vertices and $n = \max(\lceil \log_2 m \rceil, \log_2 l)$. Then, by lemma 4.8, the length of the longest cycle in $G(p')$ is $l/2$, if $l > 1$. If $m > 1$, then the length of the longest path in $G(p')$ containing no cycles is $\lceil m/2 \rceil$. By the definition of n , at least one of the values l and m is greater than 1. Thus:

$$\max\left(\left\lceil \log_2 \left\lceil \frac{m}{2} \right\rceil \right\rceil, \log_2 \frac{l}{2}\right) = \max(\lceil \log_2 m \rceil - 1, \log_2 l - 1) = n - 1$$

By the induction hypothesis, $\text{double}^{n-1}(p') = \text{double}^n(p)$ is in the final situation, and $\text{double}^{n-2}(p') = \text{double}^{n-1}(p)$ is not in the final situation. ■

The next theorem tells the behaviour of pointer jumping if the pointer array does not satisfy the condition on proper pointer jumping:

Theorem 4.15 (Improper Pointer Jumping) *If a pointer array does not satisfy the condition on proper pointer jumping then either read conflicts occur, or for all $n \geq 0$ $\text{double}^n(p)$ is not in the final situation.*

Proof: Assuming no read conflict occurs. Then the pointer array p does not satisfy the condition on proper pointer jumping iff $G(p)$ contains a cycle not of length 2^k , i.e. it contains a cycle c of length $(2m+3)2^k$ for $m, k \geq 0$. First we show by induction on k , that $G(\text{double}^k(p))$ contains a cycle of length $2m+3$. Second we show, that if $G(\text{double}^k(p))$ contains a cycle of length $2m+3$ then $G(\text{double}^{k+n}(p))$ contains a cycle of length $2m+3$ for all $n \geq 0$. This completes the proof, because by the corollary to definition 4.1 $\text{double}^l(p)$ cannot be in the final situation for $l < k$ (otherwise $\text{double}^k(p)$ would be in the final situation), and for $l \geq k$ it is not in the final situation because of the above stated properties.

(i) $G(\text{double}^k(p))$ contains a cycle of length $2m+3$.

BASE CASE: $k = 0$: trivial

INDUCTIVE CASE: $k > 0$: By lemma 4.8 $G(\text{double}(p))$ contains a cycle of length $(2m+3)2^{k-1}$. Thus, by induction hypothesis $G(\text{double}^{k-1}(\text{double}(p))) = G(\text{double}^k(p))$ contains a cycle of length $2m+3$.

(ii) Let $p' = \text{double}^k(p)$. The graph $G(\text{double}^n(p'))$ always contains a cycle of length $2m+3$.

BASE CASE: $n = 0$: proven in (i)

INDUCTIVE CASE: $n > 0$: Let $G(\text{double}^{n-1}(p'))$ containing a cycle of length $2m+3$ (induction hypothesis). Then by lemma 4.8 $G(\text{double}(\text{double}^{n-1}(p'))) = G(\text{double}^n(p'))$ contains a cycle of length $2m+3$. ■

Remark 4.16 (Improper Pointer Jumping with Concurrent Reads) *If concurrent reads are allowed then theorem 4.15 is easily modified: if a pointer array does not satisfy the condition on proper pointer jumping, then there is no iteration of double reaching the final situation. The proof is as the proof of theorem 4.15, just removing the assumption of no read conflicts.*

for all $j < n$ do in parallel <div style="display: flex; justify-content: center; align-items: center;"> <div style="text-align: center;"> if $q[j] \neq q[q[j]]$ then $q[q[j]]$ else $q[j]$ </div> <div style="margin: 0 10px;"> <hr style="width: 100%;"/> </div> <div style="text-align: center;"> <i>double</i>(q) </div> </div>
--

Figure 4.7: Introducing the *double* operation

The tools provided in this section allow to analyze the complexity of parallel algorithms automatically.

4.3 Analysis of Pointer Jumping Algorithms

In this section it is shown how the previous results can be applied to the automatic complexity analysis of algorithms based on the pointer jumping technique. This analysis method enhances the analysis of balanced binary trees, because it makes use of the fact that the program to be analyzed uses pointer jumping. The information is only used in the solution of the recurrences, thus the construction of the recurrences is the same as described in the previous chapter. However, first it is necessary to extract from the program the fact that it is based on pointer jumping. This is done by the application of the transformation of figure 4.7, which is performed after the translation to the time computing functions. The operation *double* is the same as defined in the previous section. If this transformation is applied at least once it is assumed that the program is based on pointer jumping. Then the initial pointer array must be examined whether it satisfies the condition of proper pointer jumping, and the number of iterations of *double* in order to reach the final situation has to be determined (i.e. theorem 4.14 is used). The initial pointer array is explicitly constructed by a program, or the necessary information is given as a comment in the corresponding procedure. The default is that the pointer array describes a linear list, where the last element points to itself. In order to determine whether the condition on proper pointer jumping is satisfied, a theorem prover is called. If it is satisfied, then the number n as described in theorem 4.14 is determined by proving the precondition of this theorem. The corresponding substitution yields the desired number, if it is not already specified explicitly. The extension to the pointer jumping method is discussed by finding the minimum element of a list. It is also an interesting example, because we can compare it with the time complexity of the same problem solved by the balanced binary tree technique (cp. Example 3.1).

Example 4.17 (List Minimum based on Pointer Jumping) *The program based on pointer jumping is:*

```

type nat                \* as usual including the binary min-operation *\

indexed type list(A)    \* as in chapter 2 *\

fun list_min(l:list(nat),p:list(nat)):nat = /*p is the pointer array */
  let l' = repeat(l,p,length(l)) in l'[1]

fun repeat(l:list(nat),p:list(nat),n:nat):list(nat) =
  if n=1 then l
  else let l' = for all i < length(p) do in parallel
    if not p[i]=p[p[i]] then min(l[i],l[p[i]])
    else l[i]
  in repeat(l',double(p),n/2)

fun double(p:list(nat)):list(nat) =
  for all i < length(p) do in parallel
    if not p[i]=p[p[i]] then p[p[i]] else p[i]

```

Hence, the default is assumed, i.e. the final situation is reached after $\log_2 n$ iterations (where $n = \text{length}(l) = \text{length}(p)$).

