The Automatic Worst Case Analysis of Parallel Programs: Simple Parallel Sorting and Algorithms on Graphs

Wolf Zimmermann
TR-91-045
August, 1991

Chapter 1

Introduction

Tools for automatic complexity analysis can be used for formal program development methods (such as [BMD+85, BW80, Smi88]). in order to compare intermediate results of program development and to control the direction of the development. Actually, in the recent program development methods, only correctness aspects are considered in a formal way, but no method deals efficiency aspects formally. The efficiency aspect become even more important when parallel algorithms are considered. Most of the parallel algorithms iterate a computation until a fixpoint is reached. In most of the machine models the test whether a fixpoint is reached is expensive [KR90]. Therefore the design of this kind of algorithms is based on the number of iterations necessary to reach this fixpoint, i.e. constructions like[1]:

repeat $\log n$ times
    a statement list

play an important role in the design of parallel algorithms. Information about complexity is even used in the design of parallel algorithms (and should therefore also be used in the design of parallel programs). On the other hand complexity results for parallel programs can be used in order to compare it with sequential algorithms for the same problem, and to determine the input size where parallelization is profitable.

In this report, the underlying machine model are PRAMs. Even if this model is considered as an unrealistic machine model for parallel computation, it is used most often, because it is easier to program these abstract machines than distributed memory machines. Correctness and complexity properties are easier to prove for PRAMS (and sometimes only feasible to prove it for PRAMs). Recent work consider therefore simulations of PRAMS on distributed memory architectures [MV84, AHMP87, UW87, DM89, DM90, LPP90]. It should be mentioned that none of these simulations are within a constant time factor.

The method for automatic complexity analysis for sequential programs of [Zim90a] is generalized to parallel programs. [Zim90b] describes a first approach to this generalization. In this report it is shown how some of the basic techniques can be analyzed automatically. Here we generalize the technique mentioned there in order to analyze a large class of algorithms. This is studied on parallel algorithms on graphs. All the algorithms are from [GR88], however we mention the original references.

[1] all logarithms are of base 2 troughout this report

The organization of this report is as follows: in chapter 2 we give some basic definitions and theory behind parallel algorithms. This includes a discussion about PRAMS and other parallel machine models, the definition of a functional language based on vectors with explicit parallelism, and the machine model together with the definition of time, space and processor complexities. In chapter 3 the principles of the automatic analysis method are introduced. In later chapters we refer to the different steps mentioned there. It also contains an analysis of a simple adaptive algorithm. In chapter 4, the analysis of sorting networks is shown. In chapter 5, some algorithms on graphs are analyzed. The aim of this report is to provide examples (still analyzed by hand, but by the uniform method described in chapter 3) to be tested by an implementation of the method described in this report.

Acknowledgements I am grateful to all collegues at ICSI for their support in this report. My special thanks are to Friedhelm Meyer auf der Heide who give me important hints to the part discussing the parallel architectures and the PRAM simulations. I thank also Geppino Pucci whose discussions and critics motivate me to include such a discussion. I'm grateful to Richard Karp for discussing with me some of the results in an early stage of this work. Among several other people I thank Franz Kurfess, Peter Buergisser, Thomas Lickteig, and Heinz Schmidt for discussions leading to several improvements in presentation and contents of this report.

Chapter 2

Foundations

This chapter has three tasks. First it discusses different aspects of parallel machine models and justifies our choice for a particular model. Second it defines basic notions and principles for parallel algorithms and third it defines the language in which algorithms are expressed throughout this report.

2.1 Parallel Machine Models

Parallel machines can be classified with respect to several aspects. A parallel machine consists usually of $p > 1$ processors. One distinction is whether they have a common memory or not. If they have a common memory, the architecture is called a *shared memory architecture* while the others are called *distributed memory architectures.* Distributed memory architectures are discussed in subsection 2.1.3 and shared memory architectures are discussed in subsection 2.1.4.

Another distinction is whether all processors perform at the same time the same operation or not. In the former case the machine is called *SIMD* (Single Instruction Multiple Data) and in the latter case the machine is called *MIMD* (Multiple Instruction Multiple Data). This types of machines are discussed in subsection 2.1.1

Finally there is a distinction between synchronous and asynchronous machines. These concepts are defined and discussed in subsection 2.1.2. Throughout the rest of this report we use synchronous shared restricted MIMD machines.

2.1.1 SIMD-Machines vs. MIMD-Machines

Definition 2.1 (SIMD- and MIMD-Machines) *A parallel machine is called* SIMD *iff each processor performs at the same time the same operation or is idle. Otherwise the machine is called* MIMD. ■.

Usually MIMD-machines perform some processes independently on different processors. In order to use the full power of these machines explicit communication operations

send data **to** processor i

and

receive data **from** processor i

are required. These two operations are either performed explicitly by a network or via a shared memory. On the other hand, these operations makes it usually difficult to understand programs on such machines. It is therefore also difficult to prove their correctness and to analyze their complexity.

SIMD-machines are characterized by statement of the form:

for all $i \in I$ **do in parallel** $S(i)$

This statement means that all processors P_i with an address $i \in I$ perform the command $S(i)$ while the other processors are idle. Consider now the case where $S(i)$ is a conditional statement, i.e.

$S(i) =$ **if** $B(i)$ **then** $S_1(i)$ **else** $S_2(i)$

Let $I_{true} = \{i | i \in I \wedge B(i) = true\}$ and $I_{false} = \{i | i \in I \wedge B(i) = false\}$. In order to satisfy the SIMD condition, first all processors P_i, $i \in I$ execute $B(i)$ while the other processors are idle. Second, all processors P_i, $i \in I_{true}$ execute $S_1(i)$ while the others are idle. Finally, all processors P_i, $i \in I_{false}$ execute $S_2(i)$ while the others are idle. This sequence of executions seems however unnecessary if each processor is a full processing unit (i.e. each processor can locally hold its own program). In this case the statements $S_1(i)$ and $S_2(i)$ could be performed in parallel. The correctness proof and the complexity analysis of algorithms for SIMD machines are usually simple compared to MIMD-machines.

We combine therefore the advantages of both computation models. Statements like the above conditional statement are allowed to be executed parallel, on the other hand explicit communications are not allowed. Observe that there must be an implicit communication, determining when each processor finished the computation of the statements $S(i)$. The programs on such *restricted MIMD-machines* have therefore the understandability of programs on SIMD-machines and use the advantages of MIMD-machines. Somehow, a restricted MIMD-machine can be considered as "high-level" SIMD-machine. In the rest of the report we consider therefore restricted MIMD-machines.

2.1.2 Asynchronous vs. Synchronous Machines

Consider a statement sequence

S_1: **for all** $i \in I$ **do in parallel** $S_1(i)$
S_2: **for all** $i \in I$ **do in parallel** $S_2(i)$

Statement S_1 is *executed synchronously*, if each processor P_i, $i \in I$ waits until each processor has finished the execution of statement $S_1(i)$. In contrast, if S_1 is executed *asynchronous*, then the processor P_i starts with the execution of $S_2(i)$ as soon as it finished the execution of $S_1(i)$. *Asynchronous machines* allow asynchronous execution of parallel statements while *synchronous machines* forbid them.

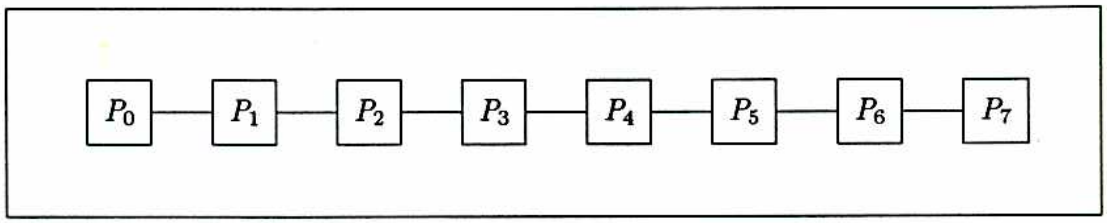


Figure 2.1: A Linear Array with 8 Processors

The advantage of synchronous models is again the simplicity of programming, correctness proofs and complexity analysis compared with asynchronous machines. On the other hand, processors spend time in waiting while eventually slower processors could profit from more work done by the faster processors. Unfortunately, correctness proofs of asynchronous machines are quite difficult, and complexity analysis is already quite difficult to analyze even for simple problems [CZ90]. We therefore restrict ourselves to synchronous machines.

2.1.3 Distributed Memory

In distributed memory architectures, memory is only local to the processors. Hence, algorithms must be designed, such that they avoid request to non-local data as often as possible, and if there is such a request, that they are possibly located close. Distributed memory architectures are distinguished in special purpose architectures and general purpose architectures. Special purpose architectures are for example VLSI-circuits, neural networks and sorting networks. We don't consider here special purpose architectures. A more detailed and more quantitative discussion of distributed memory architectures can be found in [Val90].

Distributed memory architectures can be characterized by graphs $G = (V, E)$ where $V = \{0, \ldots, p-1\}$ represents processors ($i \in V$ represents the processor P_i with address i), and (i, j) represents a *link* between processors P_i and P_j. If the graph G is directed, this link is called *one-way*, otherwise it is called *two-way*. A network G is called a *bounded-degree* network, iff for all $i \in V$, the degree is constant w.r.t. the number of processors p. Networks are assessed by the number of processors, the number of links (i.e. the size of E), the time for access to global data (i.e. the diameter of G), and the *bottlenecks* (i.e. the expected number of paths between any $i \in V$ and $j \in V$).

The probably most simple architecture is a *linear array* with p processors (see figure 2.1). Here each processor P_i is linked by two-way link to its neighbours P_{i-1} and P_{i+1}. Sometimes there is also a connection between processor P_0 and P_{p-1}. In this case the architecture is called a *ring*. These two architectures are realized in hardware (for example [Mor90]). A p-processor linear array has $p-1$ links (or p links, if it is organized as a ring), the access time to global data is $O(p)$, and the bottleneck is 1 (or 2 in the case of a ring array). Between any processor P_i and any processor P_j is just one (or in the case of a ring two) communication path, i.e. it is quite likely that communication paths overlap.

A better performance can be achieved by a *mesh* of p processors[1] (see figure 2.2). The processors are organized as two-dimensional array, each side of length $\sqrt{p}$. The processor $P_{i,j}$ is linked by two-way links to the processors $P_{i-1,j}$, $P_{i,j+1}$, $P_{i+1,j}$, and $P_{i,j-1}$. Sometimes, the additions and

[1] For simplicity we assume that p is a square number

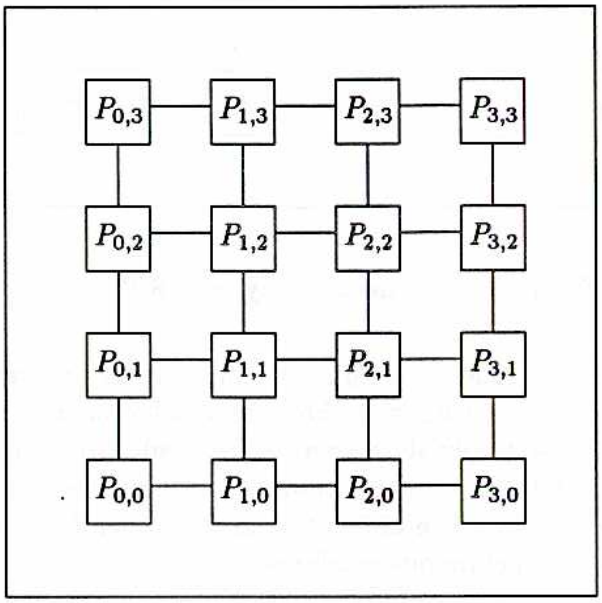


Figure 2.2: A Mesh with 16 Processors

subtractions are modulo $\sqrt{p}$ (the mesh is then a *torus*). A p-processor mesh has $2 \cdot \sqrt{p} \cdot (\sqrt{p} - 1)$ links and the access to global data costs $O(\sqrt{p})$ communications. Between two processors of a great distance there are many possible communication paths (between a processor $P_{i,j}$ and $P_{k,l}$ are $\begin{pmatrix} k-i+l-j \\ k-i \end{pmatrix}$ communication paths). Hence meshes do not have the bottlenecks of linear arrays. Meshs or torus can be easily realized as transputer networks [INM86, INM88].

A further generalization is a d-dimensional hypercupe with $p = 2^d$ processors (see figure 2.3). The processors are linked together as in a d-dimensional hypercube, i.e. two processors are linked by two way links, if the binary representation of their address differ by one bit. Observe that each processor has $\log_2 p$ links. Altogether, a d-dimensional hypercube with p processors has $p/2 \log_2 p$ links and access to global data costs $O(\log_2 p)$ communications. The bottlenecks of a linear array or a mesh become less serious. It is unlikely that communication paths overlap. The architecture of the connection machine with 65536 processors is based on 16-dimensional hypercube [Hil85].

A different style of architectures are tree-based. The most simple one is the *tree interconnection network* as shown in figure 2.4. The processors are connected like a balanced rooted binary tree (or in general a rooted r-ary tree). If such a network has p processors, then it has $p - 1$ links, and the cost of access to global data is at most $O(\log p)$ communications. The root and "high-level" nodes are however a bottleneck. If more than one global data access has to use the root, then they must be executed sequentialy. This situation is quite likely. A shuffle exchange network overcomes this problem (see figure 2.5). If it has p processors, then process P_i is linked by a one way link to processor P_j if

$$j = \begin{cases} 2\,i & \text{if } 0 \leq i < p/2 \\ 2\,i + 1 - p & \text{if } p/2 \leq i < p \end{cases}$$

Additionaly the processors P_{2i} and P_{2i+1} are linked by a two-way link. Hence, there are altogether

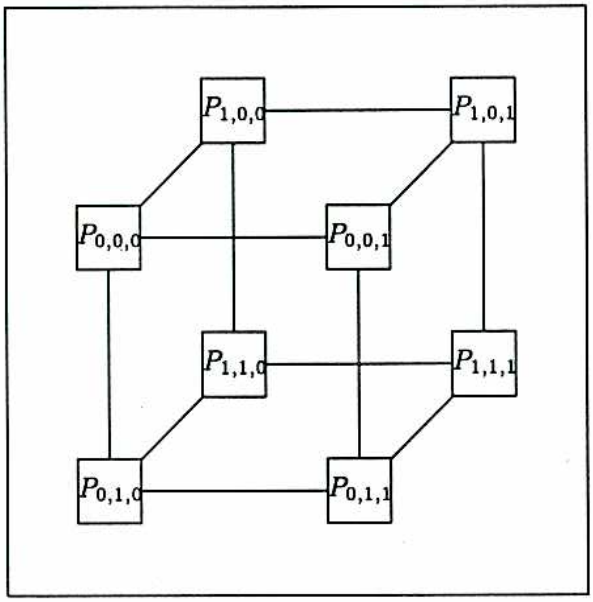


Figure 2.3: A Hypercube with 8 Processors

$3/2p$ links. A realization of a shuffle-exchange network can be found in [PHT90]. Finally, a widely used network is the butterfly network for $p_r = (r+1)\,2^r$ processors. It can be recursively described as in figure 2.6. Here B_k stands for a butterfly network with p_k processors. If addresses are assigned to the processor, then the binary address of a processor is divided in two parts, the first consisting of the r rightmost bits, the second consisting of the remaining bits. Observe that the second part represents a number α, where $0 \leq \alpha < r$. Let $b_0, \ldots b_{r-1}$ and α be these parts of processor P_i. Then P_i is connected two the processors P_j, where j consists of a first part $b_0, \ldots, b_\alpha, \ldots, b_{r-1}$ or $b_0, \ldots, \bar{b}_\alpha, \ldots, b_{r-1}$, respectively, and a second part $\alpha+1$ (see figure 2.7). Sometimes level r and level 0 are identified. Thus there are altogether $2\ r$ links, and an access to global memory costs at most $O(r)$ communications. There is no bottleneck like the root in the tree-based network. Butterflys can easily realized as transputer networks. Descriptions of hardware realization of butterfly networks can be found in [Fan86, RT86, SABS86].

Finally, a network overcoming all these problems is the complete network (i.e. the underlying graph is the complete graph). Unfortunately, a complete network with p processors has $p\ (p-1)/2$ links, and is therefore realistic only for small p.

Algorithms for distributed memory machines have to be designed in such a way, that access to memory of other processors is avoided whenever possible, and if there is such an access, then the targeted processor should be close enough to the processor sending the request. It is therefore no surprise that some problems can be very easily implemented on particular networks (for example FFT on a shuffle-exchange network), while other problems are difficult to program (e.g. matrix multiplication on a tree-based network). Even if these networks are general purpose in the sense that every problem can be programmed, they are not as universal as we wish, because the algorithms must be designed taking into account the particular architecture. In this sense, we do not consider these machines as universal.

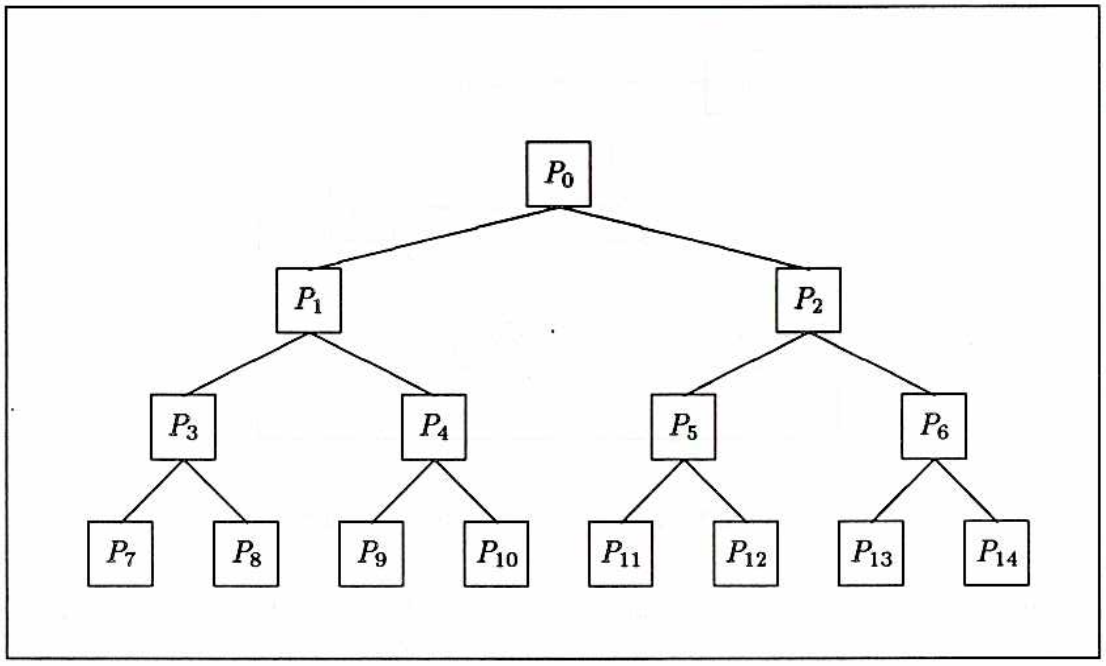


Figure 2.4: A Tree Interconnection Network with 15 Processors

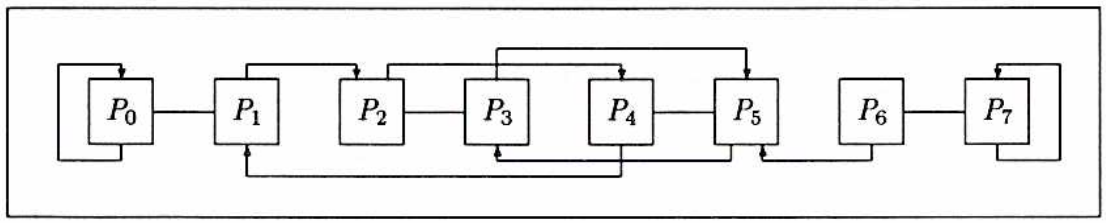


Figure 2.5: A Shuffle-Exchange Network with 8 Processors

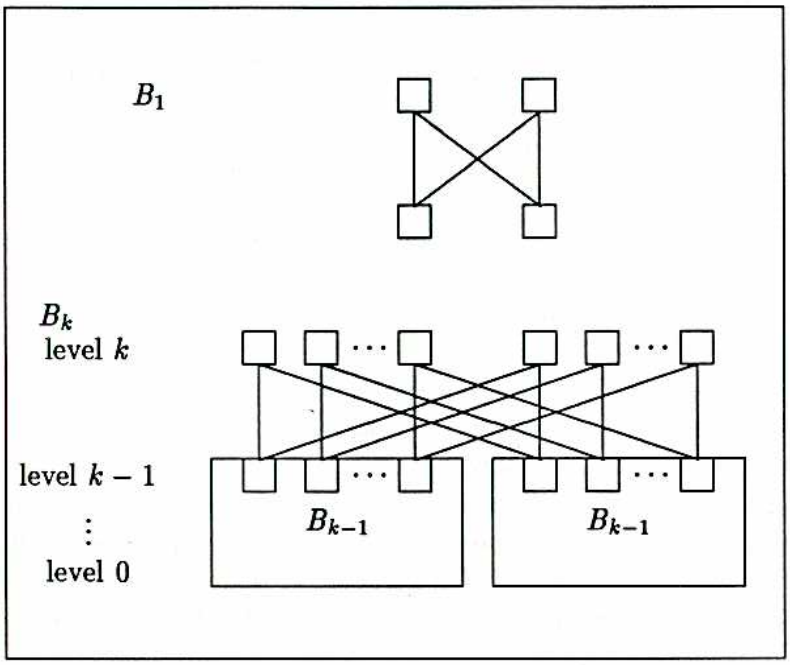


Figure 2.6: The Recursive Structure of a Butterfly-Network

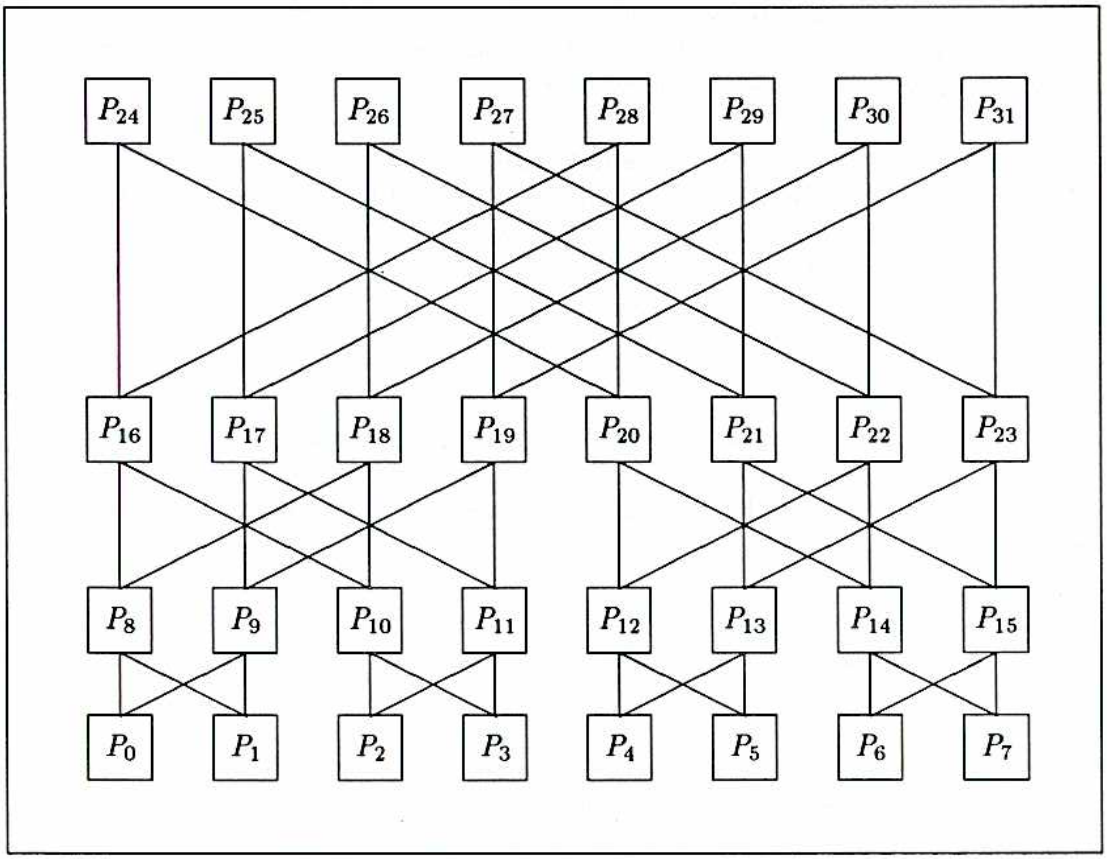


Figure 2.7: A Butterfly Network with 32 Processors

2.1.4 Shared Memory Architectures

The shared-memory architectures are all based on a PRAM. All the PRAM models are easy to program compared to distributed memory architectures, because the geography of the processors need not to be considered. In this sense PRAMS are considered as universal.

A *PRAM* (Parallel Random Access Machine) consists of a set of processors (with addresses), each of it being a RAM with local memory, and a global random access memory. A PRAM with p processors and m global memory cells is sometimes called a (p, m)-*PRAM*. A *EREW-PRAM* (Exclusive Read Exclusive Write) allows that a global memory cell i can be read (and written) only by one processor at the same time. In a *CREW-PRAM* (Concurrent Read Exclusive Write), it is allowed that different processors can read the same memory cell at the same time, but writing to the same memory cell by more than one processor at the same time is forbidden. A *CRCW-PRAM* (Concurrent Read Concurrent Write) allows also more than one processor to write into the same memory cell. They are further distinguished by their way to solve write conflicts. In a *COMMON CRCW-PRAM* a value is written into a memory cell i, only if all processor who want to write into i write the same value. In a *ARBITRARY CRCW-PRAM* one of the processors who want to write into location i is chosen arbitrarly. In a *PRIORITY CRCW-PRAM* to each processor is assigned a priority, and in the case of writing into location i, the processor with lowest priority (among the processors who want to write into i) is chosen.

PRAMs, especially the CRCW-PRAMs, are the most unrealistic machine models for parallel machines. Because of the easyness of programming them, in recent time, they are simulated on bounded-degree networks. We give some of the main results. It is distinguished between randomized and deterministic simulation of PRAMs. The first theorem is standard and involves only the simulation of concurrent reads and concurrent writes by an EREW-PRAM:

Theorem 2.2 (Simulation of Concurrent Reads and Concurrent Writes) *T steps on a CRCW-PRAM with p processors can be simulated in time $O(T \log p)$ on a EREW-PRAM.*

A proof of this theorem can be found for example in [Akl89].

The next two theorems are proven in [KU88].

Theorem 2.3 (Karlin and Upfal) *T steps on a (p, m)-PRAM, m polynomial in p, can be simulated on a bounded-degree network of p processors in expected time $\Theta(T \log p)$.*

Observe the restriction of m and that this theorem also states a lower bound. The following theorem is a generalization:

Theorem 2.4 (Karlin and Upfal) *T steps on a (p, m)-PRAM can be simulated on a bounded-degree network with p processors in time $O(T \log m)$ with probability tending to 1 as n tends to ∞.*

Both of these simulations are performed with the butterfly network. The processor must use queues in order to handle message passing. Their queue size is $O(\log p)$. A better and more simple solution reducing the queue size to $O(1)$ is given in [Ran87]:

Theorem 2.5 (Ranade) *T steps of a p-processor CRCW-PRAM can be simulated probabilistically on a bounded-degree network with p processors in expected time $O(T \log p)$.*

His network is again the butterfly network.

As a consequence of [DM90] the following theorem holds:

Theorem 2.6 (Dietzfelbinger and Meyer auf der Heide) *T steps on a PRIORITY CRCW-PRAM with p processors can be simulated probabilistically on a complete network of p processors in expected time $O(T \log p / \log\log p)$.*

The fastest known deterministic simulation is result is due to [LPP90]. The disadvantage of their result is the huge number of processors.

Theorem 2.7 (Luccio, Pietracaprina, and Pucci) *T steps on a (p,m)-PRAM can be simulated on a bounded-degree network with $O(p^2)$ processors in time $O(T \log^2 p / \log\log p)$.*

Their network is a mesh of trees. This network consists of a mesh of p^2 processing units. Each row R_j and each column C_i of the mesh is associated with a tree having as leaves the processing units in R_j and C_i and as root a processor P_i and P_j, respectively. Observe that with the exception of the tree-roots the processing units are simple switching elements, and that processor P_i is the root of both column C_i and row R_i.

However, there are deterministic simulation results, which are better (and optimal) w.r.t. the product of the number of processors and time. An optimal simulation is given in [AHMP87]:

Theorem 2.8 (Alt, Hagerup, Mehlhorn, and Preparata) *T steps on a p-processor CRCW-PRAM can be simulated deterministically on a bounded-degree network with p processors in time $O(T \log^2 p)$.*

They proved also, that any deterministic simulation scheme simulating T steps on a p-processor PRAM must take time at least $\Omega(\log^2 p / \log\log p)$.

Finally, the most remarkable results are given in [Val90]:

Theorem 2.9 (Valiant) *T steps on a EREW-PRAM with p processors can be simulated probabilistically on a bounded-degree network with q processors in expected time $O(T\ p/q)$, if $p \geq q \log q$.*

Valiant proved a similar result even in the case of a CRCW-PRAM:

Theorem 2.10 (Valiant) *T steps on a CRCW-PRAM with p processors can be simulated probabilistically on a bounded-degree network with q processors in expected time $O(T\ p/q)$, if $p \geq q^{1+\varepsilon}$ for some $\varepsilon > 0$.*

Observe that these results that the product of the number of processors and time is asymptotically the same for the algorithm on the PRAM and the simulated algorithm on the bounded-degree network. The simulations can be performed either with a hypercube or with a butterfly.

In the rest of this report we use a CREW-PRAM. This choice is justified because of the universality of the machine and the above simulation results.

2.2 Parallel Algorithms

Here several aspects of parallel algorithms are discussed. First some basic notion concerning complexity measures are dicussed. Based on this definitions some desirable properties of parallel algorithms can be defined. This motivates the introduction of a complexity class *NC*. We do not discuss further details of this complexity class. It should only be mentioned, that all algorithms discussed in this report belong to *NC*. A more extensive description of the concepts discussed here can be found in [GR88, Akl89, KR90].

Definition 2.11 (Parallel Complexity Measures) *An algorithm A on a PRAM has* time complexity $T(n)$, *if the algorithm A halts for all inputs of size n after* $T(n)$ *parallel steps. A has* processor complexity $p(n)$, *if it needs for any input of size n at most* $p(n)$ *processors to execute A. Finally, A has* space complexity $m(n)$, *if for any input of size n at most* $m(n)$ *memory cells are needed to execute A. A has work* $w(n)$, *if* $w(n) = p(n)\ t(n)$. ■

For space complexity, usually only global space is considered. Observe that the notion of work counts the number of operations of an performed by a parallel algorithm. We can therefore consider a parallel algorithm as optimal, if its work is asymptotically equal to the time complexity of the best known sequential algorithm:

Definition 2.12 (Optimal Parallel Algorithm) *A PRAM-Algorithm A solving a problem P is* optimal, *iff its work* $w(n) = O(t(n))$, *where* $t(n)$ *is the time complexity of the fastest known sequential algorithm solving problem P.* ■

Although it is desirable that a parallel algorithm is optimal, it is often difficult to achieve this property using parallel execution. Also the time complexity of a parallel algorithm should be low. On the other hand a parallel algorithm should not use to many processors. These requirements motivate the notion of an efficient parallel algorithm.

Definition 2.13 (Efficient Parallel Algorithms) *A PRAM algorithm A is said to be* efficient, *iff its time complexity* $t(n) = O(\log^k n)$ *for a* $k \geq 0$, *and its processor complexity* $p(n) = O(n^l)$ *for an* $l \geq 0$. ■

Observe, by the simulations results from the last section, the notion of efficiency is invariant under the different PRAMS and under butterfly-networks or hypercubes. This motivates the definition of the complexity class[2] NC, first studied in [Pip79]:

Definition 2.14 (NC) *The complexity class* NC *is the set of all problems, which can be solved by an efficient parallel algorithm.* ■

It is unknown whether $P = NC$ or not. This problem seems as hard as the problem $P = NP?$. Hence, the notion of NP-completeness and reductions are also defined for the $P = NC?$ problem. It makes therefore sense to talk about P-complete problems. In this report we consider only problems in NC. [GR88, KR90] contain a more extensive discussion of these aspects on parallel computation.

In practice, algorithms should be designed in such a way, that they can adapt themselves to the available number of processors. Such algorithms are called *adaptive*. This is theoretically always possible as first shown in [Bre74]. In order to discuss this result, we give also the standard proof:

[2] NC = Nick (Pippengers) Class

Lemma 2.15 (Brent) *Let A be a PRAM-algorithm with time complexity $t(n)$ and work $w(n)$. Then A can be implemented on a p-processor PRAM with parallel time $O(w(n)/p + t(n))$.*

Proof: Let $w_i(n)$ be the number of operations performed by A on step i. These operations can be simulated with p processors in time $w_i(n)/p + 1$. Thus the complexity of the algorithm on a p-processor PRAM is:

$$\sum_{i=1}^{t(n)} w_i(n)/p + 1 = w(n)/p + t(n)$$

■

Observe that this proof does not schedule the $w_i(n)$ operations explicitly to the p available processors. It assumes that scheduling is not a problem. However, this assumption is not justified in practice. Many algorithms are just made adaptive using Brents lemma. They are not easy implementable within the given requirements, and often it is even yet unknown how to schedule the operations to the p processors. We consider therefore only adaptive algorithms, if the scheduling is known.

2.3 The Language PARFL0.1

In this section we define a language PARFL0.1 for expressing CREW-PRAM algorithms. This language is in its nature functional (as e.g. that in [Zim90a] defined for sequential algorithms). The parallelism is explicit in order to allow economic processor usage – a main property of many parallel algorithms. Typing is not explicit, it has a similar structure as the vector types of FP (see for example [HC88, LeM88]) just with the difference that all the elements of a vector must be of the same type. Additionaly we introduce cartesian products. In the first subsection we introduce the syntax of PARFL, in the second subsection the semantics of PARFL is defined and finally in the third subsection, we define the complexity measures for PARFL.

2.3.1 Syntax

We start first with the definition of the types in PARFL, and giving then a context-free grammar for PARFL in EBNF. The informal idea behind the semantics of the syntactical constructs will be explained here, while the formal part is postponed to the next subsection.

The notion of vectors and types is defined as follows:

Definition 2.16 (Types of PARFL) *The set Ω of* types *is the smallest set satisfying (i), (ii), and (iii):*

(i) *The* basis types `nat` *and* `bool` *are in Ω. They are defined by the sets*

$$\texttt{bool} = \{\texttt{true}, \texttt{false}\} \qquad \texttt{nat} = \{0, 1, 2, \ldots\}$$

The operations of `bool` *are the standard logical operations* `and`, `or`, *and* `not`. *The operations on* `nat` *are the standard arithmetic operations $+, -, \cdot$ and $/$. Sometimes* max *and* min *are also included. Additionaly it contains the basic relation operators $=, <, \leq, \geq, >, \neq$.*

Projection	$v[i] = \begin{cases} v_i & \text{if } v = \langle v_0, \ldots, v_{n-1}\rangle \text{ and } 0 \leq i < n \\ \bot & \text{otherwise} \end{cases}$
Appending an element at the front	$cons(a, \langle v_0, \ldots, v_{n-1}\rangle) = \langle a, v_0, \ldots, v_{n-1}\rangle$
Appending two vectors	$\langle v_0, \ldots, v_{n-1}\rangle \circ \langle w_0, \ldots, w_{m-1}\rangle = \langle v_0, \ldots, v_{n-1}, w_0, \ldots, w_{m-1}\rangle$
First element	$hd(v) = \begin{cases} v_0 & \text{if } v = \langle v_0, \ldots, v_{n-1}\rangle \text{ and } n > 0 \\ \bot & \text{otherwise} \end{cases}$
Tail elements	$tl(v) = \begin{cases} \langle v_1, \ldots, v_{n-1}\rangle & \text{if } v = \langle v_0, \ldots, v_{n-1}\rangle \text{ and } n > 0 \\ \bot & \text{otherwise} \end{cases}$
Empty	$mt(v) = \begin{cases} true & \text{if } v = \langle\rangle \\ false & \text{otherwise} \end{cases}$
Length	$lg(\langle v_0, \ldots, v_{n-1}\rangle) = n$

Figure 2.8: Vector Operations

(ii) *If A is in Ω, the set $\langle A\rangle$ of* vectors over *A is also in Ω and defined as the smallest set satisfying:*

- *The* empty vector $\langle\rangle$ *is in* $\langle A\rangle$.
- *If* $v_0, \ldots, v_{n-1} \in A$ *then the* vector of length n $\langle v_0, \ldots, v_{n-1}\rangle$ *is in* $\langle A\rangle$.

The operations on vectors are defined in figure 2.8.

(iii) *If $A_1, \ldots A_k$ are in Ω then the* cartesiaon product $A_1 \times \cdots \times A_k$ *is also in Ω If $a_1 \in A_1, \ldots a_k \in A_k$ then $(a, \ldots, a_k) \in A_1 \times \cdots \times A_k$. The operations are just the projections $(a_1, \ldots, a_k).i = a_i$ if $1 \leq i \leq k$ and $\bot$ otherwise.*

The set of extended objects *is defined as* $\Omega^+ = \bigcup_{\omega \in \Omega} \omega \cup \{\top, \bot\}$. ■

Remark: The elements $\bot$ and $\top$ play the role of *undefined* and *nothing*, respectively. The element $\top$ occurs only in connection with control structures and means that a processor evaluating $\top$ is idle. The bottom element $\bot$ means that a program does not terminate or terminates with a run-time error. ■

The syntax of PARFL is defined by the following grammar $G = (N, T, P, \langle prog\rangle)$, where:

$N = \{\langle prog\rangle, \langle fun\rangle, \langle pars\rangle, \langle type\rangle, \langle expr\rangle, \langle simpleexpr\rangle, \langle letexpr\rangle, \langle ifexpr\rangle, \langle forallexpr\rangle, \langle selectexpr\rangle, \langle compexpr\rangle, \langle op\rangle, \langle uop\rangle\}$

$T = \{\langle id\rangle, \langle nat\rangle, \langle bool\rangle, ':', '=', '\times', '(', ')', ',', '[', ']', '.', '+', '-', '*', '/', '<', '<=', '>', '>=', '<>', '\circ$, **skip**, **let**, **in**, **if**, **then**, **else**, **forall**, **do**, **parallel**, **select**, **from**, **to**, **modify**, **and**, **or**, **not**, hd, tl, mt, $lg\}$

P is defined in figure 2.9

The terminal $\langle id\rangle$ represents any identifier. An identifier starts with a letter and is followed by any sequence of characters. The other terminals $\langle nat\rangle$ and $\langle bool\rangle$ represent any natural number or boolean. Infix-expressions are bracketed left-associative and with the standard precedence. Vector operations and operations on cartesian products have higher precedence than any infix- or unary

```
⟨prog⟩       ::= (⟨fun⟩)*
⟨fun⟩        ::= fun ⟨id⟩[⟨pars⟩]' :' ⟨type⟩' =' ⟨expr⟩
⟨pars⟩       ::= '('⟨id⟩' :' ⟨type⟩(',' ⟨id⟩' :' ⟨type⟩)* ')'
⟨type⟩       ::= bool|nat|'('⟨type⟩')'|⟨type⟩' ×' ⟨type⟩('×'⟨type⟩)*
⟨expr⟩       ::= ⟨simpleexpr⟩|⟨letexpr⟩|⟨ifexpr⟩|⟨forallexpr⟩|⟨selectexpr⟩|⟨modifyexpr⟩|⟨compexpr⟩|'('⟨expr⟩')'
⟨simpleexpr⟩ ::= ⟨id⟩|⟨nat⟩|⟨bool⟩|skip|'⟨'⟨expr⟩(',' ⟨expr⟩)* '⟩'|'('⟨expr⟩(',' ⟨expr⟩)* ')'
⟨letexpr⟩    ::= let ⟨id⟩' =' ⟨expr⟩ in ⟨expr⟩
⟨ifexpr⟩     ::= if ⟨expr⟩['=' ⟨expr⟩] then ⟨expr⟩ else ⟨expr⟩
⟨forallexpr⟩ ::= forall ⟨expr⟩' <=' ⟨id⟩' <' ⟨expr⟩ do in parallel ⟨expr⟩
⟨selectexpr⟩ ::= select ⟨expr⟩' <=' ⟨id⟩' <' ⟨expr⟩ in parallel from ⟨expr⟩
⟨modifyexpr⟩ ::= modify ['('⟨expr⟩(',' ⟨expr⟩)*[')'] ',' ⟨expr⟩' <=' ⟨id⟩' <' ⟨expr⟩ to ⟨expr⟩ from ⟨expr⟩
⟨compexpr⟩   ::= ⟨id⟩'('⟨expr⟩(',' ⟨expr⟩)* ')'|⟨expr⟩⟨op⟩⟨expr⟩|⟨expr⟩'['⟨expr⟩']'|⟨expr⟩'.'⟨expr⟩|⟨uop⟩⟨expr⟩
⟨op⟩         ::= ' +' |' −' |' *' |'/'|max|min|' <' |' <=' |' =' |' >' |' >=' |' <>' |and|or|' o'
⟨unop⟩       ::= hd|tl|mt|lg|not
```

Figure 2.9: The Syntax of PARFL

operator. Unary operators have higher precedence than infix operators. Relational operators have the lowest precedence.

The **let, if** and function application are standard expressions. Informally the parallel expressions have the following meaning. A **forall** statement delivers a vector, i.e. the statement

forall $l \leq i < r$ **do in parallel** $t(i)$

is described the value of vector v of the imperative statement:

forall $0 \leq i < r - l$ **do in parallel**
$v[i] :=\cdot\, t(i + l)$

Thus it delivers a vector. In this statement, no write-conflict can occur.

The **select**-statement select a value with a certain property from a vector, i.e. the statement

select $l \leq i < r$ **in parallel from**
if $c(i)$ **then** $t(i)$ **else skip**

is the value of the scalar k after the imperative statement:

forall $l \leq i < r$ **do in parallel**
if $c(i)$ **then** $k := t(i)$

Observe that the statement **skip** leaves some processors idle. In the select-statement, write conflicts can occur. They are resolved by one of the strategies described in subsection 2.1.4.

Finally the **modify** statement modifies some vector entries from another vector. Thus the expression

modify $t_1(i)$, $l \leq i < r$ **to** $t_2(i)$ **from** t_3

is the value of variable v after the execution of the following imperative statement (n is the length of vector t_3):

forall $0 \leq j < n$ **do in parallel**
    $v[i] := t_3(i)$;
forall $l \leq i < r$ **do in parallel**
    $v[t_1(i)] := t_2(i)$;

Write conflicts can also occur by evaluating this expression. The generalization to higher dimensions is straightforward.