1st Step: Derivation of the time functions.

```

fun time_list_min(l:list(nat),p:list(nat)):nat = 13 + time_repeat(l,p,length(l))

fun time_repeat(l:list(nat),p:list(nat),n:nat) =
  if n=1 then 6
  else 19 + max(0<=i<length(p),if not p[i]=p[p[i]] then 19 else 13) +
    + time_double(p) + time_repeat(for all ...,double(p),n/2)

fun time_double(p:list(nat)):nat =
  4 + max(0<=i<length(p),if not p[i]=p[p[i]] then 15 else 13)

```

Step 2: Normalization

The transformation of figure 3.2 is not applicable. The application of the transformation of figure 3.3 on the conditionals as argument of the max-operations in the body of time_repeat and time_double yields:

```

fun time_list_min(l:list(nat),p:list(nat)):nat = 13 + time_repeat(l,p,length(l))

```

```

fun time_repeat(l:list(nat),p:list(nat),n:nat) =
  if n=1 then 6
  else 19 + max(0<=i<length(p),time_repeat'(i,p)) + time_double(p)
    + time_repeat(for all ...,double(p),n/2)

fun time_repeat'(i:nat,p:list(nat)) = if not p[i]=p[p[i]] then 19 else 13

fun time_double(p:list(nat)):nat =
  4 + max(0<=i<length(p),time_double'(i,p))

fun time_double'(i:nat,p:list(nat)):nat = if not p[i]=p[p[i]] then 15 else 13

```

The algorithm for finding the irrelevant argument positions (figure 3.5) finds that no argument position is irrelevant, thus the transformation removing irrelevant argument positions is not applicable, and the result of the normalisation step is the above program.

3rd Step: Symbolic Evaluation

The only symbolic evaluation is for the third argument in the function `time_repeat`, because this is the only argument occurring in a condition which can be solved. Thus the following equations are obtained:

$$\begin{aligned}
time_list_min(l, p) &= 13 + time_repeat(l, p, length(p)) \\
time_repeat(l, p, 1) &= 6 \\
time_repeat(l, p, n) &= 19 + \max_{i=0}^{length(p)-1} time_repeat'(i, p) + time_repeat(for\ all\ \dots, double(p), n/2) \\
&\quad + time_double(p) \text{ where } n > 1 \\
time_repeat'(i, p) &= \begin{cases} 19 & \text{if } p[i] \neq p[p[i]] \\ 13 & \text{otherwise} \end{cases} \\
time_double(p) &= 4 + \max_{i=0}^{length(p)-1} time_double'(i, p) \\
time_double'(i, p) &= \begin{cases} 16 & \text{if } p[i] \neq p[p[i]] \\ 13 & \text{otherwise} \end{cases}
\end{aligned}$$

4th Step: Mapping onto Naturals

With the exception of the parameters on positions `time_double'/2`, `time_repeat'/2`, and the ones which are of type `nat`, the mapping `length` is chosen. Thus the following equations are obtained ($k = length(l)$, $m = length(p)$):

$$\begin{aligned}
time_list_min(k, m) &= 13 + time_repeat(k, m, k) \\
time_repeat(k, m, 1) &= 6 \\
time_repeat(k, m, n) &= 19 + \max_{i=0}^{m-1} time_repeat'(i, p) + time_repeat(k, length(double(p)), n/2) \\
&\quad + time_double(m) \text{ where } n > 1
\end{aligned}$$

$$\text{time_repeat}'(i, p) = \begin{cases} 19 & \text{if } p[i] \neq p[p[i]] \\ 13 & \text{otherwise} \end{cases}$$

$$\text{time_double}(m) = 4 + \max_{i=0}^{m-1} \text{time_double}'(i, p)$$

$$\text{time_double}'(i, p) = \begin{cases} 16 & \text{if } p[i] \neq p[p[i]] \\ 13 & \text{otherwise} \end{cases}$$

Now the output length of double has to be analyzed yielding:

$$\text{length_double}(m) = m$$

This equation completes the recurrence system.

5th Step: Solving the Recurrence System.

Until now, the information that the program is based on pointer jumping is not used. This information in fact is just needed in this step. Consider for example the equation defining $\text{time_double}'(i, p)$. In general the following analysis would be obtained:

$$\text{Best Case: } \text{time_double}'(i, p) = 13$$

$$\text{Worst Case: } \text{time_double}'(i, p) = 16$$

$$\text{Average Case: } \text{time_double}'(i, p) = 13 + 3 \text{ Prob}[p[i] \neq p[p[i]]]$$

Using this information in time_double yields:

$$\text{Best Case: } \text{time_double}(m) = 17$$

$$\text{Worst Case: } \text{time_double}(m) = 20$$

$$\text{Average Case: } \text{time_double}(m) = 27 + 3 \text{ Prob}[\forall i : 0 \leq i < m : p[i] \neq p[p[i]]]$$

But this is not the truth, the time complexity of the program behaves deterministically for each input of length n . If λ calls of double are used to reach the final situation, then in the first λ calls there is always a j such that $p[j] \neq p[p[j]]$. Thus in this case it is always $\text{time_double}(m) = 20$. After these λ calls, on the other hand, for all i holds $p[i] = p[p[i]]$. For time_repeat hold similar considerations ($\max_{i=0}^{m-1} \text{time_repeat}'(i, p) = 20$ if less than λ calls of double are performed, otherwise it is 13). Using these results yields the following recurrence, which includes also the number of calls of the operation double:

$$\text{time_repeat}(k, m, 1, \lambda) = 6$$

$$\text{time_repeat}(k, m, n, 0) = 44 + \text{time_repeat}(k, m, n/2, 0)$$

$$\text{time_repeat}(k, m, n, \lambda) = 54 + \text{time_repeat}(k, m, n/2, \lambda - 1)$$