Although in parallel imperative languages one statement for parallel execution is sufficent, in parallel functional languages the two latter expressions are necessary, because they evaluate to different types of expressions. The first parallel statement could be considered as a special case of the third. Even if the second statement could be simulated by the third, this cannot be done within a constant time factor. We therefore consider it as necessary.

2.3.2 The Semantics of PARFL0.1

The semantics of PARFL is defined in three steps. First we consider the structure of Ω^+. This will give us some insight how to express the different styles of solving write-conflicts. Then we define the static semantics by giving typing rules for expressions. Finally the operational semantics defines the dynamic behaviour of correctly typed expressions and programs,

An important fact is that Ω^+ obey several ordering structures. These structures correspond to the way how and whether write-conflicts can be solved. The following definition and lemma plays therefore a major role in defining an operational semantics:

Definition 2.17 (Structures of Ω^+) *The quadruple $(\Omega^+, \sqsubset_{EW}, \sqcap_{EW}, \sqcup_{EW})$ where $\sqsubset_{EW}$ is the transitive closure of the following relation:*

(i) $\bot \sqsubset_{EW} \omega$ *for all* $\omega \in \Omega^+$.

(ii) $\omega \sqsubset_{EW} \top$ *for all* $\omega \in \Omega^+$.

(iii) If $v_0 \sqsubset_{EW} w_0, \ldots, v_{n-1} \sqsubset_{EW} w_{n-1}$ *then* $\langle v_0, \ldots, v_{n-1} \rangle \sqsubset_{EW} \langle w_0, \ldots, w_{n-1} \rangle$.

(iv) If $a_1 \sqsubset_{EW} b_1, \ldots, a_k \sqsubset_{EW} b_k$, *then* $(a_1, \ldots, a_k) \sqsubset_{EW} (b_1, \ldots, b_k)$.

is called an exclusive write structure. *The operations $x \sqcap_{EW} y$ ($x \sqcup_{EW} y$) is the largest (smallest) element z with $z \sqsubseteq_{EW} x$ and $z \sqsubseteq_{EW} y$ ($x \sqsubseteq_{EW} z$ and $y \sqsubseteq_{EW} z$), and $x \sqcap_{EW} x = \bot$.*

A common concurrent write *structure is a quadrupe $(\Omega^+, \sqsubset_{CCW}, \sqcap_{CCW}, \sqcup_{CCW})$ where $\sqsubset_{CCW}$ is defined as the reflexive and transitive closure of a relation satisfying also (i)-(iv). The operations $x \sqcap_{CCW} y$ and $x \sqcup_{CCW} y$ are defined analogously to the exclusive-write structure (just with the difference that $x \sqcap_{CCW} x = x$.* ■

If values $x_1, \ldots, x_k$ shall be written at the same location then it is written $x_1 \sqcap \cdots \sqcap x_k$ into this location.

Remark: The other strategies to solve write conflicts (PRIORITY CRCW-PRAM and ARBITRARY CRCW PRAM) can be defined by similar partial orderings, but the domains are different. In the priority based models, Ω^+ is replaced by $\Omega^+ \times \mathbf{N}$. The second component defines the priority of the element to be written in a certain memory location. If the first component is different from $\top$ and $\bot$ then the elements are further ordered by their second component. Otherwise the definition is the same as above.

In the case of an ARBITRARY CRCW PRAM, Ω^+ is replaced by the power domain 2^{Ω^+}. Then a set is associated to each memory cell, namely the set of possible values it can contain. Concurrent writes are defined as the union of the sets to be associated with a memory cell. The ordering becomes quite complicated. ■

Lemma 2.18 (The Ordering Structure of Ω^+) *The relation $\sqsubset_{EW}$ of the exclusive write structure is a strict ordering relation with the least element $\bot$ and the largest element $\top$. The relation $\sqsubseteq_{CCW}$ of the common concurrent write structure is a ordering relation with the least element $\bot$ and the largest element $\top$.*

Proof: The minimality of $\bot$ and $\top$ is by (i) and (ii) of definition 2.17 obvious for both structures. The transitivity of both relations is clear from their definition. Observe that only $\sqsubset_{CCW}$ is reflexive. Hence $\sqsubset_{EW}$ is a strict ordering relation. It remains to show that $\sqsubset_{CCW}$ is antisymmetric, i.e. from $a \sqsubset_{CCW} b$ and $b \sqsubset_{CCW} a$ follows that $a = b$. Suppose that this is not the case. If a and b are vectors then there must be a i such that $a_i \sqsubset_{CCW} b_i$ and $b_i \sqsubset_{CCW} a_i$ and $a_i \neq b_i$. A similar argument applies for cartesian products. Hence we can restrict ourselves to the basic types. But for these types it is obvious from (i) and (ii) of definiton 2.17 that $a = b$. ■

It follows immediately:

Corollary 2.19 *The exclusive-write structure is well-defined, i.e. $x \sqcap_{EW} y$ and $x \sqcup_{EW} y$ exist. Furthermore, the common concurrent write structure is a lattice.*

Observe that the only lattice laws not satisfied by the exclusive-write structure are the adjunction laws ($a \sqcap (a \sqcup b) = a$ and $a \sqcup (a \sqcap b) = a$).

For the definition of correctly typed programs, technically a typing environment is required, i.e. to each identifier is associated a type. Because polymorphic types are allowed, we need an ordering on polymorphic types representing a "more precise" relation. Intuitively, the language of types consists of a signature representing basic types, vectors and cartesian products, and a set of variables. If a type τ_1 can be obtained from a type τ_2 by a substitution, then τ_1 is more precise than τ_2. If types have to be coerced then they are unified. If they are not unifiable, then a type error occurs. Thus we use the following notions (see for example [EM85] for the general definitions):

(i) Ω_V is the set of types with variables V. They are defined as in definition 2.16 including the properties that a variable $v \in V$ is also a type, and if τ_1 and τ_2 are types, then $\tau_1 \mapsto \tau_2$ is also a type. PARFL0.1 is restricted in its functions to first-order functions, i.e. neither τ_1 nor τ_2 contain $\mapsto$.

(ii) A *type substitution* θ is a finite list of pairs $[v/t]$ where $v \in V$, $t \in \Omega_V$, all the variables on the left are different, and no type on the right contains any variable on the left. It is interpreted as the following mapping $\Omega_V \mapsto \Omega_V$:

$$\begin{aligned} \theta v &= \begin{cases} t & \text{if } [v/t] \in \theta \\ \textbf{error} & \text{otherwise} \end{cases} \\ \theta\langle\tau\rangle &= \langle\theta\tau\rangle \\ \theta(\tau_1 \times \cdots \times \tau_k) &= \theta\tau_1 \times \cdots \times \theta\tau_k \\ \theta(\tau_1 \mapsto \tau_2) &= \theta\tau_1 \mapsto \theta\tau_2 \end{aligned}$$

(iii) A substitution θ is called a *unifier* of types τ_1 and τ_2, if $\theta\tau_1 = \theta\tau_2$. It is called the most general unifier iff for each unifier θ' there is a substition σ such that $\theta' = \sigma \circ \theta$.

A program Π can be considered as a set of functions. The type associated to a function **fun** $f(x_1 : \tau_1, \ldots, x_k : \tau_k) : \tau = t$ is $\tau_1 \times \cdots \times \tau_k \mapsto \tau$. The *type environment* θ_Π *associated to a program* Π is the substitution defined by the functions $f \in \Pi$ together with their associated type. A *type environment* θ is simply a type substitution. Let $TYPEENV$ be the set of all type environments. A type τ is called the *most general common type* of types $\tau_1, \ldots, \tau_k$, if $\tau = \theta\tau_1 = \cdots = \theta\tau_k$ where θ is the most general unifier of $\tau_1, \ldots, \tau_k$. Now we are ready to define the type of expressions (i.e. words derivable from the non-terminal $\langle expr \rangle$, denoted by $EXPR$).

Definition 2.20 (Type of Expressions) *The* type of an expression under a type enviroment *is a mapping*

$$TYPE : EXPR \times TYPEENV \mapsto \Omega_V \uplus \{\textbf{error}\}$$

inductively defined over the structure of expressions (see figure 2.10). ■

The types of the basic operations are included in the type environment. Now we can define the notion of a correctly typed program:

Definition 2.21 (Correctly Typed Program) *A program* Π *is* correctly typed, *if for each function*

fun $f(x_1 : \tau_1, \ldots, x_k : \tau_k) : \tau = e$

$TYPE[\![e]\!]\ \theta\ (\neq \textbf{error})$ *and* τ *are unifiable, where* $\theta = \theta_o \circ \theta_\Pi$, θ_o *is the environment for the basic operations, and* θ_Π *the type environment associated with* Π. ■

Only correctly typed programs are considered for interpretation.

When interpreting programs, we have to take into account the parallel nature of the language, i.e. we need to assign to the evaluation the processor evaluating a particular expression. Hence, the number of the currently unavailable processors and the address of the processor evaluating the given expression have to be provided. The adresses are natural numbers. Hence, adequate mappings from more dimensional arrays to one dimensional arrays have to be used [ASU86, WG85]. If the vector elements are of variable length then techniques for compacting sparse arrays are used. We address

$$TYPE[\![v]\!]\,\theta = \theta(v)$$

$$TYPE[\![\langle e_0,\ldots,e_{n-1}\rangle]\!]\,\theta = \begin{cases} \langle\sigma\rangle & \text{if } \sigma \text{ is the most general common type of } TYPE[\![e_0]\!]\,\theta,\ldots,TYPE[\![e_{n-1}]\!]\,\theta \\ \textbf{error} & \text{if the most general common type does not exist} \\ \textbf{error} & \text{if at least one } TYPE[\![e_i]\!]\,\theta = \textbf{error} \end{cases}$$

$$TYPE[\![(e_1,\ldots,e_k)]\!]\,\theta = \begin{cases} \textbf{error} & \text{if at least one } TYPE[\![e_i]\!]\,\theta = \textbf{error} \\ \prod_{i=1}^{k} TYPE[\![e_i]\!]\,\theta & \text{otherwise} \end{cases}$$

$$TYPE[\![\langle\rangle]\!]\,\theta = \langle A\rangle$$

$$TYPE[\![c]\!]\,\theta = \tau \text{ if } c \in \tau$$

$$TYPE[\![f(e_1,\ldots,e_k)]\!] = \begin{cases} \textbf{error} & \text{if at least one } TYPE[\![e_i]\!]\,\theta = \textbf{error} \\ & \text{or, if } \theta(f) = \tau_1 \times \cdots \times \tau_l \mapsto \tau \text{ and either } k \neq l \\ & \text{or at least one } TYPE[\![e_i]\!]\,\theta \text{ is not unifiable with } \tau_i \\ \sigma\tau & \text{if } \theta(f) = \tau_1 \times \cdots \tau_k \mapsto \tau,\ \sigma \text{ is the most general unifier} \\ & \text{of } \prod_{i=1}^{k}\tau_i \text{ and } \prod_{i=1}^{k} TYPE[\![e_i]\!]\,\theta \end{cases}$$

$$TYPE[\![\textbf{skip}]\!]\,\theta = A$$

$$TYPE[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!]\,\theta = \begin{cases} \textbf{error} & \text{if } TYPE[\![e_i]\!]\,\theta = \textbf{error} \text{ for an } i \\ TYPE[\![e_2]\!]\,\theta \circ [v/TYPE[\![e_1]\!]\,\theta] & \text{otherwise} \end{cases}$$

$$TYPE[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]\,\theta =$$
$$\begin{cases} \textbf{error} & \text{if at least one } TYPE[\![e_i]\!]\,\theta = \textbf{error} \text{ or } TYPE[\![e_1]\!]\,\theta \neq \textbf{bool} \text{ or } TYPE[\![e_i]\!]\,\theta \\ & \text{are not unifiable for } i = 2,3 \\ \tau & \text{if } \tau \text{ is the most general common type of } TYPE[\![e_i]\!]\,\theta,\ i = 2,3 \end{cases}$$

$$TYPE[\![\textbf{forall } e_1 \leq i < e_2 \textbf{ do in parallel } e_3]\!]\,\theta =$$
$$\begin{cases} \textbf{error} & \text{if at least one } TYPE[\![e_i]\!]\,\theta = \textbf{error} \\ \langle TYPE[\![e_3]\!]\,\theta\rangle & \text{otherwise} \end{cases}$$

$$TYPE[\![\textbf{select } e_1 \leq i < e_2 \textbf{ in parallel from } e_3]\!]\,\theta =$$
$$\begin{cases} \textbf{error} & \text{if at least one } TYPE[\![e_i]\!]\,\theta = \textbf{error} \\ TYPE[\![e_3]\!]\,\theta & \text{otherwise} \end{cases}$$

$$TYPE[\![\textbf{modify } e_1,\ e_2 \leq i < e_3 \textbf{ to } e_4 \textbf{ from } e_5]\!]\,\theta =$$
$$\begin{cases} \textbf{error} & \text{if at least one } TYPE[\![e_i]\!]\,\theta = \textbf{error} \\ & \text{or } TYPE[\![e_5]\!]\,\theta \neq \langle\cdot\rangle \text{ or } TYPE[\![e_5]\!]\,\theta = \langle\tau\rangle, \text{ and } \tau \text{ and } TYPE[\![e_4]\!]\,\theta \text{ are not unifiable} \\ \langle\tau_1\rangle & \text{if } TYPE[\![e_5]\!]\,\theta = \langle\tau_1\rangle \text{ and } \tau_2 \text{ is the most general common type of } \tau_1 \text{ and } TYPE[\![e_4]\!]\,\theta \end{cases}$$

The symbols for the basic operations are dealt as variables, e_i stands for any expression (i.e. $e_i \in EXPR$), c for any constant (function), v for any variable, A for a type variable, and θ for any type environment.

Figure 2.10: Typing Rules

therefore processors by tuples of natural numbers. If ϕ is such a mapping, and k is the number of unavailable processors, then processor $P_{(i_1,\ldots,i_l)}$ has address $k + \phi(i_1,\ldots,i_l)$. Before coming to the definition of the semantics of programs similar tools as in the typing are provided:

A *substitution* ρ is a finite list of pairs $[v/e]$ where v is a variable, and $e \in EXPR$. It is interpreted as a mapping $EXPR \mapsto EXPR$ defined by (v, v' denote variables, e, e_i expressions, and f any function or operation symbol):

$$\begin{aligned}
(\langle\rangle\ \rho)v &= v \\
([v'/e]\ \rho)v &= \begin{cases} e & \text{if } v = v' \\ \rho v & \text{otherwise} \end{cases} \\
\rho f(e_1,\ldots,e_k) &= f(\rho e_1,\ldots,\rho e_k) \\
\rho\langle e_1,\ldots,e_k\rangle &= \langle \rho e_1,\ldots,\rho e_k\rangle \\
\rho(e_1,\ldots,e_k) &= (\rho e_1,\ldots,\rho e_k)
\end{aligned}$$

An *environment* is a substitution. The set of environments is denoted by ENV. Furthermore let $LOCAL = \bigcup_{k\geq 0} \mathbf{N}^k$ the address space of the processors, and $PROG$ the set of correctly typed programs, where programs are considered as set of functions.

The formal operational semantics is then defined by[3]:

Definition 2.22 (Operational Semantics) *The* operational semantics *of a PARFL program is a mapping*

$$EVAL : EXPR \times PROG \times \mathbf{N} \times LOCAL \times ENV \mapsto EXPR$$

where $EVAL[\![e]\!]\ \Pi\ n\ p\ \rho$ *is inductively defined by the equations in figure 2.11.* ■

Intuitively, $EVAL[\![e]\!]\ \Pi\ n\ p\ \rho$ means that expression e is evaluated under program Π and environment ρ by processor $P_{n-\phi(p)}$, where ϕ is the above stated mapping.

Remark: From the definition of the semantics it is obvious that the underlying machine model is a CREW-PRAM. The shared memory is necessary, because each processor must have access to the evaluated expressions, the exclusive-write property is because of the use of $\sqcap_{EW}$ in the definition for the **select** and **modify** expressions. Therefore it is not difficult to generalize this definition to concurrent writes. It is just necessary to replace $\sqcap_{EW}$ by the adequate $\sqcap_X$.

The machine needs for sychronization (i.e. determining whether all processors invoked by a parallel statement are idle) a global wired-or (or wired-and). The time for this synchronization can be assumed in practice as constant [PHT90]. This allows then for example checking in constant time whether two vectors (tuples) are equal.

All vectors are stored in the shared memory, all objects of the other type are hold locally in the processors (as well as their local address). ■

The language PARFL is strict in the sense that it evaluates its arguments before they are needed. It can be made lazy if in figure 2.11 following changes would be made:

- In the case of a function call the environment would be

$$\rho' = [x_1/e_1]\cdots[x_k/e_k]\ \rho$$

[3] $e(i)$ denotes an expression containing eventually i

For variables v: $EVAL[\![v]\!]\ \Pi\ n\ p\ \rho = \rho v$.
For any basic operation op:

$$EVAL[\![op(e_1,\ldots,e_k)]\!]\ \Pi\ n\ p\ \rho = op(EVAL[\![e_1]\!]\ \Pi\ n\ p\ \rho,\ldots,EVAL[\![e_k]\!]\ \Pi\ n\ p\ \rho)$$

$EVAL[\![\textbf{skip}]\!]\ \Pi\ n\ p\ \rho = \top$
$EVAL[\![\langle e_1,\ldots,e_k\rangle]\!]\ \Pi\ n\ p\ \rho = \langle EVAL[\![e_1]\!]\ \Pi\ n\ p\ \rho,\ldots,EVAL[\![e_k]\!]\ \Pi\ n\ p\ \rho\rangle$
$EVAL[\![(e_1,\ldots,e_k)]\!]\ \Pi\ n\ p\ \rho = (EVAL[\![e_1]\!]\ \Pi\ n\ p\ \rho,\ldots,EVAL[\![e_k]\!]\ \Pi\ n\ p\ \rho)$
For a function **fun** $f(x_1:\tau_1,\ldots,x_k:\tau_k):\tau = e$:

$$EVAL[\![f(e_1,\ldots,e_k)]\!]\ \Pi\ n\ p\ \rho = EVAL[\![e]\!]\ \Pi\ n\ p\ \rho'$$

where $\rho' = [x_1/EVAL[\![e_1]\!]\ \Pi\ n\ p\ \rho]\cdots[x_k/EVAL[\![e_k]\!]\ \Pi\ n\ p\ \rho]\ \rho$.
$EVAL[\![\textbf{let}\ v = e_1\ \textbf{in}\ e_2]\!]\ \Pi\ n\ p\ \rho = EVAL[\![e_2]\!]\ \Pi\ n\ p\ \rho'$ where $\rho' = [v/EVAL[\![e_1]\!]\ \Pi\ n\ p\ \rho]\ \rho$

$$EVAL[\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!]\ \Pi\ n\ p\ \rho = \begin{cases} EVAL[\![e_2]\!]\ \Pi\ n\ p\ \rho & \text{if } EVAL[\![e_1]\!]\ \Pi\ n\ p\ \rho = \textbf{true} \\ EVAL[\![e_3]\!]\ \Pi\ n\ p\ \rho & \text{if } EVAL[\![e_1]\!]\ \Pi\ n\ p\ \rho = \textbf{false} \\ \bot & \text{otherwise} \end{cases}$$

For the rest let be

$$\begin{aligned} l &= EVAL[\![e_1]\!]\ \Pi\ n\ p\ \rho \\ r &= EVAL[\![e_2]\!]\ \Pi\ n\ p\ \rho \end{aligned}$$

$$EVAL[\![\textbf{forall}\ e_1 \le i < e_2\ \textbf{do in parallel}\ e_3(i)]\!]\ \Pi\ n\ p\ \rho =$$
$$\langle EVAL[\![e_3(l)]\!]\ \Pi\ n+r-l\ (p,0)\ \rho,\ldots,EVAL[\![e_3(r-1)]\!]\ \Pi\ n+r-l\ (p,r-l-1)\ \rho\rangle$$

$$EVAL[\![\textbf{select}\ e_1 \le i < e_2\ \textbf{in parallel from}\ e_3(i)]\!]\ \Pi\ n\ p\ \rho =$$
$$EVAL[\![e_3(l)]\!]\ \Pi\ n+r-l\ (p,0)\ \rho\sqcap_{EW}\cdots\sqcap_{EW}EVAL[\![e_3(r-1)]\!]\ \Pi\ n+r-l\ (p,r-1)\ \rho$$

Define now:

$$\begin{aligned} m &= lg\ e_4 \\ I(i) &= \{j:\ \exists l \le i < r : e_0(j) = i\} \\ b(i) &= \begin{cases} (EVAL[\![e_4]\!]\ \Pi\ n\ p\ \rho) & \text{if } I(i) = \emptyset \\ \sqcap_{EW}\{EVAL[\![e_3(j)]\!]\ \Pi\ n+r-l\ (p,j-l)\ \rho : j \in I(i)\} & \text{otherwise} \end{cases} \end{aligned}$$

Then:

$$EVAL[\![\textbf{modify}\ e_0(i),\ e_1 \le i < e_2\ \textbf{to}\ e_3(i)\ \textbf{from}\ e_4]\!]\ \Pi\ n\ p\ \rho = \langle b(0),\ldots,b(m-1)\rangle$$

Figure 2.11: Operational Semantics

- For variables the equation would be $EVAL[\![v]\!]\ \Pi\ n\ p\rho = EVAL[\![\rho(v)]\!]\ \Pi\ n\ p\ (\rho - [v/\rho(v)])$

However, the complexity analysis of lazy functions is very difficult. This was the reason why we chose a strict semantics.

2.3.3 Complexity Measures

Now, we are ready to define complexity measures. The following measures are distinguished: time complexity (proportional to the number of parallel *EVAL*s necessary for the evaluation of expressions), work (proportional to the overall number of *EVAL*s), number of processors, space complexity (maximal size of intermediate expressions needed for the evaluation of expressions), the output length, and output size of expressions.

Hence, at first complexiy measures on data are defined. One already known is the length of a vector. The size of a data corresponds to the number of memory cells. The dimension of a cartesian produkt plays the role of the length of a vector, but it is already known at compile time.

Definition 2.23 (Size of Objects) *The* size *of objects is a function* $sz : \Omega^+ - \{\top, \bot\} \mapsto \mathbf{N}$, *inductively defined as follows:*

(i) *For any object* ω *in the basic types:* $sz(\omega) = 1$.

(ii) $sz(\langle\rangle) = 1$ *and* $sz((\,)) = 1$.

(iii) *For vectors:* $sz(\langle e_0, \ldots, e_{k-1}\rangle) = sz(e_0) + \cdots + sz(e_{k-1})$

(iv) *For tuples:* $sz((e_0, \ldots, e_{k-1})) = sz(e_0) + \cdots + sz(e_{k-1})$

■

The vector length has to be generalized to different levels, i.e. if for example $v \in \langle\langle A\rangle\rangle$, then $lg\ v$ counts only the dimension of the top-level vector. However, it is sometimes necessary to count the sum of dimensions of the vectors in v, i.e. if $lg\ v = n$ then $lg(v_0) + \cdots + lg(v_{n-1})$ must sometimes be counted. Therefore the length of the vectors is generalized to the *level-length* of vectors, i.e.

Definition 2.24 (Level-Length of Vectors) *The* level-length *of a vector is a familiy of functions* $ll_k : \mathbf{N} \times \underbrace{\langle\cdots\langle}_{k\ times}\ A\ \underbrace{\rangle\cdots\rangle}_{k\ times} \mapsto \mathbf{N}$, *inductivly defined by:*

(i) $ll_1 = lg$ *and* $ll_k(1, v) = lg(v)$.

(ii) $ll_k(i, v) = \sum_{j=0}^{lg(v)-1} ll_{k-1}(i-1, v[j])$

■

The definition of the time complexity is based on the time needed for the basic operations and the execution of the control structures. Their names are defined as:

τ_{op}	for each basic operation *op* (see figure 2.8 and definition 2.16)
τ_{var}	for access to variables
τ_{call}	for function calls
τ_{let}	for the evaluation of **let** expressions
τ_{if}	for the evaluation of conditional expressions
τ_{skip}	for the evaluation of **skip**
τ_{for}	for the evaluation of a **forall** expression
τ_{select}	for the evaluation of a **select** expression
τ_{modify}	for the evaluation of a **modify** expression

If each of these basic complexities is defined to be one, then the number of *parallel* calls of *EVAL* is counted. The complexity of evaluating an expression needs again an environment as it was used in definition 2.22. Hence, the definition of the time complexity needs the same notations and definitions as the definition of the operational semantics. However, it is not necessary to use processor allocation.

Definition 2.25 (Time Complexity) *The* time *for the evaluation of an expression* $e \in EXPR$ *under a correctly typed program* $\Pi \in PROG$ *and an environment* $\rho \in ENV$ *is inductively defined (see figure 2.12) by the function*

$$TIME : EXPR \times PROG \times ENV \mapsto \mathbf{N} \uplus \{\infty\}$$

The time complexity *of a function* $f : \tau_1 \times \cdots \tau_k \mapsto \tau \in \Pi \in PROG$ *is a function* $time\_f : \tau_1 \times \cdots \tau_k \mapsto \mathbf{N} \uplus \{\infty\}$ *where for each term* $t_1 \in \tau_1, \ldots, t_k \in \tau_k$ *holds:*

$$time\_f(t_1, \ldots, t_k) = TIME[\![f(t_1, \ldots, t_k)]\!]\ \Pi\ \langle\rangle$$

The worst case time complexity *of* f *w.r.t. to measures* $m_1, \ldots, m_k$ *on* $\tau_1, \ldots, \tau_k$ *is a function* $\overline{time\_f} : \mathbf{N}^k \mapsto \mathbf{N} \uplus \{\infty\}$ *where for each* $t_i \in \tau_i$ *with* $m_i(t_i) = n_i\ (= 1, \ldots, k)$

$$time\_f(t_1, \ldots, t_k) \leq \overline{time\_f}(n_1, \ldots, n_k)$$

is valid. ■

Observe that the worst case time complexity (as it will be for the other complexities) is just defined as an upper bound for the time complexity of a function. A complexity analysis should deliver at least the asumptotic worst case. Following observations can be made in figure 2.12: first the equations does not use the fact that concurrent writes are forbidden. They can therefore also used for the time complexity of any CRCW PRAM. Second the *EVAL*s start just with one processor their evaluation. This is formally justified because in the definition of time, it is sometimes only necessary to know the *value* of the expression, and not the processor actually used in the evaluation of this expression.

The definition of the other complexities are similar to the definition of the time complexity:

For variables v: $TIME[\![v]\!]\ \Pi\ \rho = \tau_{var}$
For any basic operarion op:

$$TIME[\![op(e_1,\ldots,e_k)]\!]\ \Pi\ \rho = \tau_{op} + TIME[\![e_1]\!]\ \Pi\ \rho + \cdots + TIME[\![e_k]\!]\ \Pi\ \rho$$

$TIME[\![\textbf{skip}]\!]\ \Pi\ \rho = \tau_{skip}$
For a function **fun** $f(x_1 : \tau_1, \ldots, x_k : \tau_k) = e$:

$$TIME[\![f(e_1,\ldots,e_k)]\!]\ \Pi\ \rho = \tau_{call} + TIME[\![e_1]\!]\ \Pi\ \rho + \cdots + TIME[\![e_k]\!]\ \Pi\ \rho + TIME[\![e]\!]\ \Pi\ \rho'$$

where $\rho' = [x_1/EVAL[\![e_1]\!]\ \Pi\ 1\ ()\ \rho] \cdots [x_k/EVAL[\![e_k]\!]\ \Pi\ 1\ ()\ \rho]\ \rho'$.
$TIME[\![\textbf{let}\ v = e_1\ \textbf{in}\ e_2]\!]\ \Pi\ \rho = \tau_{let} + TIME[\![e_1]\!]\ \Pi\ \rho + TIME[\![e_2]\!]\ \Pi\ \rho'$ where $\rho' = [v/EVAL[\![e_1]\!]\ \Pi\ 1\ ()\ \rho]\ \rho$.

$$TIME[\![\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!]\ \Pi\ \rho =$$
$$\begin{cases} \tau_{if} + TIME[\![e_1]\!]\ \Pi\ \rho + TIME[\![e_2]\!]\ \Pi\ \rho & \text{if } EVAL[\![e_1]\!]\ \Pi\ 1\ ()\ \rho = \textbf{true} \\ \tau_{if} + TIME[\![e_1]\!]\ \Pi\ \rho + TIME[\![e_3]\!]\ \Pi\ \rho & \text{if } EVAL[\![e_1]\!]\ \Pi\ 1\ ()\ \rho = \textbf{false} \\ \infty & \text{otherwise} \end{cases}$$

For the rest, let be $l = EVAL[\![e_1]\!]\ \Pi\ 1\ ()\ \rho$ and $r = EVAL[\![e_2]\!]\ \Pi\ 1\ ()\ \rho$. Then:

$$TIME[\![\textbf{forall}\ e_1 \le i < e_2\ \textbf{do in parallel}\ e_3(i)]\!]\ \Pi\ \rho =$$
$$\tau_{for} + TIME[\![e_1]\!]\ \Pi\ \rho + TIME[\![e_2]\!]\ \Pi\ \rho + \max_{i=l}^{r-1} TIME[\![e_3(i)]\!]\ \Pi\ \rho$$

$$TIME[\![\textbf{select}\ e_1 \le i < e_2\ \textbf{in parallel from}\ e_3(i)]\!]\ \Pi\ \rho =$$
$$\tau_{select} + TIME[\![e_1]\!]\ \Pi\ \rho + TIME[\![e_2]\!]\ \Pi\ \rho + \max_{i=l}^{r-1} TIME[\![e_3(i)]\!]\ \Pi\ \rho$$

Let now $I = \{EVAL[\![e_0(i)]\!]\ \Pi\ 1\ ()\ \rho : l \le i < r\}$:

$$TIME[\![\textbf{modify}\ e_0(i),\ e_1 \le i < e_2\ \textbf{to}\ e_3(i)\ \textbf{from}\ e_4]\!]\ \Pi\ \rho =$$
$$\tau_{modify} + \max_{i=l}^{r-1} TIME[\![e_0(i)]\!]\ \Pi\ \rho + TIME[\![e_1]\!]\ \Pi\ \rho + TIME[\![e_2]\!]\ \Pi\ \rho + \max_{k \in I} TIME[\![e_3(k)]\!]\ \Pi\ \rho$$
$$+ TIME[\![e_4]\!]\ \Pi\ \rho$$

Figure 2.12: Time Equations

Definition 2.26 (Processor Complexity) *The* processor complexity *for the evaluation of an expression $e \in EXPR$ under a correctly typed program $\Pi \in PROG$ and environment $\rho \in ENV$ is the function*

$$PROC : EXPR \times PROG \times ENV \mapsto \mathbf{N} \uplus \{\infty\}$$

inductively defined by figure 2.13. The processor complexity *of a function $f : \tau_1 \times \cdots \times \tau_k \mapsto \tau \in \Pi \in PROG$ is a function $proc\_f : \tau_1 \times \cdots \times \tau_k \mapsto \mathbf{N} \uplus \{\infty\}$ defined by ($e_i \in \tau_i$, $i = 1, \ldots, k$):*

$$proc\_f(e_1, \ldots, e_k) = PROC[\![f(e_1, \ldots, e_k)]\!]\ \Pi\ \rho$$

. The worst case processor complexity of f w.r.t. to measures m_i on τ_i ($i = 1, \ldots, k$) is a function

$$\overline{proc\_f} : \mathbf{N}^k \mapsto \mathbf{N}$$

satisfying for all $e_i \in \tau_i$ with $m_i(e_i) = n_i$:

$$proc\_f(e_1, \ldots, e_k) \leq \overline{proc\_f}(n_1, \ldots, n_k)n$$

■

The remarks after definition 2.25 are here also valid.

The space complexity is similar to the processor complexity. Only the first three equations need to be changed. All the other equations are the same. Thus:

Definition 2.27 (Space Complexity) *The* space complexity *for the evaluation of an expression $e \in EXPR$ under a correctly typed program $\Pi \in PROG$ and environment $\rho \in ENV$ is the function*

$$SPACE : EXPR \times PROG \times ENV \mapsto \mathbf{N} \uplus \{\infty\}$$

inductively defined analgous to figure 2.13 with the exception of the first three equations which are:

For variables v: $SPACE[\![v]\!]\ \Pi\ \rho = sz(\rho(v))$
For any basic operation op:
$SPACE[\![op(e_1, \ldots, e_k)]\!]\ \Pi\ \rho = \max\{\max_{i=1}^{k} SPACE[\![e_i]\!]\ \Pi\ \rho, sz(EVAL[\![op(e_1, \ldots, e_k)]\!]\ \Pi\ 1\ ()\ \rho)\}$
$SPACE[\![\mathbf{skip}]\!]\ \Pi\ \rho = 0$

The space complexity *of a function $f : \tau_1 \times \cdots \times \tau_k \mapsto \tau \in \Pi \in PROG$ is a function $space\_f : \tau_1 \times \cdots \times \tau_k \mapsto \mathbf{N} \uplus \{\infty\}$ defined by ($e_i \in \tau_i$, $i = 1, \ldots, k$):*

$$space\_f(e_1, \ldots, e_k) = SPACE[\![f(e_1, \ldots, e_k)]\!]\ \Pi\ \rho$$

. The worst case space complexity of f w.r.t. to measures m_i on τ_i ($i = 1, \ldots, k$) is a function

$$\overline{space\_f} : \mathbf{N}^k \mapsto \mathbf{N}$$

satisfying for all $e_i \in \tau_i$ with $m_i(e_i) = n_i$:

$$space\_f(e_1, \ldots, e_k) \leq \overline{space\_f}(n_1, \ldots, n_k)n$$

■

For variables v: $PROC[\![v]\!]\ \Pi\ \rho = 1$
For any basic operarion op:

$$PROC[\![op(e_1,\ldots,e_k)]\!]\ \Pi\ \rho = \max_{i=1}^{k} PROC[\![e_i]\!]\ \Pi\ \rho$$

$PROC[\![\textbf{skip}]\!]\ \Pi\ \rho = 1$
For a function $\textbf{fun}\ f(x_1 : \tau_1,\ldots,x_k : \tau_k) = e$:

$$PROC[\![f(e_1,\ldots,e_k)]\!]\ \Pi\ \rho = \max\{\max_{i=1}^{k} PROC[\![e_i]\!]\ \Pi\ \rho, PROC[\![e]\!]\ \Pi\ \rho'\}$$

where $\rho' = [x_1/EVAL[\![e_1]\!]\ \Pi\ 1\ ()\ \rho]\cdots[x_k/EVAL[\![e_k]\!]\ \Pi\ 1\ ()\ \rho]\ \rho'$.
$PROC[\![\textbf{let}\ v = e_1\ \textbf{in}\ e_2]\!]\ \Pi\ \rho = \max\{PROC[\![e_1]\!]\ \Pi\ \rho, PROC[\![e_2]\!]\ \Pi\ \rho'\}$ where $\rho' = [v/EVAL[\![e_1]\!]\ \Pi\ 1\ ()\ \rho]\ \rho$.

$$PROC[\![\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!]\ \Pi\ \rho = \begin{cases} \max\{PROC[\![e_1]\!]\ \Pi\ \rho, PROC[\![e_2]\!]\ \Pi\ \rho\} & \text{if } EVAL[\![e_1]\!]\ \Pi\ 1\ ()\ \rho = \textbf{true} \\ \max\{PROC[\![e_1]\!]\ \Pi\ \rho, PROC[\![e_3]\!]\ \Pi\ \rho\} & \text{if } EVAL[\![e_1]\!]\ \Pi\ 1\ ()\ \rho = \textbf{false} \\ \infty & \text{otherwise} \end{cases}$$

For the rest, let be $l = EVAL[\![e_1]\!]\ \Pi\ 1\ ()\ \rho$ and $r = EVAL[\![e_2]\!]\ \Pi\ 1\ ()\ \rho$. Then:

$$PROC[\![\textbf{forall}\ e_1 \le i < e_2\ \textbf{do in parallel}\ e_3(i)]\!]\ \Pi\ \rho = \max\{PROC[\![e_1]\!]\ \Pi\ \rho, PROC[\![e_2]\!]\ \Pi\ \rho, \sum_{i=l}^{r-1} PROC[\![e_3(i)]\!]\ \Pi\ \rho\}$$

$$PROC[\![\textbf{select}\ e_1 \le i < e_2\ \textbf{in parallel from}\ e_3(i)]\!]\ \Pi\ \rho = \max\{PROC[\![e_1]\!]\ \Pi\ \rho, PROC[\![e_2]\!]\ \Pi\ \rho, \sum_{i=l}^{r-1} PROC[\![e_3(i)]\!]\ \Pi\ \rho\}$$

Let now $I = \{EVAL[\![e_0(i)]\!]\ \Pi\ 1\ ()\ \rho : l \le i < r\}$:

$$PROC[\![\textbf{modify}\ e_0(i),\ e_1 \le i < e_2\ \textbf{to}\ e_3(i)\ \textbf{from}\ e_4]\!]\ \Pi\ \rho = \max\{\sum_{i=l}^{r-1} PROC[\![e_0(i)]\!]\ \Pi\ \rho, PROC[\![e_1]\!]\ \Pi\ \rho, PROC[\![e_2]\!]\ \Pi\ \rho, \sum_{k\in I} PROC[\![e_3(k)]\!]\ \Pi\ \rho, PROC[\![e_4]\!]\ \Pi\ \rho\}$$

Figure 2.13: Processor Complexity

Other complexities more used in automatic complexity analysis are the output length, output size, and output level-length of an expression or function. We show the definition for the output length. The other measures are defined analogously (named *SIZE* and LL_k, respectively):

Definition 2.28 (Output Length) *The* output length *of an expression $e \in EXPR$ under a correctly typed program $\Pi \in PROG$ and environment $\rho \in ENV$ is the function*

$$LENGTH : EXPR \times PROG \times ENV \mapsto \mathbf{N} \uplus \{\infty\}$$

defined by $LENGTH[\![e]\!]\ \Pi\ \rho = lg(EVAL[\![e]\!]\ \Pi\ 1\ ()\ \rho)$. *The* output length *of a function* $f : \tau_1 \times \cdots \times \tau_k \mapsto \tau \in \Pi \in PROG$ *is a function* $length\_f : \tau_1 \times \cdots \times \tau_k \mapsto \mathbf{N} \uplus \{\infty\}$ *defined by* $(e_i \in \tau_i,\ i = 1, \ldots, k)$*:*

$$length\_f(e_1, \ldots, e_k) = LENGTH[\![f(e_1, \ldots, e_k)]\!]\ \Pi\ \rho$$

The worst case output length of f w.r.t. to measures m_i on τ_i $(i = 1, \ldots, k)$ is a function

$$\overline{length\_f} : \mathbf{N}^k \mapsto \mathbf{N}$$

satisfying for all $e_i \in \tau_i$ with $m_i(e_i) = n_i$:

$$length\_f(e_1, \ldots, e_k) \leq \overline{length\_f}(n_1, \ldots, n_k)n$$

■

An automatic complexity analysis system has to derive a closed form expression for the worst case complexities. It is called *correct*, if the derived complexity is a worst case complexity of the required measure for each obtained result. Observe that an automatic complexity analysis system can never be complete (i.e. the complexity of each computable function can be derived), because otherwise the halting problem would be decidable. It is therefore a goal to enable the analysis of algorithms defined by the most common algorithm design principles, but it should be clear that there will always be algorithms, which cannot be analyzed. Furthermore if an automatic complexity analysis system obtains result, they should be good in the sense that it matches the asymptotic worst case (i.e. it is asymptotically not just an upper bound).

Chapter 3

Foundations of the Analysis Method

The analysis process consists as in the case of sequential algorithms [Zim90a] of a transformational phase and an algebraic phase. In the transformational phase, the program is transformed into a set of recurrence equations, which are solved in the algebraic phase. For reasons of completeness we give the whole transformation process which consists of the rules in [Zim90a] and some additional rules dealing with the parallel computations.

Most of the recurrences occuring in the analysis process are geometric recurrences. Their solution algorithms are already implemented in Maples `rsolve` [CGG+88]. Some more difficult recurrences can be solved as in [Zim90a], or via generating functions [FV90, FS87, Fla88, FSZ91, Zim91].

Finally, the method is demonstrated by two simple examples. First the classical pointer jumping for list ranking [GR88, KR90], and the second example is an adaptive algorithm for prefix sums [GR88].

3.1 The Transformational Phase

The transformational phase of the analysis process consists of four main parts. Let Π be the program to be analyzed for the complexity measure C. It is first translated into a program Π' computing the complexity C, i.e. whenever an expression E is computed by Π, the complexity w.r.t C of this expression is computed by Π'. After this transformation the results are algebraically simplified.

The second substep consists of a normalization procedure preparing the program Π' for the third substep, which derives equations defining the complexity C of Π'. Finally these equations are translated into recurrences by adequate mappings from vectors to natural numbers.

The goal is to make the transformation into recurrences complete, i.e. recurrences are always obtained for correctly typed programs. We reject the former choice of repeating the process with the output length, output level length and output size, if necessary, and analyze instead these measures at the same time with measure C (this choice is different from previous work [Weg75, Zim90a, Zim90b]). We will see that otherwise finding adequate measurs in step 4 have to use a time consuming algorithm a second time.

Another source of making the analysis process incomplete in previous works [Weg75, Zim90a,

Zim90b] are the analysis of output measures like $hd(g(t_1,\ldots,t_k))$ or $tl(g(t_1,\ldots,t_k))$, when g is a function defined in the program Π. These methods provide no way to analyze such measures. We transform therefore first Π to eliminate such subterms.