In this recurrence the parameters k and m are irrelevant according to a similar algorithm as in figure 3.5. Thus the solution does not contain these parameters. This recurrence is in fact constructed by two recurrences. The first can be solved directly, if λ is set to 0:

$$\text{time_repeat}(1, 0) = 6$$

$$\text{time_repeat}(n, 0) = 44 + \text{time_repeat}(n/2, 0)$$

This is a recurrence in n which can be solved by standard methods:

$$\text{time_repeat}(n, 0) = 6 + 44 \log_2 n$$

This is now the base case for the second recurrence. The inductive case is:

$$time_repeat(n, \lambda) = 54 + time_repeat(n/2, \lambda - 1)$$

Now, we have a recurrence on two parameters, one behaves like a geometric recurrence, the other like an ordinary recurrence. The only thing that can be done is to look what n would be if $\lambda = 0$. If 0 recursions are done then the value of n is not changed, and for each recursion the value of n is halved, thus the desired value is a solution of the recurrence:

$$\begin{aligned} a(0) &= n \\ a(\lambda) &= 1/2 \cdot a(\lambda - 1) \end{aligned}$$

which has the solution $a(\lambda) = n/2^\lambda$. This solution method can be applied to any linear function of n in the recursion. Now the value of $a(\lambda)$ is substituted in the base case of the recurrence for $time_repeat$, and the parameter n can then be removed. This transformation yields the recurrence

$$\begin{aligned} time_repeat(0) &= 6 + 44 \log_2 n - 44 \lambda \\ time_repeat(\lambda) &= 54 + time_repeat(\lambda - 1) \end{aligned}$$

which has the solution:

$$time_repeat(n, \lambda) = 6 + 44 \log_2 n + 10 \lambda$$

It is now time to make use of the information we got from the fact that the program is based on pointer jumping. The specific value of λ is substituted for λ in the above formula. In this example, it is assumed that the pointer array defines a linear list of length n , therefore by theorem 4.14, the value $\lambda = \log_2 n$ is obtained. Finally:

$$time_repeat(n) = 6 + 54 \log_2 n$$

Therefore the time complexity is:

$$time_list_min(n) = 19 + 54 \log_2 n$$

where n is the length of the arrays l and p .

This example should be discussed. First, if this result is compared with the time complexity of the program for finding the minimum element of a list using the balanced binary tree technique (example 3.6) which was $time_list_min(n) = 6 + 24 \log_2 n$, then it must be concluded that under the assumption that each reduction step costs one time unit, and that the computation is synchronous, it is always preferable to chose the program $list_min$ based on the balanced binary tree technique.

Even more interesting is the comparison of these parallel time complexities to the sequential time complexity. The sequential algorithm is

n	sequential	balanced binary trees	pointer jumping
8	76 reductions	78 reductions	181 reductions
10	96 reductions	102 reductions	235 reductions
11	106 reductions	102 reductions	235 reductions
29	286 reductions	126 reductions	289 reductions
30	296 reductions	126 reductions	289 reductions
64	636 reductions	150 reductions	343 reductions

Table 4.1: Reduction Steps needed for List Minimum Programs

```

fun list_min(l:list(nat)):nat =
  if empty(cdr(l)) then car(l)
  else min(car(l),list_min(cdr(l)))

```

The automatic complexity analysis method for sequential programs leads to $time_min(n) = 10n - 4$ where n is the length of the list l . Table 4.1 shows for some values of n the time needed to perform the corresponding algorithms (if n is not a power of 2 then $\lceil \log_2 n \rceil$ is used). This means that even for $n = 10$ the sequential algorithm is faster than both parallel algorithms technique. The sequential algorithm is even for $n = 29$ faster than the pointer jumping algorithm.

This comparisons gives hints to save processors, i.e. in the case of a balanced binary tree technique, if the list is divided into sublists of 10 elements, and these sublists are computed sequentially the overall running time is still faster than the completely parallel program but it needs just $n/10$ processors (instead of n processors).

4.4 Example: Finding a Sublist of a List

This example shows how pointer jumping can be applied differently from the operation *double*. Instead of the condition $p[i] \neq p[p[i]]$, the program uses another condition for jumping. This program is based on the algorithm for sublist computation in [GR88]. We give in this section also a detailed average case analysis of this algorithm. This analysis leads also to some results in summation of binomial coefficients.

Assuming that three arrays are available, the contents array val , the pointer array p and the label array l . An element $val[i]$ is labeled iff $l[i] = true$, otherwise it is not labeled. In this case is $l[i] = false$. Suppose that p defines a linear list, and f points to the head of the list. The algorithm for sublist computation modifies p and f , such that p and f define the sublist of labeled elements (i.e. they occur in the same order as in the input list defined by p). In the pointer jumping, $p[i]$ is only set to $p[p[i]]$ if $val[i]$ is not labeled. If this jumping operation is repeated $\lceil \log_2 n \rceil$ times, then the only elements of the sublist which are eventually not labeled are the first and the last element in that sublist. This occurs because after iterating pointer jumping f is not changed, and therefore the first element in the sublist is just the first element of the input list. Also the last element is not changed because $p[i] = i$. Thus there must some computation be done after iterating the pointer

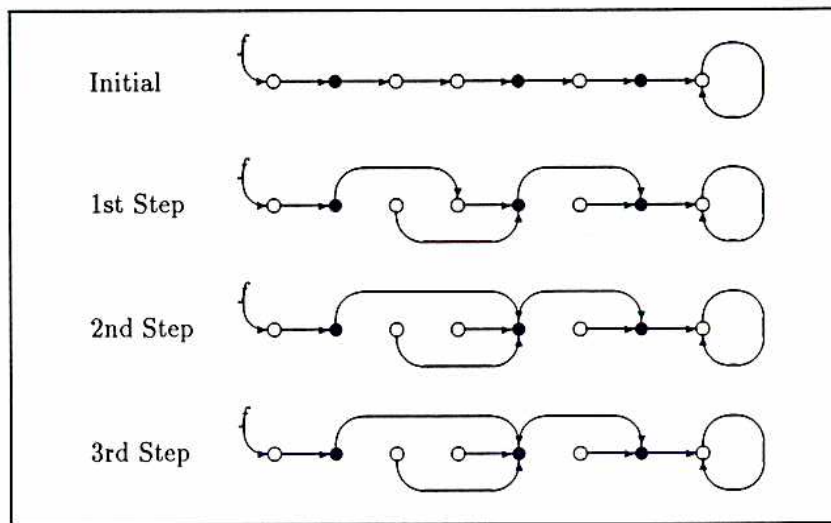


Figure 4.8: Pointer Jumping in Sublist Computation

jumping. An example is shown in figure 4.8. The labeled elements in this and the following figures are indicated by filled circles while unlabeled are indicated by unfilled circles.

If the first element is not labeled then – after the iteration is finished – the second element must be either labeled or the last element (if no element is labeled). If the last element is not labeled, then it is the only element in the list which is not labeled. Thus each element which points to non-labeled successor must point to itself. These two adjustments for the example shown in figure 4.8 are shown in figure 4.9.

Hence, the following program computes the sublist:

```
indexed type list(A)          \* as in chapter 2 *\

type pair(A,B)                \* as in section 3.6 *\

fun sublist(val:list(A),p:list(nat),l:list(bool).f:nat):pair(list(nat),nat)=
  let p' = repeat(p,label,length(p)) in
  let f' = if label[f] then f else p[f] in
  let p'' = for all i < length(p) do in parallel
    if l[p'[i]] then p'[i] else i
  in (p'',f)

fun repeat(p:list(nat),l:list(bool),n:nat):list(nat) =
  if n=1 then p
  else let p' = for all i < length(p) do in parallel
    if l[p[i]] then p[i] else p[p[i]]
  in repeat(p',l,n/2)
```

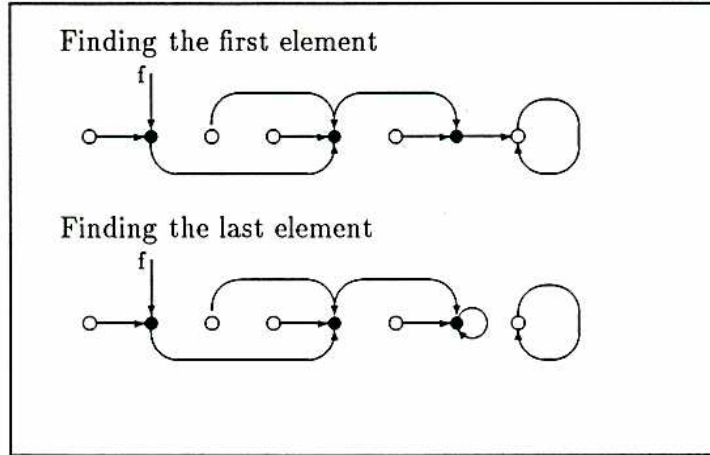


Figure 4.9: Adjustment of the First and Last Element

1st Step: Translation into time functions

After some simplifications, the time functions are:

```

fun time_sublist(val,p,l,f):nat =
  if l[f] then 22 + time_repeat(p,l,length(p)) +
    + max(0<=i<length(p),if l[repeat(p,l,length(p))[i]] then 8 else 6)
  else 24 + time_repeat(p,l,length(p)) +
    + max(0<=i<length(p),if l[repeat(p,l,length(p))] then 8 else 6)

fun time_repeat(p:list(nat),l:list(bool),n:nat):nat =
  if n = 1 then 5
  else 13 + max(0<=i<length(p),if l[p[i]] then 9 else 11)
    + time_repeat(for all ... do in parallel...,l,n/2)

```

Step 2: Normalization

The transformation of figure 3.2 is not applicable. The conditionals as an argument of the max-operation in the body of `time_sublist` and `time_repeat`, respectively, can be eliminated by the transformation of figure 3.3:

```

fun time_sublist(val,p,l,f):nat =
  if l[f] then 22 + time_repeat(p,l,length(p))
    + max(0<=i<length(p),time_sublist'(i,l,p))
  else 24 + time_repeat(p,l,length(p)) + max(0<=i<length(p),time_sublist'(i,l,p))

fun time_sublist'(i:nat,l:list(bool),p:list(nat)):nat =
  if l[repeat(p,l,length(p))[i]] then 8 else 6

```

```

fun time_repeat(p:list(nat),l:list(bool),n:nat):nat =
  if n = 1 then 5
  else 13 + max(0<=i<length(p),time_repeat'(i,l,p))
    + time_repeat(for all ... do in parallel...,l,n/2)

fun time_repeat'(i:nat,l:list(bool),p:list(nat)):nat = if l[p[i]] then 9 else 11

```

The algorithm in figure 3.5 finds that the argument position `time_sublist/1` is irrelevant. Thus, by the transformation of figure 3.4 this position is removed. Finally, the operation call in the body of `time_sublist'` has to be removed. Therefore the resulting program is obtained by applying the transformation of figure 3.9:

```

fun time_sublist(p,l,f):nat =
  if l[f] then 22 + time_repeat(p,l,length(p))
    + max(0<=i<length(p),time_sublist'(i,l,p))
  else 24 + time_repeat(p,l,length(p)) + max(0<=i<length(p),time_sublist'(i,l,p))

fun time_sublist'(i:nat,l:list(bool),p:list(nat)):nat =
  if l[access_repeat(i,p,l,length(p))] then 8 else 6

fun access_repeat(i:nat,p:list(nat),l:list(bool),n:nat) =
  if n=1 then p[i]
  else let p' = for all i < length(p) do in parallel
    if l[p[i]] then p[i] else p[p[i]]
  in access_repeat(i,p',l,n/2)

fun time_repeat(p:list(nat),l:list(bool),n:nat):nat =
  if n = 1 then 5
  else 13 + max(0<=i<length(p),time_repeat'(i,l,p))
    + time_repeat(for all ... do in parallel...,l,n/2)

fun time_repeat'(i:nat,l:list(bool),p:list(nat)):nat = if l[p[i]] then 9 else 11

```

3rd Step: Symbolic Evaluation

By the rules of figure 3.6, the following equations are obtained:

$$\begin{aligned}
 time_sublist(p,l,f) &= \begin{cases} 22 + time_repeat(p,l,length(p)) \\ \quad + \max_{i=0}^{length(p)-1} time_sublist'(i,p,l,f) & \text{if } l[f] \\ 24 + time_repeat(p,l,length(p)) \\ \quad + \max_{i=0}^{length(p)-1} time_sublist'(i,p,l,f) & \text{otherwise} \end{cases} \\
 time_sublist'(i,p,l) &= \begin{cases} 8 & \text{if } l[access_repeat(i,p,l,length(p))] \\ 6 & \text{otherwise} \end{cases} \\
 time_repeat(p,l,1) &= 5
 \end{aligned}$$

$$time_repeat(p, l, n) = 13 + \max_{i=0}^{length(p)-1} time_repeat'(i, l, p) + time_repeat(p, l, n/2)$$

$$time_repeat'(i, p, l) = \begin{cases} 9 & \text{if } l[p[i]] \\ 12 & \text{otherwise} \end{cases}$$

4th Step: Mapping onto Integers

The *length* function is applied on *time_sublist/1* and *time_repeat/1*:

$$time_sublist(m, l, f) = \begin{cases} 22 + time_repeat(m, l, m) \\ \quad + \max_{i=0}^{m-1} time_sublist'(i, p, l, f) & \text{if } l[f] \\ 24 + time_repeat(m, l, m) \\ \quad + \max_{i=0}^{m-1} time_sublist'(i, p, l, f) & \text{otherwise} \end{cases}$$

$$time_sublist'(i, p, l) = \begin{cases} 8 & \text{if } l[access_repeat(i, p, l, length(p))] \\ 6 & \text{otherwise} \end{cases}$$

$$time_repeat(m, l, 1) = 5$$

$$time_repeat(m, l, n) = 13 + \max_{i=0}^{m-1} time_repeat'(i, l, p) + time_repeat(m, l, n/2)$$

$$time_repeat'(i, p, l) = \begin{cases} 9 & \text{if } l[p[i]] \\ 12 & \text{otherwise} \end{cases}$$

5th Step: Solution of the Recurrences

We show in this section a complete average case analysis of this parallel algorithm. This analysis seems hardly automatizable, but the results obtained here are interesting because they can be compared with the estimates obtained by an automatic average case analysis method.