Thus the transformation algorithm is:

Algorithm *transform*
INPUT: A program Π, and a complexity measure C
OUTPUT: A recurrence system R, describing the complexity C of Π and a set of M required measures on argument positions

$R := \emptyset$
$M := \emptyset$
$\Pi' :=$ *remove_incompleteness*(Π);
transform Π according to C by the method of subsection 3.1.1 [Let this program be Π_0]
let k be the maximal nesting depth of vectors in Π;
transform $\Pi \cup \Pi'$ according to *LENGTH*; [This program is defined to be Π_1]
for $i = 2,\ldots,k$ **do**
  transform $\Pi \cup \Pi'$ according to LL_i; [This program is defined to be Π_i]
end for
transform $\Pi \cup \Pi'$ according to *SIZE*; [This program is defined to be Π_{k+1}]
$\hat{\Pi} := \Pi_0 \cup \Pi_1 \cup \cdots \cup \Pi_{k+1}$;
normalize $\hat{\Pi}$ by the method of subsection 3.1.2 [Let Π'' be the normalized program
derive a set of equations E from Π'' by applying the method of subsection 3.1.3
derive from E and Π' by the method of subsection 3.1.4 the recurrence equations R,
and the required measures to M;
output R

■

During the transformation process equations are created. In order to provide a notion of correctness, it is necessary to extend definition 2.22 to equations. Let Π be a program and E be a set of equations, $n \in \mathbf{N}$, $p \in LOCAL$, $\rho \in ENV$, and $f(t_1,\ldots,t_n) \in EXPR$. Each equation of E has the form $LHS \equiv RHS$, where $LHS, RHS \in EPXR$ and LHS is of the form $g(u_1,\ldots,u_m)$, where the t_i consist only of variables and constructing basic operation (i.e. they don't contain operations like *fst*, *tl*, etc.) Then the evaluation of expressions in figure 2.11 is replaced by

$$EVAL[\![f(t_1,\ldots,t_n)]\!]\ (\Pi \cup E)\ n\ p\ \rho = \begin{cases} EVAL[\![RHS]\!]\ (\Pi \cup E)\ n\ p\ \rho'' & \text{if there is a } LHS \equiv RHS \in E \text{ and } \sigma \text{ s.t. } f(t_1',\ldots,t_n') = \sigma(LHS) \\ EVAL[\![B]\!]\ (\Pi \cup E)\ n\ p\ \rho'\rho & \text{if there is no matching equation and} \\ & \textbf{fun } f(x_1:\tau_1,\ldots,x_n:\tau_n):\tau = B \in \Pi \\ \textbf{error} & \text{otherwise} \end{cases}$$

where $t_i' := EVAL[\![t_i]\!]\ (\Pi \cup E)\ n\ p\ \rho$, $\rho' = \sigma\rho$, and $\rho'' = [x_1/t_1'] \cdots [x_n/t_n']\rho$. Observe that the equations are applied by pattern matching, and that they have priority over functions. The latter is done for formal reasons to ensure that when equations are added, that they are actually really used in the evaluation of terms. Furthermore, recurrences are special equations, they contain only variables of type *nat*, and the type of the LHS and RHS is also *nat*.

The idea of subsection 3.1.4 is to apply one of the complexity measures *lg*, *sz*, or ll_k to the argument positions of the functions defined by E. When this is done the E become recurrences. In order to obtain adequate results these mappings must be kept during the analysis process. Informally,

algorithm *transform* is correct, if the semantics of the output R together with M describe under $EVAL$ the same as C.

Observe that if the analysis process is complete (i.e. it terminates always) it makes sense to give upper bounds for their running time. The parameters we need there is the length of the program, the maximal nesting depth (outside of conditionals), and the maximal natural number occuring in the program. We will see that the running time of the analysis is linear in the length of the program while it grows very fast (even faster than exponential) in the other two measures. Hence, as expected, not the length of the program makes the analysis process difficult, but its structure.

We define in advance the algorithm *remove_incompleteness* because it is somehow a preprocessing step. After the application of *remove_incompleteness*, the program will not contain any expressions like $hd(g(t_1,\ldots,t_n))$ or $tl(g(t_1,\ldots,t_n))$. The algorithm preserves the semantics, and so w.r.t. the output measures, the transformed program will deliver the same result.

The algorithm *remove_incompleteness* replaces expressions like these discussed above. During this removal process, new functions must added to the program Π.

This algorithm (and also the normalization step) makes use of program transformation rules. There are two type of transformation rules. Rules of the first type just transform a subexpression into another one, if eventually some syntactic conditions are satisfied. They have the form

$$\frac{construct_1}{construct_2}(parameters) \qquad conditions \qquad \text{type 1}$$

The constructs are program schemes, i.e. they denote partial syntax trees containing some non-terminals, here denoted by calligraphic letters $\mathcal{A},\mathcal{B},\mathcal{C},\ldots$. The conditions may contain some of these non-terminals, and the parameters may guide the application. Examples of rules guided by a parameter are rules 7 and rules 8. These type of rules is applied to a program Π as follows:

Algorithm *apply_rule_1*
INPUT: A program Π, and a rule of type 1
OUTPUT: A program Π'

[Match a subtree of the syntaxtree of Π with $construct_1$]
Find a subtree t of the syntaxtree of Π and a tree substitution σ s.t. $t = \sigma(construct_1$.
if no such t and σ exists **then output** Π **end if**
if $\sigma(cond) = true$ **then**
    replace in the syntaxtree of Π the subtree t by $\sigma(construct_2)$;
end if;
output Π

■

A subtree t as described in the above algorithm is called the *redex* of rule r.

The second type of rules enhance the program by new functions. They have therefore the form:

$$\frac{construct_1}{construct_2}(parameters) \qquad conditions$$
$$def$$

where the notations are as above and *def* is a function definition. These kind of rules are applied to a program Π as the rules of type 1. Additionaly in the **then**-part in algorithm *apply_rule_1*, a statement $\Pi := \Pi \cup \{\sigma(def)\}$ is added.

We are now ready to define algorithm *remove_incompleteness*.

Algorithm *remove_incompleteness*

INPUT:	A program Π
OUTPUT:	A program Π', s.t. Π' does not contain any sub term of the form $\mu(g(\cdots))$ where $g \in \Pi'$ and Π' is under *EVAL* equivalent to Π.
VARIABLE:	A representing new functions to be created

$A := \emptyset$;
while rule 1 is applicable in Π **do** [loop 1]
  apply rule 1 in R on a subterm $\mu(f(t_1, \ldots, t_n)$;
  if $\mu\_f \notin \Pi$ **then** $A := A \cup \{\mu\_f\}$; **end if**
end while;
while $A \neq \emptyset$ **do** [loop 2]
  take a $\mu\_f \in A$;
  $A := A - \{\mu\_f\}$;
  Let **fun** $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = B \in \Pi$ be the corresponding definition
  [Eliminate leading calls of *tl*]
  Let $F = \{f_1, \ldots, f_k\}$ be the mutually recursive closure defined by definition 3.1;
  if $\exists f \in F$: rule 4 applicable **then**
    for each $f \in F$ **do** [loop 2.1]
      $\Pi := \Pi - \{$**fun** $f(x_1 : \tau_1, \ldots, x_k : \tau_k) : \tau = B\}$;
      while rule 4 is applicable on B with F,*op*(B), and m **do** [loop 2.1.1]
        apply rule 4 on B with F,$top(B)$, and m;
      end while;
      For each $t \in top(B)$ where rule 4 was not applied, replace t by $tl*(t, m)$;
      Let B' be the transformed body;
      $\Pi := \Pi \cup \{$**fun** $f(x_1 : \tau_1, \ldots, x_n : \tau_n, m : nat) : \tau = B'\}$;
      while rule 5 is applicable in Π with f**do** [loop 2.1.2]
        apply rule 5 with f;
      end while;
    end for;
    $$\Pi := \Pi \cup \left\{ \begin{array}{ll} \textbf{fun} & tl*(l : \langle A \rangle, m : nat) : \langle A \rangle = \\ & \textbf{if } m = 0 \textbf{ then } l \textbf{ else } tl*(tl(l), m-1) \end{array} \right\};$$
  end if;
  Apply rule 2 on the new definition of f;
  Let **fun** $\mu\_f(pars) : \tau' = B'$ be the result;
  while rule 3 is applicable in B' **do** [loop 2.2]
    apply rule 3 in B' with μ;
    simplify the result according to figures 2.8 and 3.5;
  end while;
  Let B'' the result of these transformations;
  $\Pi := \Pi \cup \{$**fun** $\mu\_f(pars) : \tau' = B''\}$;

while rule 1 is applicable in Π on a subterm $\nu(g(t_1,\ldots,t_p))$
for a function body B_k **do** [loop 2.3]
apply rule 1 in Π on the subterm $\nu(g(t_1;\ldots,t_p))$;
if $\nu\_g \notin \Pi$ **then** $A := A \cup \{\nu\_g\}$; **end if**;
end while;
end while;
output Π and R

■

Loop 1 collects all the new functions to be created The new functions are created in loop 2. When creating a function μ_f then μ is applied symbolically to the body of f (loop 2.2). But then it could be necessary to create new functions. This can only happen, when μ is applied to a term t which is a function call. Therefore the only terms where new functions must be created are "top-level" terms, i.e. if

if $\mathcal{C}$ **then** $\mathcal{E}_1$ **else** $\mathcal{E}_2$

is the body of f, then the "top-level" terms are $\mathcal{E}_1$ and $\mathcal{E}_2$, and μ is applied to them by rule 1. If the $\mathcal{E}_i$ are themselves conditionals then the notion of "top-level" is defined recursively, otherwise a new function may only only to be created if $\mathcal{E}_i = \mu_1(\cdots(\mu_k(g(\cdots))\cdots)$. Loop 2.1 is necessary for the termination of algorithm *remove_incompleteness*. Consider for example the function

```
fun f(l:<nat>,n:nat):<nat> =
  if n=0 then l
  else tl(f(cons(n,l),n-1))
```

If hd_f must be created, then we would have after performing loop 2.2:

```
fun hd_f(l:<nat>,n:nat):nat =
  if n=0 then hd(l)
  else hd(tl(f(cons(n,l),n-1)))
```

Therefore tl_f has to be created. After the execution of loop 2.3. This function would be

```
fun tl_f(l:<nat>,n:nat):<nat> =
  if n=0 then tl(l)
  else tl(tl_f(cons(n,l),n-1)))
```

But now tl_tl_f has to be created. Loop 2.1 transforms Π into a program without such situations. For example it transforms the above function into

```
fun f(l:<nat>,n:nat,m:nat):<nat> =
  if n=0 then tl*(l,m)
  else f(cons(n,l),n-1,m+1)

fun tl*(l:<A>,m:nat):<A>) =
  if m=0 then l
  else tl*(tl(l),m+1}
```

and every other function call `f(t,u)` is replaced by `f(t,u,0)`.

Algorithm *remove_incompleteness* would not terminate in this situation, if it wouldn't be avoided by loop 2.1. In appendix A we prove that such situations cannot occur with the other basic operations, if the program is correctly typed.

To summarize the thoughts, we have to consider expressions of the form $\mu_1(\cdots(\mu_k(g(\cdots))\cdots)$ on a "top-level", and they may lead algorithm *remove_incompleteness* it g is a recursive call. We therefore base the definition of mutually recursive on top-level expressions. Algorithm *remove_incompleteness* will be discussed in more detail in appendix A.

Definition 3.1 (Top-Level Recursive Closure) **(a)** *The set* $top(\mathcal{E})$ *of* top-level expressions *of an expression* $\mathcal{E}$ *is defined as follows:*

- $top(g(t_1,\ldots,t_k)) = \{g(t_1,\ldots,t_k)\}$, $g \in \Pi$,
- $top(\mu_1(\cdots(\mu_k(g(t_1,\ldots,t_k)))\cdots)) = \{\mu_k(g(t_1,\ldots,t_k)\}$, $k \geq 1$, $g \in \Pi$, μ_i *basic operations.*
- $top(\textbf{if } \mathcal{C} \textbf{ then } \mathcal{E}_1 \textbf{ else } \mathcal{E}_2) = top(\mathcal{E}_1) \cup top(\mathcal{E}_2)$
- $top(t) = \emptyset$, *otherwise*

(b) *The* top-level calling graph $T_\Pi = (V, E)$ *is defined by:*

- $V = \{f | \textbf{fun } f(x_1 : \tau_1,\ldots,x_k : \tau_k) \in \Pi = B\}$,
- $E = \{(f,g) | \textbf{fun } f(x_1 : \tau_1,\ldots,x_k : \tau_k) = B \in \Pi, \exists t_1,\ldots,t_n,\mu : g(t_1,\ldots,t_n) \in top(B) \vee \mu(g(t_1,\ldots,t_n))\}$

(c) *A set of function* $F = \{f_1,\ldots,f_m\}$ *are called* top-level mutually recursive *in* Π *if they are a strongly connected component in* T_Π.

The used rules are:

Rule 1 (Elimination of Basic Operations)

$$\frac{\mu(f(\mathcal{T}_1,\ldots,\mathcal{T}_n)}{\mu\_f(\mathcal{T}_1,\ldots,\mathcal{T}_n)}(\mu, f) \qquad \mu \neq (\cdot)[\cdot] \text{ is a basic function, } f \in \Pi \text{ of arity } n$$

$$\frac{f(\mathcal{T}_1,\ldots,\mathcal{T}_n)[i]}{access\_f(\mathcal{T}_1,\ldots,\mathcal{T}_n,i)}(\mu, f) \qquad \mu = (\cdot)[\cdot],\ f \in \Pi \text{ of arity } n$$

This rule performs the elimination of basic operation μ in connection with f. In order to keep equivalence, a new function μ_f must be created by applying symbolically μ to the body of f.

Rule 2 (Folding μ into f)

$$\frac{\textbf{fun } f(\mathcal{P}) : \mathcal{T} = \mathcal{E}}{\textbf{fun } \mu\_f(\mathcal{P}) : \mathcal{U} = \mu(\mathcal{E})}(\mu, f) \qquad \mu \neq (\cdot)[\cdot] \text{ is a basic operation of type } \mathcal{T} \mapsto \mathcal{U}, \text{ and } f \in \Pi$$

$$\frac{\textbf{fun } f(\mathcal{P}) : \langle \mathcal{T} \rangle = \mathcal{E}}{\textbf{fun } access\_f(\mathcal{P}, i : nat) : \mathcal{T} = \mathcal{E}[i]}(\mu, f) \qquad \mu = (\cdot)[\cdot], \text{ and } f \in \Pi$$

The next rule propagates μ through expressions:

Rule 3 (Propagation of μ in Expressions)

$$\frac{\mu(\textbf{if } \mathcal{C} \textbf{ then } \mathcal{E}_1 \textbf{ else } \mathcal{E}_2)}{\textbf{if } \mathcal{C} \textbf{ then } \mu(\mathcal{E}_1) \textbf{ else } \mu(\mathcal{E}_2)}$$

$$\frac{(\textbf{if } \mathcal{C} \textbf{ then } \mathcal{E}_1 \textbf{ else } \mathcal{E}_2)[\mathcal{I}]}{\textbf{if } \mathcal{C} \textbf{ then} \mathcal{E}_1[\mathcal{I}] \textbf{ else } \mathcal{E}_2[\mathcal{I}]}$$

$$\frac{\mu(\textbf{let } \mathcal{X} = \mathcal{E}_1 \textbf{ in } \mathcal{E}_2)}{\textbf{let } \mathcal{X} = \mathcal{E}_1 \textbf{ in } \mu(\mathcal{E}_2)}$$

$$\frac{(\textbf{let } \mathcal{X} = \mathcal{E}_1 \textbf{ in } \mathcal{E}_2)[\mathcal{J}]}{\textbf{let } \mathcal{X} = \mathcal{E}_1 \textbf{ in } \mathcal{E}_2[\mathcal{J}]}$$

$$\frac{\mu(\textbf{for all } \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ do in parallel } \mathcal{E}(\mathcal{I}))}{\textbf{for all } \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ do in parallel } \mu(\mathcal{E}(\mathcal{I}))}$$

$$\frac{(\textbf{for all } \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ do in parallel } \mathcal{E}(\mathcal{I}))[\mathcal{J}]}{\mathcal{E}(\mathcal{J})}$$

$$\frac{\mu(\textbf{select } \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ in parallel from } \mathcal{E}(\mathcal{I}))}{\textbf{select } \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ in parallel from } \mu(\mathcal{E}(\mathcal{I}))}$$

$$\frac{(\textbf{select } \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ in parallel from } \mathcal{E}(\mathcal{I})[\mathcal{J}]}{\textbf{select } \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ in parallel from } \mu(\mathcal{E}(\mathcal{I})[\mathcal{J}]}$$

$$\frac{\mu(\textbf{modify } \mathcal{A}[\mathcal{I}],\ \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ to } \mathcal{E}(\mathcal{I}) \textbf{ from } \mathcal{V})}{\textbf{modify } \mathcal{A}[\mathcal{I},\ \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ to } \mu(\mathcal{E}(i)) \textbf{ from } \mu(\mathcal{V})}$$

$$\frac{(\textbf{modify } \mathcal{A}[\mathcal{I}],\ \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ to } \mathcal{E}(\mathcal{I}) \textbf{ from } \mathcal{V})[\mathcal{J}]}{\mathcal{E}(\mathcal{K})} \qquad \text{if } \mathcal{A}[\mathcal{K}] = \mathcal{J}$$

$$\frac{(\textbf{modify } \mathcal{A}[\mathcal{I}],\ \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ to } \mathcal{E}(\mathcal{I}) \textbf{ from } \mathcal{V})[\mathcal{J}]}{\mathcal{V}[\mathcal{J}]} \qquad \text{if } \forall \mathcal{L} \leq i < \mathcal{R} : \ \mathcal{A}[\mathcal{K}] \neq \mathcal{J}$$

Observe that it is impossible to simplify a access to the result of the modify statement, if the index is just symbolic, because it is impossible to decide whether it is one of the modifying indices or not.

The following rule eliminates *tl*s surrounding top-level expressions in mutually recursive functions.

Rule 4 (Elimination of *tl*s)

$$\frac{tl^k(f(\mathcal{P}))}{f(\mathcal{P}, n+k)}(F, T, n) \qquad f \in F \wedge k \geq 0 \wedge f(\mathcal{P}) \in T \vee tl(f(\mathcal{P})) \in T$$

Finally the last rule replaces all other function calls of f with a new call where the last argument is 0, i.e. it has not yet surrounding *tls*.

Rule 5 (New Initializations)

$$\frac{f(\mathcal{A}_1,\ldots,\mathcal{A}_k)}{f(\mathcal{A}_1,\ldots,\mathcal{A}_k,0)}(def) \qquad def = \mathbf{fun}(\mathcal{X}_1:\mathcal{T}_\infty,\ldots,\mathcal{X}_k:\mathcal{T}_k):\mathcal{T}=\mathcal{B}$$

Rules 4 together with the new function created in loop 2.1, and rule 5 corresponds to the transformation [BW80, page 295].

Lemma 3.2 *Algorithm remove_incompleteness is correct and terminates.*

Proof: Suppose the algorithm terminates. Rule 1 is always applied together with rule 2. Therefore it is just a fold transformation with the new function definition. This transformation is therefore correct (see e.g. [BD77]). The other transformations in rule 3 just propagate a basic function μ through expressions. Hence, these rules also preserve semantics. Finally the body of loop 2.1 together with rules 4 together with the new function created and the execution of loop 2.1.1. preserve semantics because they correspond to the application of the transformation in [BW80, page 295] with several additional foldings.

It remains now to show, that algorithm *remove_incompleteness* terminates. This proof is difficult, because when new functions are created by rule 2, new redeces of rule 1 may occur. This proof is long and requires some additional proofs. It is therefore given in appendix A ■

3.1.1 Transformation in Complexity Computing Programs

A translation process **TC** w.r.t. a complexity measure C is given to translate a program Π into a complexity computing program Π'. For each function $f(x_1,\ldots,x_n)\in\Pi$ a function $C\_f(x_1,\ldots,x_n)$ such that $C_f(t_1,\ldots,t_n)$ yields the complexity C of computing $f(t_1,\ldots,t_n)$. The result of the translation if afterwards being simplified by standard algebraic simplification and by some additional simplification rules taking into account that the *worst case* is analyzed. The main algorithm is:
Algorithm *translate*
INPUT: A program Π, a set of equations R, and a complexity measure C.
OUTPUT: A program Π' and a set of recurrence equations R' satisfying theorems 3.3 – 3.32

case C **of**
TIME: $\Pi := \Pi \cup \{\mathbf{TTIME}[\![def]\!] \mid def\in\Pi\}$;
PROC: $\Pi := \Pi \cup \{\mathbf{TPROC}[\![def]\!] \mid def\in\Pi\}$;
SPACE: $\Pi := \Pi \cup \{\mathbf{TSPACE}[\![def]\!] \mid def\in\Pi\}$;
LENGTH: $\Pi := \Pi \cup \{\mathbf{TLENGTH}[\![def]\!] \mid def\in\Pi\}$;
$R := \{LHS = \mathbf{TLENGTH}[\![RHS]\!] \mid LHS = RHS\in R\}$;
SIZE: $\Pi := \Pi \cup \{\mathbf{TSIZE}[\![def]\!] \mid def\in\Pi\}$;
$R := \{LHS = \mathbf{TSIZE}[\![RHS]\!] \mid LHS = RHS\in R\}$;
LL_k: $\Pi := \Pi \cup \{\mathbf{TLL}_k[\![def]\!] \mid def\in\Pi\}$;
$R := \{LHS = \mathbf{TLL}_k[\![RHS]\!] \mid LHS = RHS\in R\}$;

$$\mathbf{TTIME}[\![v]\!] = \begin{cases} \tau_{var} & \text{if } v \text{ is a variable} \\ \tau_v & \text{if } v \text{ is a constant} \end{cases}$$

$$\mathbf{TTIME}[\![\mathbf{fun}\ f(x_1 : T_1, \ldots, x_k : T_k) : T = B]\!] = \mathbf{fun}\ time\_f(x_1 : T_1, \ldots, x_k : T_k) : nat = \mathbf{TTIME}[\![B]\!]$$

$$\mathbf{TTIME}[\![g(t_1, \ldots, t_n)]\!] = \begin{cases} \tau_g + \mathbf{TTIME}[\![t_1]\!] + \cdots + \mathbf{TTIME}[\![t_n]\!] & \text{if } g \text{ is a basic operation} \\ \tau_{call} + \mathbf{TTIME}[\![t_1]\!] + \cdots + \mathbf{TTIME}[\![t_n]\!] + time\_g(t_1, \ldots, t_n) & \text{otherwise} \end{cases}$$

$$\mathbf{TTIME}[\![\mathbf{skip}]\!] = \tau_{skip}$$

$$\mathbf{TTIME}[\![\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2]\!] = \tau_{let} + \mathbf{TTIME}[\![e_1]\!] + \mathbf{TTIME}[\![e_2]\!]$$

$$\mathbf{TTIME}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] = \mathbf{if}\ e_1\ \mathbf{then}\ \tau_{if} + \mathbf{TTIME}[\![e_1]\!] + \mathbf{TTIME}[\![e_2]\!]\ \mathbf{else}\ \tau_{if} + \mathbf{TTIME}[\![e_1]\!] + \mathbf{TTIME}[\![e_3]\!]$$

$$\mathbf{TTIME}[\![\mathbf{forall}\ e_1 \le i < e_2\ \mathbf{do\ in\ parallel}\ e_3(i)]\!] = \tau_{for} + \mathbf{TTIME}[\![e_1]\!] + \mathbf{TTIME}[\![e_2]\!] + \max_{i=e_1}^{e_2-1} \mathbf{TTIME}[\![e_3(i)]\!]$$

$$\mathbf{TTIME}[\![\mathbf{select}\ e_1 \le i < e_2\ \mathbf{in\ parallel\ from}\ e_3(i)]\!] = \tau_{select} + \mathbf{TTIME}[\![e_1]\!] + \mathbf{TTIME}[\![e_2]\!] + \max_{i=e_1}^{e_2-1} \mathbf{TTIME}[\![e_3(i)]\!]$$

$$\mathbf{TTIME}[\![\mathbf{modify}\ e_0(i),\ e_1 \le i < e_2\ \mathbf{to}\ e_3(i)\ \mathbf{from}\ e_4]\!] = \tau_{modify} + \max_{i=e_1}^{e_2-1} \mathbf{TTIME}[\![e_0(i)]\!] + \mathbf{TTIME}[\![e_1]\!] + \mathbf{TTIME}[\![e_2]\!] + \max_{i=e_1}^{e_2-1} \mathbf{TTIME}[\![e_3(i)]\!] + \mathbf{TTIME}[\![e_4]\!]$$

Figure 3.1: Translation to Time Functions

end case;
simplify Π and R algebraically and by the rules in figure 3.5;
output Π and R;

The translation schemes $\mathbf{TTIME}[\![\cdot]\!]$, $\mathbf{TPROC}[\![\cdot]\!]$, $\mathbf{TSPACE}[\![\cdot]\!]$, and $\mathbf{TLENGTH}[\![\cdot]\!]$ are defined in figures 3.1, 3.3, 3.2, and 3.4 respectively. The definitions of $\mathbf{TSIZE}[\![\cdot]\!]$ and $\mathbf{TLL}_k[\![\cdot]\!]$ are analogous to the definition of $\mathbf{TLENGTH}[\![\cdot]\!]$. ■

Together with algebraic simplification, the simplification rules of figure 3.5 are designed for the worst case complexity. Another simplification considers the basic operations on vectors i.e. rules like for example

$$lg(tl(v)) = lg(v) - 1 \text{ and } lg(fr(v)) = lg(v) - 1$$

are applied. The simplification ends if no simplification rule can be applied. It is easy but long to prove the following theorem by induction on expressions.

$$\mathbf{TSPACE}[\![v]\!] = \begin{cases} size(v) & \text{if } v \text{ is a variable} \\ 1 & \text{if } v \text{ is a constant} \end{cases}$$

$$\mathbf{TSPACE}[\![\mathbf{fun}\ f(x_1 : T_1, \ldots, x_k : T_k) : T = B]\!] = \mathbf{fun}\ space\_f(x_1 : T_1, \ldots, x_k : T_k) : nat = \mathbf{TSPACE}[\![B]\!]$$

$$\mathbf{TSPACE}[\![g(t_1, \ldots, t_n)]\!] =$$
$$\begin{cases} \max(\mathbf{TSPACE}[\![t_1]\!], \ldots, \mathbf{TSPACE}[\![t_n]\!]) & \text{if } g \text{ is a basic operation} \\ \max(\mathbf{TSPACE}[\![t_1]\!], \ldots, \mathbf{TSPACE}[\![t_n]\!], space\_g(t_1, \ldots, t_n)) & \text{otherwise} \end{cases}$$

$$\mathbf{TSPACE}[\![\mathbf{skip}]\!] = 0$$

$$\mathbf{TSPACE}[\![\mathbf{let}\ v = e_1\ \mathbf{in}\ e_2]\!] = \max(\mathbf{TSPACE}[\![e_1]\!], \mathbf{TSPACE}[\![e_2]\!]\ [v/e_1])$$

$$\mathbf{TSPACE}[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!] =$$
$$\mathbf{if}\ e_1\ \mathbf{then}\ \max(\mathbf{TSPACE}[\![e_1]\!], \mathbf{TSPACE}[\![e_2]\!])\ \mathbf{else}\ \max(\mathbf{TSPACE}[\![e_1]\!], \mathbf{TSPACE}[\![e_3]\!])$$

$$\mathbf{TSPACE}[\![\mathbf{forall}\ e_1 \le i < e_2\ \mathbf{do\ in\ parallel}\ e_3(i)]\!] =$$
$$\max\left(\mathbf{TSPACE}[\![e_1]\!], \mathbf{TSPACE}[\![e_2]\!]) + \sum_{i=e_1}^{e_2-1} \mathbf{TSPACE}[\![e_3(i)]\!]\right)$$

$$\mathbf{TSPACE}[\![\mathbf{select}\ e_1 \le i < e_2\ \mathbf{in\ parallel\ from}\ e_3(i)]\!] =$$
$$\max\left(\mathbf{TSPACE}[\![e_1]\!], \mathbf{TSPACE}[\![e_2]\!]) + \sum_{i=e_1}^{e_2-1} \mathbf{TSPACE}[\![e_3(i)]\!]\right)$$

$$\mathbf{TSPACE}[\![\mathbf{modify}\ e_0(i),\ e_1 \le i < e_2\ \mathbf{to}\ e_3(i)\ \mathbf{from}\ e_4]\!] =$$
$$\max\left(\sum_{i=e_1}^{e_2-1} \mathbf{TSPACE}[\![e_0(i)]\!], \mathbf{TSPACE}[\![e_1]\!], \mathbf{TSPACE}[\![e_2]\!], \sum_{i=e_1}^{e_2-1} \mathbf{TSPACE}[\![e_3(i)]\!], \mathbf{TSPACE}[\![e_4]\!]\right)$$

Figure 3.2: Translation to Space Functions

$\mathbf{TPROC[\![}v\mathbf{]\!]} = 1$ if v is a variable or a constant

$$\mathbf{TPROC[\![fun}\ f(x_1 : T_1, \ldots, x_k : T_k) : T = B\mathbf{]\!]} = \mathbf{fun}\ proc\_f(x_1 : T_1, \ldots, x_k : T_k) : nat = \mathbf{TPROC[\![}B\mathbf{]\!]}$$

$$\mathbf{TPROC[\![}g(t_1, \ldots, t_n)\mathbf{]\!]} =$$
$$\begin{cases} \max(\mathbf{TPROC[\![}t_1\mathbf{]\!]}, \ldots, \mathbf{TPROC[\![}t_n\mathbf{]\!]}) & \text{if } g \text{ is a basic operation} \\ \max(\mathbf{TPROC[\![}t_1\mathbf{]\!]}, \ldots, \mathbf{TPROC[\![}t_n\mathbf{]\!]}, proc\_g(t_1, \ldots, t_n)) & \text{otherwise} \end{cases}$$

$$\mathbf{TPROC[\![skip]\!]} = 1$$

$$\mathbf{TPROC[\![let}\ v = e_1\ \mathbf{in}\ e_2\mathbf{]\!]} = \max(\mathbf{TPROC[\![}e_1\mathbf{]\!]}, \mathbf{TPROC[\![}e_2\mathbf{]\!]})$$

$$\mathbf{TPROC[\![if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\mathbf{]\!]} =$$
$$\mathbf{if}\ e_1\ \mathbf{then}\ \max(\mathbf{TPROC[\![}e_1\mathbf{]\!]}, \mathbf{TPROC[\![}e_2\mathbf{]\!]})\ \mathbf{else}\ \max(\mathbf{TPROC[\![}e_1\mathbf{]\!]}, \mathbf{TPROC[\![}e_3\mathbf{]\!]})$$

$$\mathbf{TPROC[\![forall}\ e_1 \le i < e_2\ \mathbf{do\ in\ parallel}\ e_3(i)\mathbf{]\!]} =$$
$$\max\left(\mathbf{TPROC[\![}e_1\mathbf{]\!]}, \mathbf{TPROC[\![}e_2\mathbf{]\!]}) + \sum_{i=e_1}^{e_2-1} \mathbf{TPROC[\![}e_3(i)\mathbf{]\!]}\right)$$

$$\mathbf{TPROC[\![select}\ e_1 \le i < e_2\ \mathbf{in\ parallel\ from}\ e_3(i)\mathbf{]\!]} =$$
$$\max\left(\mathbf{TPROC[\![}e_1\mathbf{]\!]}, \mathbf{TPROC[\![}e_2\mathbf{]\!]}) + \sum_{i=e_1}^{e_2-1} \mathbf{TPROC[\![}e_3(i)\mathbf{]\!]}\right)$$

$$\mathbf{TPROC[\![modify}\ e_0(i),\ e_1 \le i < e_2\ \mathbf{to}\ e_3(i)\ \mathbf{from}\ e_4\mathbf{]\!]} =$$
$$\max\left(\sum_{i=e_1}^{e_2-1} \mathbf{TPROC[\![}e_0(i)\mathbf{]\!]}, \mathbf{TPROC[\![}e_1\mathbf{]\!]}, \mathbf{TPROC[\![}e_2\mathbf{]\!]}, \sum_{i=e_1}^{e_2-1} \mathbf{TPROC[\![}e_3(i)\mathbf{]\!]}, \mathbf{TPROC[\![}e_4\mathbf{]\!]}\right)$$

Figure 3.3: Translation to Processor Functions

$$\textbf{TLENGTH}[\![v]\!] = \begin{cases} \mathit{length}(v) & \text{if } v \text{ is a variable} \\ 0 & \text{if } v \text{ is a constant} \end{cases}$$

$$\textbf{TLENGTH}[\![\textbf{fun } f(x_1 : T_1, \ldots, x_k : T_k) : T = B]\!] = \textbf{fun } \mathit{length\_f}(x_1 : T_1, \ldots, x_k : T_k) : \mathit{nat} = \textbf{TLENGTH}[\![B]\!]$$

$$\textbf{TLENGTH}[\![g(t_1, \ldots, t_n)]\!] = \begin{cases} \mathit{lg}(g(t_1, \ldots, t_n)) & \text{if } g \text{ is a basic operation} \\ \mathit{length\_g}(t_1, \ldots, t_n) & \text{otherwise} \end{cases}$$

$$\textbf{TLENGTH}[\![\textbf{skip}]\!] = 0$$

$$\textbf{TLENGTH}[\![\textbf{let } v = e_1 \textbf{ in } e_2]\!] = \textbf{TLENGTH}[\![e_2[v/e_1]]\!]$$

$$\textbf{TLENGTH}[\![\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!] = \textbf{if } e_1 \textbf{ then TLENGTH}[\![e_2]\!] \textbf{ else TLENGTH}[\![e_3]\!]$$

$$\textbf{TLENGTH}[\![\textbf{forall } e_1 \leq i < e_2 \textbf{ do in parallel } e_3(i)]\!] = e2 - e1$$

$$\textbf{TLENGTH}[\![\textbf{select } e_1 \leq i < e_2 \textbf{ in parallel from } e_3(i)]\!] = \max_{i=e_1}^{e_2-1} \textbf{TLENGTH}[\![e_3(i)]\!]$$

$$\textbf{TLENGTH}[\![\textbf{modify } e_0(i),\ e_1 \leq i < e_2 \textbf{ to } e_3(i) \textbf{ from } e_4]\!] = \textbf{TLENGTH}[\![e_4]\!]$$

Figure 3.4: Translation to Length Functions

$$e_2 \leq e_3 \Rightarrow \max(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = e_3$$

$$e_2 > e_3 \Rightarrow \max(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) = e_2$$

$$(\forall i : e_1 < i < e_2 \Rightarrow e_3(i) = e_3(e_1)) \Rightarrow \max_{i=e_1}^{e_2-1} e_3(i) = e_3(e_1)$$

$$(\textbf{forall } l \leq i < r \textbf{ do in parallel } t(i))[j] = t(j)$$

Figure 3.5: Some Additional Simplification

Theorem 3.3 (Correctness and Completeness of Algorithm *translate*) *Algorithm translate terminates for each correcly typed program* Π *and complexity measure* $C \in \{TIME, PROC, SPACE, LENGTH, SIZE, LL_k\}$. *After termination holds for each expression* $e \in EXPR$ *and* $\rho \in ENV$:

(a) $TIME[\![e]\!]\ \Pi\ \rho \leq EVAL[\![\mathbf{TTIME[}e\mathbf{]}]\!]\ \Pi'\ 1\ ()\ \rho$
if $C = TIME$ *and* $\Pi' = \Pi \cup \{\mathbf{TTIME[}def\mathbf{]}\ |def \in \Pi\}$

(b) $SPACE[\![e]\!]\ \Pi\ \rho \leq EVAL[\![\mathbf{TSPACE[}e\mathbf{]}]\!]\ \Pi'\ 1\ ()\ \rho$
if $C = SPACE$ *and* $\Pi' = \Pi \cup \{\mathbf{TSPACE[}def\mathbf{]}\ |def \in \Pi\}$

(c) $PROC[\![e]\!]\ \Pi\ \rho \leq EVAL[\![\mathbf{TPROC[}e\mathbf{]}]\!]\ \Pi'\ 1\ ()\ \rho$
if $C = PROC$ *and* $\Pi' = \Pi \cup \{\mathbf{TPROC[}def\mathbf{]}\ |def \in \Pi\}$

(d) $LENGTH[\![e]\!]\ \Pi\ \rho \leq EVAL[\![\mathbf{TLENGTH[}e\mathbf{]}]\!]\ \Pi'\ 1\ ()\ \rho$
if $C = LENGTH$ *and* $\Pi' = \Pi \cup \{\mathbf{TLENGTH[}def\mathbf{]}\ |def \in \Pi\}$

(e) $SIZE[\![e]\!]\ \Pi\ \rho \leq EVAL[\![\mathbf{TSIZE[}e\mathbf{]}]\!]\ \Pi'\ 1\ ()\ \rho$
if $C = SIZE$ *and* $\Pi' = \Pi \cup \{\mathbf{TSIZE[}def\mathbf{]}\ |def \in \Pi\}$

(f) $LL_k[\![e]\!]\ \Pi\ \rho \leq EVAL[\![\mathbf{TLL}_k\mathbf{[}e\mathbf{]}]\!]\ \Pi'\ 1\ ()\ \rho$
if $C = LL_k$ *and* $\Pi' = \Pi \cup \{\mathbf{TLL}_k\mathbf{[}def\mathbf{]}\ |def \in \Pi\}$

Moreover simplifications of Π' *do not change the properties (a) - (f).*

3.1.2 Normalization

The goal of the normalization step is to bring the output of the transformation into an equivalent form suitable to further processing. The result of this step is a program containing no nested conditionals, having removed some irrelevant argument positions from complexity functions, and contain no conditional statement inside a maximum operation. The normalization is performed by applying some program transformation steps. Only the last transformation is performed by a fold-unfold tactic. Thus the normalization algorithm is as follows:

Algorithm *normalize*

INPUT: A program Π
OUTPUT: An equivalent program Π' satisfying definition 3.10.

while rule 6 is applicable **do** [loop 1]
    Apply rule 6;
end while;
$I :=$ *irrelevant_positions*(Π)
while $I \neq \emptyset$ **do** [loop 2]
    take an $f/i \in I$;
    apply rule 7 on Π with f/i;
    while rule 8 is applicable on Π with f/i **do** [loop 2.1]
        apply rule 8 on Π with f/i;

end while;
end while;
while rule 9 is applicable **do** [loop 3]
  let f' be a new symbol [i.e. not defined by Π or by figure 2.8]
  apply rule 9 on Π with f'
end while;
output Π

■

The elimination of nested conditionals is given by the following rule:

Rule 6 (Elimination of nested conditionals)

$$\frac{\textbf{if } \mathcal{C}_1 \textbf{ then if } \mathcal{C}_2 \textbf{ then } \mathcal{E}_1 \textbf{ else } \mathcal{E}_2 \textbf{ else } \mathcal{E}_3}{\textbf{if } \mathcal{C}_1 \wedge \mathcal{C}_2 \textbf{ then } \mathcal{E}_1 \textbf{ else if } \mathcal{C}_1 \wedge \neg\mathcal{C}_2 \textbf{ then } \mathcal{E}_2 \textbf{ else } \mathcal{E}_3}$$

The correctness of the rule (i.e. an application of rule 6 to a program Π does not change the semantics of Π) is obvious, and is for expample proven in a similar setting in [BW80]. After one application of rule 6 to a program Π, the number of the redeces in Π decreases obviously by one. Hence:

Lemma 3.4 *Loop 1 in algorithm* normalize *terminates, and after its termination the program* Π' *does not contain nested conditionals and is equivalent to the program* Π' *before loop 1*

The translation to complexity functions makes some of the parameters superflous. This can be easily seen by consideration of the rules for variables, because in some of them variables just disappear. This effect is the idea behind loop 2 of algorithm *normalize*. A more formal definition of this property is:

Definition 3.5 (Irrelevant Argument Positions) *Let*

$$\textbf{fun } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = B$$

a function definition in a program Π *(for example that after loop 1). An argument position* f/i $(1 \leq i \leq n)$ *is called* irrelevant *iff for all terms* $t_1, \ldots, t_n, u_i$ *of type* $\tau_1, \ldots, \tau_n, \tau_i$, *for all* $n \in \mathbf{N}$, *for all* $p \in LOCAL$ *and all* $\rho \in ENV$ *holds:*

$$EVAL[\![f(t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots, t_n)]\!]\ \Pi\ n\ p\ \rho = EVAL[\![f(t_1, \ldots, t_{i-1}, u_i, t_{i+1}, \ldots, t_n)]\!]\ \Pi\ n\ p\ \rho.$$

Otherwise, f/i *is called* relevant. *An argument position* f/i occurs in a term t, *iff* x_i *is a subterm of* t.

It is clear, that in general it is not decidable whether an argument position is irrelevant or not. However, it is possible to give an algorithm yielding a set of irrelevant argument positions of a program Π, but not necessarely all of them. This algorithm is an extension of that in [Weg75] and has been successfully used in the automatic complexity analysis of sequential algorithms [Zim90b].

Algorithm *irrelevant_positions*

INPUT: A program Π
OUTPUT: A set I of irrelevant positions in Π

$R := \emptyset$; [R is the set of eventually relevant position]
repeat
$R := R \cup \{f/i | f/i$ occurs in a non-recursive subexpression of $\Pi\}$
$\cup \{f/i | f/i$ occurs in a condition in $\Pi\}$
$\cup \{f/i | f/i$ occurs in an argument at a position in $R\}$
until R does not change;
output $\{f/i | f/i$ is an argument position in $\Pi\} - R$

■

In [Weg75] and in the generalized version in [Zim90b], this algorithm is used as a definition. It is straightforward to show that this algorithm is correct.

Lemma 3.6 (Correctness of Algorithm *irrelevant_positions*) *Algorithm irrelevant_positions applied onto a program* Π *terminates and outputs a set of irrelevant argument positions in* Π

Proof: The termination is obvious because there is only a finite number of argument positions in a program Π, and no iteration deletes argument positions from R. If no argument position is added to R, then the loop terminates. Thus there is only one execution of the loop body which adds no argument position to R. This property together with the finiteness of the number of argument positions ensures the termination.

Let I be the output of algorithm *irrelevant_positions*. It remains now to show that if $f/i \in I$, then f/i is in fact irrelevant. Observe first that if $f/i \in I$ and x is the parameter on f/i, then

(i) In the body of f, x doesn't occur in a condition, and

(ii) x doesn't occur in a non-recursive subexpression, and

(iii) x occurs only a subterm on a recursive call of a function g on the i-th position, if $g/i \in I$.

We prove now by induction on the number of (mutually) recursive function calls the following equivalent statement:

> For each $f/i \in I$ (**fun** $f(x_1 : \tau_1, \ldots, x_n : \tau_n) = B \in \Pi$ then for each $n \in \mathbf{N}$, $p \in LOCAL$, $\rho \in ENV$:
>
> $$EVAL[\![B]\!]\ \Pi\ n\ p\ \rho = EVAL[\![B]\!]\ \Pi\ n\ p\rho'$$
>
> where ρ' is any environment satisfying $\rho'(x) = \rho(x)$ for all $x \neq x_i$. Especially the values $\rho(x_i)$ and $\rho'(x_i)$ can be different.

INDUCTION BASIS: The number of mutually recursive function calls is 0.
Then (i) and (ii) are satisfied and there is no access to x_i during the evaluation. $EVAL[\![x_i]\!]\ \Pi\ n'\ p'\ \rho$ is never called. Thus, in this case the result of the evaluation doesn't depend on $\rho(x_i)$.

INDUCTION STEP: The same arguments as used in the induction basis show that the only possibility where the claim is not satisfied could be recursive function calls. Let

$$EVAL[\![g(t_1,\ldots,t_k)]\!]\ \Pi\ n'\ p'\ \rho$$

be such a function call and x_i be a subterm of t_j. By property (iii), it must be $g/j \in I$. Furthermore let be

$$\textbf{fun}\ g(y_1:\sigma_1,\ldots,y_k:\sigma_k):\sigma = B'$$

be the function definition of g in Π. Then by induction hypothesis:

$$EVAL[\![B']\!]\ \Pi n'\ p'\ \hat{\rho}\rho = EVAL[\![B']\!]\ \Pi n'\ p'\ \bar{\rho}\rho$$

and

$$EVAL[\![B']\!]\ \Pi n'\ p'\ \hat{\rho}\rho' = EVAL[\![B']\!]\ \Pi n'\ p'\ \bar{\rho}\rho'$$

where $\hat{\rho} = [y_1/EVAL[\![t_1]\!]\ \Pi\ n'\ p'\ \rho]\cdots[y_j/EVAL[\![t_j]\!]\ \Pi\ n'\ p'\ \rho]\cdots[y_k/EVAL[\![t_k]\!]\ \Pi\ n'\ p'\ \rho]$ and $\bar{\rho} = [y_1/EVAL[\![t_1]\!]\ \Pi\ n'\ p'\ \rho]\cdots[y_j/EVAL[\![t_j]\!]\ \Pi\ n'\ p'\ \rho']\cdots[y_k/EVAL[\![t_k]\!]\ \Pi\ n'\ p'\ \rho]$. If x_i is a subterm of B' then $x_i \in \{y_1,\ldots,y_k\}$, for if not then the evaluation would not be correctly typed in the sense of definition 2.21. By definition we of the access to variables in the environment we have therefore:

$$(\hat{\rho}\rho)(x_i) = \hat{\rho}(x_i) = (\hat{\rho}\rho')(x_i)$$

and therefore

$$EVAL[\![B']\!]\ \Pi n'\ p'\ \hat{\rho}\rho = EVAL[\![B']\!]\ \Pi n'\ p'\ \hat{\rho}\rho'$$

Thus

$$EVAL[\![g(t_1,\ldots,t_k)]\!]\ \Pi\ n'\ p'\ \rho = EVAL[\![g(t_1,\ldots,t_k)]\!]\ \Pi\ n'\ p'\ \rho'$$

■

By definition, irrelevant argument positions can be removed safely from a program, i.e. the corresponding rules for function definition and function application are therefore

Rule 7 (Remove Irrelevant Argument Positions From Function Definitions)

$$\frac{\textbf{fun}\ f(\mathcal{P}_1,\ldots,\mathcal{P}_{i-1},\mathcal{P}_i,\mathcal{P}_{i+1},\ldots,\mathcal{P}_n):\mathcal{T}=\mathcal{E}}{\textbf{fun}\ f(\mathcal{P}_1,\ldots,\mathcal{P}_{i-1},\mathcal{P}_{i+1},\ldots,\mathcal{P}_n):\mathcal{T}=\mathcal{E}}(f/i)$$

Rule 8 (Remove Irrelevant Argument Positions From Function Calls)

$$\frac{f(\mathcal{E}_1,\ldots,\mathcal{E}_{i-1},\mathcal{E}_i,\mathcal{E}_{i+1},\ldots,\mathcal{E}_n)}{f(\mathcal{E}_1,\ldots,\mathcal{E}_{i-1},\mathcal{E}_{i+1},\ldots,\mathcal{E}_n)}(f/i)$$

It is nearly obvious from definition 3.5 and from lemma 3.6:

Lemma 3.7 *Loop 2 of algorithm* normalize *terminates, and after its termination, the program* Π' *is equivalent to the program before the loop and the argument positions in* I *are removed.*

Proof: Suppose first that the inner loop always terminates. The body of the outer loop is executed once for each element in I. Now let $f/i \in I$. Then after execution of one iteration of the inner loop, the number of redeces of rule 8 with the argument f/i is reduced by one. Therefore the inner loop also terminates.

In order to prove the correctness suppose first that after application of rule 7, the original function is still in the program. Let $f(t_1,\ldots,t_n)$ be a redex of rule 8. Suppose further that $EVAL$ chooses the function with the approbriate arguments. Let $f/i \in I$. Because of lemma 3.6, the argument position f/i is irrelevant. But then, by the definition 3.5:

$$EVAL[\![f(t_1,\ldots,t_{i-1},t_i,t_{i+1},\ldots,t_n)]\!]\ \Pi'\ n\ p\ \rho = EVAL[\![f(t_1,\ldots,t_{i-1},t_{i+1},\ldots,t_n)]\!]\ \Pi'\ n\ p\ \rho$$

for any $n \in \mathbf{N}$, $p \in LOCAL$, and $\rho \in ENV$. Hence, after the execution of the inner loop the semantics of the program is not changed. Because then the argument position f/i is eliminated from each function call, all calls of f have now $n-1$ arguments, and the original function f with n arguments is never called. Thus the old function definition of f with n arguments can be removed from the program without changing its semantics. Hence, one execution of the body of loop 2 does not change the semantics of the program and removes the selected argument position. ■

The third important transformation is to eliminate conditionals as arguments of the maximum-operation. The most easy transformation is to create a new function having the free variables of the condional expression as parameters and the conditional expression itsself as the body. Then the argument of the maximums-operation is replaced by a function call:

Rule 9 (Elimination of Conditionals in Maximums)

$$\frac{\max_{i=l}^{r}\ \textbf{if}\ \mathcal{C}\ \textbf{then}\ \mathcal{E}_1\ \textbf{else}\ \mathcal{E}_2}{\begin{array}{l}\max_{i=l}^{r} f(x_1,\ldots,x_n)\\ \textbf{fun}\ f(x_1:\tau_1,\ldots,x_n:\tau_n):\tau = \textbf{if}\ \mathcal{C}\ \textbf{then}\ \mathcal{E}_1\ \textbf{else}\ \mathcal{E}_2\end{array}}(f)$$

$x_1,\ldots,x_n$ are the free variables in $\mathcal{C}$, $\mathcal{E}_1$ and $\mathcal{E}_2$. τ_i are the appropriate types of the variables x_i, and τ is the unified type of $\mathcal{E}_1$ and $\mathcal{E}_2$

This transformation consists just of folding a conditional with the new function definition. Fold-transformations are correct as often proven in literature (e.g. [BD77, BW80]). Furthermore, in each execution of loop 3, the number of redeces of rule 9 decreases by one. Thus:

Lemma 3.8 *Loop 3 in algorithm* normalize *terminates, and after its termination the program* Π' *is equivalent to the program before loop 3.*

Putting lemmas 3.4 to 3.8 together we have:

Theorem 3.9 (Correctness of *normalize*) *Algorithm* normalize *terminates, and after its termination its output R and Π' is equivalent to its input and satisfy definition 3.10*

After execution of algorithm *normal* Π and R is said to be in normal form, i.e.

Definition 3.10 (Normal Form) *A program Π is called in normal form, iff*

- *Each function body is of the form*

  if C_1 **then** $\mathcal{E}_1$
  else if C_2 **then** $\mathcal{E}_2$
  $\vdots$
  else $\mathcal{E}_k$

- *No function body contains let-expressions*
- *No conditional is inside a maximum-operation*

This step transformes a program in a very nice form for further processing, and is crucial for making the transformational phase complete.

3.1.3 Derivation of Symbolic Equations

The goal is now to transform the program into an equivalent set of equations. The notion of equivalence is based on definition 3.11. The transformation is performed by evaluating the functions with symbolic arguments and hoping that the conditions evaluate to true or false. If this is not possible then conditional equations are created.

The main algorithm of this subsection uses the definition of a term value. A value of a term t is the set of terms, where each variable in t is substituted by any term of its type. More formally the value of a term t containing variables $x_1, \ldots, x_n$ or types $T_1, \ldots, T_n$, respectiveley is definded by:

$$val(t) = \bigcup_{i=1}^{n} \bigcup_{t_i \in T_i} \{[x_1/t_1] \cdots [x_n/t_n]t\}$$

The value of a finite set S of terms is the union of the value of its members.

After transforming the program obtained by the normalization phase into equations, the symbolic evaluation is as follows:

Algorithm *symbolic_evaluation*

INPUT: A set of equations E

OUTPUT: An equivalent set of equations E', s.t. each RHS in E' of the form **if** C **then** $\cdots$ implies that there are no finite sets S_1 and S_2 whose values form a partition of a certain type and have the following property:

- $\forall(t_1,\ldots,t_n) \in val(S_1) : EVAL[\![C]\!]\ \emptyset\ 1\ ()\ [x_1/t_1]\cdots[x_n/t_n] = true$
- $\forall(t_1,\ldots,t_n) \in val(S_2) :$

  $EVAL[\![C]\!]\ \emptyset\ 1\ ()\ [x_1/t_1]\cdots[x_n/t_n] = false$

where $x_1,\ldots,x_n$ are the variables in C.

(1) $E' := \{LHS = RHS \in E | RHS \neq \textbf{if} \cdots\}$;
(2) $E := E - E'$;
[The set E' is the set to be examined]
(3) **while** $E \neq \emptyset$ **do** [loop 1]
(4) **let** $LHS = \textbf{if}\ C\ \textbf{then}\ E_1\ \textbf{else}\ E_2 \in E$
(5) remove this equation from E;
(6) **let** $\{x_1,\ldots,x_n\}$ be the variables of C with types $T_1,\ldots,T_n$;
(7) **if** C contains a non-basic operation $\vee$
(8) C contains a subterm of type *nat* which contains more than one variable or
(9) is not linear in its variables [see lemma 3.14 $\vee$
(10) C contains a condition of the form $f(n) < g(m)$ where $n \neq m$ are variables $\vee$
(11) C contains a term of the form $a[t]$ where t is not constant
(12) **then** [This equation is not symbolically evaluable]
(13) $E' := E' \cup \{LHS = \textbf{if}\ C\ \textbf{then}\ E_1\ \textbf{else}\ E_2\}$;
(14) **else**
(15) **assume** C in disjunctive normal form;
(16) find a finite set S of terms such that [see algorithm *instantiate_condition*]
(i) $\forall(t_1,\ldots,t_n) \in val(S) : EVAL[\![C]\!]\ \emptyset\ 1\ ()\ [x_1/t_1]\cdots[x_n/t_n] = true$
(ii) $\forall(t_1,\ldots,t_n) \in T_1 \times \cdots \times T_n - val(S) : EVAL[\![C]\!]\ \emptyset\ 1\ ()\ [x_1/t_1]\cdots[x_n/t_n] = false$
[The following step can be performed by algorithm *complete*]
(17) Find a set S' s.t. $val(S') = T_1 \times \cdots \times T_n - val(S)$;
[The following condition can be checked by algorithm *consistent*]
(18) **if** S and S' don't form a partition of $T_1 \times \cdots \times T_n$
(19) **then** $E' := E' \cup \{LHS = \textbf{if}\ C\ \textbf{then}\ E_1\ \textbf{else}\ E_2\}$;
(20) **else** [The following two steps can be performed by algorithm *minimize*
(21) Find a minimal subset $S_1 \subseteq S$ s.t. $val(S_1) = val(S)$;
(22) Find a minimal subset $S_2 \subseteq S'$ s.t. $val(S_2) = val(S')$;
(23) **for each** $(s_1,\ldots,s_n) \in S_1$ **do** [loop 2]
(24) $\sigma := [x_1/s_1]\cdots[x_n/s_n]$;
(25) $E' := E' \cup \{\sigma LHS = \sigma E_1\}$;
(26) **end for**;
(27) **for each** $(s_1,\ldots,s_n) \in S_2$ **do** [loop 3]
(28) $\sigma := [x_1/s_1]\cdots[x_n/s_n]$;
(29) **if** E_2 is a conditional **then**
(30) $E := E \cup \{\sigma LHS = \sigma E_2\}$;

(31) **else** $E' := E' \cup \{\sigma LHS = \sigma E_2\}$; **end if**;
(32) **end for**;
(33) **end if**;
(34) **end if**;
(35) **end while**;
(36) **output** E'

■

Before performing any correctness proof and complexity analysis of algorithm *symbolic_eval*, we define the algorithms used there, and analyze their complexity.

First, the precise definition of the semantics of equations is given. Observe, that a function can now be defined by more than one equation. The semantics is then closed to the standard semantics of strict functional languages, i.e. if evaluating a term, the equation with a matching LHS is chosen. If there are more equations with a matching LHS, then the most specific equation is chosen:

Definition 3.11 (Semantics of Equations) **(a)** *Let* $f(t_1, \ldots, t_n) = RHS \in E$ *be an equation. Then* f *is a* by E defined symbol.

(b) *Let* t *be a term,* E *be a set of equations. The* set of variables $V(t)$ of t *is inductively defined as follows:*

$$V(x) = \begin{cases} \{x\} & \text{if } x \text{ is not a by } E \text{ defined symbol and not a basic operation} \\ \emptyset & \text{otherwise} \end{cases}$$

$$V(f(t_1, \ldots, t_n)) = V(t_1) \cup \cdots \cup V(t_n)$$

(c) *Let* t *be a term,* E *be a set of equations. The set*

$$A(t) = \{LHS = RHS | \exists \text{substition } \sigma : \sigma\ LHS = t\}$$

is called the set of applicable (or matching) equations in E for t. *The equation* $LHS = RHS \in A(t)$ *s.t. for all* $LHS' = RHS'$ *there is a substitution* σ' *with* $\sigma'\ LHS' = LHS$ *is called the* most specific applicable equation.

(d) *In figure 2.11 the evaluation rule for function application is replaced by:*

$$EVAL[\![f(t_1, \ldots, t_n)]\!]\ E\ n\ p\ \rho = EVAL[\![RHS]\!]\ E\ n\ p\ \sigma\rho$$

where $LHS = RHS$ *is the most specific applicable equation in* E, *and* $\sigma\ LHS = f(t_1, \ldots, t_n)$.

Suppose now that a condition C has variables $V(C) = \{x_1, \ldots, x_n\}$ and x_i is type T_i. Then algorithm *symbolic_eval* has to find a partition of $S_{true} \uplus S_{false} = T_1 \times \cdots \times T_n$ such that for all $(t_1, \ldots, t_n) \in S_{true}$ the condition C evaluatesw to *true* under the environment $[x_1/t_1] \cdots [x_n/t_n]$. Similarly C must evaluate to *false*, if it is instantiated with terms in S_{false}. This partition has to be defined by finite sets of terms whose values are S_{true} and S_{false}, respectively. Furthermore these two sets of terms should be as small as possible. Thus we define

Definition 3.12 (Finite Description of A Partition of Type A) *Two sets of terms S_1 and S_2* describe a partition of A, *iff $val(S_1) \uplus val(S_2) = A$. They are called a* finite description of partition of A, *if both S_1 and S_2 are finite. Finally, such a set S_i is called* minimal, *if for any different terms $s_1, s_2 \in S_i$, there is no substititon σ such that $\sigma\ s_1 = s_2$ or $\sigma\ s_2 = s_1$.*

Thus a set S of terms is minimal, iff for any proper subset $S' \subset S$ also $val(S') \subset val(S)$. We start now with defining a decision algorithm for the above definition, and obtain then from this algorithm a new algorithm computing a partition. The next two lemmas provide important properties for finite descriptions of partitions of natural numbers. We will see that such descriptions can contain only constant or linear terms. These lemmas are based on well-known theorems of number theory. It is assumed that the basic operations on natural numbers are addition, multiplication, and computing the remainder.

Lemma 3.13 *Let $A_i = \{a_i\ n + b_i \mid n \geq 0\}$, $1 \leq i \leq k$. Then*

$$\exists n_0 \{n | n \geq n_0\} \subseteq \bigcup_{i=1}^{k} A_i \Leftrightarrow \exists a \forall 0 \leq i < a \exists j \exists n \in A_j : n \equiv i \bmod a$$

This is a standard number theoretic result, and proven by defining a to the least common multiple of $a_1, \ldots, a_k$. In this case $n_0 = \max_{1 \leq i \leq k} b_i$ is sufficient.

Remark: This lemma provides a methodology for checking whether a set of at most linear terms describe $\mathbf{N}$. First extract the linear terms $a_1\ n + b_i, 1 \leq i \leq k$ of this set, and choose $a = \mathrm{lcm}(a_1, \ldots, a_k)$. Then check whether

$$\bigcup_{i=1}^{k} \{b_i + j\ a_i\ | 0 \leq j < a/a_i\} = \{0, \ldots, a-1\}$$

If this is not the case, then by lemma 3.13, this set cannot describe $\mathbf{N}$. Otherwise, check whether all terms less than $b = \max(b_1, \ldots, b_k)$ can be obtained by S either by constant terms in S or by choosing finite values of n and evaluating with these values the linear terms (checking this for $n < b$ would be sufficient. ■

The other important result from number theory is

Lemma 3.14 *Let be $A_i = \{P_i(n) \mid n \geq 0\}$, $1 \leq i \leq k$, where the P_i are polynomials (or faster increasing functions). If there is a $n_0 \in \mathbf{N}$ s.t:*

$$\{n | n \geq n_0\} \subseteq A_1 \cup \cdots \cup A_k$$

then there are A_{i_j}, $1 \leq j \leq l$ such that P_{i_j} is linear and

$$\{n | n \geq n_1\} \subseteq A_{i_1} \cup \cdots \cup A_{i_l}$$

for a $n_1 \in \mathbf{N}$.

The proof is based on the fact that functions growing faster than linear functions grow so fast that only a finite set of linear functions can cover the gaps left by this fast growing function. But then it can be shown, that this finite set of linear functions cover already each large natural number.

This lemmas allows to exclude higher degree polynomials from consideration in finding finite descriptions of partitions of **N**. Moreover if there are polynomials in one of the two sets, then it follows immediately that the values these sets are either not disjoint, or their union is not **N**. Thus, if some polynomials occur in a condition C, this condition cannot be evaluated symbolically in finite steps, and hence a conditional equation is created.

The following algorithm, based on lemmas 3.13 and 3.14, checks whether the value of a given set S of terms describe a type A. Later this algorithm is modified, s.t. it returns a set S' s.t $S \cup S'$ describes A.

Algorithm *complete*

INPUT: A finite set $S = \{t_1, \ldots, t_n\}$ of terms, and a type T, where all t_i are of type T
OUTPUT: *true*, if $T = val(S)$, *false* otherwise.

(1) **if** $S = \{a\}$ **then output**(*true*); **end if**
(2) **case** T **of**
(3) nat : **assume** $S = \{a_i\ n + b_i | 1 \leq i \leq k\} \cup \{c_j | 1 \leq j \leq l\}$;
(4) **if** $k = 0$ **then output**(*false*); **end if**;
(5) $b := \max\{b_1, \ldots, b_k\}$;
(6) $a := \mathrm{lcm}(a_1, \ldots, a_k)$;
(7) $C := \{c_j | 1 \leq j \leq l\} \cup \{a_i\ n + b_i | 1 \leq i \leq k, a_i\ n + b_i \leq b\}$;
(8) **if** $\bigcup_{i=1}^{k} \{b_i + j\ a_i \bmod a | 0 \leq j < a/a_i\} = \{0, \ldots, a-1\}$
(9) **then output**($C = \{0, \ldots, b-1\}$);
(10) **else output**(*false*); **end if**;
(11) $\langle B \rangle$: **if** $S = \{\langle\rangle\} \vee \langle\rangle \notin S\}$ **then output**(*false*); **end if**;
(12) **assume** $S = \{\langle\rangle, cons(a_{1,1}, l_1), \ldots, cons(a_{k_1,1}, l_1), \ldots, cons(a_{1,m}, l_m), \ldots, cons(a_{k_m,m}, l_m)\}$;
(13) **output** $\left(\bigwedge_{i=1}^{m} complete(\{a_{1,i}, \ldots, a_{k_i,i}\}, B) \wedge complete(\{l_1, \ldots, l_m\}, \langle B \rangle) \right)$;
(14) $B \times C$: **assume** $S = \{(a_1, b_{1,1}), \ldots, (a_1, b_{k_1,1}), \ldots, (a_m, b_{1,m}), \ldots, (a_m, b_{k_m,m})\}$;
(15) **output** $\left(\bigwedge_{i=1}^{m} complete(\{b_{1,i}, \ldots, b_{k_i,i}\}, C) \wedge complete(\{a_1, \ldots, a_m\}, B) \right)$;
(16) **end case**;

■

Lemma 3.15 (Correctness of Algorithm *complete***)** *Algorithm* complete *terminates and returns* true *if and only if* $val(S) = A$.

Proof: The termination follows from the complexity results in lemma 3.16. Thus, suppose that algorithm *complete* terminates on input S and A.

The correctness is proven by induction over the structure of types:

CASE 1: $A = nat$. The correctness follows from lemmas 3.13 and 3.14

CASE 2: $A = \langle B \rangle$. This case is proven by induction on the cardinality of S.
CASE 2.1: $card(S) = 1$. If $S = \{l\}$ then $val(S) = \langle B \rangle$, and algorithm *complete* returns *true*. Otherwise $S = \{\langle\rangle\}$ or $S = \{cons(t_1, t_2)\}$ for some terms t_1, t_2. In both cases algorithm *complete* returns *false*.
CASE 2.2: $card(S) > 1$. Then, by the induction hypothesis for the induction over the type structure, for all $1 \leq i \leq m$:

$$complete(\{a_{1,i}, \ldots, a_{k_i,i}\}, B) = true \Leftrightarrow val(\{a_{1,i}, \ldots, a_{k_i,i}\}) = B$$

By the induction hypothesis on $card(S)$ we have:

$$complete(\{l_1, \ldots, l_m\}, \langle B \rangle) = true \Leftrightarrow val(\{l_1, \ldots, l_m\}) = \langle B \rangle$$

Suppose now $complete(S, \langle B \rangle) = true$. Then it follows from the above observations:

$$\begin{aligned} val(S) &= \{\langle\rangle\} \cup \bigcup_{i=1}^{m} \bigcup_{j=1}^{k_i} \{cons(a'_j, l'_i) | a'_j \in val(a_j), l'_j \in val(l_j)\} \\ &= \{\langle\rangle\} \cup \bigcup_{i=1}^{m} \{cons(a, l_i) | a \in B\} \\ &= \{\langle\rangle\} \cup \{cons(a, l) | a \in B,\ l \in \langle B \rangle\} \\ &= \langle B \rangle \end{aligned}$$

Suppose now $complete(S, \langle B \rangle) = false$. Then there is one i such that $complete(\{a_{1,i}, \ldots, a_{k_i,i}\}, B) = false$ or $complete(\{l_1, \ldots, l_m\}, \langle B \rangle) = false$. In the former case is a $b_i \in B - val(\{a_{1,i}, \ldots, a_{k_i,i}\})$. Then for any $l \in val(l_i)$, we have $cons(b_i, l) \notin val(S)$. A similar argument show in the latter case, that $val(S) \neq \langle B \rangle$.

CASE 3: $A = B \times C$. This can be proven similarly to the case $A = \langle B \rangle$. ■

Lemma 3.16 (Time Complexity of Algorithm *complete*) . *Let $S = \{t_1, \ldots, t_n\}$ be a set of terms and $m = sz(t_1) + \cdots + sz(t_n)$ Let S' be the set of maximal subterms of type nat in S, and S' be partitioned as line (3) of algorithm* complete, *b defined as in line (4), and a defined as in line (5). Furthermore let m be the cardinality of S'. Then algorithm $complete(S, A)$ terminates after*

$$O(m\ n\ a \log(n\ a) + m\ n\ b\ \log(n\ b)) \text{ steps}$$

Proof: Define $T(n, m)$ be the number of steps algorithm $complete(S, A)$ needs to terminate, where n and m are the above numbers. We first discuss the complexities of each line. Line (1) costs clearly only $O(1)$ steps. The partition of a set with m elements in line (3) costs $O(n \log n)$ steps by sorting according to a lexicographic ordering on the coefficient tuples. Line (4) costs a constant amount of time. If κ is the number of linear terms, then line (5) costs $O(\kappa)$ time steps, and line (6) costs $O(\kappa \log \max\{a_1, \ldots, a_\kappa\})$ time steps. Both The computation in line (7) costs $O(\kappa\ b)$. For the test in line (8) use 2-3-trees. First insert the elements $0, \ldots, a-1$, and then compute all $b_i + j\ a_i \bmod a$ and delete them from the tree. If an empty set remains, then the test was successfull. Altogether a inserts and at most $\kappa\ a$ deletes are performed according to [AHU74, page 163], this

needs $O(\kappa\ a\ \log(\kappa\ a))$ steps. Similarly, if line (9) is evaluated together with line (7), this needs $O(\kappa\ b\ \log(\kappa\ b))$ steps. Line (10) needs a constant amount of time. Thus, line (3)–(10) need

$$O(n\ a\ \log(n\ a) + n\ b\ \log(n\ b)) \text{ time steps}$$

We have therefore:

$$T(n,0) \leq c_0\ (n\ a\ \log(n\ a) + n\ b\ \log(n\ b)$$

for some suitable constant c_0.

Line (11) costs a constant amount of time. For arranging the order in line (12), count first the length of each vector, and then sort the elements of S according to the lexicographic ordering on $(lg(t_i), x)$ where x is the most inner list, and $\langle\rangle \leq x$. Thus line (12) can be done in $O(n\ logn)$ steps. The combination in line (13) requires at most $O(n)$ steps, the cardinality of the recursive sets is less than n, and the sum of their sizes is less than $m-1$. We have therefore for line (11) - (13) (and similar arguments hold for lines (14)–(15):

$$\begin{aligned} T(n,0) &\leq c_0\ (n\ a\ \log(n\ a) + n\ b\ \log(n\ b) \\ T(n,m) &\leq c_1\ n\ \log n + \sum_{i=1}^{l} T(k_i, j_i) \end{aligned}$$

for some suitable constants c_0, c_1, and for all $1 \leq i \leq l$ is $k_i < l$, and $j_1 + \cdots + j_l \leq m - l$[1]. Then it is easy to proof by induction m:

$$T(n,m) \leq c_1\ m\ n\ \log n + c_0\ (m+1)\ (n\ a\ \log(n\ a) + n\ b\ \log(n\ b)$$

■

Now, the algorithm *complete* is modified to an algorithm *complete'*, such that it outputs a set of terms completing the input set, i.e. $complete'(S, A) = S'$ such that $val(S \cup S') = A$. In the case of natural numbers, instead of the test of line (8) we compute the difference of these sets. From the remaining elements, the suitable terms are constructed. The same is done in line (9), and the resulting set is the union of these two sets. It is important to see, that we can use the same data-structure (2-3-Trees) to construct these two sets. Thus, the asymptotic complexity doesn't change in this case. In the case of lists and cartesian products, the recursive calls output completing sets for subterms. These sets are used to compute the output set S'.

Algorithm *complete'*

INPUT:	A set S of terms, and a type A, s.t. the type of $t \in S$ is A. All maximal subterms of type *nat* in S are at most linear.
OUTPUT:	A set S' such that $val(S \cup S') = A$

(1) **if** $S = \{a\}$ **then output**$(\emptyset)$; **end if**;
(2) **case** A **of**
(3) *nat*: **assume** $S = \{a_i\ n + b_i\ |1 \leq i \leq k\} \cup \{c_0, \ldots, c_s\}$;

[1] l is the number of recursive calls of complete and corresponds to m in algorithm *complete*

(4) **if** $k = 0$ **then**
(5) $m := \max S$; **output**$(\{0, \ldots, m\} - S) \cup \{n + m + 1\})$;
(6) **else** $b := \max\{b_1, \ldots, b_k\}$;
(7) $a := \mathrm{lcm}(a_1, \ldots, a_k)$;
(8) $H := \{0, \ldots, a-1\} - \bigcup_{i=1}^{k} \{(b_i + j\ a_i) \bmod a \mid 0 \leq j < a/a_i\}$;
(9) $L := \{0, \ldots, b-1\} - \{s | s \in val(S), s \leq b\}$;
(10) **output**$(L \cup \{a \cdot n + b + c | c \in H\})$;
(11) **end if**;
(12) $\langle B \rangle$: **if** $\langle\rangle \notin S$ **then** $S' := \{\langle\rangle\}$;
(13) **else if** $S = \{\langle\rangle\}$ **then output**$\{cons(a, l)\})$;
(14) **else** $S' := \emptyset$; **end if**;
(15) $S := S \cup S'$;
(16) **assume** $S = \{\langle\rangle, cons(a_{1,1}, l_1), \ldots, cons(a_{k_1,1}, l_1), \ldots, cons(a_{1,r}, l_r), \ldots, cons(a_{k_r,r}, l_r)\}$;
(17) **for** $i = 1, \ldots, r$ **do**
(18) $S' := S' \cup \{cons(a, l_i) | a \in complete'(\{a_{1,i}, \ldots, a_{k_i,i}\}, B)\}$;
(19) **end for**;
(20) $S' := \{cons(a, l) | l \in complete'(\{l_1, \ldots, l_r\}, \langle B \rangle)\}$;
(21) **output**(S');
(22) $B \times C$: $S' := \emptyset$;
(23) **assume** $S = \{(a_1, b_{1,1}), \ldots, (a_1, b_{k_1,1}), \ldots, (a_r, b_{1,r}), \ldots, (a_r, b_{k_r,r})\}$;
(24) **for** $i = 1, \ldots, r$ **do**
(25) $S' := S' \cup \{(a_i, b) | b \in complete'(\{b_{1,i}, \ldots, b_{k_i,i}\}, B)\}$;
(26) **end for**;
(27) $S' := S' \cup \{(a, b) | b \in complete'(\{a_1, \ldots, a_r\}, A)\}$;
(28) **output** S';
(29) **end case**;

■

The proof of the correctness or algorithm *complete'* can be obtained by simply modifying the proof of algorithm *complete*, and is therefore omitted here.

Corollary 3.17 (Time Complexity of Algorithm *complete'***)** . *Let* $S = \{t_1, \ldots, t_n\}$ *be a set of terms and* $m = sz(t_1) + \cdots + sz(t_n)$ *Let* S' *be the set of maximal subterms of type nat in* S*, and* S' *be partitioned as line (3) of algorithm* complete*, b defined as in line (4), and a defined as in line (5). Furthermore let m be the cardinality of* S'*. Then algorithm complete*(S, A) *terminates after*

$$O(m\ n\ a \log(n\ a) + m\ n\ b\ \log(n\ b)) \text{ steps}$$

Proof: It is obvious that in the case $A = nat$ the complexity is the same as in algorithm *complete*. If we can show that lines (16) - (20) cost $O(n \log n)$ time then it follows immediately that the asymptotic complexities of algorithm *complete* and algorithm *complete'* are the same. By the choice of 2-3-trees, we know that the recursive calls return 2-3-trees. Lines (16) - (20) perform altogether $n + 1$ union-operations. These operations cost according to [AHU74, page 263] $O(n\ \log n)$ steps.
■

The next algorithm checks for two given sets S_1 and S_2, whether $val(S_1) \cap val(S_2) = \emptyset$. This property is called the *consistency* of S_1 and S_2. All subterms of type *nat* are assumed to be linear in one

variable or being constant. Otherwise zeros of multivariate polynomials have to be determined. One example of this problem is Fermats problem, which is not yet solved in general. Hence, the restriction to linear and constant terms. When we use the algorithm in algorithm *symbolic_eval* it is ensured, that all subterms of type *nat* are either constant or linear. Again, lemmas 3.13 and 3.14 are useful to deal with natural numbers.

Algorithm *consistent*

INPUT: Two sets $S_1 = \{t_1, \ldots, t_n\}$, and $S_2 = \{u_1, \ldots, u_m\}$ such that each subterm of type *nat* is either constant or linear in its variables, and each t_i and u_j are of the same type A.

OUTPUT: If $val(S_1) \cap val(S_2) = \emptyset$ then *true* else *false*.

(1) **if** $S_1 = \{a\} \vee S_2 = \{a\}$ and a is a variable **then output**(*false*) **end if**;
(2) **case** A **of**
(3) *nat*: **assume** $S_1 = \{a_i \cdot n + b_i | 1 \leq i \leq k\} \cup \{c_1, \ldots, c_{n-k}\}$;
(4) **assume** $S_2 = \{\alpha_i \cdot n + \beta_i | 1 \leq i \leq l\} \cup \{\gamma_1, \ldots, \gamma_{m-l}\}$;
(5) $b := \max\{b_1, \ldots, b_k, c_1, \ldots, c_{n-k}, \beta_1, \ldots, \beta_l, \gamma_1, \ldots, \gamma_{m-l}\}$;
(6) $a := \mathrm{lcm}(a_1, \ldots, a_k, \alpha_1, \ldots, \alpha_l)$;
(7) $R_1 := \bigcup_{i=1}^{k} \{(b_i + j\ a_i) \bmod a\ | 0 \leq j < a/a_i\}$;
(8) $R_2 := \bigcup_{i=1}^{k} \{(\beta_i + j\ \alpha_i) \bmod a\ | 0 \leq j < a/\alpha_i\}$;
(9) **if** $R_1 \cap R_2 \neq \emptyset$ **then output**(*false*); **end if**;
(10) $A_1 := \{n | n \in val(S_1), n \leq b\}$;
(11) $A_2 := \{n | n \in val(S_2), n \leq b\}$;
(12) **if** $A_1 \cap A_2 = \emptyset$ **then output**(*true*) **else output**(*false*) **end if**;
(13) $\langle B \rangle$: **if** $\langle\rangle \in S_1 \wedge \langle\rangle \in S_2$ **then output**(*false*) **end if**;
(14) $S_1 := S_1 - \{\langle\rangle\}$;
(15) $S_2 := S_2 - \{\langle\rangle\}$;
(16) **for all** $t \in S_1$ **do** [all terms are supposed to have different variables]
(17) **for all** $u \in S_2$ **do**
(18) **if** t and u do not contain a non-constant subterm of type *nat* **then**
(19) **if** t and u can be unified **then output**(*false*);
(20) **else output**(*true*); **end if**
(21) **else** replace each non-constant maximal subterm of type *nat* by
(22) a new variable x, and store the replaced terms in a substitution ρ;
(23) **if** t and u are unifiable **then**
(24) $\sigma := unify(t, u)$;
(25) **for each** $[x/y] \in \sigma$ **do**
(26) **if** x and y are variables, and
(27) $consistent(\rho(x), \rho(y))$ **then output**(*true*) **end if**;
(28) **end for**;
(29) **output**(*false*);
(30) **end if**;
(31) **end if**;
(32) **end for**;
(33) **end for**;
(34) **output**(*true*);
(35) $B \times C$: **for all** $t \in S_1$ **do** [all terms are supposed to have different variables]

```
(36)          for all u ∈ S2 do
(37)             if t and u do not contain a non-constant subterm of type nat then
(38)                if t and u can be unified then output(false);
(39)                else output(true); end if
(40)             else replace each non-constant maximal subterm of type nat by
(41)                a new variable x, and store the replaced terms in a substitution ρ;
(42)                if t and u are unifiable then
(43)                   σ := unify(t, u);
(44)                   for each [x/y] ∈ σ do
(45)                      if y is not a variable, or
(46)                         not consistent(ρ(x), ρ(y))
(47)                      then output(false) end if;
(48)                   end for;
(49)                end if;
(50)             end if;
(51)          end for;
(52)       end for;
(53)       output(true);
(54) end case;
```

■

Lemma 3.18 (Correctness of Algorithm *consistent*) *Algorithm consistent terminates and outputs true, iff* $val(S_1) \cap val(S_2) = \emptyset$.

Proof: The termination is obvious. Consider now the case where S_1 and S_2 contain term of type *nat*:

Suppose $val(S_1) \cap val(S_2) \neq \emptyset$. Because both S_1 and S_2 contain only linear or constant terms, it must hold for an n_0, a, k, and l:

$$\{s | s \in val(S_1),\ s \leq n_0\} \cap \{s | s \in val(S_2),\ s \leq n_0\}$$

or $S_1 = \bigcup_{i=1}^{k} \{n | n \equiv b_i \bmod a, n > n_0\}$, $S_2 = \bigcup_{i=1}^{l} \{n | n \equiv c_i \bmod a, n > n_0\}$ where $\{b_1, \ldots, b_k\} \cap \{c_1, \ldots, c_l\}$. In the first case, algorithm *consistent* outputs *false* because of line (12), in the second case it also outputs *false* by line (9).

Suppose now, that the algorithm outputs *true*. Then we have $R_1 \cap R_2 = \emptyset$ and $A_1 \cap A_2 = \emptyset$. Then we have by (5)-(8) and (10)-(11)

$$S_1 = A_1 \cup \bigcup_{i=1}^{k} \{n | n \equiv b_i + j\ a_i \bmod a,\ 1 \leq j < a/a_i,\ n \geq b\}$$

$$S_2 = A_2 \cup \bigcup_{i=1}^{l} \{n | n \equiv \beta_i + j\ \alpha_i \bmod a,\ 1 \leq j < \alpha/a_i,\ n \geq b\}$$

Because $R_1 \cap R_2 = \emptyset$, $A_1 \cap A_2 = \emptyset$, and by definition $A_1 \cap R_2 = \emptyset$ and $A_2 \cup R_1 = \emptyset$, it is $S_1 \cap S_2 = \emptyset$.

Consider now the case where the terms in S_1 and S_2 are of type $\langle B \rangle$ for some B. Suppose the algorithm outputs *false*. Then one of the following two cases occur:

CASE 1: There are two terms $t \in S_1$, and $u \in S_2$, none of them containing any non-constant subterm of type *nat*, and t and u can be unified. Let σ be the most general unifier of t and u. Then it is

$$\emptyset \neq val(\sigma\ t) \cap val(\sigma u) \subseteq val(t) \cap val(u) \subseteq val(S_1) \cap val(S_2)$$

and therefore the intersection of the values of S_1 and S_2 cannot be empty.

CASE 2: Two terms $t \in S_1$ and $u \in S_2$ contain non-constant subterms of type *nat*. Then by lines (21)–(22) each such subterm of type *nat* is replaced by a new variable, and the substitution ρ can recover its values. Let t' and u' be these modified terms. The algorithm outputs *false* if t' and u' can be unified, and for each pair $[x/y]$ in the most general unifier σ of t' and u' holds: if x and y are variables, then $consistent(\rho(x), \rho(y)) = \mathit{false}$ and hence $val(\rho(x)) \cap val(\rho(y)) \neq \emptyset$. Then:

$$\emptyset \neq val(\rho(\sigma(t'))) \cap val(\rho(\sigma(u'))) \subseteq val(t) \cap val(u) \subseteq val(S_1) \cap val(S_2)$$

Suppose now $val(S_1) \cap val(S_2) \neq \emptyset$. Then there are two terms $t \in S_1$, $u \in S_2$ such that $val(S_1) \cap val(S_2) \neq \emptyset$. By definition of *val*, there must be an environment ρ such that $\rho\ t = \rho\ u$. Hence, ρ is a unifier of t and u. If neither t nor u contain any non-constant subterm of type *nat*, then the algorithm outputs *false* by line (19). Otherwise the by lines (21)–(22) modified terms t' and u' can also be unified. Let σ be their most general unifier. Suppose now the algorithm would output *true*. Then there would be subterms on the same occurence, such that their values represent two disjoint sets. But then $val(t) \cap val(u) = \emptyset$ in contradiction to the assumption. Hence the algorithm outputs also in this case *false*.

The cartesian product is proven analogously to vectors. ■

Lemma 3.19 (Complexity of Algorithm *consistent*)
Let be $S_1 = \{t_1, \ldots, t_n\}$, $S_2 = \{u_1, \ldots, u_m\}$.

(a) *If all* t_i *and* u_j *are of type nat then let* b *the largest number occuring in any of the* t_i *and* u_j *and let* a *be the least common multiple of all coefficients of variables among the* t_i *and* u_j. *The time complexity of algorithm consistent with input* S_1 *and* S_2 *is then*

$$O((n+m)\ (a \log((n+m)\ a) + b\ \log((n+m)\ b)))$$

b *If the* t_i *and* u_j *are vectors, let* b *be the largest natural number occuring in* S_1 *and* S_2, $r = sz(t_1) + \cdots + sz(t_n)$, *and* $s = sz(u_1) + \cdots + sz(u_m)$. *Then algorithm consistent with input* S_1 *and* S_2 *terminates after*

$$O(b^2 \log b\ (m\ r + n\ s)) \text{ steps.}$$

Proof:

(a) Lines (3)–(4) need as in the proof of lemma 3.16 $O((n+m)\ \log(n+m)$ steps. Line (5) can be performed in $O(n+m)$ steps. Using Euclids algorithm, line (6) can be performed in time $O((k+l)\log b)$. Using 2-3-trees lines (7) – (9) can performed in $O((k+l)\ a\ \log((k+l)\ a))$ steps and lines (10) – (12) can be performed in $O((k+l)\ b\ \log((k+l)\ b)$ steps. Summing up these complexities yields the stated result.

(b) We consider here the case of vectors. The complexities in the case of cartesian products is obtained similarly.

Using $2-3-trees$, the test in line (13) requires $O(\log n + \log m)$ steps. Similarly, lines (14)–(15) require $O(\log n + \log m)$ steps. Define now $r_t = sz(t)$ and $s_u = sz(u)$. Consider now the body of the loops in lines (18) – (31) for two particular terms t and u. The test in line (18) can be easily performed in time $O(r_t + s_u)$. According to [MM82], line (19) can be performed in $O(r_t + s_u)$ steps. Line (21) needs also $O(r_t + s_u)$ time. Also lines (23)–(24) need that time. In line (26), it can be seen by (a), that at most $O(b_{t,u}^2 \log b_{t,u})$ steps are needed, where $b_{t,u}$ is the largest natural number occuring in b and u. Because the length of the substititution σ is certainly bounded by $r_t + s_u$, the loop (25)–(27) requires at most $O((r_t + s_u)\ b_{t,u}^2 \log b_{t,u})$ steps, and is therefore dominating in lines (18)–(31). Because

$$\sum_{t \in S_1} \sum_{u \in S_2} (r_t + s_u)\ b_{t,u}^2 \log b_{t,u} \leq b^2 \log b \sum_{t \in S_1} \sum_{u \in S_2} (r_t + s_u) = b^2 \log b\ (m\ r + n\ s)$$

the time coplexity of algorithm *consistent* is in the case of vectors $O(b^2 \log b\ (m\ r + n\ s))$.

■

The next algorithm minimizes a given set S, i.e. it removes terms as long as the value of the set is not changed. Its application in algorithm *symbolic_eval* will lead to a smaller set of equations. Its main idea is based on the following observation. If a set S contains to terms s and t such that s matches t then s can be removed from S without changing the value of S.