Suppose in this step, that the labels are uniformly distributed over *bool* = {*true*, *false*}, and that they are independently chosen for each $i < length(p)$. Thus, there are m i.i.d random variables L_i over {*true*, *false*} where $Prob[L_i = true] = Prob[L_i = false] = 1/2$. The computation of

$$\max_{i=0}^{m-1} time_repeat'(i, l, p)$$

yields:

Best Case: 9

Worst Case: 12

Average Case: $Prob[\forall i : L_{p[i]} = true] 9 + (1 - Prob[\forall i : L_{p[i]} = false]) 12$

The computation of $q = Prob[\forall i : L_{p[i]} = true]$ is based on the assumption that initially p defines a linear list. Based on this assumption, it is easy to see that only $f \neq p[i]$ for all i . Thus

$$q = \prod_{i=0, i \neq p[i]}^{m-1} Prob[L_{p[i]} = true] = \left(\frac{1}{2}\right)^{m-2}$$

Hence, the average execution time for the first iteration is

$$\tau_1(m) = 25 - \frac{3}{2^{m-2}}$$

The following considerations are based on the following fact:

Lemma 4.18 (Pointer Jumping based on Labels) *After one execution of*

```
for all i < length(p) do in parallel
  if label[p[i]] then p[i] else p[p[i]]
```

the distance from an element to the next labeled element is halved.

Proof: similar to lemma 4.9 ■

An easy consequence of this lemma is that just $\lceil \log_2 l \rceil$ iterations are necessary to reach a stable situation (i.e. each element points to a labeled element), where l is the maximum number of consecutive labels in the list. Thus, in the second step q is given by (without loss of generality $p[i] = i + 1$):

$$q = \text{Prob}[\forall 0 \leq i < j \leq m-1 : L_i = \text{true} \wedge L_j = \text{true} \Rightarrow j - i \leq 2 \wedge i \leq 1 \wedge j \geq m-2]$$

Theorem 4.19 (Stable Situation after 1 Iteration) *The probability that after one execution of*

```
for all i < length(p) do in parallel
  if label[p[i]] then p[i] else p[p[i]]
```

the pointer array p is in a stable situation (i.e. every further application of this operation does not change p) is:

$$q = \frac{F_{m+2}}{2^m}$$

where F_n is the n -th Fibonacci number.

Proof: Consider first the conditional probability q_k , such that after one iteration of pointer jumping the pointer array is in a stable situation, k elements are labeled. Then it is easy to compute q :

$$q = \frac{1}{2^m} \sum_{k=0}^m \binom{m}{k} q_k$$

because there are $\binom{m}{k}$ labelings with k labels. Let $c_{k,m}$ be the number of such labelings (i.e. there are at most one consecutive false-label). Then, the computation is even easier:

$$q = \frac{1}{2^m} \sum_{k=0}^m c_{k,m}$$

We transform now the probability space in order to simplify the computation of $c_{k,m}$. Let $C \subseteq \{1, \dots, m\}$ be the indices of the k labeled elements incremented by one, and (a_1, \dots, a_k) be the elements of C in increasing order. This sequence must have the following properties:

- (i) $1 \leq a_1 \leq 2$
- (ii) $1 \leq a_{i+1} - a_i \leq 2$ for $1 \leq i < k$
- (iii) $a_k \geq m - 1$

If we define

$$\Delta_i := \begin{cases} a_1 & \text{if } i = 0 \\ a_{i+1} - a_i & \text{if } 1 \leq i < k \\ m - a_k & \text{if } i = k \end{cases}$$

then the above properties are equivalent to:

- (i) $1 \leq \Delta_i \leq 2$ for $0 \leq i < k$ and $0 \leq \Delta_k \leq 1$
- (ii) $\sum_{i=0}^k \Delta_i = m$

Thus, the number $c_{k,m}$ equals the number of possible choices of the vector $(\Delta_0, \dots, \Delta_k)$ subject to the above constraints. Define

$$\begin{aligned} N_{k,j} &:= \{i \in \mathbb{N} \mid k \leq i \leq j\} \\ g_{k,j}(z) &:= \sum_{i=k}^j z^i \end{aligned}$$

Hence, if $k \leq m$,

$$\begin{aligned} c_{k,m} &= [z^m] g_{1,2}^k(z) g_{0,1}(z) \\ &= [z^m] z^k (1+z)^{k+1} \\ &= \binom{k+1}{m-k} \end{aligned}$$

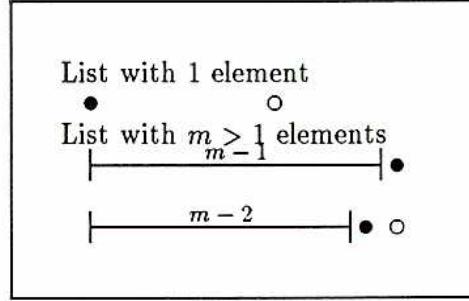


Figure 4.10: Valid Labellings for Termination after 2 Steps

According to [GKP89], it is:

$$\sum_{k=0}^n \binom{n-k}{k} z^k = \frac{1}{\sqrt{1+4z}} \left(\left(\frac{1+\sqrt{1+4z}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{1+4z}}{2} \right)^{n+1} \right)$$

Therefore, the following is obtained:

$$\sum_{k=0}^m \binom{k+1}{m-k} = \sum_{k=0}^m \binom{m+1-k}{k} = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{m+2} - \left(\frac{1-\sqrt{5}}{2} \right)^{m+2} \right) = F_{m+2}$$

where F_k is the k -th Fibonacci-number. This completes the proof. ■

In the second step, the average execution time of the iteration in *repeat* is therefore:

$$12 - 3 \cdot \frac{F_{m+2}}{2^m}$$

This result has also a combinatorial interpretation. This is demonstrated by figure 4.10. If $m = 0$, then there is one labeling: the empty labeling. If $m = 1$ there are two valid labelings: either the element is labeled or it is not. Consider now an m -element list. Suppose that the last one is labeled. Then the other $m-1$ elements can be labeled arbitrarily with a valid labeling. If the last element is not labeled, then the second last must be. Otherwise the labeling would not be valid. But then the first $m-2$ elements could be labeled with any valid labeling. Therefore the number a_m of labelings obtains the recurrence:

$$\begin{aligned} a_0 &= 1 \\ a_1 &= 2 \\ a_m &= a_{m-1} + a_{m-2} \end{aligned}$$

which has as its solution exactly the $m+2$ -nd Fibonacci-number.