Algorithm *minimize*

INPUT: A set S of terms.
OUTPUT: A minimal subset $S' \subseteq S$ such that $val(S) = val(S')$

(1) $V := S$; $E := \emptyset$; $S' := S$;
(2) **for all** $t \in V$ **do**
(3) **for all** $s \in V - \{t\}$ **do**
(4) **if** t matches $s \wedge \neg s$ matches t **then** $E := E \cup \{(s,t)\}$; **end if**;
(5) **end for**;
(6) **end for**;
(7) **for all** $(s,t) \in E$ **do**
(8) $S' := S' - \{s\}$;
(9) **end for**;
(10) **output** S';

■

Lemma 3.20 (Correctness and Complexity of Algorithm *minimize*) *Algorithm minimize outputs after $O(n^2\ m)$ steps for each input set S with $card(S) = n$ and $m = \max_{t \in S} sz(t)$ a minimal set S' such that $val(S') = val(S)$.*

Proof: For the correctness define a relation

$$s \prec t :\Leftrightarrow t \text{ matches } s$$

It is well-known that this relation forms a partial order. Hence the loop (2)–(6) yield a graph where two vertices are connected by an edge if they are ordered by $\prec$. Observe that the test in line (4) and the exclusion of t in line (3) excludes reflexive edges from $G = (V, E)$. Thus G is a directed acyclic graph. Because G is acyclic there are terms t which do not precede any other term in G. These terms are selected in the loop in lines (7)–(9). Thus for each $s \in S - S'$ there is $t \in S'$ such that s matches t. Hence the s in line (8) can be removed without changing the value of S'.

Consider now the complexity. The test in line (4) costs $O(\max(sz(s), sz(t)))$ steps, which is $O(m)$. Furthermore line (4) is executed $n^2 - n$ times, and thus lines (1)–(6) needs $O(n^2\ m)$ steps. Lines (7)–(9) need $O(k)$ steps where k is the number of elements in E. It is $k \leq n^2$, and hence lines (7)–(10) need time $O(n^2)$. Altogether the algorithm needs time $O(n^2\ m)$. ■

The remaining algorithm used by *symbolic_eval* is the algorithm ***instantiate_condition***. The input of this algorithm is a condition C in disjunctive normal form and a tuple $(x_1, \ldots, x_n)$ of variables which is of type $T_1 \times \cdots \times T_n$. It outputs a set S of symbolic terms with the following property:

$$\begin{array}{l} \forall (s_1, \ldots, s_n) \in val(S) : EVAL[\![C]\!]\ \emptyset\ 1\ ()\ [x_1/s_1] \cdots [x_n/s_n] = true \\ \forall (s_1, \ldots, s_n) \in T_1 \times \cdots \times T_n - val(S) : EVAL[\![C]\!]\ \emptyset\ 1\ ()\ [x_1/s_1] \cdots [x_n/s_n] = false \end{array} \tag{3.1}$$

Algorithm *instantiate_condition*

INPUT: A condition C in disjunctive normalform, containing only basic operations, and a tuple $(x_1, \ldots, x_n)$ of its variables.
OUTPUT: A set S with property (3.1)

(1) **assume** $C = C_1 \vee \cdots \vee C_k$;
(2) **if** $k > 1$ **then output**$(\bigcup_{i=1}^{k}$ *instantiate_condition*$(C_i, (x_1, \ldots, x_n)))$; **end if**;
(3) **assume** $C = L_1 \wedge \cdots \wedge L_m$;
(4) **if** $m > 1$ **then**
(5) $\quad S_1 :=$ *instantiate_condition*$(L_1, (x_1, \ldots, x_n))$;
(6) $\quad S_2 :=$ *instantiate_condition*$(L_2 \wedge \cdots \wedge L_m, (x_1, \ldots, x_n))$;
(7) $\quad$ **output**(*combine*(S_1, S_2)); [It is $val(combine(S_1, S_2)) = val(S_1) \cap val(S_2)$]
(8) **end if**;
(9) **case** C **of**
(10) $\quad a_1 \cdot x_j + b_1 < a_2 \cdot x_j + b_2$:
(11) $\quad\quad$ **if** $a_1 < a_2$ **then**
(12) $\quad\quad\quad$ **if** $b_1 \leq b_2$ **then output**$(\{(x_1, \ldots, x_n)\})$ [condition always satisfied]
(13) $\quad\quad\quad$ **else output**$\{(x_1, \ldots, x_{j-1}, x_j + \lfloor (b_1 - b_2)/(a_2 - a_1) \rfloor + 1, x_{j+1}, \ldots, x_n)\})$; **end if**;
(14) $\quad\quad$ **else if** $a_1 = a_2$ **then**
(15) $\quad\quad\quad$ **if** $b_1 < b_2$ **then output**$(\{(x_1, \ldots, x_n)\})$ **else output**$(\emptyset)$; **end if**;
(16) $\quad\quad$ **else if** $b_1 < b_2$ **then output**$(\{(x_1, \ldots, x_{j-1}, y, x_{j+1}, x_n) | 0 \leq y < \lceil (b_2 - b_1)/(a_1 - a_2) \rceil\})$;
(17) $\quad\quad$ **else output**$(\emptyset)$; **end if**; **end if**;
(18) $\quad$ **not** $a_1 \cdot x_j + b_1 < a_2 \cdot x_j + b_2$:
(19) $\quad\quad$ **if** $a_1 < a_2$ **then**
(20) $\quad\quad\quad$ **if** $b_1 \leq b_2$ **then output**$(\emptyset)$;
(21) $\quad\quad\quad$ **else output**$\{(x_1, \ldots, x_{j-1}, y, x_{j+1}, \ldots, x_n) | 0 \leq y \leq \lfloor (b_1 - b_2)/(a_2 - a_1) \rfloor\})$; **end if**;
(22) $\quad\quad$ **else if** $a_1 = a_2$ **then**
(23) $\quad\quad\quad$ **if** $b_1 < b_2$ **then output**$(\emptyset)$; **else output**$(\{(x_1, \ldots, x_n)\})$; **end if**;
(24) $\quad\quad$ **else if** $b_1 < b_2$ **then output**$(\{(x_1, \ldots, x_{j-1}, x_j + \lceil (b_2 - b_1)/(a_1 - a_2) \rceil, x_{j+1}, x_n)\})$;

(25) **else output**$(\{(x_1,\ldots,x_n)\})$; **end if**; **end if**;
(26) $a_k\ x^k{}_j + \cdots + a_1 \cdot x_j + a_0 = b_k\ x^k{}_j + \cdots + b_1 \cdot x_j + b_0$:
(27) **if** $(a_k,\ldots,a_0) = (b_k,\ldots,b_0)$ **then output** $\{(x_1,\ldots,x_n)\}$; **end if**;
(28) $C := \{x | x \text{ divides } (a_0 - b_0) \wedge x \geq 0\}$;
(29) $S := \{x | x \in C \wedge a_k\ x^k + \cdots + a_0 = b_k\ x^x + \cdots + b_0\}$;
(30) **output** $\{(x_1,\ldots,x_{j-1},x,x_{j+1},\ldots,x_n) | x \in S\}$;
(31) **not** $a_k\ x^k{}_j + \cdots + a_1 \cdot x_j + a_0 = b_k\ x^k{}_j + \cdots + b_1 \cdot x_j + b_0$:
(32) **if** $(a_k,\ldots,a_0) = (b_k,\ldots,b_0)$ **then output** $\emptyset$; **end if**;
(33) $C := \{x | x \text{ divides } (a_0 - b_0) \wedge x \geq 0\}$;
(34) $S := \{x | x \in C \wedge a_k\ x^k + \cdots + a_0 = b_k\ x^x + \cdots + b_0\}$;
(35) **if** $S = \emptyset$ **then output** $\{(x_1,\ldots,x_n)\}$; **end if**;
(36) **output** $\{(x_1,\ldots,x_{j-1},y,x_{j+1},\ldots,x_n) | y \notin S \wedge y < \max S \text{ or } y = x_j + \max S + 1\}$;
(37) $mt(t)$: *instantiate_condition*$(t = \langle\rangle, (x_1,\ldots,x_n)$
(38) **not** $mt(t)$: *instantiate_condition*(**not** $t = \langle\rangle, (x_1,\ldots,x_n)$
(39) $hd(t_1) = t_2$: **let** l be a new variable;
(40) $(S_0,S_1,\ldots,S_n) :=$ *instantiate_condition*$(t_1 = cons(t_2,l),(l,x_1,\ldots,x_n))$;
(41) **output** $\{(u_1,\ldots,u_n) | \exists t_i \in S_i, t_0 \in S_0 : u_i = [l/t_0]t_i\}$;
(42) **not** $hd(t_1) = t_2$: **let** l be a new variable;
(43) $(S_0,S_1,\ldots,S_n) :=$ *instantiate_condition*(**not** $t_1 = cons(t_2,l),(l,x_1,\ldots,x_n)$);
(44) **output** $\{(u_1,\ldots,u_n) | \exists t_i \in S_i, t_0 \in S_0 : u_i = [l/t_0]t_i\}$;
(45) $t_1[i] = t_2$: **let** $a_0,\ldots,a_{i-1},l$ be new variables;
(46) $(S_1,\ldots,S_n,U_0,\ldots,U_{i-1},S_{n+1}) :=$
(47) *instantiate_condition*$(t_1 = cons(a_0,\ldots,cons(a_{i-1},cons(t_n,l))\cdots),(x_1,\ldots,x_n,a_o,\ldots,a_{i-1},l))$;
(48) **output** $([a_0/u_0]\cdots[a_{i-1}/u_{i-1}][l/t_{n+1}]t_k | t_k \in S_k, a_j \in U_j, t_{n+1} \in S_{n+1} | k = 1,\ldots,n)$;
(49) **not** $t_1[i] = t_2$: **let** $a_0,\ldots,a_{i-1},l$ be new variables;
(50) $(S_1,\ldots,S_n,U_0,\ldots,U_{i-1},S_{n+1}) :=$
(51) *instantiate_condition*(**not** $t_1 = cons(a_0,\ldots,cons(a_{i-1},cons(t_n,l))\cdots),(x_1,\ldots,x_n,a_o,\ldots,a_{i-1},l)$);
(52) **output** $([a_0/u_0]\cdots[a_{i-1}/u_{i-1}][l/t_{n+1}]t_k | t_k \in S_k, a_j \in U_j, t_{n+1} \in S_{n+1} | k = 1,\ldots,n)$;
(53) $tl(t_1) = t_2$: **let** a be a new variable;
(54) $(S_0,S_1,\ldots,S_n) :=$ *instantiate_condition*$(t_1 = cons(a,t_2),(a,x_1,\ldots,x_n))$;
(55) **output** $\{(u_1,\ldots,u_n) | \exists t_i \in S_i, t_0 \in S_0 : u_i = [a/t_0]t_i\}$;
(56) **not** $tl(t_1) = t_2$: **let** a be a new variable;
(57) $(S_0,S_1,\ldots,S_n) :=$ *instantiate_condition*(**not** $t_1 = cons(a,t_2),(a,x_1,\ldots,x_n)$);
(58) **output** $\{(u_1,\ldots,u_n) | \exists t_i \in S_i, t_0 \in S_0 : u_i = [a/t_0]t_i\}$;
(59) $cons(t_1,t_2) = \langle\rangle$: **output** $(\emptyset,\ldots,\emptyset)$;
(60) **not** $cons(t_1,t_2) = \langle\rangle$: **output** $(x_1,\ldots,x_n)$;
(61) $cons(a_1,l_1) = cons(a_2,l_2)$: *instantiate_condition*$(a_1 = a_2 \wedge l_1 = l_2,(x_1,\ldots,x_n))$;
(62) **not** $cons(a_1,l_1) = cons(a_2,l_2)$: *instantiate_condition*$(\neg a_1 = a_2 \vee \neg l_1 = l_2,(x_1,\ldots,x_n))$;
(63) $t_1.i = t_2$: **assume** t_1 is a k-tuple **where** $i \leq k$;
(64) **let** $a_0,\ldots,a_{i-1},a_{i+1},\ldots,a_k$ be new variables;
(65) $(S_1,\ldots,S_n,A_1,\ldots,A_{i-1},A_{i+1},\ldots,A_k) :=$
instantiate_condition$(t_1 = (a_1,\ldots,a_{i-1},t_2,a_{i+1},\ldots,a_k),(x_1,\ldots,x_n,a_0,\ldots,a_{i-1},a_{i+1},\ldots,a_k))$;
(66) **output** $(\{[a_0/u_0]\cdots[a_{i-1}/u_{i-1}][a_{i+1}/u_{i+1}]\cdots[a_k/u_k]t_m | t_m \in S_m, u_j \in A_j\} | m = 1,\ldots,n)$;
(67) **not** $t_1.i = t_2$: **assume** t_1 is a k-tuple **where** $i \leq k$;
(68) **let** $a_0,\ldots,a_{i-1},a_{i+1},\ldots,a_k$ be new variables;
(69) $(S_1,\ldots,S_n,A_1,\ldots,A_{i-1},A_{i+1},\ldots,A_k) :=$
instantiate_condition$(\neg t_1 = (a_1,\ldots,a_{i-1},t_2,a_{i+1},\ldots,a_k),(x_1,\ldots,x_n,a_0,\ldots,a_{i-1},a_{i+1},\ldots,a_k))$;

(70) **output** $(\{[a_0/u_0]\cdots[a_{i-1}/u_{i-1}][a_{i+1}/u_{i+1}]\cdots[a_k/u_k]t_m | t_m \in S_m, u_j \in A_j\} | m = 1,\ldots,n)$;

(71) $(t_1,\ldots,t_k) = (s_1,\ldots,s_k)$: *instantiate_condition* $\left(\bigwedge_{i=1}^{k} t_i = s_i, (x_1,\ldots,x_n)\right)$;

(72) **not** $(t_1,\ldots,t_k) = (s_1,\ldots,s_k)$: *instantiate_condition* $\left(\bigvee i = 1^k \neg t_i = s_i, (x_1,\ldots,x_n)\right)$;

(73) **end case**;

■

We complete this section with the algorithm *combine*. This algorithm computes the intersection of two sets and is derived from algorithm *consistent* which checks whether the intersection of two sets is empty or not.

Algorithm *combine*

INPUT: Two sets $S_1 = \{t_1,\ldots,t_n\}$, and $S_2 = \{u_1,\ldots,u_m\}$ such that each subterm of type *nat* is either constant or linear in its variables, and each t_i and u_j are of the same type A.

OUTPUT: A set S such that $val(S) = val(S_1) \cap val(S_2)$

(1) **if** $S_1 = \{a\}$ a is a variable **then output** S_2; **end if**;
(2) **if** $S_2 = \{a\}$ a is a variable **then output** S_1; **end if**;
(3) **case** A **of**
(4) *nat*: **assume** $S_1 = \{a_i \cdot n + b_i | 1 \leq i \leq k\} \cup \{c_1,\ldots,c_{n-k}\}$;
(5) **assume** $S_2 = \{\alpha_i \cdot n + \beta_i | 1 \leq i \leq l\} \cup \{\gamma_1,\ldots,\gamma_{m-l}\}$;
(6) $b := \max\{b_1,\ldots,b_k,c_1,\ldots,c_{n-k},\beta_1,\ldots,\beta_l,\gamma_1,\ldots,\gamma_{m-l}\}$;
(7) $a := \mathrm{lcm}(a_1,\ldots,a_k,\alpha_1,\ldots,\alpha_l)$;
(8) $R_1 := \bigcup_{i=1}^{k}\{(b_i + j\ a_i) \bmod a\ |0 \leq j < a/a_i\}$;
(9) $R_2 := \bigcup_{i=1}^{k}\{(\beta_i + j\ \alpha_i) \bmod a\ |0 \leq j < a/\alpha_i\}$;
(10) $R := R_1 \cap R_2$;
(11) $A_1 := \{n | n \in val(S_1), n \leq b\}$;
(12) $A_2 := \{n | n \in val(S_2), n \leq b\}$;
(13) $A := A_1 \cap A_2$;
(14) **output** $A \cup \{a\ n + b | a \in R\}$;
(15) $\langle B\rangle$: **if** $\langle\rangle \in S_1 \wedge \langle\rangle \in S_2$ **then** $S := \{\langle\rangle\}$ **else** $S := \emptyset$; **end if**;
(16) $S_1 := S_1 - \{\langle\rangle\}$;
(17) $S_2 := S_2 - \{\langle\rangle\}$;
(18) **for all** $t \in S_1$ **do**
(19) **for all** $u \in S_2$ **do**
(20) **if** t and u do not contain a non-constant subterm of type *nat* **then**
(21) **if** t and u can be unified **then**
(22) $\sigma := unify(t,u)$;
(23) $S := S \cup \{\sigma t\}$;; **end if**;
(24) **else** replace each non-constant maximal subterm of type *nat* by
(25) a new variable x, and store the replaced terms in a substitution ρ;
(26) **if** t and u are unifiable **then**
(27) $\sigma := unify(t,u)$; $S' := \sigma t$;
(28) **for each** $[x/y] \in \sigma$ **do**

```
(29)                        if x and y are variables then
(30)                        S' := {[y/u]t|t ∈ S', u ∈ combine({ρ(x)}, {ρ(y)})};
(31)                    end for;
(32)                    S := S ∪ S';
(33)                end if;
(34)            end if;
(35)        end for;
(36)    end for;
(37)    output S;
(38) B × C: S := ∅;
(39)    for all t ∈ S1 do
(40)        for all u ∈ S2 do
(41)            if t and u do not contain a non-constant subterm of type nat then
(42)                if t and u can be unified then
(43)                    σ := unify(t, u); S := S ∪ {σt}; end if
(44)            else replace each non-constant maximal subterm of type nat by
(45)                a new variable x, and store the replaced terms in a substitution ρ;
(46)                if t and u are unifiable then
(47)                    σ := unify(t, u); S' := {σt};
(48)                    for each [x/y] ∈ σ do
(49)                        if y is not a variable then
(50)                            S' := {[y/u]t|t ∈ S', u ∈ combine({ρ(x)}, {ρ(y)})};
(51)                    end for;
(52)                    S := S ∪ S';
(53)                end if;
(54)            end if;
(55)        end for;
(56)    end for;
(57)    output S;
(58) end case;
```

■

The correctness proof is similar to algorithm *consistent*. The proof for naturals is nearly the same, and the proof in the case of lists uses the fact that if σ is the most general unifier of two terms s and t then $val(\sigma t) = val(s) \cap val(t)$. Hence the proof in this case is also almost the same as in algorithm *consistent*. We omit therefore the proof of correctness for algorithm *combine*.

For the time complexity of algorithm *combine*, observe that the only change to algorithm *consistent* is maintaining the set S and S' in the case of cartesian products, and computing the output in the case of natural numbers. In the case of naturals, compared to algorithm *consistent* there is an additional union operation, and a compuatation of second set. This second set costs $O(a)$ computations. Thus the complexity of algorithm *combine* is the same as the complexity of algorithm *consistent* in the case of natural numbers.

In the case of vectors, we have $n \cdot m$ unions in line (23) and (32), where $n = card(S_1)$ and $m = card(S_2)$. Using 2-3-trees, this operations cost according to [AHU74, page 163] $O(n\ m(\log n + \log m)$ steps. All the other operations cost altogether the same as algorithm *consistent*. Observe that for large sets S_1 and S_2 we have $n\ m\ (\log n + \log m) \leq (m\ r + n\ s)$ where $r = sz(S_1)$ and $s = sz(S_2)$.

Therefore

Lemma 3.21 (Correctness and Complexity of Algorithm *combine*) *Let be* $S_1 = \{t_1, \ldots, t_n\}$ *and* $S_2 = \{u_1, \ldots, u_m\}$.

(a) *If all* t_i *and* u_j *are of type nat, then let* b *the largest number occuring in* S_1 *and* S_2, *and* a *be the least common multiple of the coefficients of variables among the* t_i *and* u_j. *Then, after* $O((n+m)\ (a \log((n+m)\ a) + b\ \log((n+m)\ b)))$ *steps, algorithm combine*(S_1, S_2) *outputs a set* S, *such that* $val(S) = val(S_1) \cap val(S_2)$.

(b) *If all* t_i *and* u_j *are vectors (or cartesian products, respectiveley), let* b *be the largest natural number occuring in* S_1 *and* S_2, $r = sz(t_1) + \cdots + sz(t_n)$, *and* $s = sz(u_1) + \cdots + sz(u_m)$. *Then, after at most* $O(b^2 \log b\ m\ r + n\ s)$ *steps, algorithm combine*(S_1, S_2) *outputs a set* S *such that* $val(S) = val(S_1) \cap val(S_2)$.

We are now ready to prove the partial correctness of algorithm *instantiate_condition*, and to compute its time complexity.

Lemma 3.22 (Partial Correctness of Algorithm *instantiate_condition*) *If algorithm instantiate_condition terminates on a condition* C *and a tuple of variables* $(x_1, \ldots, x_n)$, *then it outputs a set* S *satisfying (3.1).*

Proof: We prove the lemma by induction on the number of recursive calls of *instantiate_condition.*

BASE CASE: 0 recursive calls. These cases are the cases of lines (10), (18), (26), (34), (75) and (76). It is therefore sufficient just to consider the variable x_j involved in these cases. All the other variables

play no role.

(a) $a_1\ x_j + b_1 < a_2\ x_j + b_2$: If $a_1 < a_2$ then (a) is equivalent to $n \geq \lfloor (b_1 - b_2)/(a_2 - a_1) \rfloor + 1$. Hence, in this case *instantiate_condition* outputs the correct result. Similarly, if $a_1 > a_2$, then (a) is equivalent to $n \leq \lceil (b_2 - b_1)/(a_2 - a_1) \rceil - 1$. This is never satisfied if $b_2 \leq b_1$, and therefore, the output is also correct in this case. Finally, if $a_1 = a_2$, then the condition is true iff $b_1 < b_2$. Thus the output is also correct in this case. The negation of (a) is proven analogously.

(b) $(a_1\ x_j + b_1) \bmod a_2 = b_2$. If $b_2 \geq a_2$, then this condition is never true. Hence the algorithm is correct in this case. Consider now the case $\gcd(a_1, a_2) > 1$. Then there is a natural number q such that $q\ a_1 \equiv 0 \bmod a_2$, and therefore $q\ b_1 \equiv q\ b_2 \bmod a_2$. This is the case if, and only if $\gcd(b_2 - b_1, a_2) = q$. Hence, the algorithm outputs in lines (29) and (30) the correct result.

Now let c and d defined as in line (31), and $\gcd(a_1, a_2) = 1$. Then (b) is equivalent to $n \equiv c\ (b_2 - b_1) \bmod a_2$, and it outputs therefore also in line (32) the correct result. The negation of (b) is proven analogously.

(c) $cons(t_1, t_2) = \langle\rangle$: This case is never true, hence the output is correct. The negation of (c) is proven analogously.

INDUCTIVE CASE: There are more than k recursive calls of *instantiate_condition.* Here, the cases of lines (42), (49), (56), (57), (58), (61), (65), (69), (72), (75), (76), (77), (78), (79), (83), (87), and (88) have to be considered.

(a) $(a_1 \cdot x_j + b_1) \bmod c_1 = (a_2 \cdot x_j + b_2) \bmod c_2$: Let c, d_1 and d_2 defined as in lines (5), (51), and (52), respectively. Then it holds for all $0 \leq i < \max(c_1, c_2)$:

$$(a_1\ cdot x_j + b_1) \bmod c_1 = i \Leftrightarrow d_1\ a_1\ x_j + d_1\ b_1 \equiv d_1\ i \bmod c \Leftrightarrow (d_1\ a_1\ x_j + d_1\ b_1) \bmod c = (d_1\ i) \bmod c$$

$$(a_2\ cdot x_j + b_2) \bmod c_2 = i \Leftrightarrow d_2\ a_2\ x_j + d_2\ b_2 \equiv d_2\ i \bmod c \Leftrightarrow (d_2\ a_2\ x_j + d_2\ b_2) \bmod c = (d_2\ i) \bmod c$$

Hence the condition is equivalent to that in the recursive call in line (47). By induction hypothesis, the algorithm returns the correct result. The negation of (a) is proven analogously.

(b) $mt(t)$. This condition is equivalent to the condition $t = \langle\rangle$, and the correctness follows from the induction hypothesis. The negation if proven analogously.

(c) $hd(t_1) = t_2$. This condition is equivalent to the condition $t_1 = cons(t_2, l)$ for some l. Then the induction hypothesis applied on line (59) implies:

$$EVAL[\![t_1 = cons(t_2, l)]\!]\ \emptyset\ [l/t_0] \cdots [x_n/t_n] = true \Leftrightarrow (t_0, \ldots, t_n) \in val(S_0) \times \cdots \times val(S_n)$$

Thus:

$$EVAL[\![hd(t_1) = t_2]\!]\ \emptyset [x_1/u_1] \cdots [x_n/u_n] = true \Leftrightarrow (u_1, \ldots, u_n) \in val(S)$$

where S is the output defined in line (60). The negation of (c) is proven analogously.

(d) $t_1[i] = t_2$ and its negation are proven similar to case (c).

(e) $tl(t_1) = t_2$ and its negation are proven similar to case (c).

(f) $cons(a_1, l_1) = cons(a_2, l_2)$ is equivalent to $a_1 = a_2 \wedge l_1 = l_2$. Hence the induction hypothesis imply the correctness.

(g) $t_1.i = t_2$ and its negation are proven similar to case (c).

(h) $(t_1, \ldots, t_k) = (s_1, \ldots, s_k)$ is equivalent to $s_1 = t_1 \wedge \cdots \wedge s_k = t_k$, and therefore the induction hypothesis implies the correctness.

■

The termination follows from the following lemma:

Lemma 3.23 (Time Complexity of Algorithm *instantiate_condition*) *Let*

$$C = \bigvee_{i=1}^{r} \bigwedge_{j=1}^{s_i} L_{ij}$$

be a condition in disjunctive normal form and let $p = s_1 + \cdots + s_r$ *be the number of literals in* C. *Furthermore let* m *be the maximal natural number occuring in condition* C *(*$m = 2$, *if there is no such maximal number or the maximal number is less than 2),* k *be the maximal number of tls, hds,*

and $(\cdot)[i]$ *occuring in a literal* L*, and let* q *be the maximal number of cons and* $(\cdot,\cdot)$ *occuring in a literal* L*. Then algorithm instantiate_condition*$(C,(x_1,\ldots,x_n))$ *terminates after*

$$O\left(p\ \log m\ m^{2\cdot p\cdot 2^{(q+km)/2}+2} g(p,1,g^{2^q}((q+k\ m)/2,m,n+k\ m))\right)$$

steps, where $g(q,m,n)$ *is defined by the recurrence*[2]*:*

$$\begin{aligned} g(0,m,n) &= n \\ g(q,m,n) &= m^{2^q}\ g(q-1,m,n)^{g(q-1,m,n)} \end{aligned}$$

Proof: See appendix B ■

The following corollary summarizes the results of appendix B:

Corollary 3.24 (Complexity Results of Alogrithm *instantiate_condition*) *If* k *is the maximal subterm size, when* **if** $\cdot$ **then** $\cdot$ **else** $\cdot$ *is not counted, and the* **let** *abbreviations are expanded, in a set of equations* E*, and* m *is the maximal natural number occuring in* E*, then we have for any condition* C *occuring in* E*, and tuples* $(x_1,\ldots,x_n)$ *of variables:*

(a) *The running time of algorithm* instantiate_condition$(C,(x_1,\ldots,x_n))$ *is:*

$$O\left(k\ \log m\ m^{2\cdot k\cdot 2^{(m+1)k/2}+2} g(k,1,g^{2^k}((m+1)\ k/2,m,(m+1)\ k/2))\right)$$

(b) $sz(instantiate\_condition(C,(x_1,\ldots,x_n))) \leq m^{k\cdot 2^{(m+1)k/2}}\ g(k,1,g^{2^k}((m+1)\ k/2,m,(m+1)\ k/2))$

(c) $card(instantiate\_condition(C,(x_1,\ldots,x_n))) \leq m^{k\cdot 2^{(m+1)k/2}}$

(d) *The maximal size of term in* instantiate_condition$(C,(x_1,\ldots,x_n))$ *is:*

$$g(k,1,g^{2^k}((m+1)\ k/2,m,(m+1)\ k/2))$$

where $g(q,m,n)$ *is defined as in lemma 3.23.*

Finally, the correctness and complexity results of algorithm *symbolic_eval* are proven based on the above lemmas and corollaries.

Theorem 3.25 (Correctness of Algorithm *symbolic_eval*) *Let be* E' = symbolic_evaluation(E)*. Then for each* $t \in EXPR$*,* $n \in \mathbf{N}$*,* $p \in LOCAL$*, and* $\rho \in ENV$ *it holds:*

$$EVAL[\![t]\!]\ E\ n\ p\ \rho = EVAL[\![t]\!]\ E'\ n\ p\ \rho$$

Proof: We show that one iteration of loop 1 doesn't change the semantics, i.e. if E is changed to F and E' to F' by one execution of the body of loop 1, then for any $t \in EXPR$, $n \in \mathbf{N}$, $p \in LOCAL$, and $\rho \in ENV$:

$$EVAL[\![t]\!]\ E \cup E'\ n\ p\ \rho = EVAL[\![t]\!]\ F \cup F'\ n\ p\ \rho$$

[2]Observe that for fixed m and n: $A(2,q) \leq g(q,m,n) \leq A(3,q)$ where $A(i,j)$ is Ackermanns function

From this and the termination (proven in theorem 3.26), follows the correctness of algorithm *symbolic_evaluation*, because at the beginning is $E' = \emptyset$ and at the end is $E = \emptyset$.

Consider now the selected equation of line (4):

$$e := LHS = \textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2$$

It is immediate that lines (13) and (19) do not change the semantics of $E \cup E'$.

Consider now the equations:

$$\begin{aligned} F_1 &= \{\sigma LHS = \sigma E_1 | \sigma = [x_1/s_1] \cdots [x_n/s_n], (s_1, \ldots, s_n) \in S_1\} \\ F_2 &= \{\sigma LHS = \sigma E_2 | \sigma = [x_1/s_1] \cdots [x_n/s_n], (s_1, \ldots, s_n) \in S_2\} \end{aligned}$$

where S_1 and S_2 are the sets obtaine in line (21) and (22), respectively. We know form the condition in line (18) that they describe disjoint sets of terms. It is $F \cup F' = (E - e) \cup E' \cup F_1 \cup F_2$. We show now by induction on the number of $EVAL$ steps to evaluate t, that for any $t \in EXPR$, $n \in \mathbf{N}$, $p \in LOCAL$, and $\rho' \in ENV$:

$$EVAL[\![t]\!]\ E \cup E'\ n\ p\ \rho' = EVAL[\![t]\!]\ F \cup F'\ n\ p\ \rho'$$

The only interesting case to consider is if the equation e is applicable. Thus the base case is trivial, and many of the induction cases are trivial, because equations in $(E \cup E') \cap (F \cup F')$ are applied.

If e is applicable, then ther is a substitution σ' such that $\sigma' LHS = t$, and one application of $EVAL$ yields:

$$EVAL[\![\textbf{if } C \textbf{ then } E_1 \textbf{ else } E_2]\!]\ E \cup E'\ n\ p\ \sigma'\rho'$$

CASE 1: $EVAL[\![C]\!]\ \emptyset 1()\sigma'\rho' = true$. Then we have immediately that:

$$EVAL[\![C]\!]\ E \cup E'\ n\ p\ \sigma'\rho' = true$$

and after applying the rule for conditionals, that

$$EVAL[\![t]\!]\ E \cup E'\ n\ p\ \rho' = EVAL[\![E_1]\!]\ E \cup E'\ n\ p\ \sigma'\rho'$$

By lemmas 3.22 and lm:minim we know that there is a $\sigma \in \{[x_1/s_1] \cdots [x_n/s_n] | (s_1, \ldots, s_n) \in S_1\}$ and a $\tau \in ENV$ such that $\sigma'\rho' = \tau\sigma\rho'$. Hence

$$\begin{aligned} & EVAL[\![t]\!]\ F \cup F'\ n\ p\ \rho' = \\ &= \{\sigma LHS = \sigma E_1 \text{ is applicable}, \tau t = \sigma LHS\} \\ &\quad EVAL[\![\sigma E_1]\!]\ F \cup F'\ n\ p\ \tau\rho' \\ &= \{\text{See above remark} \sigma'\rho' = \tau\sigma\rho'\} \\ &\quad EVAL[\![E_1]\!]\ F \cup F'\ n\ p\ \sigma'\rho' \\ &= \{\text{inductionhypothesis}\} \\ &\quad EVAL[\![E_1]\!]\ E \cup E'\ n\ p\ \sigma'\rho' \\ &= \{\text{see above}\} EVAL[\![t]\!]\ E \cup E'\ n\ p\ \rho' \end{aligned}$$

CASE 2: $EVAL[\![C]\!]\ \emptyset 1()\sigma'\rho' = false$. Analgously to case 1 using lemmas 3.22, lm:minim and corollary 3.17. ■

Finally we give the time complexity.

Theorem 3.26 (Time Complexity of Algorithm *symbolic_evaluate*) *Let E be a set of equations, k and m be defined as in corollay 3.24. Further let $g(q,m,n)$ be the function defined in lemma 3.23, and define $G(q,n) = g(q,1,n)\ \log g(q,m,n)$. Let n be the length of E (i.e. the number of symbols in E). Then* symbolic_evaluation(E) *requires at most*

$$O\left(n\ m^{2\cdot k\cdot 2^{(m+1)k/2}+1}\ \log m\ 2\ k\ 2^{(m+1)k/2} G(k, g^{2^k}((m+1)\ k/2, m, (m+1)\ k/2))\right)$$

time steps.

Proof: It is easy to see, that the body of loop 1 is executed at most $O(n)$ times, lines (4)–(11) cost $O(k)$ and line (15) cost at most $O(2^k)$. From corollary 3.24 we conclude that line (16) costs

$$O\left(k\ \log m\ m^{2\cdot k\cdot 2^{(m+1)k/2}+2}\ g(k,1,g^{2^q}((m+1)\ k/2, m, (m+1)\ k/2))\right)$$

We can also conclude form corollary 3.24 and 3.17, that line (17) costs

$$O\left(m^{2\cdot k\cdot 2^{(m+1)k/2}+2}\ \log m\ 2\cdot k\cdot 2^{(m+1)k/2}\ G(k, g^{2^k}((m+1)\ k/2, m, (m+1)\ k/2))\right)$$

The running time of loops 2 and 3 is $O(sz(S_1))$ and $O(sz(S_2))$ where S_1 and S_2 are the sets defined lines (21) and (22), respectively. Hence the running times of these loops are dominated by the running time of line (17). One execution of a loop body costs therefore time

$$O\left(m^{2\cdot k\cdot 2^{(m+1)k/2}+2}\ \log m\ 2\cdot k\cdot 2^{(m+1)k/2}\ G(k, g^{2^k}((m+1)\ k/2, m, (m+1)\ k/2))\right)$$

Together with the fact that this body is executed n times, the theorem is proven. ■

Observe (from appendix B) that these estimates are grossly pessimistic. The high complexity arises also from the fact that the term σt where σ is the most general unifier of t and u, is bounded by r^r where $r = sz(t)$. This could happen, but only in very special cases. We conjecture that the running time is usuually polynomial in the above defined k and m. Furhtermore if k and m are constant compared to n, the length of the program, the overall running time is still linear in the length of the program. The result shows that not the length of the program, but a difficult structure of the program make symbolic evaluation difficult.

3.1.4 Derivation of Recurrences

Finally, recurrences are obtained from the equations derived in the pervious subsection. On each argument position, one of the complexity measures *lg*, *sz*, or ll_k is applied, depending on the structure of the equations. For each argument position, a new argument is introduced, containing the length, size, or level-length of the corresponding argument. Finally, irrelevant argument positions are removed by a slight extension of algorithm *irrelevant_position* (this extension is left to the reader). Observe, that in contrast to previous work [Weg75, Zim90a, Zim90b] conditional recurrences contain some more variables, needed to obtain a worst-case complexity. The correctness of this step is given by the following condition:

For each term $f(t_1, \ldots, t_k)$ where f is defined by E, for each $\rho \in ENV$, $p \in LOCAL$, and $n \in \mathbf{N}$ holds:

$$EVAL[\![f(t_1, \ldots, t_k)]\!]\ E \cup \Pi\ n\ p\ \rho = EVAL[\![f(t_{i_1}, \ldots, t_{i_s}, M_1(t_1), \ldots, M_k(t_k))]\!]\ R \cup \Pi\ n\ p\ \rho \quad (3.2)$$

Here R and M are the recurrences and complexity measures for the arguments obtained by this step, $(t_{i_1}, \ldots, t_{i_l})$ is a (possibly empty) sublist of $t_1, \ldots, t_k$, and $f/i \equiv M_i \in M$, where $f/i = M_i$ means that argument position f/i is mapped by measure M_i onto natural numbers. For each $f(u_1, \ldots, u_s, v_1, \ldots, v_k) = RHS \in R$ holds: if x is a variable occuring in RHS and occuring in a u_i, then the only place where x can occur is in a condition or in a function call to Π. This is the same place where function calls to Π can be.

Algorithm *create_recurrences*
INPUT: A set of equations E, and a program Π.
OUTPUT: A set of recurrences R, and measures M satisfying (3.2)

1. $(M, R) := derive\_mappings(E)$;
2. $R := derive\_recurrences(E, M)$;
3. $(R, M') := remove\_free\_vars(R)$;
4. $I := irrelevant\_positions(R)$;
5. $R := remove\_irr\_pos(R, I)$;
6. **output** R and $M \uplus M'$

■

The algorithm *irrelevant_positions* is just a slight modification from that of subsection 3.1.2. Algorithm *remove_irr_pos* is loop 2 of algorithm *normalize* in subsection 3.1.2. Hence it remains to find the adequate mappings and if they are found the derivation of the recurrences, given the fact that the adequate mappings are already provided.

We start with defining algorithm *derive_recurrences*. The algorithm runs in two phases. First each function f of arity k is enhanced to a function with arity $2 \cdot k$, where the argument on position $k+i$ is $\mu(t_i)$ when t_i is the $i-th$ argument of f and $f/i = \mu$ in M. Then expressions like $\mu(x)$ where μ is a measure and x is a variable are replaced by a new variable. Moreover the resulting set of equations should satisfy (3.2) where $t_{i_j} = t_j$ for all $j = 1, \ldots, k$.

Observe that for the resulting equation system most of the arguments in the first half of a function f become irrelevant. Only the argument positions remain relevant, if they are important for the value of a condition in a conditonal equation.

Algorithm *derive_recurrences*
INPUT: A set of equations E and a measure set M.
OUTPUT: A set of equations R such (3.2) is satisfied where always $t_{i_j} = t_j$ for all $j = 1, \ldots, k$.

(1) $R := \emptyset$;
(2) **for each** $LHS = RHS \in E$ **do**
(3) **assume** $LHS = f(t_1, \ldots, t_k)$;
(4) **assume** $f/1 = \mu_1, \ldots, f/k = \mu_k \in M$;
(5) $LHS' := f(t_1, \ldots, t_k, \mu_1(t_1), \ldots, \mu_k(t_k))$;
(6) simplify LHS' according to rule 10
(7) **repeat**
(8) simplify RHS according to rule 10
(9) simplify RHS according to rule 11
(10) $RHS' :=$ **RECUR**$[\![RHS]\!]\ M$;
(11) **until** rule 11 and rule 10 are not applicable
(12) $R := R \cup \{LHS' = RHS\}$;
(13) **end for**;
(14) **for each** variable x occuring in R **do**
(15) **for each** measure μ occuring in R
(16) **let** n be a new variable;
(17) store the pair $(\mu(x), n)$ in a look-up table A;
(18) **end for**;
(19) **end for**;
(20) scan through R and replace each occurences of $\mu'(y)$ by $A[\mu'(y)]$;
(21) **output** R;

■

The transformation scheme **RECUR**$[\![\cdot]\!]\ M$ is given in figure 3.6. It adds new parameters to each function call according to M. The rules 10 just apply simplification rules for the measures:

Rule 10 (Simplification of Terms with Measures)

$$\frac{lg(cons(\mathcal{A}, \mathcal{L}))}{1 + lg(\mathcal{L})} \qquad \frac{lg(\langle\rangle)}{0} \qquad \frac{lg(tl(\mathcal{L}))}{lg(\mathcal{L}) - 1} \frac{lg(\mathcal{X})}{\mathcal{X}} \text{ if } \mathcal{X} \text{ is of type nat}$$

$$\frac{ll_k(cons(\mathcal{A}, \mathcal{L}))}{1 + ll_{k-1}(\mathcal{A}) + ll_k(\mathcal{L})} \qquad \frac{ll_k(\langle\rangle)}{0}$$

$$\frac{sz(cons(\mathcal{A}, \mathcal{L}))}{1 + sz(\mathcal{A}) + sz(\mathcal{L})} \qquad \frac{sz(\langle\rangle)}{0}$$

$$\frac{lg(\textbf{forall } \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ do in parallel} \mathcal{T}(\mathcal{I}))}{\mathcal{R} - \mathcal{L}}$$

$$\frac{ll_{k+1}(\textbf{forall } \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ do in parallel} \mathcal{T}(\mathcal{I}))}{\mathcal{R} - \mathcal{L} + \sum_{\mathcal{I}=\mathcal{L}}^{\mathcal{R}} ll_{k-1}(\mathcal{T}(\mathcal{I})}$$

$$\frac{sz(\textbf{forall } \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ do in parallel} \mathcal{T}(\mathcal{I}))}{\mathcal{R} - \mathcal{L} + \sum_{\mathcal{I}=\mathcal{L}}^{\mathcal{R}} sz(\mathcal{T}(\mathcal{I})}$$

RECUR[[**if** C **then** T_1 **else**T_2]] M = **if** C **then** **RECUR**[[T_1]] M **else**RECUR[[T_2]] M

RECUR[[$g(t_1, \ldots, t_k)$]] M =

$$\begin{cases} g(t_1, \ldots, t_k, \mu_1(\textbf{RECUR}[\![t_1]\!]\ M), \ldots, \mu_k(\textbf{RECUR}[\![t_k]\!]\ M)) & \text{if } g/i = \mu_i \in M, 1 \leq i \leq k \\ g(\textbf{RECUR}[\![t_1]\!]\ M, \ldots, \textbf{RECUR}[\![t_k]\!]\ M) & \text{if } g \text{ is a basic operation on } nat \\ g(t_1, \ldots, t_k) & \text{otherwise} \end{cases}$$

RECUR[[**forall** $l \leq i < r$ **do in parallel** $t(i)$]] M = **forall** $l \leq i < r$ **do in parallel** $t(i)$

RECUR[[**select** $l \leq i < r$ **in parallel from** $t(i)$]] M = **select** $l \leq i < r$ **in parallel from** $t(i)$

RECUR[[**modify** $e_1(i)$, $l \leq i < r$ **to** $e_2(i)$ **from** e_3]] M = **modify** $e_1(i)$, $l \leq i < r$ **to** $e_2(i)$ **from** e_3

Figure 3.6: The Transformation **RECUR**

$$\frac{\mathcal{M}(\textbf{select } \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textit{ in parallel from } \mathcal{T}(\mathcal{I}))}{\max_{\mathcal{I}=\mathcal{L}}^{\mathcal{R}} \mathcal{M}(\mathcal{I})}$$

$$\frac{lg(\textbf{modify } \mathcal{E}_1(\mathcal{I}),\ \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ to } \mathcal{E}_2(\mathcal{I}) \textbf{ from } \mathcal{E}_3)}{lg(\mathcal{E}_3)}$$

$$\frac{ll_{k+1}(\textbf{modify } \mathcal{E}_1(\mathcal{I}),\ \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ to } \mathcal{E}_2(\mathcal{I}) \textbf{ from } \mathcal{E}_3)}{ll_{k+1}(\mathcal{E}_3) + \sum_{\mathcal{I}=\mathcal{L}}^{\mathcal{R}} ll_k(\mathcal{E}_2(\mathcal{I})) - ll_k(\mathcal{E}_\ni[\mathcal{E}_2(\mathcal{I})])}$$

$$\frac{sz(\textbf{modify } \mathcal{E}_1(\mathcal{I}),\ \mathcal{L} \leq \mathcal{I} < \mathcal{R} \textbf{ to } \mathcal{E}_2(\mathcal{I}) \textbf{ from } \mathcal{E}_3)}{sz(\mathcal{E}_3) + \sum_{\mathcal{I}=\mathcal{L}}^{\mathcal{R}} sz(\mathcal{E}_2(\mathcal{I})) - sz(\mathcal{E}_\ni[\mathcal{E}_2(\mathcal{I})])}$$

The rule 11 replaces each call of $\mu(f(\cdots))$ by μ_f, which is available in E. Therefore it makes R closed under its defined functions.