This consideration is easy to generalize in order to determine the number $a_m^{(s)}$ of valid labelings if the maximum number of consecutive unlabeled elements is $s-1$. The base cases are therefore

obvious: $F_i^{(s)} = 2^i$ for $0 \leq i < s$. The case $m \geq s$ is obtained as follows: If the last element is labeled, then the other $m-1$ elements can be labeled arbitrarily with a valid labeling. If the second last element is labeled but the last is not, then the other $m-2$ elements can be labeled arbitrarily with any valid labelings. In general, if the last $i-1$ elements are unlabeled and the i -th last element is labeled, then the other $m-i$ elements can be labeled arbitrarily with a valid labeling. Clearly i must be less than s , otherwise, the labeling would not be valid. We have therefore the following recurrence:

$$\begin{aligned} F_i^{(s)} &= 2^i \quad i < s \\ F_m^{(s)} &= F_{m-1}^{(s)} + \cdots + F_{m-s+1}^{(s)} \end{aligned}$$

We call the numbers defined by this recurrence the *generalized Fibonacci numbers*. They grow for large s nearly as 2^m :

Theorem 4.20 (Generalized Fibonacci Numbers) *For large $s \geq 2$ holds:*

$$2^m (1 - 2^{-s})^m \leq F_m^{(s)} \leq 2^m$$

Proof: The upper bound is easy to see: there are at most 2^m labelings. Thus the number of valid labelings never exceeds 2^m . Consider

$$\lambda^s = \lambda^{s-1} + \cdots + 1$$

and set $\lambda = 2$, then the difference between the LHS and RHS is exactly 1. For the prove of the lower bound we define $\alpha := 1 - 2^{-s}$ and $\varepsilon := 2^{-s}$. The lower bound is proven by induction on m :

BASE CASES: Trivial

INDUCTIVE CASE: By induction hypothesis it is:

$$F_m^{(s)} \geq \alpha^{m-1} 2^{m-1} + \cdots + \alpha^{m-s+1} 2^{m-s+1}$$

It is therefore sufficient to prove: $\sum_{i=0}^{s-1} \alpha^i 2^i \geq 2^s \alpha^s$

In this proof, the following fact is used:

$$\frac{1}{1-\varepsilon} \geq 1 + \varepsilon \tag{4.1}$$

Thus:

$$\begin{aligned} \sum_{i=0}^{s-1} 2^i \alpha^i &= \alpha^s \sum_{i=0}^{s-1} 2^i \left(\frac{1}{1-\varepsilon} \right)^{s-i} \\ &\geq \{(4.1)\} \end{aligned}$$

$$\begin{aligned}
& \alpha^s \sum_{i=0}^{s-1} 2^i (1+\varepsilon)^{s-i} \\
&= \alpha^s (1+\varepsilon)^s \sum_{i=0}^{s-1} \left(\frac{2}{1+\varepsilon} \right)^i \\
&= \{ \text{Summation Formula for } 1 + \dots + x^n \} \\
& \quad \alpha^s \frac{(2^s - (1+\varepsilon)^s)(1+\varepsilon)}{1-\varepsilon} \\
&\geq \{(4.1)\} \\
& \quad \alpha^s (2^s (1+\varepsilon)^2 - (1+\varepsilon)^3) \\
&= \alpha^s (2^s + 1 - 2 \cdot 2^{-s} - 3 \cdot 2^{-2s} - 2^{-3s}) \\
&\geq \alpha^s 2^s
\end{aligned}$$

The last inequality holds only if $s \geq 2$. This completes the proof. ■

As a side-effect, by using the generating function method to determine the number of valid labelings, the following formula is obtained:

$$\sum_{k=0}^m \sum_{i=\lceil \frac{m-k}{s} \rceil}^{m+1} \binom{k+1}{i} \binom{m-s}{k} (-1)^i = F_m^{(s)}$$

Proof: Consider the valid labelings with k labels. Let (a_1, \dots, a_k) be the indices of labeled elements in increasing order. Define as above

$$\Delta_i = \begin{cases} a_1 & \text{if } i = 0 \\ a_{i+1} - a_i & \text{if } 1 \leq i < k \\ m - a_k & \text{if } i = k \end{cases}$$

Then Δ_i must satisfy the following properties:

$$\text{(i) } 1 \leq \Delta_i \leq s \text{ for } 0 \leq i < k \text{ and } 0 \leq \Delta_k \leq s-1$$

$$\text{(ii) } \sum_{i=0}^k \Delta_i = m$$

Hence, the number $c_{k,m}^{(s)}$ of valid labelings using k labels equals to the number of vectors $(\Delta_0, \dots, \Delta_k)$ satisfying the above properties. By the generating function method, this number is

$$\begin{aligned}
c_{k,m}^{(s)} &= [z^m] z^k (1+z+\dots+z^{s-1})^{k+1} \\
&= [z^{m-k}] \left(\frac{1-z^s}{1-z} \right)^{k+1} \\
&= \sum_{i=\lceil \frac{m-k}{s} \rceil}^{m+1} \binom{k+1}{i} \binom{m-s}{k} (-1)^i
\end{aligned}$$

The number of all such labelings is on the one hand

$$\sum_{k=0}^m c_{k,m}^{(s)} = \sum_{k=0}^m \sum_{i=\lceil \frac{m-k}{s} \rceil}^{m+1} \binom{k+1}{i} \binom{m-s}{k} (-1)^i$$

and on the other hand (by definition) $F_m^{(s)}$. This proves the stated claim. ■

By the above considerations the probability p_l , that there is no change in the pointer array p after l iteration is given by $F_m^{2^l}/2^m$, and therefore the following estimates hold:

$$(1 - 2^{-(2^l)})^m \leq p_l \leq 1$$

Hence, it approache very rapidly to 1 as l increases. There are many unnecessary iterations in the average. With this result, the time complexity of *repeat* is computed as follows:

Best Case:

$$time_repeat(1) = 5$$

$$time_repeat(n) = 22 + time_repeat(n/2)$$

yielding $time_repeat(n) = 5 + 22 \log_2 n$

Worst Case:

$$time_repeat(1) = 5$$

$$time_repeat(n) = 25 + time_repeat(n/2)$$

yielding $time_repeat(n) = 5 + 25 \log_2 n$

Average Case:

$$time_repeat(1, l) = 5$$

$$time_repeat(n, l) = 25 - 3 p_l + time_repeat(l+1, n/2)$$

The average case is then:

$$time_repeat(0, n) = 5 + 25 \log_2 n - \frac{3}{2^n} \sum_{i=0}^{\log_2 n} F_n^{(2^i)}$$

With theorem 4.20 the following estimates for the average case are obtained:

$$5 + 22 \log_2 n \leq time_repeat(n) \leq 5 + 25 \log_2 n - 3 \sum_{i=0}^{\log_2 n} (1 - 2^{-2^i})^n$$

and with theorem 4.24 for any $\delta > 0$:

$$\text{time_repeat}(n) \leq 5 + 22 \log_2 n + 3(1 + \delta) \log_2 \log_2 n + \varepsilon(n)$$

where $\varepsilon(n) \rightarrow 0$ as $n \rightarrow \infty$.

Finally, we have to consider the time complexity of *sublist*. First, for *time_sublist'*(i, p, l), it has to be considered how often `label[access_repeat($i, p, l, \text{length}(p)$)]` becomes *true*. By construction of `access_repeat` it holds:

Corollary 4.21 *If $p' = \text{repeat}(p, l, \text{length}(p))$ then for all $0 \leq i < \text{length}(p)$:*

$$p'[i] = \text{access_repeat}(i, p, l, \text{length}(p))$$

It is easy to prove by induction the following lemma:

Lemma 4.22 *Let be $p' = \text{repeat}(p, l, \text{length}(p))$ where p defines a linear list. Then the following implication holds:*

$$p'[i] = \text{false} \implies p'[i] = p'[p'[i]]$$

Proof: Let $G(p)$ be the graph defined by the pointer array p , and $d(p)$ be the length of the longest path $(a_0, \dots, a_{d(p)})$ in $G(p)$, where $a_2, \dots, a_{d(p)-1}$ are all labeled with *false*, and if $a_{d(p)} = p[a_{d(p)}]$ then $a_{d(p)}$ be labeled arbitrarily, otherwise it is labeled *true*. Consider now the pointer array defined by:

$p' = \text{for all } i < \text{length}(p) \text{ do in parallel}$
 if `label[p[i]]` then $p[i]$ else $p[p[i]]$

Then it is the same proof as in the classical pointer jumping to show: $d(p') = d(p)/2$. Let now be $m = \text{length}(p)$. Then after $\log_2 m$ iterations of the above operation (and therefore after the execution of *repeat*), it is $d(p) = 1$. Thus for all i it is either `label[p[i]] = true` or $p[i] = p[p[i]]$. ■

Corollary 4.23 *The probability, that `l[access_repeat($i, p, l, \text{length}(p)$)] = true` is $1/2^{m-1}$.*

This allows therefore the computation of the average case:

Best Case: $\text{time_sublist}(n) = 33 + 22 \log_2 n$

Worst Case: $\text{time_sublist}(n) = 37 + 25 \log_2 n$

Average Case: $\text{time_sublist}(n) = 36 + 25 \log_2 n - \frac{3}{2^n} \sum_{i=0}^{\log_2 n} F_n^{(2^i)} - 2^{2-n}$

$$\text{time_sublist}(n) \leq 36 + 22 \log_2 n + 3(1 + \delta) \log_2 \log_2 n + \varepsilon(n)$$

where $\delta > 0$ arbitrarily, and $\varepsilon(n) \rightarrow 0$ as $n \rightarrow \infty$

As known from the considerations for the generalized Fibonacci numbers the average case is closer to the best case instead of the worst case:

Theorem 4.24 (Number of Iterations) *Let N be the random variable defining the number of iterations on p necessary to reach a stable situation. Then the expected value of N is estimated by:*

$$E[N] \leq (1 + \delta) \log_2 \log_2 n + \varepsilon(n)$$

where $\delta > 0$ arbitrarily, and $\varepsilon(n) \rightarrow 0$ as $n \rightarrow \infty$. Thus it holds $E[N] = O(\log \log n)$.

Proof: After at most $\log_2 n$ iterations, the final situation is reached. Thus we have

$$0 \leq N \leq \log_2 n \quad (4.2)$$

It holds also (by theorem 4.20):

$$\text{Prob}[N \geq l] \leq 1 - \left(1 - 2^{(-2^l)}\right)^n \quad (4.3)$$

By using the Markov inequality for random variables $0 \leq X \leq s$:

$$\forall a \geq 0 : E[X] \leq a + \text{Prob}[X \geq a] s$$

Thus, by (4.2) the preconditions of the Markov inequality are satisfied and by the estimation (4.3) the following estimate for the expected number of iterations is obtained:

$$E[N] \leq a + \log_2 n \left(1 - \left(1 - 2^{(-2^a)}\right)^n\right) \quad (4.4)$$

Let now δ be any positive value greater than 0. If we take

$$a = (1 + \delta) \log_2 \log_2 n$$

the following inequation is obtained:

$$\begin{aligned} E[N] &\leq (1 + \delta) \log_2 \log_2 n + \log_2 n \left(1 - \left(1 - 2^{(-2^{((1+\delta) \log_2 \log_2 n)})}\right)^n\right) \\ &= (1 + \delta) \log_2 \log_2 n + \log_2 n \left(1 - \left(1 - \frac{1}{n^{\log_2^\delta n}}\right)^n\right) \end{aligned} \quad (4.5)$$

$$= (1 + \delta) \log_2 \log_2 n + \sum_{k \geq 1} \binom{n}{k} (-1)^{k+1} \frac{\log_2 n}{n^{k \cdot \log_2^\delta n}} \quad (4.6)$$

where (4.5) is obtained by using:

$$2^{-(2^{(1+\delta) \log_2 \log_2 n})} = \frac{1}{n^{\log_2^\delta n}}$$

and (4.6) is obtained by the power series of $(1 - x)^n$.