Rule 11

$$\frac{\mathcal{M}(\mathcal{F}(\mathcal{P}))}{\mathcal{M}\_\mathcal{F}(\mathcal{P})} \qquad \mathcal{M} \in \{lg, sz, ll_k\} \mathcal{F} \in \Pi$$

Lemma 3.27 (Correctness of Algorithm *derive_recurrences***)** *If M is a correct output of algorithm derive_mappings(E), then algorithm derive_recurrences(E, M) outputs a recurrence system R satisfying (3.2) where $s = k$ for any function f.*

Proof: We prove the following, stronger result:

For any $t \in EXPR$, $n \in \mathbf{N}$, $p \in LOCAL$, and $\rho \in ENV$ holds:

$$EVAL[\![t]\!]\ E \cup \Pi\ n\ p\ \rho = EVAL[\![t']\!]\ R \cup \Pi\ n\ p\ \sigma\rho \tag{3.3}$$

where t' is obtained by t by first applying lines (7)-(11) to t, and then line (20), and $[n/\mu(\rho(x))] \in \sigma \Leftrightarrow (\mu(x), n) \in A$

The correctness is a special case of (3.3). The only difference between the correctness and case 5 is that in case 5 internal variables of algorithm *derive_recurrences* are used to express the same fact. The stronger claim (3.3) is proven by induction over the structure of t. In the proof we omit the parameter n and p because they play no role.

CASE 1: $t = x$ is a variable. Then $t = t'$ and (3.3) follows immediately.
CASE 2: $t = \mu(x)$ where x is a variable. Then $t' = n$ where n is a new variable, and by lines (14)–(19), $(\mu(x), n) \in A$. Thus:

$$EVAL[\![n]\!]\ R \cup \Pi\ \sigma\rho = \sigma(n) = \mu(\rho(x)) = EVAL[\![\mu(x)]\!]\ E \cup \Pi\ \rho$$

CASE 3: $t = g(t_1, \ldots, t_k)$ where g is an arithmetic operation. Then by **RECUR[·]** M, $t' = g(t'_1, \ldots, t'_k)$ and the (3.3) follows immediately from the induction hypothesis.
CASE 4: $t = g(u_1, \ldots, u_k)$ where **fun** $g(x_1 : \tau_1, \ldots, x_k : \tau_k) : \tau = B \in \Pi$. Then $g/i \notin M$ for any $1 \leq i \leq k$. Then by transformation **RECUR[·]** M, $t' = t$. Moreover rules 10 and rules 11 cannot be applied, because in this case none of the u_i cannot contain any function call to E (or R). The same holds for B, because g was a function in Π. Thus $t' = t$ also after line (20). Hence the induction hypothesis proves (3.3).
CASE 5: $t = g(u_1, \ldots, u_k)$ where g is defined by E. Let $g/i = \mu_i \in M$ for $i = 1, \ldots, k$ and $v_i = (\mu_i(u_i))'$. Then by the definition of **RECUR[·]** M, $t' = g(u_1, \ldots, u_m, v_1, \ldots, v_m)$. Moreover the u_i do not contain any call to a function defined by E. Hence:

$$EVAL[\![u_i]\!]\ E \cup \Pi\ \rho = EVAL[\![u_i]\!]\ R \cup \Pi\ \rho = EVAL[\![u_i]\!]\ R \cup \Pi\ \sigma\rho$$

From the induction hypothesis follows that

$$EVAL[\![v_i]\!]\ R \cup \Pi\ \sigma\rho = EVAL[\![\mu_i(u_i)]\!]\ E \cup \Pi\rho$$

Let $LHS = RHS \in E$ such that there is a $\bar{\sigma}$ with:

$$\bar{\sigma} LHS = g(EVAL[\![u_1]\!]\ E \cup \Pi\ \rho, \ldots, EVAL[\![u_k]\!]\ E \cup \Pi\ \rho)$$

Consider now the corresponding $LHS' = RHS' \in R$. Then this rule is by line (6) applicable to t'. Let σ' such that:

$$\sigma' LHS' = g(EVAL[\![u_1]\!]\ R \cup \Pi\ \sigma\rho, \ldots, EVAL[\![u_k]\!]\ R \cup \Pi\ \sigma\rho, EVAL[\![v_1]\!]\ R \cup \Pi\ \sigma\rho, \ldots, EVAL[\![v_k]\!]\ R \cup \Pi\ \sigma\rho)$$

Hence $\sigma' = \tilde{\sigma}\bar{\sigma}$ because the evaluation of the u_i depend only on ρ, and contain by lines (14)–(20) different variables than the v_i. By induction hypothesis we have

$$EVAL[\![RHS]\!]\ E \cup \Pi\ \bar{\sigma}\rho = EVAL[\![RHS']\!]\ R \cup \Pi\ \hat{\sigma}\bar{\sigma}\rho$$

where $[n/\mu(\bar{\sigma}\rho(x))] \in \hat{\sigma} \Leftrightarrow (\mu(x), n) \in A$. Hence, it remains to prove that $\hat{\sigma} = \tilde{\sigma}\sigma$. Consider a variable n, where $(\mu(x), n) \in A$. If $\hat{\sigma}(n) = \perp$, then $\sigma(n) = \perp$. Thus the only possibility that $\hat{\sigma}(n) \neq \tilde{\sigma}\sigma(n)$ is if $\tilde{\sigma}(n) \neq \perp$. But then n is defined by LHS' and therefore by definition of $\hat{\sigma}$, $\hat{\sigma}(n) = \mu(\bar{\sigma}\rho(x)$ where $(\mu(x), n) \in A$, contradicting the assumption of its undefinedness. Hence, $\tilde{\sigma}\sigma(n)$ is defined, iff $\hat{\sigma}(n)$ is defined. Consider now the following cases:
First case: $\sigma(n) = t$ and $\tilde{\sigma}(n) = \perp$. It must be $(\mu(x), n) \in A$. Therefore $\hat{\sigma}(n) = \mu(\rho(x))$

(otherwise x would occur in LHS and LHS', and therefore n in LHS' by the correctness of algorithm *derive_mappings*). By the definition of σ it is also $\sigma(n) = \rho(x)$.
Second case: $\bar{\sigma}(n) = t \neq \bot$. Then $\bar{\sigma}\sigma(n) = t$. Let $(\mu(x), n) \in A$. Because in this case x occurs in LHS and the correctness of algorithm *derivve_mappings*, n occurs in LHS'. Hence $\bar{\sigma}(n) = \mu(\bar{\sigma}(x)) = \hat{\sigma}(n)$.
CASE 6 $t =$ **if** C **then** t_1 **else** t_2. Then by transformation **RECUR**, $t' =$ **if** C **then** t'_1 **else** t'_2. The induction hypothesis and the correctness of algorithm *derive_recurrences* prove (3.3). CASE 7 t is not one of the above cases. Then it is a parallel operator. The same arguments as in cases 4 or 6 prove (3.3) ■

Lemma 3.28 (Complexity of Algorithm *derive_recurrences*)
Algorithm derive_recurrences(E, M) terminates after $O(k \cdot n)$ steps where k is the maximal nesting depth in E, and n is the number of symbols in E (or the length of E)

Proof: It is immediate to see that the operations in the loop of lines (14)–(19) can be performed in $O(k\ n)$. The look-up table can be implemented as an array because it is possible to determine k and the number of variables in advance by a simple scan through E. This scan requires just $O(n)$ time. Line (20) is also a scan through E and requires therefore time $O(n)$.

Consider now the loop of lines (7)–(11). Each execution of the body requires time $O(r)$ where r is the length of RHS. Furthermore the loop is executed at most k times, because in iteration i rule 11 is applied only to terms on nesting depth i. Hence the cost of loop (7)–(11) is $O(k \cdot r)$. Moreover lines (3)–(6) require time proportional to the length of LHS. Thus summing this up for all equations lead for the loop of lines (2)–(13) to a time proportional to $O(k\ n)$. ■

Now we turn to the algorithm *derive_mappings*. The idea behind this algorithm is to try to get adequate mappings first with all argument positions *lg*. A mapping is adequate if apart from conditions, when these mappings are applied to each argument positions and lines (14)–(20) are applied to the resulting equation system, it is closed. More specifically, if an RHS contains free variables or symbols not defined by E, then these variables in symbols occur only in conditions. If this property is not satisfied by applying M, then the level of the mappings on argument positions contradicting it are increased by 1. If the maximal possible level is achieved, then sz is chosen.

Algorithm *derive_mappings*
INPUT: A set of equations E
OUTPUT: A set of mappings M, and set of equations R, such that if $LHS = RHS \in R$ contains free variables or undefined function symbols, these occur only in conditions.

(1) $M := \{f/i = lg |\ f$ defined by E and f/i is argument position $\}$;
(2) $R := \emptyset$;
(3) **repeat**
(4) **for each** $LHS = RHS \in E$ **do**
(5) **assume** $LHS = f(t_1, \ldots, t_k)$, $f/i = \mu_i \in M$, $1 \leq i \leq k$;
(6) $LHS' := f(\mu_1(t_1), \ldots, \mu_k(t_k))$;
(7) simplify LHS' according to rule 10;
(8) **repeat**
(9) simplify RHS according to rule 10;
(10) simplify RHS according to rule 11;

$$\overline{\mathbf{RECUR}}[\![\textbf{if } C \textbf{ then } T_1 \textbf{ else} T_2]\!]\ M = \textbf{if } C \textbf{ then } \overline{\mathbf{RECUR}}[\![T_1]\!]\ M \textbf{ else} \overline{\mathbf{RECUR}}[\![T_2]\!]\ M$$

$$\overline{\mathbf{RECUR}}[\![g(t_1,\ldots,t_k)]\!]\ M =$$

$$\begin{cases} g(\mu_1(\overline{\mathbf{RECUR}}[\![t_1]\!]\ M),\ldots,\mu_k(\overline{\mathbf{RECUR}}[\![t_k]\!]\ M)) & \text{if } g/i = \mu_i \in M, 1 \le i \le k \\ g(\overline{\mathbf{RECUR}}[\![t_1]\!]\ M,\ldots,\overline{\mathbf{RECUR}}[\![t_k]\!]\ M) & \text{if } g \text{ is a basic operation on } nat \\ g(t_1,\ldots,t_k) & \text{otherwise} \end{cases}$$

$$\overline{\mathbf{RECUR}}[\![\textbf{forall } l \le i < r \textbf{ do in parallel } t(i)]\!]\ M = \textbf{forall } l \le i < r \textbf{ do in parallel } t(i)$$

$$\overline{\mathbf{RECUR}}[\![\textbf{select } l \le i < r \textbf{ in parallel from } t(i)]\!]\ M = \textbf{select } l \le i < r \textbf{ in parallel from } t(i)$$

$$\overline{\mathbf{RECUR}}[\![\textbf{modify } e_1(i),\ l \le i < r \textbf{ to } e_2(i) \textbf{ from } e_3]\!]\ M = \textbf{modify } e_1(i),\ l \le i < r \textbf{ to } e_2(i) \textbf{ from } e_3$$

Figure 3.7: The Transformation $\overline{\mathbf{RECUR}}$

(11) $RHS := \overline{\mathbf{RECUR}}[\![RHS]\!]\ M$;
(12) **until** rule 11 and rule 10 are not applicable;
(13) $R := R \cup \{LHS' = RHS\}$;
(14) **if** RHS contains a free variable x or unknown function symbol
(15) occuring not in a condition **then**
(16) **for** $i = 1,\ldots,k$ **where** t_i contains x **do**
(17) **if** $f/i = lg$ **then** $M := (M - \{f/i = lg\}) \cup \{f/i = ll_2\}$; **end if**;
(18) **if** $f/i = ll_k$ **then** $M := (M - \{f/i = ll_k\}) \cup \{f/i = ll_{k+1}\}$; **end if**;
(19) $R := \emptyset$;
(20) **end for**;
(21) **end if**;
(22) **end for**;
(23) **until** M does not change [equivalently: $r \neq \emptyset$]
(24) Replace each $f/i = ll_k$ with maximal level by $f/i = sz$;
(25) **output** M and R;

■

The transformation $\overline{\mathbf{RECUR}}[\![\cdot]\!]\ M$ is similar to figure 3.6 and defined by figure 3.7. It may be the case that some variables representing $lg(x)$ for variables x occur free in R. This can only occur the case when lg was explicitely used in the program. In this case the corresponding argument position is abstracted with two different mappings. For an example see the adaptive prefix sum algorithm in the following section. Algorithm *remove_free_vars* covers this situation. It adds in R the new arguments, and puts in M' the new mappings. All this can be achieved in time $O(n)$ by three scans through the program. First it selects the free variables together with their positions (can be obtained from A used in algorithm *derive_recurrences*) then it adds the corresponding parameter in each program, and updates A. Finally the replacements as in step (20) of algorithm *derive_recurrences* are made.

Lemma 3.29 (Correctness of Algorithm *derive_mappings***)** *Algorithm derive_mappings is correct.*

Proof: If M doesn't change, then $R \neq \emptyset$. Hence the correctness of algorithm *derive_recurrences* follows from its termination (because then the condition in line (14) is never satisfied). Let m be the maximal nesting depth in E. $f/i = sz$ for each argument positions would lead to recurrences as shown in [Zim90a]. If the loop of lines (4)–(22) is repeated then at least one ll_k is changed to ll_{k+1}. But for each argument position this can happen at most m times. Hence the algorithm terminates. ■

Corollary 3.30 (Complexity of Algorithm *derive_mappings***)** *If n is the number of symbols in E and k is the maximal nesting depth in E, then algorithm derive_mappings(E) terminates after $O(k \cdots n)$ steps.*

Proof: One execution of the loop of lines (4)–(22) cost time $O(n)$, because it scans one times through E. If the k-th iteration does not succeed, then all $f/i = ll_k$ are set to $f/i = ll_{k+1}$. Thus the loop of lines (3)–(23) require time $O(k\ n)$. Line (24) is certainly executable in time $O(n)$. ■

For this step we have therefore by lemmas 3.27, 3.29, 3.6, 3.7, corollaries 3.28, 3.30, and the obvious fact that algorithm *irrelevant_positions* and loop 2 in algorithm *normalize* cost time $O(n)$:

Theorem 3.31 (Correctness and Complexity of Algorithm *create_recurrences***)** *Let n be the number of symbols in E and k the maximal nesting depth of E. Then create_recurrences(E) outputs after $O(n \cdot k)$ steps a system of recurrences satisfying (3.2).*

3.1.5 Summary

We have now shown, that the method in this section derives correctly a system of recurrences describing the desired complexity C. Moreover for each correctly typed program Π, this algorithm terminates (in the worst case after a huge number of steps).

Theorem 3.32 (Correctness of Algorithm *translate***)** *Algorithm translate(Π, C) outputs a system of recurrences R and measures M such that for each closed $t \in EXPR$:*

$$C[\![t]\!]\ \Pi\ \varepsilon \leq EVAL[\![(\mathbf{TC}[\![t]\!])']\!]\ R \cup \Pi\ \varepsilon$$

where R contains symbols not defined by R only at conditions or at argument positions whose variables occur only in condition, and t' is obtained from t w.r.t. M as in the proof of lemma 3.27.

The proof of this theorem follows directly from the correctness of the algorithms *remove_incompleteness*, *transform*, *symbolic_eval*, and *create_recurrences*.

From the corresponding complexity results, together with a few simple observations, we get:

Theorem 3.33 (Time Complexity of Algorithm *transform***)** *Let n be the size of the program Π, k its maximal nesting depth, and m the largest natural number occuring in Π ($m = 1$ if no such number appears). Then algorithm transform(Π) terminates after $O(f(n,k,m)$ steps where $f(k,m,n)$ is defined by:*

- $f(k,m,n) = n \cdot h(k,m) \cdot g(k,1,g^{2^k}((m+1)\ k/2, m, (m+1)\ k/2)) \cdot G(k, g^{2^k}((m+1)\ k/2, m, (m+1)\ k/2))$

- $h(k,m) = k^3 \ \exp(2 \ \log(m) \ k \ 2^{(m+1)k/2} + 1) \ \log(m) \ k \ 2^{(m+1)k/2}$
- *$g(q,m,n)$ is defined by the recurrence*

$$\begin{aligned} g(0,m,n) &= n \\ g(q,m,n) &= m^{2^q} \ g(q-1,m,n)^{g(q-1,m,n)} \end{aligned}$$

- $G(q,n) = g(q,1,n) \log g(q,1,n)$

Proof: The size of Π' is less or equal $k\ n$. Its nesting depth does not increase compared to Π. It is easy to see that algorithm *translate* yields the size $O(n)$ for Π_0 and size $O(k\ n)$ for Π_i, if $i > 0$. Furthermore the maximal nesting depth is not increased. Hence Π'' has size $O(n\ k^2)$ and maximal nesting depth k. Algorithm *normalize* does not change any of these bounds. The maximal natural number occuring in Π is only important in conditions, and there it is not changed by any of these algorithms. Then by theorem 3.26 the size of E is at most

$$O(n \cdot h(k,m) \cdot G(k, g^{2^k}((m+1)\ k/2, m, (m+1)\ k/2))$$

Furthermore, by lemma B.2, the maximal nesting depth increases to

$$g(k, 1, g^{2^k}((m+1)\ k/2, m, (m+1)\ k/2))$$

Thus by *create_recurrences*(E) needs the stated time, which even dominates the time needed for the execution of *symbolic_eval*(Π''). ■

The complexity result of algorithm *transform* shows that the difficulty in analyzing the complexity of the program is not its length but its structure. This matches the practical experience. An algorithm based purely on structural induction can be analyzed very fast, while analyzing programs like *quicksort* requires already a few minutes on a SUN 4. However all the estimates here are grossly pessimistic (arising in fact from the exponential upper bound for the output size of unification). Usualy this does not happen very often, but cannot be excluded. A more refined analysis of algorithm *transform* seems very difficult to manage. In fact one has to find bounds on the fact that the output size of a unification is exponential. We conjecture that this cannot be happen always. If it would, we could really have a program with these bounds. In most of the practical cases m and k can be assumed to be constant compared to the size of the program, and therefore algorithm *transform* is linear in the size of the program in most practical cases.

For the solution of recurrences we refer to [Zim90a]. If they are solved with generating functions, techniques from [FV90, FS87, Fla88, FSZ91, Zim91] can be used. The solution of particular recurrences is discussed in the example sections. Also dealing with difficulties (as e.g. pointer jumping in [Zim90b]) are discussed with the particular example. One big difference between parallel and sequential algorithms is, that in the sequential case, the recurrences have "arithmetic" character, while in the parallel case they have "geometric" character.

3.2 Basic Examples

We mainly introduce here some basic examples used in further analysis examples. The first example is basic pointer jumping, while the second is prefix sum computation. Here we use an adaptive and a non-adaptive algorithm. For discussion of these algorithms see [GR88] For each of the analysis, we assume that the basic complexities are one, thus counting the number of *EVALs*.

3.2.1 Pointer Jumping

The idea behind this algorithm is to determine the distance of each element in a list to its end. For this purpose, we define a vector p containing adresses. This vector has the property that the graph $G = (\{0, \ldots, lg(\mathrm{p}) - 1\}, \{(i, \mathrm{p}[i]) | 0 \leq i < lg(\mathrm{p})\}$ is a chain ending at cycle of length 1. The idea behind pointer jumping is to set initialize a vector `dist` of the same length as p all elements but the last to 1 (the last element is set to 0), and jump, i.e. one sets in parallel `p[i]` to `p[p[i]]` and adds the corresponding values in `dist`. The algorithm is repeated until the array p does not change. This algorithm terminates because the length of the longest chains halves at each iteration. If each chain defined by p has length 1 then it remains stable. The program is:

```
fun rank(p:<nat>):<nat> =
  let dist = forall 0 <= i < lg(p) do in parallel
       if p[i] = i then 0 else 1
  in repeat(p,dist)

fun repeat(p:<nat>,dist:<nat>):<nat> =
  let n = lg(p) in
  let d = forall 0 <= i < n do in parallel
            if p[i] <> p[p[i]] then dist[i] + dist[p[i]]
                               else dist[i]
  in let p' = forall 0 <= i < n do in parallel
                if p[i] <> p[p[i]] then p[p[i]] else p[i]
  in if p=p' then d else repeat(p',d)
```

Here, algorithm *remove_incompleteness* leaves the program unchanged, because rule 1 is not applicable. The maximal nesting depth of the program is 1, hence from the output measures it is sufficient to analyze the *lg* of the list. Then analysis w.r.t. *size* yields the same as *lg* in this case and is therefore omitted. However keep in mind, that an automatic analysis will also produce these functions. The program $\hat{\Pi}$ in algorithm *transform* will become:

```
fun time_rank(p:<nat>):<nat> = 13 +
  time_repeat(p,forall 0 <= i < lg(p) do in parallel if p[i]=i then 0 else 1)

fun time_repeat(p:<nat>,dist:<nat>):<nat> =
  if p = forall 0 <= i < lg(p) do in parallel
   if p[i] <> p[p[i]] then p[p[i]] else p[i]
  then 53
  else 55 +
       time_repeat(forall 0 <= i < lg(p) do in parallel
                      if p[i] <> p[p[i]] then p[p[i]] else p[i],
                   forall 0 <= i < lg(p) do in parallel
                      if p[i] <> p[p[i]] then dist[i] + dist[p[i]]
                                         else dist[i])

fun length_rank(p:<nat>):nat =
  length_repeat(p,forall 0 <= i < lg(p) do in parallel if p[i]=i then 0 else 1)
```

```
fun length_repeat(p:<nat>,dist:<nat>):nat =
  if p = forall 0 <= i < lg(p) do in parallel
   if p[i] <> p[p[i]] then p[p[i]] else p[i]
  then lg(p)
  else length_repeat(forall 0 <= i < lg(p) do in parallel
                       if p[i] <> p[p[i]] then p[p[i]] else p[i],
                     forall 0 <= i < lg(p) do in parallel
       if p[i] <> p[p[i]] then dist[i] + dist[p[i]] else p[i])
```

Rule 6 is not applicable to this program. Algorithm *irrelevant_positions* applied to the above program yields {time_repeat/2, length_repeat/2}. Again rule 9 is not applicable. Hence the normalized program Π'' will be:

```
fun time_rank(p:<nat>):<nat> = 13 + time_repeat(p)

fun time_repeat(p:<nat>):<nat> =
  if p = forall 0 <= i < lg(p) do in parallel
            if p[i] <> p[p[i]] then p[p[i]] else p[i]
  then 53
  else 55 + time_repeat(forall 0 <= i < lg(p) do in parallel
                          if p[i] <> p[p[i]] then p[p[i]] else p[i])

fun length_rank(p:<nat>):nat = length_repeat(p)

fun length_repeat(p:<nat>):nat =
  if p = forall 0 <= i < lg(p) do in parallel
           if p[i] <> p[p[i]] then p[p[i]] else p[i]
  then lg(p)
  else length_repeat(forall 0 <= i < lg(p) do in parallel
                       if p[i] <> p[p[i]] then p[p[i]] else p[i])
```

In algorithm *symbolic_evaluation* the first and third equation are directly in E by (1). For the second and fourth function, the property of line (11) is satisfied (i.e. the condition contains term p[i] and i is not constant). Hence the equation system E will be:

```
time_rank(p) = 13 + time_repeat(p)

time_repeat(p) =
  if p = forall 0 <= i < lg(p) do in parallel
           if p[i] <> p[p[i]] then p[p[i]] else p[i]
  then 53
  else 55 + time_repeat(forall 0 <= i < lg(p) do in parallel
                          if p[i] <> p[p[i]] then p[p[i]] else p[i])

length_rank(p) = length_repeat(p)
```

```
length_repeat(p) =
  if p = forall 0 <= i < lg(p) do in parallel
           if p[i] <> p[p[i]] then p[p[i]] else p[i]
  then lg(p)
  else length_repeat(forall 0 <= i < lg(p) do in parallel
                       if p[i] <> p[p[i]] then p[p[i]] else p[i])
```

Algorithm *derive_mappings* applied to these equation yields the set of mappings

$$M = \{\texttt{time\_rank}/1 = lg, \texttt{time\_repeat}/1 = lg, \texttt{length\_rank}/1 = lg, \texttt{length\_repeat}/1 = lg\}$$

With these mappings algorithm *derive_recurrences* derives:

```
time_rank(p,n) = 13 + time_repeat(p,n)

time_repeat(p,n) =
  if p = forall 0 <= i < n do in parallel
   if p[i] <> p[p[i]] then p[p[i]] else p[i]
  then 53
  else 55 + time_repeat(forall 0 <= i < n do in parallel
                          if p[i] <> p[p[i]] then p[p[i]] else p[i],n)

length_rank(p,n) = length_repeat(p,n)

length_repeat(p,n) =
  if p = forall 0 <= i < n do in parallel
   if p[i] <> p[p[i]] then p[p[i]] else p[i]
  then n
  else length_repeat(forall 0 <= i < n do in parallel
                       if p[i] <> p[p[i]] then p[p[i]] else p[i],n)
```

The set of irrelevant positions of these equations is empty and hence this is the final recurrence system. In this case this system is difficult to solve, because it is unclear whether the conditional recurrence `time_repeat` and `length_repeat` have a solution. Here we must get an interactive support of the user in order to solve these two recurrences. They are well defined because initially the vector `p` has the above discussed structure. In general it is easy to see, that these recurrences may for certain `p` be undefined [Zim90b]. The user has to add the general observation, that the length of the longest chain is halfed, it is initially lg(p), and the terminating case is equivalent to the fact that the length of the longest chain in `p` is 1. Thus he has to transform the recurrences into:

```
time_rank(p,n) = 13 + time_repeat(p,n,n)

time_repeat(p,n,1) = 53
time_repeat(p,n,m) =
  55 + time_repeat(forall 0 <= i < n do in parallel
                     if p[i] <> p[p[i]] then p[p[i]] else p[i],n,m/2)
```

```
length_rank(p,n) = length_repeat(p,n,n)

length_repeat(p,n,1) = n
length_repeat(p,n,m) =
  length_repeat(forall 0 <= i < n do in parallel
                  if p[i] <> p[p[i]] then p[p[i]] else p[i],n,m/2)
```

Now, algorithm *irrelevant_positions* can be applied again yielding that the positions `time_rank/1`, `time_repeat/1`, `length_rank/1`, and `length_repeat/1` are irrelevant. Thus the recurrence system

```
time_rank(n) = 13 + time_repeat(n,n)

time_repeat(n,1) = 53
time_repeat(n,m) = 55 + time_repeat(n,m/2)

length_rank(n) = length_repeat(n,n)

length_repeat(n,1) = n
length_repeat(n,m) = length_repeat(n,m/2)
```

has to be solved. By an enhancement of Maples `rsolve` this recurrence system yields the solution:

Semi-Automatic Theorem 3.34 (Complexity of `rank`) *Algorithm* `rank(p)` *has time complexity*

$$\texttt{time\_rank}(n) = 66 + 55\ \log_2 n$$

and its output length is

$$\texttt{length\_rank}(n) = n$$

where $n = lg(\texttt{p})$

Even it this not a good example for the *automatic* complexity analysis, the *semi-automatic* way, i.e. the automatic derivation of the recurrences is still possible. It is easier for the user to support the solution of the recurrences than to give this information already at the beginning. Probably at this place some heuristics play an important role to determine the number of iterations. For example the heuristic that when `p[i]` is set to `p[p[..p[i]..]]` (`p` is iterated c times, where c is constant), the system could propose the user with the hint that a chain of length m is reduced to length m/c. Providing more examples could suggest heuristics to support the user in solving recurrences, if the algebra system is not powerful to solve it automatically. The difficulty in this example is the use of vector access for addressing elements in a vector (i.e. it contains expressions like `p[p[i]]` and `dist[p[i]]`. Hence, the communication is *data-dependent*[3]. In most of algorithms with *data-dependent* communications, it is difficult to solve the obtained recurrence system automatically.

[3]This notion was due to Klaus-Jörn Lange, private communication

They have usually a similar flavour than the recurrence system in this example. When a program does not contain expressions like `t[s[i]]` then the communication is independent of the contents of the vector. In this case we speak about ***data-independent*** communication. Algorithms with data-independent communications are usually easier to analyze automatically. The next example is a program with data-independent communication.

3.2.2 Prefix Sums

This technique is also a standard design principle for parallel algorithms. The task is, given an input vector $\langle a_0, \ldots, a_{n-1} \rangle$, compute a vector $\langle s_0, \ldots, s_{n-1} \rangle$ where $s_i = a_0 + \cdots + a_i$. The idea is to compute in parallel the sum $a_i + a_{i+1}$ for all even i, then apply the function recursively to this vector, and adjust the result, i.e. all s_i where i is odd can be found on position $(i-1)/2$ in the result of the recursive call. If i is odd then the value a_{i+1} has to be subtracted from the value at position $i/2$ of the result of the recursive call. For simplicity we assume that the length of a is an integral power of 2. For a more complete discussion see [GR88, Akl89].

```
fun prefix_sum(a:<nat>):<nat> =
  if mt(a) then <>
  else if mt(tl(a)) then a
  else let n = lg(a)/2 in
       let a' = forall 0 <=i < n do in parallel a[2*i] + a[2*i+1] in
       let s' = prefix_sum(a') in
       forall 0 <= i < 2*n do in parallel
          if i mod 2 = 0 then s'[i/2] - a[i+1] else s'[(i-1)/2]
```

Algorithm *remove_incompleteness* does not change this program, because rule 1 is not applicable. All vectors involved are of lenght 1, hence the output length and output size coincide, and the output level length for $k \geq 2$ need not to be computed. By algorithm *translate*, $\hat{\Pi}$ will be the following program:

```
fun time_prefix_sum(a:<nat>):nat =
  if mt(a) then 4
  else if mt(tl(a)) then 8
  else 55 + time_prefix_sum(forall 0 <= i < lg(a)/2 do in parallel a[2*i] + a[2*i+1])

fun length_prefix_sum(a:<nat>):nat =
  if mt(a) then 0
  else if mt(tl(a)) then lg(a)
  else lg(a)
```

The algorithm *normalize* leaves this program unchanged, because rule 6 is not applicable, there are no irrelevant argument positions, and rule 9 is also not applicable. Algorithm *symbolic_evaluation* transforms the above program into the following set of equations:

```
time_prefix_sum(<>) = 4
time_prefix_sum(<c>) = 8
time_prefix_sum(cons(c1,cons(c2,a))) =
  55 + time_prefix_sum(forall 0 <= i < lg(a)/2+1 do in parallel
                         cons(c1,cons(c2,a))[2*i] + cons(c1,cons(c2,a))[2*i + 1])
```

```
length_prefix_sum(<>) = 0
length_prefix_sum(<c>) = 1
length_prefix_sum(cons(c1,cons(c2,a))) = lg(a) + 2
```

Algorithm *derive_mappings* yields `time_prefix_sums/1` = *lg* and `length_prefix_sum/1` = *lg*. Thus *derive_recurrences* transforms the above equations into

```
time_prefix_sum(<>,0) = 4
time_prefix_sum(<c>,1) = 8
time_prefix_sum(cons(c1,cons(c2,a)),n+2) =
  55 + time_prefix_sum(forall 0 <= i < n/2+1 do in parallel
                         cons(c1,cons(c2,a))[2*i] + cons(c1,cons(c2,a))[2*i + 1],n/2+1)

length_prefix_sum(<>,0) = 0
length_prefix_sum(<c>,1) = 1
length_prefix_sum(cons(c1,cons(c2,a)),n+2) = n + 2
```

In this system the argument positions `time_prefix_sum/1` and `length_prefix_sum/1` are irrelevant. Thus after their removal, the output of algorithm *transform* is the following recurrence system:

```
time_prefix_sum(0) = 4
time_prefix_sum(1) = 8
time_prefix_sum(n+2) = 55 + time_prefix_sum(n/2+1)

length_prefix_sum(0) = 0
length_prefix_sum(1) = 1
length_prefix_sum(n+2) = n + 2
```

This system can be solved by using standard methods (as e.g. implemented in Maple), and we obtain

Automatic Theorem 3.35 (Complexity of Prefix Sums) *The time complexity of the program* `prefix_sum(a)` *is:*

$$\texttt{time\_prefix\_sum}(n) = \begin{cases} 4 & \text{if } n = 0 \\ 55\ \log_2(n) + 8 & \text{otherwise} \end{cases}$$

and the output length of `prefix_sum(a)` *is:*

$$\texttt{length\_prefix\_sum}(n) = n$$

where $n = lg(\texttt{a})$.

Thus this program can be analyzed completely automatic. This program uses also n processors if its input is of length n. It is also possible to analyze an adaptive version of the prefix sums, i.e. the number of processors allowed to use is an additional parameter of this function. Here, first sums of small pieces (of size n/p) are computed sequentialy (in parallel using p processors). Then the prefix

sum of this reduced vector can be computed by the above algorithm (because then prefix_sum needs only p processors). Finally an adjustment has to be done for the n/p pieces. This can also be done sequentialy using only p processors to evaluate each piece of n/p in parallel. We assume here again for simplicity that both p and n are integral powers of 2.

```
fun adapt_prefix_sum(a:<nat>,p:nat):<nat> =
  let n = lg(a) in
  let a1 = forall 0 <= i < p do in parallel sum(a,i*n/p,n/p) in
  let s = prefix_sum(a1) in
  let a2 = forall 0 <= i < p do in parallel adjust(a,s,i,i*n/p,n/p) in
  flat(a2)

fun sum(a:<nat>,l:nat,m:nat):nat =
  if m=0 then 0
  else a[l] + sum(a,l+1,m-1)

fun adjust(a:<nat>,s:<nat>,i:nat,l:nat,m:nat):<nat> =
  if m=0 then <>
  else if m=1 then <s[i]>
  else let k = adjust(a,s,i,l+1,m-1) in
       cons(hd(k)-a[l],k)

fun flat(a:<<nat>>) : <nat>
  if mt(a) then <>
  else if mt(tl(a)) then hd(a)
  else flat(forall 0 <= i < lg(a)/2 do a[2*i] o a[2*i+1])
```

The function prefix_sum is as above. Rule 1 is not applicable. Hence *remove_incompleteness* leaves the program unchanged. Observe that for each function only its output length makes sense, because the are just plain vectors. For function sum none of the output measures need to be analyzed, because it is already of type nat. Thus the program $\hat{\Pi}$ is:

```
fun time_adapt_prefix_sum(a:<nat>,p:nat):nat =
  42 + max(0<=i<p,time_sum(a,i*lg(a)/p,lg(a)/p))
     + time_prefix_sum(forall 0 <= i < p do in parallel sum(a,i*lg(a)/p,lg(a)/p))
     + max(0<=i<p,time_adjust(a,prefix_sum(forall 0 <= i < p do in parallel
                                          sum(a,i*lg(a)/p,lg(a)/p)),i,i*lg(a)/p,lg(a)/p))
     + time_flat(forall 0 <= i < p do in parallel
                 adjust(a,prefix_sum(forall 0 <= i < p do in parallel
                                     sum(a,i*lg(a)/p,lg(a)/p)),i,i*lg(a)/p,lg(a)/p))

fun time_sum(a:<nat>,l:nat,m:nat):nat =
  if m=0 then 5
  else 16 + time_sum(a,l+1,m-1)

fun time_adjust(a:<nat>,s:<nat>,i:nat,l:nat,m:nat):nat =
  if m=0 then 5
  else if m=1 then 11
  else 28 + time_adjust(a,s,i,l+1,m-1)

fun time_flat(a:<<nat>>):nat =
  if mt(a) then 4
```

```
  else if mt(tl(a)) then 9
  else 26 + time_flat(forall 0 <= i < lg(a)/2 do a[2*i] o a[2*i+1])

fun length_adapt_prefix_sum(a:<nat>,p:nat):nat =
  length_flat(forall 0 <= i < p do in parallel
                adjust(a,prefix_sum(forall 0 <= i < p do in parallel
                                      sum(a,i*lg(a)/p,lg(a)/p)),i,i*lg(a)/p,lg(a)/p))

fun length_adjust(a:<nat>,s:<nat>,i:nat,l:nat,m:nat):nat =
  if m=0 then 0
  else if m=1 then 1
  else 1 + length_adjust(a,s,i,l+1,m-1)

fun length_flat(a:<<nat>>):nat =
  if mt(a) then 0
  else if mt(tl(a)) then lg(hd(a))
  else length_flat(forall 0 <= i < lg(a)/2 do a[2*i] o a[2*i+1])
```

The functions `time_prefix_sum` and `length_prefix_sum` are omitted, because these are already discussed above. Below, we do not explicitely mention the results of the particular step arising form `prefix_sum`. Now for the execution of algorithm keep in mind that rule 6 is not applicable, that algorithm *irrelevant_positions* delivers the irrelevant positions `time_adjust`/i and `length_adjust`/i for $i = 1, 2, 3, 4$, and `time_sum`/i for $i = 1, 2$. Again rule 9 is again not applicable. Hence after *normalize* the program Π'' is:

```
fun time_adapt_prefix_sum(a:<nat>,p:nat):nat =
  42 + time_sum(lg(a)/p)) + time_adjust(lg(a)/p))
     + time_prefix_sum(forall 0 <= i < p do in parallel sum(a,i*lg(a)/p,lg(a)/p))
     + time_flat(forall 0 <= i < p do in parallel
                   adjust(a,prefix_sum(forall 0 <= i < p do in parallel
                                         sum(a,i*lg(a)/p,lg(a)/p)),i,i*lg(a)/p,lg(a)/p))

fun time_sum(m:nat):nat =
  if m=0 then 5
  else 16 + time_sum(m-1)

fun time_adjust(m:nat):nat =
  if m=0 then 5
  else if m=1 then 11
  else 28 + time_adjust(m-1)

fun time_flat(a:<<nat>>):nat =
  if mt(a) then 4
  else if mt(tl(a)) then 9
  else 26 + time_flat(forall 0 <= i < lg(a)/2 do a[2*i] o a[2*i+1])

fun length_adapt_prefix_sum(a:<nat>,p:nat):nat =
  length_flat(forall 0 <= i < p do in parallel
                adjust(a,prefix_sum(forall 0 <= i < p do in parallel
                                      sum(a,i*lg(a)/p,lg(a)/p)),i,i*lg(a)/p,lg(a)/p))

fun length_adjust(m:nat):nat =
  if m=0 then 0
  else if m=1 then 1
  else 1 + length_adjust(m-1)

fun length_flat(a:<<nat>>):nat =
  if mt(a) then 0
```

```
  else if mt(tl(a)) then lg(hd(a))
  else length_flat(forall 0 <= i < lg(a)/2 do a[2*i] o a[2*i+1])
```

An application of algorithm *symbolic_evaluation* onto this program yields:

```
time_adapt_prefix_sum(a,p) = 42 + time_sum(lg(a)/p)) + time_adjust(lg(a)/p))
     + time_prefix_sum(forall 0 <= i < p do in parallel sum(a,i*lg(a)/p,lg(a)/p))
     + time_flat(forall 0 <= i < p do in parallel
                 adjust(a,prefix_sum(forall 0 <= i < p do in parallel
                                     sum(a,i*lg(a)/p,lg(a)/p)),i,i*lg(a)/p,lg(a)/p))

time_sum(0) = 5
time_sum(m+1) = 16 + time_sum(m)

time_adjust(0) = 4
time_adjust(1) = 11
time_adjust(m+2) = 28 + time_adjust(m+1)

time_flat(<>) = 4
time_flat(<c>) = 9
time_flat(cons(c1,cons(c2,a))) =
  26 + time_flat(forall 0 <= i < lg(a)/2+1 do cons(c1,cons(c2,a))[2*i] o cons(c1,cons(c2,a))[2*i+1])

length_adapt_prefix_sum(a,p) =
  length_flat(forall 0 <= i < p do in parallel
                 adjust(a,prefix_sum(forall 0 <= i < p do in parallel
                                     sum(a,i*lg(a)/p,lg(a)/p)),i,i*lg(a)/p,lg(a)/p))

length_adjust(0) = 0
length_adjust(1) = 1
length_adjust(m+2) = 1 + length_adjust(m+1)

length_flat(<>) = 0
length_flat(<c>) = lg(c)
length_flat(cons(c1,cons(c2,a))) =
  length_flat(forall 0 <= i < lg(a)/2+1 do cons(c1,cons(c2,a))[2*i] o cons(c1,cons(c2,a))[2*i+1])
```

Now algorithm *derive_mappings* delivers with the exception for `length_flat/1` for each argument position *lg*, while for the other it delivers ll_2 which is equivalent to *sz*. Thus algorithm *create_recurrences* delivers the following recurrence system:

```
time_adapt_prefix_sum(n,p) = 42 + time_sum(n/p) + time_adjust(n/p)
      + time_prefix_sum(p) + time_flat(p)

time_sum(0) = 5
time_sum(m+1) = 16 + time_sum(m)

time_adjust(0) = 4
time_adjust(1) = 11
time_adjust(m+2) = 28 + time_adjust(m+1)

time_flat(0) = 4
time_flat(1) = 9
time_flat(n+2) = 26 + time_flat(n/2+1)
```

```
length_adapt_prefix_sum(n,p) = length_flat(n,p*length_adjust(n/p))

length_adjust(0) = 0
length_adjust(1) = 1
length_adjust(m+2) = 1 + length_adjust(m+1)

length_flat(0,0) = 0
length_flat(1,n) = n
length_flat(n+2,2+n1 + n2 ) = length_flat(n/2+1,2+n1 + n2)
```

where the last equation was obtained by simplfiying the sum arising from the parallel statement. It is immediate to obtain the following solutions via standard methods:

$$\begin{aligned}
\texttt{time_sum}(m) &= 5 + 16\ m \\
\texttt{time_adjust}(m) &= 28\ m - 17 + 21\ 0^m \\
\texttt{time_flat}(m) &= \begin{cases} 4 & \text{if } m = 0 \\ 26\ \log_2(m) + 9 & \text{otherwise} \end{cases} \\
\texttt{length_adjust}(m) &= m \\
\texttt{length_flat}(n, m) &= m
\end{aligned}$$

Together with the previous results we obtain therefore automatically:

Automatic Theorem 3.36 (Complexity of Adaptive Prefix Sums) *The time complexity of algorithm* `adapt_prefix_sum(a,p)` *is:*

$$\texttt{time\_adaptive\_prefix\_sum}(n, p) = 44\ \frac{n}{p} + 26\ \log_2(n/p) + 55\ \log_2(p) + 33 + 21\ 0^{n-2p}$$

and its output length is:

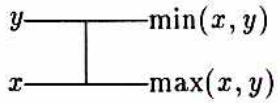
$$\texttt{length\_adaptive\_prefix\_sum}(n, p) = n$$

where $n = lg(a)$.