It remains now to prove that

$$\varepsilon(n) := \sum_{k \geq 1} \binom{n}{k} (-1)^{k+1} \frac{\log_2 n}{n^{k \cdot \log_2^\delta n}} \rightarrow 0$$

as $n \rightarrow \infty$. This is done by observing that:

$$\begin{aligned} \varepsilon(n) &\leq \sum_{k \geq 1} \left(\frac{n e}{k} \right)^k \frac{\log_2^k n}{n^{k \cdot \log_2^\delta n}} \\ &= \sum_{k \geq 1} k^{-k} \left(\frac{e \log_2 n}{n^{\log_2^\delta n - 1}} \right)^k \end{aligned}$$

where we made use of the inequality

$$\binom{n}{k} \leq \left(\frac{n e}{k} \right)^k$$

Furthermore, the powerseries $\sum_{k \geq 1} k^{-k} x^k$ converges for all $x \in \mathbb{R}$. Thus its radius of convergence is ∞ . Because a powerseries converges uniformly within its radius of convergence, it holds:

$$\begin{aligned} \lim_{n \rightarrow \infty} \sum_{k \geq 1} k^{-k} \left(\frac{e \log_2 n}{n^{\log_2^\delta n - 1}} \right)^k &= \sum_{k \geq 1} k^{-k} \lim_{n \rightarrow \infty} \left(\frac{e \log_2 n}{n^{\log_2^\delta n - 1}} \right)^k \\ &= \sum_{k \geq 0} k^{-k} \cdot 0 \\ &= 0 \end{aligned}$$

Because always $\varepsilon(n) \geq 0$ it follows that $\lim_{n \rightarrow \infty} \varepsilon(n) = 0$. This completes the proof. ■

It seems that using inequalities to estimate the average case are very powerful tools, because most of them make no use of particular distribution assumption, and an automatic complexity analysis system need not to care about these distributions.

Chapter 5

Conclusions

In this report we introduced a generic method for analyzing the worst case complexity of parallel programs (e.g. list minimum based on balanced binary tree technique and pointer jumping, prefix sums and polynomial evaluation). All these programs are not optimal in the sense that their work is asymptotically more than the time complexity of the best sequential algorithm. Thus the next step is to fix the number of processors to p and evaluate the worst case complexity as a function $T(n, p)$ of the input size n and processor number p , and then choose $p < O(n)$ such that $T(n, p(n)) = O(T(n))$ where $T(n)$ is the complexity using an unbounded number of processors (i.e. as evaluated by the method of this report). This extension allows then to analyze programs based on the algorithms with a reduced number of processors.

The complexity analysis method should also be checked by applying it to more complicated examples like graph algorithms (using for example the Euler-Tour technique), sorting algorithms, algorithms based on the pebble game, and algebraic algorithms.

The automatic average case analysis, as shown by the example of sublist computation, seems to be very difficult. Because literature about average case analysis of parallel algorithms is very rare, it is too early to design even an automatic estimation technique. It should be mentioned that it seems to us that algorithms based on the balanced binary tree technique and divide-and-conquer technique have at least asymptotically the same worst-case and average-case complexity. However in pointer jumping algorithms (and similar in pebble game algorithms) this seems different. Therefore, manual estimations of average cases have to be done for these algorithms.

Another result of this report is the very general applicability of the pointer jumping technique. It is not necessary to apply it to linear lists, but as shown in chapter 4 it can also be applied to a set of linear list at some circles. If concurrent reads are allowed, then it can be even applied to “suns” as long as the length of their circles is a power of 2. It should be investigated whether these more general pointer structures can lead to new efficient parallel algorithms (especially for asynchronous PRAMs).

Acknowledgements

I would like to thank my colleagues at the International Computer Science Institute. My special thanks are to Geppino Pucci. I profit from a lot of fruitful discussions about the pointer jumping technique, and get therefore a deep view of these kind of algorithms. Especially the average case analysis of the sublist computation was influenced by these discussions. I also thank Michael Luby and Thomas Lickteig for discussing the results of this average case analysis. I thank Ivan Havel for discussing the conditions on performing pointer jumping. I thank Frans Kurfess for reading a draft of this report, and giving me a lot of improving comments. Finally, I thank Richard Karp for discussing the final results of this report.

Bibliography

- [CZ90] R. Cole and O. Zajicek. The expected advantage of asynchrony. In *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 85 – 94, 1990.
- [Ech88] R. Echahed. On completeness of narrowing strategies. In *Proceedings of the 13th CAAP '88*. Springer Lecture Notes in Computer Science 299, 1988.
- [EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1, Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1985.
- [Fla88] P. Flajolet. Mathematical methods in the analysis of algorithms and datastructures. In E. Boerger, editor, *Trends in Theoretical Computer Science*, chapter 6, pages 225–304. Computer Science Press, 1988.
- [FO88] P. Flajolet and A. Odlyzko. Singularity analysis of generating functions. Technical Report 826, INRIA, April 1988.
- [FSZ88] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-epsilon-omega: An assistant algorithms analyzer. In *Proceedings of the AAECC'88*. Lecture Notes in Computer Science 357, Springer, 1988.
- [FSZ91] P. Flajolet, B. Salvy, and P. Zimmermann. Average case analysis of algorithms. *Theoretical Computer Science*, to appear, 1991.
- [GKP89] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.
- [GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.
- [HC88] T. Hickey and J. Cohen. Automating program analysis. *Journal of the ACM*, 35(1):185 – 220, 1988.
- [HH80] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. In *Proceedings of the 21th Annual Symposium on Foundations of Computer Science*. IEEE, 1980.
- [KR88] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, University of California at Berkeley, March 1988.

- [LeM88] Daniel LeMetayer. Ace: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248 – 266, 1988.
- [Weg75] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528 – 539, 1975.
- [Zim89] Paul Zimmermann. Alas: Un systeme d'analyse algebrique. Technical Report 968, INRIA, Januar 1989.
- [Zim90a] Paul Zimmermann. *Séries génératrice et analyse automatique d'algorithmes*. PhD thesis, Ecole Polytechnique Supérieure Paris, in preparation, 1990.
- [Zim90b] Wolf Zimmermann. *Automatische Komplexitätsanalyse funktionaler Programme*. Informatik-Fachberichte. Springer, 1990.
- [ZZ89] Paul Zimmermann and Wolf Zimmermann. The automatic complexity analysis of divide-and-conquer algorithms. Technical Report 1134, INRIA, December 1989.