Chapter 4

Batchers Sorting Algorithms

Two simple (an in practice the most efficient) parallel sorting algorithms are analyzed. These algorithms were first introduced by Batcher [Bat68]. Both algorithms are discussed in [GR88], the odd-even-sort is also discussed in [Akl89], while the second algortihm bitonic sort is also discussed in [KR90]. Both bitonic-sort and odd-even sort are based on merging algorithms. These algorithms using n processors and time $O(\log^2 n)$ to sort a vector of n elements. Hence they are not optimal. Optimal algorithms are for example the AKS sorting network [AKS83] and Cole's algorithm [Col88]. Both of these algorithms are also discussed in [GR88]. The AKS sorting has huge constants, so that it cannot be used for practical applications. Cole's algorithm seems more practical, but its implementiation would require a huge program, so that we do not discuss it here. We picturize sorting networks by a compare exchange module:



4.1 Odd-Even-Sorting

The basic idea behind this sorting algorithm is the following merge algorithm. If the two vectors to be merged have length one then the result is an ordered vector of length containing the two elements of the input vector. If the input vectors have length n split both of them into two vectors of length $n/2$ consisting of elements on its odd and even positions, respectively. Then, recursively merge in parallel the two vectors obtained from the odd positions (this vector is called *odd*) and the two vectors obtained from the even positions (this vector is called *even*). Then the smallest element is the smallest element of the even positions, and the largest element is the largest element of the odd positions. Then for the odd positions i of the odd vector they need only to be compared with position $i-1$ in the even vector. In figure 4.1 an odd-even sorting network is picturized together with an example. Positions $0-3$ and $4-7$ represent the two vectors to be merged.

The program in PARFL is as follows:

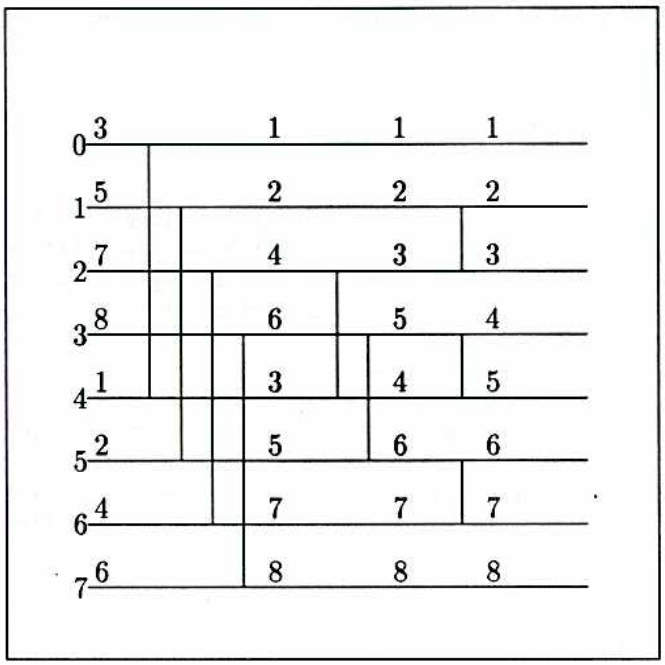


Figure 4.1: Odd-Even Merge for Eight Elements

```
fun odd_even_sort(s:<nat>):<nat> =
  if mt(s) then <>
  else if mt(tl(s)) then s
  else let n = lg(s) in
       let s1 = forall 0 <= i < n/2 do in parallel s[i] in
       let s2 = forall 0 <= i < n/2 do in parallel s[n/2+i] in
       let ss = <s1,s2> in
       let ss' = forall 0 <= i < 2 do in parallel odd_even_sort(ss[i]) in
       odd_even_merge(ss'[0],ss'[1])

fun odd_even_merge(s:<nat>,s':<nat>):<nat> =
  if lg(s)=1 then <min(hd(s),hd(s')),max(hd(s),hd(s'))>
  else let n = lg(s) in
       let e = forall 0 <= i < n/2 do in parallel s[2*i] in
       let e' = forall 0 <= i < n/2 do in parallel s'[2*i] in
       let o = forall 0 <= i < n/2 do in parallel s[2*i+1] in
       let o' = forall 0 <= i < n/2 do in parallel s'[2*i+1] in
       let r = <e,o> in
       let r' = <e',o'> in
       let ss = forall 0 <= i < 2 do in parallel odd_even_merge(r[i],r'[i]) in
       forall 0 <= i < 2*n do in parallel
         if i=0 then ss[0][0]
         else if i=2*n-1 then ss[1][n-1]
         else if i mod 2 = 1 then min(ss[0][(i+1)/2],ss[1][(i-1)/2])
         else max(ss[0][i/2],ss[1][i/2-1])
```

Algorithm *remove_incompleteness* leaves this program unchanged, because rule 1 is not applicable. The program uses just plain vectors, thus the only measure on the output is *lg* (and equals therefore to *sz*). Hence, the program $\hat{\Pi}$ will be

```
fun time_odd_even_sort(s:<nat>):nat =
  if mt(s) then 4
  else if mt(tl(s)) then 8
  else 54 + max(time_odd_even_sort(forall 0 <= i < lg(s)/2 do in parallel s[i]),
                time_odd_even_sort(forall 0 <= i < lg(s)/2 do in parallel s[lg(s)/2+i]))
         + time_odd_even_merge(odd_even_sort(forall 0 <= i < lg(s)/2 do in parallel s[i]),
                               odd_even_sort(forall 0 <= i < lg(s)/2 do in parallel s[lg(s)/2+i]))

fun time_odd_even_merge(s:<nat>,s':<nat>):nat =
  if lg(s)=1 then 21
  else 124 + max(time_odd_even_merge(forall 0 <= i < lg(s)/2 do in parallel s[2*i],
                                     forall 0 <= i < lg(s)/2 do in parallel s'[2*i]),
                 time_odd_even_merge(forall 0 <= i < lg(s)/2 do in parallel s[2*i+1],
                                     forall 0 <= i < lg(s)/2 do in parallel s'[2*i]+1))

fun length_odd_even_sort(s:<nat>):nat =
  if mt(s) then 0
  else if mt(tl(s)) then lg(s)
  else length_odd_even_merge(odd_even_sort(forall 0 <= i < lg(s)/2 do in parallel s[i]),
                             odd_even_sort(forall 0 <= i < lg(s)/2 do in parallel s[lg(s)/2+i]))

fun length_odd_even_merge(s:<nat>,s':<nat>):nat = if lg(s) = 1 then 2 else 2*lg(s)
```

Rule 6 is not applicable. The irrelevant positions determined by algorithm *irrelevant_position* are `time_odd_even_merge/2` and `length_odd_even_merge/2`. Rule 9 is not applicable. Therefore algorithm *normalize* outputs the following program:

```
fun time_odd_even_sort(s:<nat>):nat =
  if mt(s) then 4
  else if mt(tl(s)) then 8
  else 54 + max(time_odd_even_sort(forall 0 <= i < lg(s)/2 do in parallel s[i]),
                time_odd_even_sort(forall 0 <= i < lg(s)/2 do in parallel s[lg(s)/2+i]))
         + time_odd_even_merge(odd_even_sort(forall 0 <= i < lg(s)/2 do in parallel s[i]))

fun time_odd_even_merge(s:<nat>):nat =
  if lg(s)=1 then 21
  else 124 + max(time_odd_even_merge(forall 0 <= i < lg(s)/2 do in parallel s[2*i])
                 time_odd_even_merge(forall 0 <= i < lg(s)/2 do in parallel s[2*i+1]))

fun length_odd_even_sort(s:<nat>):nat =
  if mt(s) then 0
  else if mt(tl(s)) then lg(s)
  else length_odd_even_merge(odd_even_sort(forall 0 <= i < lg(s)/2 do in parallel s[i]))

fun length_odd_even_merge(s:<nat>):nat = if lg(s) = 1 then 2 else 2*lg(s)
```

Algorithm *symbolic_evaluation* delivers then the following systems of equations:

```
time_odd_even_sort(<>) = 4
time_odd_even_sort(<c>) = 8
time_odd_even_sort(cons(c1,cons(c2,s))) =
   54 + max(time_odd_even_sort(forall 0 <= i < lg(s)/2 + 1 do in parallel cons(c1,cons(c2,s))[i]),
            time_odd_even_sort(forall 0 <= i < lg(s)/2 + 1 do in parallel s[lg(s)/2+i]))
```

```
      + time_odd_even_merge(odd_even_sort(forall 0 <= i < lg(s)/2 + 1 do in parallel
                                          cons(c1,cons(c2,s))[i]))

time_odd_even_merge(<c>) = 21
time_odd_even_merge(cons(c1,cons(c2,s))) =
  124 + max(time_odd_even_merge(forall 0 <= i < lg(s)/2+1 do in parallel
                                cons(c1,cons(c2,s))[2*i])
            time_odd_even_merge(forall 0 <= i < lg(s)/2+1 do in parallel
                                cons(c1,cons(c2,s))[2*i+1]))

length_odd_even_sort(<>) = 0
length_odd_even_sort(<c>) = 1
length_odd_even_sort(cons(c1,cons(c2,s))) =
  length_odd_even_merge(odd_even_sort(forall 0 <= i < lg(s)/2+1 do in parallel
                                      cons(c1,cons(c2,s))[i]))

length_odd_even_merge(<c>) = 2
length_odd_even_merge(cons(c1,cons(c2,s))) = 2*lg(s) + 4
```

Algorithm *derive_mappings* gives for each argument position the mapping *lg*. Thus algorithm *create_recurrences* outputs:

```
time_odd_even_sort(0) = 4
time_odd_even_sort(1) = 8
time_odd_even_sort(n+2) =
   54 + time_odd_even_sort(n/2+1)
      + time_odd_even_merge(length_odd_even_sort(n/2+1))

time_odd_even_merge(1) = 21
time_odd_even_merge(n+2) = 124 + time_odd_even_merge(n/2+1)

length_odd_even_sort(0) = 0
length_odd_even_sort(1) = 1
length_odd_even_sort(n+2) = length_odd_even_merge(length_odd_even_sort(n/2+1))

length_odd_even_merge(1) = 2
length_odd_even_merge(n+2) = 2*n + 4
```

Thus they have the solutions (obtained automatically by Computer Algebra):

$$
\begin{aligned}
\texttt{length\_odd\_even\_merge}(n) &= 2\ n \\
\texttt{length\_odd\_even\_sort}(n) &= n \\
\texttt{time\_odd\_even\_merge}(n) &= 124\ \log_2(n) + 21 \\
\texttt{time\_odd\_even\_sort}(n) &= \begin{cases} 0 & \text{if } n = 0 \\ 62\ \log_2^2(n) + 13\ \log_2(n) + 8 & \text{otherwise} \end{cases}
\end{aligned}
$$

Hence we have

Automatic Theorem 4.1 (Complexity of Odd-Even Sort) *The worst case time complexity of the execution of* `odd_even_sort(s)` *is:*

$$\texttt{time\_odd\_even\_sort}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 62\ \log_2^2(n) + 13\ \log_2(n) + 8 & \text{otherwise} \end{cases}$$

and its output length is at most

$$\texttt{length\_odd\_even\_sort}(n) = n$$

where $n = lg(s)$.

When we apply algorithm *transform* with measure *proc* i.e. counting the number of processors used by the program, then it outputs the system of recurrences:

```
proc_odd_even_sort(0) = 1
proc_odd_even_sort(1) = 1
proc_odd_even_sort(n+2) = max(n/2+1,2*proc_odd_even_sort(n/2+1),proc_odd_even_merge(n/2+1))

proc_odd_even_merge(1) = 1
proc_odd_even_merge(n+2) = max(2*n+4,2*proc_odd_even_merge(n/2+1))
```

These kind of recurrences can be solved, by solving it with each argument of the maximum operator seperately, and taking the maximum solution. Thus:

Automatic Theorem 4.2 (Processor Complexity of Odd-Even Merge) *The execution of program* `odd_even_merge(s)` *requires at most the following number of processors:*

$$proc_odd_even_sort(n) = \begin{cases} 1 & \text{if } n = 0 \\ n & \text{otherwise} \end{cases}$$

where $n = lg(s)$.

4.2 Bitonic Sorting

The merge step of bitonic sort has as input only one vector, where the first half and second half of the vector is already ordered. The first half is interleaved with the second half, and then the elements are pairwise compared with the second half. Then the process is recursively applied to the first and the second half on the resulting vector. For a sorting network for eight elements see figure 4.2. Observe here its regularity compared to odd-even-merge. The second half is presented in opposite order.

The following program does the task (for simplicity we assume that n is an integral power of two):

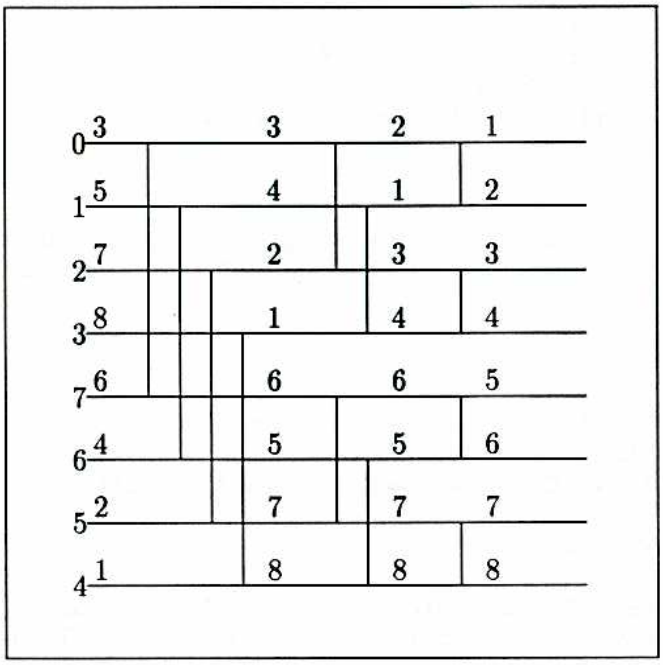


Figure 4.2: A Bitonic Sorting Network

```
fun bitonic_sort(v:<nat>):<nat> =
  if lg(v) <= 1 then v
  else let n = lg(v) in
       let v1 = forall 0 <= i < n/2 do in parallel v[i] in
       let v2 = forall n/2 <= i < n do in parallel v[i] in
       let w = <v1,v2> in
       let w1 = forall 0 <= i < 2 do in parallel bitonic_sort(w[i])
       let w2 = forall 0 <= i < n/2 do in parallel w[1][n/2-1-i]
       bitonic_merge(w1[0] o w2)

fun bitonic_merge(v:<nat>):<nat> =
  if lg(v) = 1 then v
  else let n = lg(v) in
       let w1 = forall 0 <= i < n/2 do in parallel min(v[i],v[n/2+i]) in
       let w2 = forall 0 <= i < n/2 do in parallel max(v[i],v[n/2+i]) in
       let w = <w1,w2> in
       let v' = forall 0 <= i < 2 do in parallel bitonic_merge(w[i]) in
       v'[0] o v'[1]
```

Again, rule 1 is not applicable and therefore algorithm *remove_incompleteness* leaves the program unchanged. Also the maximal output level of vector is 1. Hence it is sufficient to analyze just the output length. The program $\hat{\Pi}$ is then:

```
fun time_bitonic_sort(v:<nat>):nat =
  if lg(v) <= 1 then 6
  else 64 + max(time_bitonic_sort(forall 0 <= i < lg(v)/2 do in parallel v[i]),
```

```
            time_bitonic_sort(forall lg(v)/2 <= i < lg(v) do in parallel v[i]))
        + time_bitonic_merge(
            bitonic_sort(forall 0 <= i < lg(v)/2 do in parallel v[i]) o
            forall 0 <= i < lg(v)/2 do in parallel
              bitonic_sort(forall lg(v)/2 <= i < lg(v) do in parallel v[i])[lg(v)/2-1-i])

fun time_bitonic_merge(v:<nat>):nat =
  if lg(v)=1 then 5
  else 58 + max(time_bitonic_merge(forall 0 <= i < lg(v)/2 do in parallel min(v[i],v[n/2+i]),
              time_bitonic_merge(forall 0 <= i < lg(v)/2 do in parallel max(v[i],v[n/2+i]))

fun length_bitonic_sort(v:<nat>):nat =
  if lg(v)<=1 then lg(v) else
  length_bitonic_merge(
    bitonic_sort(forall 0 <= i < lg(v)/2 do in parallel v[i]) o
    forall 0 <= i < lg(v)/2 do in parallel
      bitonic_sort(forall lg(v)/2 <= i < lg(v) do in parallel v[i])[lg(v)/2-1-i])

fun length_bitonic_merge(v:<nat>):nat =
  if lg(v)=1 then lg(v) else
  length_bitonic_merge(forall 0 <= i < lg(v)/2 do in parallel min(v[i],v[n/2+i]) +
  length_bitonic_merge(forall 0 <= i < lg(v)/2 do in parallel max(v[i],v[n/2+i])
```

Rule 6 and rule 9 are not applicable. Also algorithm *irrelevant_positions* does not find an irrelevant argument position. Hence algorithm *normalize* leaves this program unchanged. Algorithm *symbolic_evaluation* yields then the following equations:

```
time_bitonic_sort(<>) = 6
time_bitonic_sort(<c>) = 6
time_bitonic_sort(cons(c1,cons(c2,v))) = 64 +
  max(time_bitonic_sort(forall 0 <= i < lg(v)/2+1 do in parallel cons(c1,cons(c2,v))[i]),
      time_bitonic_sort(forall lg(v)/2+1<=i<lg(v)+2 do in parallel cons(c1,cons(c2,v))[i]))
  + time_bitonic_merge(
      bitonic_sort(forall 0<=i<lg(v)/2+1 do in parallel cons(c1,cons(c2,v))[i]) o
      forall 0<=i<lg(v)/2 do in parallel
        bitonic_sort(forall lg(v)/2+1<=i<lg(v)+2 do in parallel
                       cons(c1,cons(c2,v))[i])[lg(v)/2-i])

time_bitonic_merge(<c>) = 5
time_bitonic_merge(cons(c1,cons(c2,v))) = 58 +
  max(time_bitonic_merge(forall 0<=i<lg(v)/2+1 do in parallel
                       min(cons(c1,cons(c2,v))[i],cons(c1,cons(c2,v))[n/2+i]),
      time_bitonic_merge(forall 0<=i<lg(v)/2 do in parallel
                       max(cons(c1,cons(c2,v))[i],cons(c1,cons(c2,v))[n/2+i]))

length_bitonic_sort(<>) = 0
length_bitonic_sort(<c>) = 1
length_bitonic_sort(cons(c1,cons(c2,v))) =
  length_bitonic_merge(
     bitonic_sort(forall 0 <= i < lg(v)/2+1 do in parallel cons(c1,cons(c2,v))[i]) o
     forall 0<=i<lg(v)/2+1 do in parallel
       bitonic_sort(forall lg(v)+1/2<=i<lg(v)+2 do in parallel cons(c1,cons(c2,v))[i])[lg(v)/2-i])

length_bitonic_merge(<c>) = 1
lenght_bitonic_merge(cons(c1,cons(c2,v))) =
  length_bitonic_merge(
    forall 0 <= i < lg(v)/2+1 do in parallel
      min(cons(c1,cons(c2,v))[i],cons(c1,cons(c2,v))[n/2+i]) +
```

```
length_bitonic_merge(
  forall 0 <= i < lg(v)/2+1 do in parallel
    max(cons(c1,cons(c2,v))[i],cons(c1,cons(c2,v))[n/2+i])))
```

Now, algorithm *derive_mappings* derives for each argument position the mapping *lg*. Hence algorithm *create_recurrences* outputs the following system of recurrences.

```
time_bitonic_sort(0) = 6
time_bitonic_sort(1) = 6
time_bitonic_sort(n+2) = 64 + time_bitonic_sort(n/2+1) +
  time_bitonic_merge(length_bitonic_sort(n/2+1) + n/2 + 1)

time_bitonic_merge(1) = 5
time_bitonic_merge(n+2) = 58 + time_bitonic_merge(n/2+1)

length_bitonic_sort(0) = 0
length_bitonic_sort(1) = 1
length_bitonic_sort(n+2) =
  length_bitonic_merge(length_bitonic_sort(n/2+1) + n/2 + 1)

length_bitonic_merge(1) = 1
length_bitonic_merge(n+2) = 2 * length_bitonic_merge(n/2+1)
```

This recurrence system can be solved automatically, because they are standard. Thus:

Automatic Theorem 4.3 (Complexity of Bitonic Sorting) *The time complexity of program* `bitonic_sort(v)` *is at most*

$$\texttt{time\_bitonic\_sort}(n) = \begin{cases} 6 & \text{if } n = 0 \\ 29\ \log_2^2(n) + 102\ \log_2 n + 6 & \text{otherwise} \end{cases}$$

and its output length is at most

$$\texttt{length\_bitonic\_sort}(n) = n$$

where $n = lg(v)$.

Chapter 5

Algorithms on Graphs

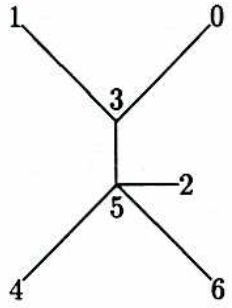
This chapter contains selected examples from parallel algorithms on graphs. Most of these algorithms are based on descriptions of [GR88] and [KR90]. Usually, algorithms on graphs can only be analyzed semi-automatically unless powerful heuristics are developed, because their communication is data dependent. Hence this chapter serves also as an open problem section to determine the difficulties in solving recurrences obtained from these algorithms.

5.1 Algorithms based on the Euler-Tour Technique

The Euler-Tour technique is based on finding an Euler tour on trees. In general an Euler tour on a directed graph is a closed path visiting each edge. When we consider trees, each tree edge is replaced by two antiparallel edges. The program is based on the algorithm in [KR90], and runs in parallel constant time using $O(n)$ processors. The origin of this technique is by Tarjan and Vishkin [TV85]. In [KR90], the following is assumed: first the tree is given by adjacency lists. Second there is for each edge a pointer to its antiparallel edge. A version of this technique not requiring such information is in [GR88]. Then the algorithm runs in constant time. In order to keep the spirit of this method, we represent a graph as follows: the adjacency lists are defined by two vectors `e` and `p` where `p[i]` points to the successor of `e[i]`. The adjacency lists are represented as in the pointer jumping example. They need not to be consequetive in `p`. Then the addresses of the first elements of each adjacency list are given by a vector `v`, where `v[i]` contains the index of the first element in the adjacency list of vertex `i`. Finally, the vector `cr` contains the addresses of the antiparallel edges, i.e. `cr[i]` contains the address of the antiparallel edge of `e[i]`. The basic idea of the algorithm is the following. First make the lists circular (this is easily achieved in constant time, because `e[cr[i]]` contains the source of an edge, by `v[e[cr[i]]]`, the address of the first element of the adjacency list containing `e[i]` can be found, and the end of an adjacency list is equivalent to the condition `p[i]=i`). Then an Euler tour can be obtained by setting the successor of edge (i,j) the the successor of edge (j,i) in the circular lists. The resulting list is a circular Euler tour. Hence the program is as follows:

```
fun euler_tour_on_trees(v:<nat>,e:<nat>,p:<nat>,cr:<nat>) = <nat x nat> x <nat> =
     let n = lg(v) in
     let circ = forall 0 <= i < 2*n-2 do in parallel
                   if p[i] = i then v[e[cr[i]]] else p[i]
  in (forall 0 <= i < 2*n-2 do in parallel (e[i],e[cr[i]]),
      forall 0 <= i < 2*n-2 do in parallel p[cr[i]])
```

The following example is from [GR88]. We translate it into our representation.



Let us assume the following representation:

	0	1	2	3	4	5	6
v	5	1	10	8	9	4	2

	0	1	2	3	4	5	6	7	8	9	10	11
e	3	3	5	1	2	3	6	5	0	5	5	4
p	11	1	2	7	0	5	6	7	3	9	10	6
cr	7	3	6	1	10	8	2	0	5	11	4	9

During the computation the following vectors are created.

	0	1	2	3	4	5	6	7	8	9	10	11
circ	11	1	2	7	0	5	4	8	3	9	10	6
result.1	(5,3)	(1,3)	(6,5)	(3,1)	(5,2)	(0,3)	(5,6)	(3,5)	(3,0)	(4,5)	(2,5)	(5,4)
result.2	8	7	4	1	10	3	2	11	5	6	0	9

The result represents the following circular list:

$(5,3) \rightarrow (3,0) \rightarrow (0,3) \rightarrow (3,1) \rightarrow (1,3) \rightarrow (3,5) \rightarrow (5,4) \rightarrow (4,5) \rightarrow (5,6) \rightarrow (6,5) \rightarrow (5,2) \rightarrow (2,5) \rightarrow (5,3)$

The analysis process yields automatically:

Automatic Theorem 5.1 (Complexity of Euler Tour on Trees) *The time complexity of program* `euler_tour_on_trees(v,e,p,cr)` *is at most*

$$\texttt{time\_euler\_tour\_on\_trees} = 57$$

its output length is at most:

$$\texttt{length\_euler\_tour\_on\_trees}(n) = 2\ n - 2$$

its output size is at most:

$$\texttt{size\_euler\_tour\_on\_trees}(n) = 6\ n - 6$$

and the number of processors used is at most

$$\texttt{proc\_euler\_tour\_on\_trees}(n) = 2\ n - 2$$

where $n = lg(v)$.

Now we turn to applications [GR88, KR90]. First, a tree can be rooted at vertex i if the circular list is broken at edge (i,j). The tour represents then a depth first traversal. Then it makes sense to determine the father relation. The father relation can be obtained as follows. First rank the the list representing an Euler-Tour. Then i is a father of j if the rank of edge (i,j) is greater than that of (j,i), because edge (i,j) comes then before edge (j,i) in the depth-first traversal of the tree. Consider the above example and suppose that the Euler-Tour is broken at edge $(5,3)$ (i.e. 5 is the root of the tree). For example the rank of edge $(5,3)$ is 11 while that of edge $(3,5)$ is 6. Hence vertex 5 is father of 3. The program is therefore as follows (`r` is the desired root):

```
fun father(v:<nat>,e:<nat>,p:<nat>,cr:<nat>,r:nat):<nat> =
  let tour = euler_tour_on_trees(v,e,p,cr) in
  let n = lg(v) in
  let p1 = forall 0 <= i < 2*n-2 do in parallel
     if tour.2[i] = v[r] then i else tour.2[i]
  in \* p1 is now a list starting at v[r] *\
     let rnk = rank(p1) in
     modify v[e[i]], 0 <= i < 2*n -2 to
                if rnk[i] > rnk[cr[i]] then e[cr[i]] else skip
     from v
```

A recurrence system is obtained automatically. The complexity of list ranking can only be obtained semi-automatically. We get then together with automatic theorem 5.1:

Semi-Automatic Theorem 5.2 *The execution time of program* `father(v,e,p,cr)` *is at most*

$$\texttt{time\_father}(n) = 55\ \log_2 n + 239$$

its output length is at most

$$\texttt{length\_father}(n) = n$$

and it needs at most

$$\texttt{proc\_father}(n) = 2 * n - 2$$

processors, where $n = lg(v)$.

Observe, that the above program is executable on a CREW PRAM, because in the modify statement no write conflict can occur.

The next application is to determine the number of descencands of each node. If r_1 is the rank of edge $(i, father(i))$ and r_2 is the edge of $(father(i), i)$ then $(r_2 - r_1 + 1)/2$ is the number of descendands of vertex i. For example the rank of edge $(3,5)$ is 6, that of $(5,3)$ is 11. Hence, the number of descendands is 3 (this number includes the vertex itsself). The correctness of this step follows also from the fact, that the euler-tour starting at r is a depth first traversal of the tree,

because then the difference represents the number of edges visited during the traversal, which is precisely one more than the double of number of proper descendands of node i. Thus the algorithm is similar to above. It is more efficient not to call function *father*, but using instead the equivalent test $rank(father(i), i) > rank(i, father(i))$.

```
fun nd(v:<nat>,e:<nat>,p:<nat>,cr:<nat>,r:nat):<nat> =
  let tour = euler_tour_on_trees(v,e,p,cr) in
  let n = lg(v) in
  let p1 = forall 0 <= i < 2*n-2 do in parallel
  in \* p1 is now a list starting at v[r] *\
     if tour.2[i] = v[r] then i else tour.2[i]
     let rnk = rank(p1) in
     modify v[e[i]], 0 <= i < 2*n -2 to
                if rnk[i] > rnk[cr[i]] then (rnk[i] - rnk[cr[i]] + 1)/2 else skip
     from v
```

This function runs for the same reason as `father` on a CREW PRAM. The analysis is because of the call of pointer jumping semi-automatic:

Semi-Automatic Theorem 5.3 *The execution time of program* `nd(v,e,p,cr)` *is at most*

$$\texttt{time\_nd}(n) = 55\ \log_2 n + 247$$

its output length is at most

$$\texttt{length\_nd}(n) = n$$

and it needs at most

$$\texttt{proc\_nd}(n) = 2 * n - 2$$

processors, where $n = lg(v)$.

It is also easy to determine a preorder and postorder numbering of the vertices. If just the edges from father to the sons are counted then their position yields an "inverse preorder" i.e. the preorder of vertex i is $n - k$ where k is the rank of the edge (j, i) where (j, i) is an edge from father to son. The program uses pointer jumping just with a different initialization, namely all father son edges are initialized with 1 and the others with 0.

```
fun pre(v:<nat>,e:<nat>,p:<nat>,cr:<nat>,r:nat):<nat> =
  let tour = euler_tour_on_trees(v,e,p,cr) in
  let n = lg(v) in
  let p1 = forall 0 <= i < 2*n-2 do in parallel
  in \* p1 is now a list starting at v[r] *\
     let rnk = rank(p1) in
     let fs = forall 0 <= i < 2*n-2 do in parallel
                 if rnk[i] > rnk[cr[i]] then 1 else 0
```

```
in \* fs[i] is 1 iff it is a father son edge *\
   let fsrnk = repeat(p,fs) in
   modify v[e[i]] 0 <= i < 2*n - 2 to
     if e[i] = r then 0 else if fs[i] = 1 then n - fsrnk[i] else skip
   from v
```

In the analysis of `repeat(p,d)` we derived in the previous chapter semi-automatically that the time complexity is $\texttt{time\_repeat}(n) = 55 \log_2 n + 53$ where $n = lg(p)$. Hence the complexity results are

Semi-Automatic Theorem 5.4 *The execution time of program* `pre(v,e,p,cr)` *is at most*

$$\texttt{time\_pre}(n) = 110\ \log_2 n + 365$$

its output length is at most

$$\texttt{length\_pre}(n) = n$$

and it needs at most

$$\texttt{proc\_pre}(n) = 2 * n - 2$$

processors, where $n = lg(v)$.

The post-order numbering can be obtained analogouly, weighting the son-father edges to 1 and the father-son edges to 0. Finally it can be determined whether node i is ancestor of node j. This is the case iff $pre(i) \leq pre(j) < pre(i) + nd(i)$. This program uses $\max(n^2, 2\ n - 2)$ processors.

```
fun anc(v:<nat>,e:<nat>,p:<nat>,cr:<nat>,r:nat):<<nat>> =
  let tour = euler_tour_on_trees(v,e,p,cr) in
  let n = lg(v) in
  let p1 = forall 0 <= i < 2*n-2 do in parallel
  in \* p1 is now a list starting at v[r] *\
     let rnk = rank(p1) in
     let fs = forall 0 <= i < 2*n-2 do in parallel
                 if rnk[i] > rnk[cr[i]] then 1 else 0
  in \* fs[i] is 1 iff it is a father son edge *\
     let fsrnk = repeat(p,fs) in
     let pre = modify v[e[i]] 0 <= i < 2*n - 2 to
                 if e[i] = r then 0 else if fs[i] = 1 then n - fsrnk[i] else skip

  in let nd = modify v[e[i]], 0 <= i < 2*n -2 to
                if fs[i]=1 then (rnk[i] - rnk[cr[i]] + 1)/2 else skip
              from v
  in forall 0 <= i < n do in parallel
       forall 0 <= j < n do in parallel
         pre[i] <= pre[j] and pre[j] < pre[i] + nd[i]
```

Hence, its complexities can also derived semi-automatically:

Semi-Automatic Theorem 5.5 *The execution time of program* `anc(v,e,p,cr)` *is at most*

$$\texttt{time\_anc}(n) = 110\ \log_2 n + 422$$

its output length is at most

$$\texttt{length\_anc}(n) = n$$

its output size is at most

$$\texttt{size\_anc}(n) = n^2$$

and it needs at most

$$\texttt{proc\_anc}(n) = \max(n^2, 2\ n - 2)$$

processors, where $n = lg(v)$.

If we use a heuristic for pointer-jumping (e.g. the heuristic discussed in the previous chapter) then these algorithms could be analyzed automatically.

5.2 All Pair Shortest Paths

The problem is described as follows: Given a weighted graph $G = (V, E, W)$, represented by a weight matrix W defined as follows:

$$W[i,j] = \begin{cases} w(i,j) & \text{if there is an edge with weight } w(i,j) \text{ in } G \\ n^2\ \max_{(i,j)\in E} w(i,j) & \text{otherwise} \end{cases}$$

where $n = |V|$. The output is a weight-matrix W' s.t. $W[i,j]$ is the shortest path between vertices i and j. The program below is based on the description of [GR88] and first published by Kucera [Kuc82].

```
fun shortest_path(w:<<nat>>):<<nat>> =
  let w' = it_shortest_paths(w) in
  let n = lg(w) in
  forall 0<=i<n do in parallel
    forall 0<=j<n do in parallel
      if i=j then 0 else w'[i][j]

fun it_shortest_path(w:<<nat>>):<<nat>> =
  let n=lg(w) in
  let w2 = forall 0 <= i < n do in parallel
             forall 0 <= j < n do in parallel
```

```
              forall 0 <= k < n do in parallel w[i][k] + w[k][j]
  in let w' = forall 0 <= i < n do in parallel
               forall 0 <= j < n do in parallel
                   vmin(cons(w[i][j],w2[i][j])
  in if w=w' then w
  else it_shortest_paths(w')

fun vmin(x:<nat>):nat =
  if mt(tl(x)) then hd(x)
  else vmin(forall 0 <= i < lg(x)/2 do in parallel min(x[2*i],x[2*i+1])
```

Now we want to analyze the time complexity of the function **shortest_path**. Rule 1 is not applicable. Therefore algorithm *remove_incompleteness* leaves the program unchanged. Now, the output length as well as the output size may important because each function (besides **vmin**) outputs vectors of vectors. We then the following program $\hat{\Pi}$:

```
fun time_shortest_path(w:<<nat>>):nat =
  22 + time_it_shortest_path(w)

fun time_it_shortest_path(w:<<nat>>):nat =
  if w = forall 0 <= i < lg(w) do in parallel
           forall 0 <= j < lg(w) do in parallel
             vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                 w[i][k] + w[k][j]))
  then 49 + time_vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                 w[i][k] + w[k][j]))
  else 50 + time_vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
       + time_it_shortest_path(
             forall 0 <= i < lg(w) do in parallel
               forall 0 <= j < lg(w) do in parallel
                 vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                     w[i][k] + w[k][j]))

fun time_vmin(x:<nat>):nat =
  if mt(tl(x)) then 6
  else 24 + time_vmin(forall 0 <= i < lg(x)/2 do in parallel min(x[2*i],x[2*i+1])

fun length_shortest_path(w:<<nat>>):nat = lg(w)

fun length_it_shortest_path(w:<<nat>>):<nat> =
  if w = forall 0 <= i < lg(w) do in parallel
           forall 0 <= j < lg(w) do in parallel
             vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                 w[i][k] + w[k][j]))
  then lg(w)
  else length_it_shortest_path(
             forall 0 <= i < lg(w) do in parallel
```

```
              forall 0 <= j < lg(w) do in parallel
                vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                    w[i][k] + w[k][j]))

fun size_shortest_path(w:<<nat>>):nat = lg(w) * lg(w)

fun size_it_shortest_path(w:<<nat>>):<nat> =
  if w = forall 0 <= i < lg(w) do in parallel
           forall 0 <= j < lg(w) do in parallel
             vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                 w[i][k] + w[k][j]))
  then sz(w)
  else size_it_shortest_path(
             forall 0 <= i < lg(w) do in parallel
               forall 0 <= j < lg(w) do in parallel
                 vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                    w[i][k] + w[k][j]))
```

Now, rule 6 is not applicable. Furthermore algorithm *irrelevant_position* outputs the empty set. Hence each argument position is relevant. Finally, rule 9 is also not applicable. Hence algorithm *normalize* leaves this program unchanged. Algorithm *symbolic_eval* leaves with the exception of `time_vmin` all functions unchanged, because the condition

```
w = forall 0 <= i < lg(w) do in parallel
      forall 0 <= j < lg(w) do in parallel
        vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
```

contains a function call. Thus the resulting equation system is now:

```
time_shortest_path(w) = 22 + time_it_shortest_path(w)

time_it_shortest_path(w) =
  if w = forall 0 <= i < lg(w) do in parallel
           forall 0 <= j < lg(w) do in parallel
             vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                 w[i][k] + w[k][j]))
  then 49 + time_vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                 w[i][k] + w[k][j]))
  else 50 + time_vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
       + time_it_shortest_path(
             forall 0 <= i < lg(w) do in parallel
               forall 0 <= j < lg(w) do in parallel
                 vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                    w[i][k] + w[k][j]))

time_vmin(<c>) =  6
time_vmin(cons(c1,cons(c2,x))) =
```

```
  24 + time_vmin(forall 0 <= i < lg(x)/2+1 do in parallel
                   min(cons(c1,cons(c2,x))[2*i],cons(c1,cons(c2,x))[2*i+1])

length_shortest_path(w) = lg(w)

length_it_shortest_path(w) =
  if w = forall 0 <= i < lg(w) do in parallel
           forall 0 <= j < lg(w) do in parallel
             vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                 w[i][k] + w[k][j]))
  then lg(w)
  else length_it_shortest_path(
             forall 0 <= i < lg(w) do in parallel
               forall 0 <= j < lg(w) do in parallel
                 vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                     w[i][k] + w[k][j]))

size_shortest_path(w) = lg(w) * lg(w)

size_it_shortest_path(w) =
  if w = forall 0 <= i < lg(w) do in parallel
           forall 0 <= j < lg(w) do in parallel
             vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                 w[i][k] + w[k][j]))
  then sz(w)
  else size_it_shortest_path(
             forall 0 <= i < lg(w) do in parallel
               forall 0 <= j < lg(w) do in parallel
                 vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
                                     w[i][k] + w[k][j]))
```

By algorithm *derive_mappings* all the mappings are *lg*, and in size_shortest_paths it is *sz*. In line (3) of algorithm *create_recurrence* additionaly, a parameter with *lg* is added. The algorithm *irrelevant_positions* discovers that the first argument position of time_it_shortest_path, length_it_shortest_path, size_it_shortest_path, and time_shortest_path is relevant. Thus the resulting recurrence system is:

```
time_shortest_path(w,n) = 22 + time_it_shortest_path(w,n)

time_it_shortest_path(w,n) =
  if w = forall 0 <= i < n do in parallel
           forall 0 <= j < n do in parallel
             vmin(cons(w[i][j],forall 0 <= k < n do in parallel
                                 w[i][k] + w[k][j]))
  then 49 + time_vmin(n+1)
  else 50 + time_vmin(n+1)
       + time_it_shortest_path(
```

```
            forall 0 <= i < n do in parallel
              forall 0 <= j < n do in parallel
                vmin(cons(w[i][j],forall 0 <= k < n do in parallel
                                    w[i][k] + w[k][j]),n)

time_vmin(1) =  6
time_vmin(n+2) = 24 + time_vmin(n/2+1)

length_shortest_path(n) = n

length_it_shortest_path(w,n) =
  if w = forall 0 <= i < n do in parallel
           forall 0 <= j < n do in parallel
             vmin(cons(w[i][j],forall 0 <= k < n do in parallel
                                 w[i][k] + w[k][j]))
  then n
  else length_it_shortest_path(
            forall 0 <= i < n do in parallel
              forall 0 <= j < n do in parallel
                vmin(cons(w[i][j],forall 0 <= k < n do in parallel
                                    w[i][k] + w[k][j]))

size_shortest_path(n) = n * n

size_it_shortest_path(w,m,n) =
  if w = forall 0 <= i < n do in parallel
           forall 0 <= j < n do in parallel
             vmin(cons(w[i][j],forall 0 <= k < n do in parallel
                                 w[i][k] + w[k][j]))
  then m
  else size_it_shortest_path(
            forall 0 <= i < n do in parallel
              forall 0 <= j < n do in parallel
                vmin(cons(w[i][j],forall 0 <= k < n do in parallel
                                    w[i][k] + w[k][j]),m,n)
```

For solving the recurrences for `x_it_shortest_path`, $\mathtt{x} \in \{\mathtt{size}, \mathtt{time}, \mathtt{length}\}$, we have now similar problems than in pointer jumping. Here we have operations on matrices, an addition $\oplus$ = min and a multiplication $\otimes$ = +. This operations on matrices are defined by classical matrix addition and multiplication. With these operations the effect of the change in the first argument positions of the functions `x_it_shortest_path` is described by $M \oplus M \otimes M$. Define now an operation $M \odot N = M \oplus M \otimes N$. If M is a $n \times n$ matrix, then $M^n = M^{n+1} = \cdots$ where the power is based on $\odot$. This property is easy to prove by induction. The change in the first argument position is from M to $M \odot M$. Hence $\lceil \log_2 n \rceil$ recursive calls are necessary. This can be described adding an additional argument position, initially it has the value $lg(\mathtt{w})$, and then at each recursive call its value is halfed. The condition

```
w = forall 0 <= i < lg(w) do in parallel
```

```
forall 0 <= j < lg(w) do in parallel
  vmin(cons(w[i][j],forall 0 <= k < lg(w) do in parallel
```

become now equivalent to the test wether this additional parameter is 1.

What was done by this definition? We defined a structure $(\Omega, \oplus, \otimes)$ which is algebraically a semiring. Then we have a measure m on this structure (here it was the dimension of the matrices). The next step was defining a binary operation $\odot$ based on $\oplus$ and $\otimes$ and showing that for a function f over natural numbers $\underbrace{M \odot \cdots \odot M}_{f(m(M)) \text{ times}}$ is idempotent w.r.t. $\odot$. Furthermore the operation $\odot$ is based on the program, such that the argument W changes to $W \odot W$, and the terminating condition is $W = W \odot W$. Then we were able to conclude that after $\log_2 f(m(M))$ iterations, the terminating condition become true. Hence adding a new parameter q representing $f(m(M))$ halfing this at each recursive step, initializing it with $f(m(M))$ and replacing the terminating condition by the equivalent condition $q = 1$ leads to a ordinary recurrence. We call such a structure a parallel *iteration scheme* and require the user to provide such a scheme to support data-dependent communication. Further research should obtain automatically from a program such iteration schemes.

Here we obtain then (after eliminating further irrelevant argument positions) the following recurrence system (which can then be solved by computer algebra systems).

```
time_shortest_path(n) = 22 + time_it_shortest_path(n,n)

time_it_shortest_path(n,1) = 49 + time_vmin(n+1)
time_it_shortest_path(n,q) = 50 + time_vmin(n+1) + time_it_shortest_path(n,q/2)

time_vmin(1) =  6
time_vmin(n+2) = 24 + time_vmin(n/2+1)

length_shortest_path(n) = n

length_it_shortest_path(n,1) = n
length_it_shortest_path(n,q) = length_it_shortest_path(n,q/2)

size_shortest_path(n) = n * n

size_it_shortest_path(m,1) = m
size_it_shortest_path(m,q) = size_it_shortest_path(m,q/2)
```

We get then

Semi-Automatic Theorem 5.6 *The time complexity of* `shortest_path(w)` *is:*

$$\texttt{time\_shortest\_path}(n) = 24\ \log_2(n+1)\log_2(n) + 24\ \log_2(n+1) + 50\log_2(n) + 71$$

or equivalently by using the Taylor development of $\log_2(n)$

$$\texttt{time\_shortest\_path}(n) = 24\ \log_2^2(n) + 74\log_2 n + 71 + O(\log^2 n/n)$$

where $n = lg(w)$.

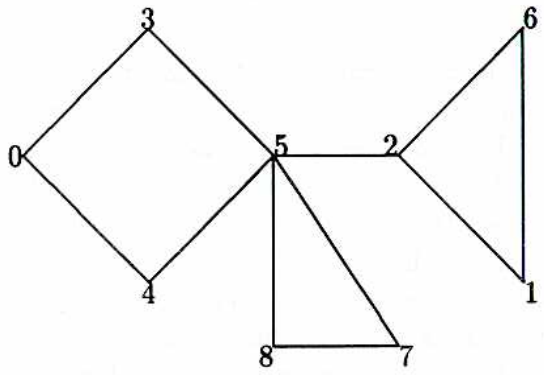
5.3 Finding Connected Components

We follow this section the algorithm described by Hirschberg, Chandra, and Sarvate [HCS79]. A discussion of this algorithm can also be found in [GR88]. Let $G = (V, E)$ be a graph. A *connected component* of G is maximal subgraph $C = (U, F)$ where $U \subseteq V$ and $F = \{(i,j) \in E | i,j \in U\}$ such that there is for each i and j a path between i and j in U. The idea behind the algorithm is component merging. Initially each vertex is assumed to be its own component. Then the following process is iterated: If there is q vertex in one component, and another vertex in another component, such that there is an edge between these two vertices, then merge these components into a single component. In [HCS79] a vector D is used to describe the components. At the end $D(i)$ contains the smallest vertex number of the connected component it belongs to. The graph is represented by a symmetric adjacency matrix ($A[i,j] = \mathit{true} \Leftrightarrow (i,j) \in E$). Their algorithm is as follows ($n = |V|$):
Algorithm *Connected Components*

(1) **forall** $0 <= i < n$ **do in parallel** $D(i) := i$
(2) **repeat**
(3) **forall** $0 <= i < n$ **do in parallel**
(4) $S := \{D(j) | A(i,j) = \mathit{true} \wedge D(j) \neq D(i)\}$;
(5) **if** $S = \emptyset$ **then** $C(i) := D(i)$ **else** $C(i) := \min(S)$;
(6) **forall** $0 <= i < n$ **do in parallel**
(7) $T := \{C(j) | D(j) = i \wedge C(j) \neq i\}$;
(8) **if** $T = \emptyset$ **then** $C(i) := D(i)$ **else** $C(i) := \min(T)$;
(9) **forall** $0 <= i < n$ **do in parallel** $D(i) := C(i)$;
(10) **repeat**
(11) $C(i) := C(C(i))$;
(12) **until** C does not change
(13) **forall** $0 <= i < n$ **do in parallel** $D(i) := \min(C(i), D(i))$;
(14) **until** D does not change

■

The algorithm detects in line (3)-(5) whether there are two vertices in different components connected by an edge. The resulting vector $C(i)$ contains in this case the lowest component to which $D(i)$ is connected and leaves $D(i)$ unchanged if the component $D(i)$ is not connected to another component. In vector C the value of two vertices j and k with the same component number in D may differ. They are set to their minimal value in C in lines (6)-(8). Hence, after line (9) the if i and j were in the same components at line (3), they are also in the same components after line (9). In [HCS79] is shown that the vector C interpreted as a pointer structure, has nearly a tree structure: there is are only loops of length 2 in C, and all other paths lead to one of the nodes in that loop. All the vertices in this tree belong to the same component. Thus performing pointer jumping in lines (10)-(12) on vector C let all vertices point to a root (which is one of the vertices in a loop of length 2. Observe that this root need not to be the smallest vertex number in the component. This is adjusted in line (13). The algorithm is best understood by performing an example. In this example we index the arrays C and D by the iteration number, the C, and D after line (k) is indexed by an upper index. The example is from [GR88]:

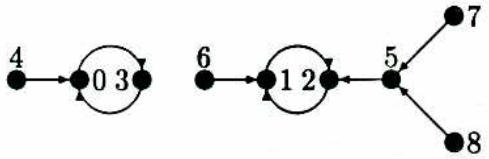


The algorithm produces the following states.

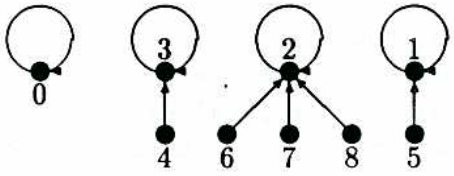
	0	1	2	3	4	5	6	7	8
D_0	0	1	2	3	4	5	6	7	8
C_1^5	3	2	1	0	0	2	1	5	5
C_1^8	3	2	1	0	0	2	1	5	5
D_1^9	3	2	1	0	0	2	1	5	5
C_1^{12}	0	1	2	3	3	1	2	2	2
D_1^{13}	0	1	1	0	0	1	1	1	1
C_2^5	0	1	1	1	1	0	1	1	1
C_2^8	1	0	1	0	0	1	1	1	1
D_2^9	1	0	1	0	0	1	1	1	1
C_2^{12}	0	1	0	1	1	0	0	0	0
D_2^{13}	0	0	0	0	0	0	0	0	0

If we interprete these vectors as pointers and draw a graph $G(v) = (\{0, \ldots, 8\}, \{(i, v[i]) | 0 \leq i \leq 8\})$ associated to a vector v then the states are:

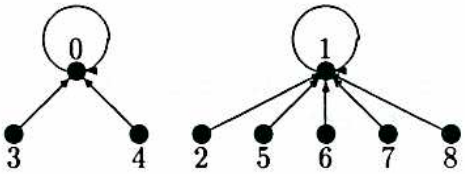
$C_1^5 = C_1^8$



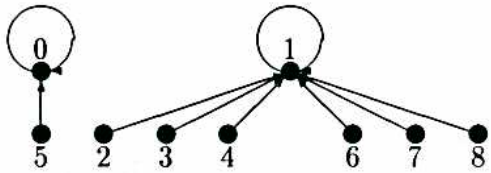
$C_1^{12} = D_1^9$



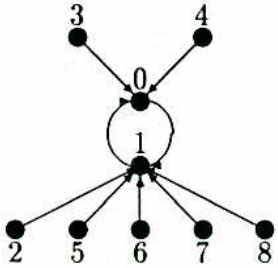
D_1^{13}



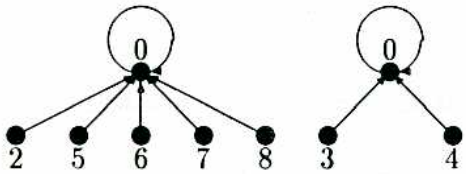
C_2^5



C_2^8



C_2^{12}



The program in PARFL is as follows:

```
fun connected_components(a:<<bool>>):<nat> =
  it_cc(a,forall 0 <= i < lg(a) do in parallel i)

fun it_cc(a:<<bool>>,d:<nat>):<nat> =
  let n = lg(a) in
  let t = forall 0<=j<n do in parallel
            forall 0<=i<n do in parallel
              if a[i][j] and d[i]<>d[j] then d[j] else n
  in let t1 = forall 0<=j<n do in parallel vmin(t[j])
  in let c1 = forall 0<=j<n do in parallel
                if t1[j]=n then d[j] else t1[j]
  in let t2 = forall 0<=j<n do in parallel
                forall 0<=i<n do in parallel
                  if d[j]=i and c1[j]<>i then c1[j] else n
  in let t3 = forall 0<=j<n do in parallel vmin(t2[j])
  in let c2 = forall 0<=j<n do in parallel
                if t3[j]=n then d[j] else t3[j]
  in let c = repeat(c2)
  in let d'= forall 0<=j<n do in parallel min(c[j],c2[c[j]])
  in if d=d' then d else it_cc(a,d')

fun repeat(c:<nat>):<nat> =
  let c'= forall 0<=i<lg(c) do in parallel c[c[i]] in
  if c=c' then c else repeat(c')

fun vmin(x:<nat>):nat =
  if mt(tl(x)) then hd(x)
  else vmin(forall 0<=i<lg(x)/2) do in parallel min(x[2*i],x[2*i+1])
```

Rule 1 is not applicable, and hence algorithm *remove_incompleteness* leaves the program unchanged. For expressing $\hat{\Pi}$ we abbreviate `t`, `t1`, `c1`, `t2`, `t3`, `c2`, `c`, and `d'` as in the program above by the let expression. This is just for clarity. In the automatic complexity analysis they are really expanded. Then $\hat{\Pi}$ is:

```
fun time_connected_components(a:<<bool>>):nat =
  7 + time_it_cc(a,forall 0 <= i < lg(a) do in parallel i)

fun time_it_cc(a:<<bool>>,d:<nat>):nat =
  if d=d' then
    122 + max(0<=j<lg(a),time_vmin(t[j]))
        + max(0<=j<lg(a),time_vmin(t2[j])) + time_repeat(c2)
  else
    124 + max(0<=j<lg(a),time_vmin(t[j]))
        + max(0<=j<lg(a),time_vmin(t2[j])) + time_repeat(c2)
        + time_it_cc(a,d')
```

```
fun time_repeat(c:<nat>):nat =
  if c=forall 0<=i<lg(c) do in parallel c[c[i]] then 15
  else 16 + time_repeat(forall 0<=i<lg(c) do in parallel c[c[i]])

fun time_vmin(x:<nat>):nat =
  if mt(tl(x)) then 6
  else 22 + time_vmin(forall 0<=i<lg(x)/2) do in parallel min(x[2*i],x[2*i+1])

fun length_connected_components(a:<<bool>>):nat =
  length_it_cc(a,forall 0 <= i < lg(a) do in parallel i)

fun length_it_cc(a:<<bool>>,d:<nat>):nat =
  if d=d' then lg(d) else length_it_cc(a,d')

fun length_repeat(c:<nat>):nat =
  if c=forall 0<=i<lg(c) do in parallel c[c[i]] then lg(c)
  else length_repeat(forall 0<=i<lg(c) do in parallel c[c[i]])
```

Algorithm *normalize* leaves the above program unchanged. In algorithm *symbolic_evaluation*, the only function which can be evaluated symbolically is **vmin**. Hence the equation system is:

```
time_connected_components(a) =
  7 + time_it_cc(a,forall 0 <= i < lg(a) do in parallel i)

time_it_cc(a,d) =
  if d=d' then 122 + max(0<=j<lg(a),time_vmin(t[j]))
                   + max(0<=j<lg(a),time_vmin(t2[j])) + time_repeat(c2)
  else 124 + max(0<=j<lg(a),time_vmin(t[j]))
        + max(0<=j<lg(a),time_vmin(t2[j])) + time_repeat(c2)
        + time_it_cc(a,d')

time_repeat(c) = if c=forall 0<=i<lg(c) do in parallel c[c[i]] then 15
  else 16 + time_repeat(forall 0<=i<lg(c) do in parallel c[c[i]])

time_vmin(<c>) = 6
time_vmin(cons(c1,cons(c2,x))) =
  22 + time_vmin(forall 0<=i<lg(x)/2+1 do in parallel
                   min(cons(c1,cons(c2,x))[2*i],cons(c1,cons(c2,x))[2*i+1])

length_connected_components(a) =
  length_it_cc(a,forall 0 <= i < lg(a) do in parallel i)

length_it_cc(a,d) = if d=d' then lg(d) else length_it_cc(a,d')

length_repeat(c) =
  if c=forall 0<=i<lg(c) do in parallel c[c[i]] then lg(c)
  else length_repeat(forall 0<=i<lg(c) do in parallel c[c[i]])
```

Now, algorithm *create_recurrences* outputs the following system of recurrences (each mapping is *lg*):

```
time_connected_components(a,n) =
  7 + time_it_cc(a,forall 0 <= i < n do in parallel i,n)

time_it_cc(a,d,n) =
  if d=d' then 122 + 2*time_vmin(n) + time_repeat(c2,n)
  else 124 + 2*time_vmin(n) + time_repeat(c2,n) + time_it_cc(a,d',n)

time_repeat(c,n) = if c=forall 0<=i<n do in parallel c[c[i]] then 15
  else 16 + time_repeat(forall 0<=i<n do in parallel c[c[i]],n)

time_vmin(1) = 6
time_vmin(n+2) = 22 + time_vmin(n/2+1)

length_connected_components(a,n) =
  length_it_cc(a,forall 0 <= i < n do in parallel i,n)

length_it_cc(a,d,n) = if d=d' then n else length_it_cc(a,d',n)

length_repeat(c,n) =
  if c=forall 0<=i<n do in parallel c[c[i]] then n
  else length_repeat(forall 0<=i<n do in parallel c[c[i]],n)
```

The recurrences for `x_repeat` and `x_it_cc` cannot be solved without help of the user. Even if we had already a heuristic for the function `repeat` we do it within a parallel iteration scheme. In this case this a monoid $(\Omega_n, \cdot)$ where[1]

$$\Omega_n = \{c : \mathbf{N}_n \mapsto \mathbf{N}_n | (\mathbf{N}_n, \{(i, c(i)) | i \in \mathbf{N_n}\}) \text{ contains at most cylces of length } 2\}$$

The operation $\cdot$ is function composition. It is easy to see that c^n is idempotent and that $c^i = c\dot{c}$. Hence $\log_2 n$ recursive calls are necessary to reach idempotency. The iteration scheme of `it_cc` is more complicated. Here we give a non-deterministic scheme (i.e. it contains a computation as given by the program, but it may contain more computations), and show that the fixpoint is achieved for any computation after $\log_2 n$ steps. We define a *generalized* $G = (V, E)$ where $V \subseteq 2^{\mathbf{N}_n}$ is a partition of $\mathbf{N}_n$, i.e. the parallel iteration scheme is a monoid $(\Phi_n, \cdot)$ with:

$$\Phi_n = \{(V, E) | \ V \text{ is a partition of } \mathbf{N}_n \ \}$$

The partition V reflects the component merging. In one stage the size of V is at least reduced by a factor of two. It is possible to merge m components by pairwise union into $m/2$ components. Sometimes the algorithm merge even more components, but this is the minimum. The operation $\cdot$ is based on the following partial order:

$$V_1 \sqsubseteq V_2 :\Leftrightarrow \forall C \in V_1 \exists D \in V_2 : C \subseteq D$$

[1] $\mathbf{N}_n = \{0, \ldots, n-1\}$

The bottom element is $\bot_n = \{\{0\}, \ldots, \{n-1\}$ (which is also the intialization). The top element is $\top_n = \{\mathbf{N}_n\}$. Let $V_1 \sqcup V_2$ be the least upper bound of V_1 and V_2. The merging of components w.r.t. E can be described non-deterministically by the following operation:

$$V/E = \{C_1 \cup C_2 | C_1, C_2 \in V \wedge \exists i \in C_1, j \in C_2 : (i,j) \in E\}$$

The elements C_1 and C_2 are chosen s.t. V/E are also a partition. Observe, that not the minimal neighbours were merged but just one neighbour. The effect of one iteration can now be described by $G := (V/E, E)$. In order to find the operation $\cdot$ we define for $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ that

$$G_1 \cdot G_2 = ((V_1 \sqcup V_2)/E, E)$$

Then $\bot_n^n$ is idempotent w.r.t. E and in each iteration $G := G^2$. Hence the body of `it_cc` terminates after at most $\log_2 n$ recursive calls. Hence the final recurrence system is:

```
time_connected_components(n) = 7 + time_it_cc(n,n)

time_it_cc(n,1) = 122 + 2*time_vmin(n) + time_repeat(n)
time_it_cc(n,m) = 124 + 2*time_vmin(n) + time_repeat(n) + time_it_cc(n,m/2)

time_repeat(1) = 15
time_repeat(m) = 16 + time_repeat(m/2)

time_vmin(1) = 6
time_vmin(n+2) = 22 + time_vmin(n/2+1)

length_connected_components(n) = length_it_cc(n,n)

length_it_cc(n,1) = n
length_it_cc(n,m) = length_it_cc(n,m/2)

length_repeat(n,1) = n
length_repeat(n,m) = length_repeat(n,m/2)
```

Hence

Semi-Automatic Theorem 5.7 *The time complexity of* `connected_components(a)` *is at most*

$$\texttt{time\_connected\_components}(n) = 60\ \log_2^2(n) + 211\ \log_2(n) + 156$$

where $n = lg(\texttt{a})$.

Appendix A

Discussion of Algorithm remove_incompleteness

Here we prove the termination of algorithm *remove_incompleteness*, and obtain then its complexity. The main part in this proof is lemma A.5 which connects correctly typed programs to the impossibility of some programs. In fact, this connection will ensure the termination of algorithm *remove_incompleteness*.

We start first with the easy termination proofs:

Lemma A.1 (Termination of Loop 1) *Loop 1 terminates. After its termination* Π *does not contain any expression of the form* $\mu(g(\cdots))$ *where* μ *is a basic operation and* $g \in \Pi$, *and the set* A *contains new functions to be created.*

Proof: The number of redeces of rule 1 is decreased by one after one execution of the body of loop 1. Hence, loop 1 terminates. Furthermore, after its termination there is no redex of rule 1 and therefore R cannot contain a basic operation. If an expression $\mu(f(\cdots))$ is transformed, then μ_f is added to the set A, if μ_f is not already defined. Thus the set A contains the new functions to be created by loop 2. ■

For the termination proof of loop 2, we prove first the termination of the loops inside loop 2.

Lemma A.2 (Termination of Loop 2.1) *Loop 2.1 terminates on each set* F *of top-level mutually recursive functions. After its termination, each top-level expression of the function bodies of* F *are of the form*

$\mu(g(\cdots))$, *where* $g \in F$, *and* $\mu \neq tl$ *is a basic operation*

or $g(\cdots)$, *where* $g \in F$, *or* $\mu(h(\cdots))$ *and* $h \notin F$ *(*μ *is a basic operation).*

Proof: Observe first, that loops 2.1.1 and 2.1.2 terminate, because the number of redeces of rules 4 and 5 inside a function body of F and Π, respectively, decreases by one after each execution of the bodies.

After execution of loop 2.1.1 on B, there is no top-level expression of the form $tl(g(\cdots))$, where $g \in F$. This loop 2.1.1. is executed for each $f \in F$. Therefore there is no function body of F with a top-level expression of the form $tl(g(\cdots))$, where $g \in F$. ■

Lemma A.3 (Termination of Loop 2.2) *Loop 2.2 terminates, and after its termination on B', there is no subexpression of the following forms*

- μ(**if** $\cdot$ **then** $\cdot$ **else** $\cdot$), μ *basic operation.*
- μ(**let** $\cdot$ = $\cdot$ **in** $\cdot$), μ *basic operation.*
- μ(**forall** $\cdot$ **do in parallel** $\cdot$), μ *basic operation.*
- μ(**select** $\cdot$ **in parallel from** $\cdot$), μ *basic operation.*
- μ(**modify** $\cdot$ **to** $\cdot$ **from** $\cdot$), $\mu \neq$ *access basic operation.*

in B'.

Proof: The number of redeces of rule 3 is reduces by one after one execuution of the body of loop 2.2. After its termination B' contains no redex of rule 3. Hence the second result. ■

Lemma A.4 (Termination of Loop 2.3) *Loop 2.3 terminates and after its termination, no expression of the form $\mu(g(\cdots))$ occur. The set A contains new functions to be created.*

Proof: The number of redeces of rule 5 decreases by one after one execution of the body of loop 2.3. Therefore loop 2.3 terminates, and after its termination there is no redex of rule 5. If a new function has to be created it is added to A in the **if**-statement. ■

We prove now, that a situation like in the example discussed after algorithm *remove_incompleteness* cannot occur with any from *tl* different basic operation:

Lemma A.5 (Correctly Typed Programs) *Let Π be a correctly typed program, and let $F = \{f_1, \ldots, f_k\}$ be a set of top-level mutually recursive functions. Then any top-level expression in a body of $f \in F$ is a subexpression of an expression of the form $\mu_1(\cdots(\mu_k(g(\cdots))\cdots)$, where either any $\mu_i = tl$ or $g \notin F$.*

Proof: Suppose, that there is a body of a function $f \in F$ which has a top-level expression inside an expression of the form $\nu_1(\cdots(\nu_k(g(\cdots))\cdots)$ where $g \in F$ and one $\nu\_i = hd$. With out loss of generality let be $f = f_1$ and $g = f_2$. Define furthermore $\mu_1 = \nu_1 \circ \cdots \circ \nu_2$. Let $f_1, \ldots, f_k$ without loss of generality be a top-level calling sequence, i.e. f_i contains a call $\mu_i(f_{i+1}(\cdots))$ on a top-level position and f_k contains a call $\mu_k(f_1(\cdots))$ on a top-level positions. Furthermore, let the types of the f_i be of the form $A_i \mapsto B_i$. Because the μ_i are compositions of basic operations their types are of the form

$$\mu_i : \langle^{r_i} E\rangle^{r_i} \mapsto E$$

where $r_1 \geq 1$ because μ_1 contains one *hd* operation. The output type of f_{i+1} must be the input type of μ_i, and on the other hand the output type of f_i must be the output type of μ_i. Therefore we obtain the equation system

$$B_i = E_i, 1 \leq i \leq k,\ B_{i+1} = \langle^{r_i} E_i\rangle^{r_i},\ 1 \leq i < k, B_1 = \langle^{r_k} E_k\rangle^{r_k}$$

This equation system leads to an equation $E_1 = \langle^r E_1\rangle^r$, where $r > 1$. But there is no type E_1 satisfying this equation.

The contradicitions for the other basic operations are derived in a similar way. ■

From lemma A.2 and lemma A.5 we immediatly obtain:

Corollary A.6 (Property after Loop 2.1) *Let F be the set of top-level mutually recursive functions before loop 2.1. Then all top-level expressions are of the form $f_i(\cdots)$, if $f_i \in F$.*

Now all the technical properties are prepared for the proof of termination of loop 2. We suppose without loss of generality, that mutually recursive functions are processed together. By corollary A.6 while transforming a function f, new functions can only be added to A if the corresponding is not in not in the top-level mutually recursive closure of f. We therefore consider the top-level graph reduced to its strongly connected components. It is well-known that this reduced graph is acyclic. Therefore during execution of loop 2, a point is reached where no new functions are added to A. Hence we know together with lemmas A.2, A.3, and A.4:

Lemma A.7 (Termination of Loop 2) *Loop 2 of algorithm* remove_incompleteness *terminates.*

Together with lemma A.1, this proves the termination of algorithm *remove_incompleteness.*

We turn now to the discussion of the time complexity of algorithm *remove_incompleteness.* Let n be the length of the program $\Pi \cup R$ (i.e. the number of symbols occuring in $\Pi \cup R$), and k be the maximal nesting depth w.r.t basic oeperations of an expression $\mu_1(\ldots(\mu_k(f(\cdots))\cdots)$. We know from corollary A.6, that for each $f \in \Pi$, new functions $\mu_k\text{-}f, \ldots, \mu_1\text{-}\ldots\text{-}\mu_k\text{-}f$ are created. Let m be the number of basic operations (here $m = 4$), and l be the number of functions in Π. Then at most $k\ m^k\ l$ new functions are created, i.e. the resulting program has length $O(k\ m^k\ n)$. This lenght is obviously propotional to the running time of algorithm *remove_incompleteness.* Hence we have:

Lemma A.8 (Time Complexity of Algorithm *remove_incompleteness*) *If n is the length of $\Pi \cup R$, k is the maximal nesting depth of basic operations in Π, and m is the number of basic operations, the running time of algorithm* remove_incompleteness *is $O(k\ m^k\ n)$.*

Observe that m is a constant, and that in practice k is also constant. Therefore in practice, the running time of this algorithm is linear in its input size.

Appendix B

The Complexity of Algorithm instantiate_condition

In order to compute the complexity in lemma 3.23 we have to prove several lemmas about the output cardinalities and size of the sets involved. We start with simplifying algorithm *combine* for the case where all the leading coefficients of linear terms equal to 1. In this case, lines (4)–(14) can be replaced by **output** $S_1 \cap S_2$. But if $n \leq m$ this costs just time $O(n \log m)$. Also line (50) just requires constant time. Then we obtain easily from the proof of lemma 3.21:

Corollary B.1 (Time Complexity of *combine* in Special Case) *Let be* $S_1 = \{t_1 \ldots, t_n\}$ *and* $S_2 = \{u_1, \ldots, u_m\}$ *such that each subterm of type nat is either constant or linear with leading coefficient 1. Without loss of generality suppose* $n \leq m$. *Then*

(a) *If all* t_i *and* u_j *are of type nat, then algorithm* combine(S_1, S_2) *needs time* $O(n \log m)$.

(b) *If* t_i *and* u_j *are vectors (or tuples, respectively) then algorithm* combine(S_1, S_2) *needs time* $O(m\ r + n\ s)$ *where* $r = sz(S_1)$ *and* $s = sz(S_2)$.

For the output cardinality we have:

Lemma B.2 (Output Cardinaliyt of *combine*) *If* $n = card(S_1)$ *and* $m = card(S_2)$. *Then*

$$card(\text{combine}(S_1, S_2)) \leq n\ m$$

Proof: In the case of natural numbers, it is easy to see that: $card(combine(S_1, S_2)) \leq \min(n, m) \leq n \cdot m$. Otherwise, in line (43), at most one element is added whenever it is executed. From line (50) it also follows that $card(S') \leq 1$ because in this case $card(combine(\{\rho(x)\}.\{\rho(y)\})) \leq 1$. Hence line (52) also adds at most one element. In each execution of lines (41)–(54), at most one of lines (43) and (52) are executed. Because lines (41)–(54) are executed $n\ m$ times, we have proven the claim. ■

Lemma B.3 (Output Size of *combine*) *Define* $n = card(S_1)$, $m = card(S_2)$, *and* $r = \max\{sz(t) | t \in S_1 \cup S_2\}$. *Then:*

$$sz(\text{combine}(S_1, S_2)) \leq n\ m\ r^r$$

Proof: Define first for two terms t and u that $r = sz(t)$, $s = sz(u)$, and σ to be the most general unifier of t and u. Without loss of generality let be $r \geq s$. Then it is well-known that

$$sz(\sigma t) \leq \left(\frac{r}{s}\right)^s \leq r^r$$

Thus for any term $t \in combine(S_1, S_2)$, it is $sz(t) \leq r^r$. This together with lemma B.2 yields the stated claim. ■

We consider now algorithm *instantiate_condition*. From now we define for any condition C in DNF p to be the number of literals in C, k to be the maximal number of operations in $\{hd, tl, (\cdot).i, (\cdot)[i]\}$ in a literal of C, q to be the maximal number of operations in $\{cons, (\cdot,\cdot)\}$ in a literal of C, and m to be the largest natural number occuring in C ($m = 1$ if there is no natural number). The first complexity we consider is the output cardinality of *instantiate_condition*$(C, (x_1, \ldots, x_n))$:

Lemma B.4 (Output Cardinality of *instantiate_condition*) *The output cardinality of* instantiate_condition *is:*

$$card(instantiate\_condition(C, (x_1, \ldots, x_n))) \leq m^{p \cdot 2^{(q+k\ m)/2}}$$

Proof: Define $C(k, q, m) = card(instantiate\_condition(L, (x_1, \ldots, x_n)))$ where L is a literal. If $k = 0$ and $q = 0$, then line (16) (or line (21)) dominate, with an output cardinality of m. Thus

$$C(0, 0, m) \leq m$$

The value k decreases in the recursive calls in lines (40), (43), (47), (51), (55), (57), (65), and (69). Each of these operations increase q. If we make the (reasonable) assumption, that $C(k, q, m)$ is non-decreasing[1] in each of its arguments. Then lines (66) and (70) dominate the others. Observe further, that the output cardinality of in these lines equals to the output cardinality of the recursive calls in lines (65) and (69) respectively. In lines (65) and (69), the number of new variables increases at most by m. Hence:

$$C(k, q, m) \leq C(k - 1, q + m, m)$$

When q decreases, consider first line (61). There are two recursive calls, both of them with at most $q - 2$ operations in $\{cons, (\cdot,\cdot)\}$. Because line (61) requires a combination, we obtain with lemma B.2:

$$C(k, q, m) \leq c^2(k, q, m)$$

Line (62) involves just a union of two sets and is therefore dominated by the output cardinality of line (61). Lines (71) and (72) are dealt analogously. Thus, the inequations

$$\begin{aligned} C(0, 0, m) &\leq m \\ C(k, q, m) &\leq C(k - 1, q + m, m) \\ C(k, q, m) &\leq C^2(k, q - 2, m) \end{aligned}$$

[1] The claim is in fact a non-decreasing function

Thus

$$C(k,q,m) \leq C(0,q+k\cdot m,m) \leq C^{\sqrt{2}^{q+k\cdot m}}(0,0,m) \leq m^{\sqrt{2}^{q+k\cdot m}}$$

In lines (3)–(7) at most all p literals are combined, hence

$$card(instantiate\_condition(C,(x_1,\ldots,x_n))) \leq C^p(k,q,m) \leq m^{p\cdot\sqrt{2}^{q+k\cdot m}}$$

■

The output size of *instantiate_condition* is growing extremely fast:

Lemma B.5 (Output Size of *instantiate_condition*) *The output size of* instantiate_condition *is:*

$$sz(instantiate\_condition(C,(x_1,\ldots,x_n))) \leq m^{p\cdot\sqrt{2}^{q+k\cdot m}}\ g(p,1,g^{2^k}((q+k\ m)/2,m,n+k\ m))$$

where the function $g(q,m,n)$ is defined by the recurrence

$$\begin{aligned} g(0,m,n) &= n \\ g(q,m,n) &= m^{2^q}\ g(q-1,m,n)^{g(q-1,m,n}} \end{aligned}$$

Proof: The organization of this proof is similar to the proof of lemma B.4. We define first:

$$S(k,q,m,n) := \max\{sz(t)|t \in instantiate\_condition(L,(x_1,\ldots,x_n))\}$$

where L is a literal. Clearly, if $k=q=0$, then $S(0,0,m,n) \leq n$. If k is decreased then the recursive calls in lines (40), (43), (47), (51), (55), (57), (65), and (69) have to be considered. If they output a maximal term size S, the maximal term size in the output of these cases is bounded by S^2. Again, by the assumption that $S(q,k,m,n)$ is non-decreasing in its arguments, we obtain

$$S(q,k,m,n) \leq S^2(q,k,m,n)$$

If q is decrease, then $S(k,q,n,m)$ is dominated by the output in line (61), because in this case, two sets are combined, both of them having a maximal terms size of at most $S(k,q-2,m,n)$. Hence by lemma B.3:

$$S(k,q,m,n) \leq C^2(k,q,m)\cdot S(k,q-2,n,m)^{S(k,q-2,m,n)}$$

We conclude first that $S(k,q,n,m) \leq S^{2^k}(0,q+k\cdot m,m,n+k\cdot m)$, and then with the definition of g and considering the third equation with $k=0$:

$$S(k,q,m,n) \leq g^{2^k}((q+k\ m)/2,m,n+k\ m))$$

Now, at most p *combines* are performed, i.e. the maximum term size of the output it most

$$g(p,1,S(k,q,n,m)) \leq g(p,1,g^{2^k}((q+k\ m)/2,m,n+k\ m)))$$

Because there are at most $m^{p \cdot \sqrt{2}^{q+k \cdot m}}$ terms in the output, the claim follows. ■

We conjecture, that usually, the size of the output is polynomial in its parameters, but we know from the results of the size of unification, that the size really explodes, even if probably not as much as here stated, when the analysis is done more carefully. We have also from the proof:

Corollary B.6 (Maximal Term Size in *instantiate_condition*) *The maximal term size of instantiate condition is given by:*

$$\max\{sz(t) | t \in \mathit{instantiate\_condition}(C, (x_1, \ldots, x_n))\} \leq g(p, 1, g^{2^k}((q + k\ m)/2, m, n + k\ m))$$

In order to get a feeling for the rate of growth of g, we state without proof that for fixed m and n:

$$A(2, q) \leq g(q, m, n) \leq A(3, q)$$

where $A(i, j)$ is Ackermann's function.

We have now everything prepared for analyzing the time complexity of algorithm *instantiate_condition*. Sets are supposed to be represented as 2-3-trees. We consider also first the running time $T(k, q, m, n)$ for literals. Then it is immediate that in the case $k = q = 0$ at most m insertions are performed as it can be seen from the dominating lines (16) and (21). This requires time $O(m \log m)$ and the maximum term size is n. Therefore:

$$T(0, 0, n, m) \leq c\ n\ m\ \log m + d \text{ where } c \text{ and } d \text{ are suitable constants}$$

If k is decreasing, then the case in lines (45)–(48) is dominating (or alternatively, lines (49)–(52), lines (63)–(66), or lines (67)–(70)). The computation of the output in line (48) is the same as the size of the output of line (46), (47). Hence:

$$T(k, q, m, n) \leq m^{\sqrt{2}^{q+k \cdot m}}\ g^{2^k}((q + k \cdot m)/2, m, n + k \cdot m) + T(k - 1, q + m, n + m, m)$$

When q is decreasing, consider line (61). This line dominates line (62) because it requires a *combine* operation. Lines (71) and (72) are dealt analogously. The time for the two recursive calls is bounded by $2 \cdot T(k, q - 2, n, m)$, because both, $a_1 = a_2$ and $l_1 = l_2$, contain at most $q - 2$ operations in $\{cons, (\cdot, \cdot)\}$. Their output cardinality is by lemma B.2 bounded by $m^{\sqrt{2}^{q+k \cdot m}}$, their maximal term size is by lemma B.6 bounded by $g^{2^k}((q + k\ m)/2, m, n + k\ m))$. Thus the combination time is:

$$O(m^2\ \log m\ m^{\sqrt{2}^{q+k \cdot m}}\ g^{2^k}((q + k \cdot m)/2, m, n + k \cdot m)$$

Hence there is a constant δ such that

$$T(k, q, m, n) \leq \delta\ m^2\ \log m\ m^{\sqrt{2}^{q+k \cdot m}}\ g^{2^k}((q + k \cdot m)/2, m, n + k \cdot m) + 2\ T(k, q - 2, n, m)$$

Altogether, we get

$$T(k, q, m, n) \leq k \cdot m^{\sqrt{2}^{q+k \cdot m}}\ g^{2^k}((q + k \cdot)/2, m, n + k \cdot m) + T(0, q + k\ m, n + k\ m, m)$$

$$T(0,q,n,m) \leq \delta\ q/2\ 2^{q/2}\ \log m\ m^{\sqrt{2}^q}\ g(q/2,m,n) + 2^{q/2}\ T(0,0,n,m)$$

The first term dominated the second hence:

$$T(k,q,m,n) = O(k\ m^{\sqrt{2}^{q+k\cdot m}}\ g^{2^k}((q+k\cdot m)/2, m, n+k\cdot m))$$

Evaluating the p literals requires at most time $O(p\cdot T(k,q,m,n))$, and combining at most p literals require at most time

$$O(p\ m^2\ logmm^{2\cdot p\cdot\sqrt{2}^{q+k\ cdotm)/2}}\ g(p,1,g^{2^k}((q+k\cdot m)/2, m, n+k\cdot m))$$

which dominates the time for evaluating the p literals. Thus the time complexity of *instantiate_condition*$(C,(x_1,\ldots,x_n))$ is

$$O(p\ m^2\ logmm^{2\cdot p\cdot\sqrt{2}^{q+k\ cdotm)/2}}\ g(p,1,g^{2^k}((q+k\cdot m)/2, m, n+k\cdot m))$$

Bibliography

[AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal on Computing*, 16:808 – 835, 1987.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Akl89] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1989.

[AKS83] M. Ajtai, J. Komlos, and E. Scemeredi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS 1968 Spring Joint Computer Conference*, pages 307–314, 1968.

[BD77] R. M. Burstall and J. Darlington. A transformation systems for developing recursive programs. *Journal of the ACM*, 24:44 – 67, 1977.

[BMD+85] F.L. Bauer, M.Broy, W. Dosch, F. Geiselbrechtinger, W. Hesse, R. Gnatz, B. Krieg-Brueckner, A. Laut, T. Matzner, B. Moeller, F. Nickle, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, and H. Woessner. *The Munich Project CIP*, volume 183 of *Lecture Notes in Computer Science.* Springer, 1985.

[Bre74] R.P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, pages 201 – 206, 1974.

[BW80] F.L. Bauer and H. Woessner. *Algorithmische Sprache und Programmentwicklung.* Springer-Verlag, 1980.

[CGG+88] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M. Watt. *MAPLE Reference Manual.* Symbolic Computation Group, Dept. of Computer Science, University of Waterloo, Canada, 5 edition, Maerz 1988.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770 – 785, 1988.

[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.

[DM89] M. Dietzfelbinger and F. Meyer auf der Heide. An optimal parallel dictionary. In *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 360 – 368, 1989.

[DM90] M. Dietzfelbinger and F. Meyer auf der Heide. How to distribute a dictionary in a complete network. Technical Report 68, Universität-Gesamthochschule Paderborn, Fachbereich Mathematik-Informatik, April 1990.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1, Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science. Springer Verlag, 1985.

[Fan86] M. Fanty. A Connectionist Simulator for the BBN Butterfly Multiprocessor. Technical report, Computer Science Department, University of Rochester, Rochester, January 1986.

[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.

[FS87] P. Flajolet and J.-M. Steyaert. A complexity calculus for recursive tree algorithms. *Mathematical Systems Theory*, 19:301 – 331, 1987.

[FSZ91] P. Flajolet, B. Salvy, and P. Zimmermann. Average case analysis of algorithms. *Theoretical Computer Science*, 1991.

[FV90] P. Flajolet and J.S. Vitter. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Vol. A*, pages 431–524. MIT-Press, 1990.

[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.

[HCS79] D.S. Hirschberg, A.K. Chandra, and D.V. Sarwate. Computing connected components on parallel computers. *CACM*, 22(8):461 – 464, 1979.

[Hil85] W.D. Hillis. *The Connection Machine.* MIT Press, 1985.

[INM86] INMOS. *Transputer Architecture.* INMOS Ltd., 1986.

[INM88] INMOS. *Transputer Databook.* INMOS Ltd., 1988.

[KR90] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Vol. A*, pages 871–941. MIT-Press, 1990.

[KU88] A. Karlin and E. Upfal. Parallel hashing: An efficient implementation of shared memory. *Journal of the ACM*, pages 876 – 892, 1988.

[Kuc82] L. Kucera. Parallel computation and conflicts in memory access. *Information Processing Letters*, 14:93–96, 1982.

[LeM88] Daniel LeMétayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248 – 266, 1988.

[LPP90] F. Luccio, A. Pietracaprina, and G. Pucci. A new scheme for the deterministic simulations of prams in vlsi. *Algorithmica*, 5:529 – 544, 1990.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258 – 282, 1982.

[Mor90] Nelson Morgan. The ring array processor (rap): Algorithms and architecture. Technical Report TR-90-047, International Computer Science Institute, 1990.

[MV84] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of prams by parallel machines with restricted garanularity of parallel memories. *Acta Informatica*, 21:339 – 374, 1984.

[PHT90] M. Philippsen, C. Herter, and W. F. Tichy. The triton project. Technical Report 33/90, Universität Karlsruhe, Fakultät für Informatik, December 1990.

[Pip79] Nicholas Pippenger. On simultaneous resource bounds. In *Proceedings on the 20th Symposium on Foundations of Computer Science*, pages 307 – 311, 1979.

[Ran87] A. Ranade. How to emulate shared memory. In *Proceedings on the 28th Symposium on Foundations of Computer Science*, pages 185 – 194, 1987.

[RT86] R. Rettberg and R. Thomas. Contention is no Obstacle to Shared-Memory Multiprocessing. *Communications of the ACM*, 29:1202–1212, 1986.

[SABS86] S. Steinberg, D. Allen, L. Bagnall, and C. Scott. The Butterfly TM Lisp System. *AI Languages and Architectures*, pages 730–741, 1986.

[Smi88] D. Smith. Kids - a knowledge-base software development system. Technical report, Kestrel-Institute, 1988.

[TV85] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14:862 – 874, 1985. also in: Foundations of Computer Science 1984, pp. 12–20.

[UW87] E. Upfal and A. Widgerson. How to share memory in a distributed system. *Journal of the ACM*, 34:116 – 127, 1987.

[Val90] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Vol. A*, pages 945–971. MIT-Press, 1990.

[Weg75] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528 – 539, 1975.

[WG85] W. Waite and G. Goos. *Compiler Construction.* Texts and Monographs in Computer Science. Springer, 1985.

[Zim90a] Wolf Zimmermann. *Autmatische Komplexitätsanalyse funktionaler Programme.* Informatik-Fachberichte. Springer, 1990. Disseration at the Universität Karlsruhe, Fakultät für Informatik.

[Zim90b] Wolf Zimmermann. Automatic worst case analysis of parallel programs. Technical Report TR-90-066, International Computer Science Institute, December 1990.

[Zim91] Paul Zimmermann. *Séries génératrice et analyse automatique d'algorithmes.* PhD thesis, Ecole Polytechnique Superieur Paris, 1991.