Real-Time Properties of Indirect Recursive Procedures

Johann Blieberger
Institut für Rechnergestützte Automation
Technische Universität Wien
Treitlstr. 1, A-1040 Wien
E-mail: blieb@auto.tuwien.ac.at

Received March 31, 1995; accepted July 15, 1996

The purpose of this paper is to show that indirect recursive procedures can be used for implementing real-time applications without harm, if a few conditions are met. These conditions ensure that upper bounds for space and time requirements can be derived at compile time. Moreover they are simple enough such that many important recursive algorithms can be implemented.

In addition, our approach allows for concentrating on essential properties of the parameter space during space and time analysis. This is done by morphisms that transfer important properties from the original parameter space to simpler ones, which results in simpler formulas of space and time estimates.

1. INTRODUCTION

The most significant difference between real-time systems and other computer systems is that the system behavior must not only be correct but the result of a computation must be available within a predefined deadline. It has turned out that a major progress in order to guarantee the timeliness of real-time systems can only be achieved if the *scheduling problem* is solved properly. Most scheduling algorithms assume that the runtime of a task is known a priori (cf. e.g. [LL73, HS91, Mok84]). Thus the *worst-case execution time* of a task plays a crucial role.

The most difficult tasks in estimating the timing behavior of a program are to determine the number of iterations of a certain loop and to handle problems originating from the use of recursion. A solution to the first problem has been given in [Bli94], direct recursion has been treated in [BL96], indirect recursion will be studied in this paper.

If recursive procedures are to be used in implementing real-time applications, several problems occur:

1. It is not clear, whether a recursive procedure completes or not.
2. If it completes, it must be guaranteed that its result is delivered within a predefined deadline.

3. Since most real-time systems are embedded systems with limited storage space, the result of a recursive procedure must be computed using a limited amount of stack space.

In view of these problems most designers of real-time programming languages decide to forbid recursion in their languages, e.g. RT-Euclid (cf. [KS86, HS91]), PEARL (cf. [DIN82]), Real-Time Concurrent C (cf. [GR91]), and the MARS-approach (cf. [KDK$^+$89, PK89]).

Other so-called real-time languages allow recursions to be used, but do not provide any help to the programmer in order to estimate time and space behavior of the recursive procedures, e.g. Ada (cf. [Ada95]) and PORTAL (cf. [Bus85]). Interestingly, a subset of Ada (cf. [For93]) designed for determining the worst-case timing behavior forbids recursion. The same applies to SPARK ([Bar97]), a subset of Ada for high integrity systems. PORTAL uses RECURSION resources and terminates a recursive computation if the resource is exhausted. Although it is not clear from the description, one can suspect that a RECURSION resource is equivalent to an area of memory that contains the stack space. Both Ada and PORTAL cannot handle the time complexity of recursive procedures. The on-going discussion on RT-Java (cf. e.g. [Nil96]) does not touch recursive procedures, too.

Other approaches do not address recursion at all (cf. e.g. [MACT89, Sha89, Par93, ITM90]), others (cf. e.g. [PK89]) propose to replace recursive algorithms by iterative ones or to transform them into non-recursive schemes by applying program transformation rules. Certainly, if a *simple* iterative version of a recursive algorithm exists and it is also superior in space and time behavior, it should be used instead of a recursive implementation. On the other hand there are the following reasons why recursive algorithms should be implemented by recursive procedures:

- The space and time behavior of transformed programs are by no means easier to investigate than their recursive counterparts, since the stack has to be simulated and because they contain while-loops. In general, the number of iterations of these loops cannot be determined at compile time.
- A recursive algorithm originates from recursiveness in the problem domain. From the view of software engineering, a program reflecting the problem domain is considered better than others not doing so (cf. e.g. [Boo91]).
- Often recursive algorithms are easier to understand, to implement, to test, and to maintain than non-recursive versions.

Our approach is different in that we do not forbid recursion, but instead constrain recursive procedures such that their space and time behavior either can be determined at compile time or can be checked at runtime. Thus timing errors can be found either at compile time or are shifted to logical errors detected at runtime. Hence all three problems above are solved by our approach. In particular, problem (1.) can be tested at runtime (cf. Section 7) and problems (2.) and (3.) can be solved at compile time or tested at runtime (cf. sections 5 and 4, respectively).

The constraints mentioned above are more or less simple conditions. If they can be proved to hold, the space and time behavior of the recursive procedure can be estimated easily.

Compared to the paper on direct recursion ([BL96]) this paper requires a much more delicate analysis. Even defining the very important concept of *recursion* *depth* is much more complex than for direct recursion. The results on space and time effort are much harder to derive than their "direct" counterparts. Although this paper can be seen as a strict generalization of [BL96], the reader may like to contact [BL96] as an introduction before going into the details of analyzing indirect recursion.

The basic idea of our approach is to employ monotonical properties of the recursive procedures in order to determine worst-case space and time behavior. The major focus is on "local" monotonical properties which means that space and time behavior can be found (or estimated) without having to analyze the whole recursive call-chain.

The rest of the paper is structured as follows: Section 2 presents important definitions and preliminary results. Section 3 defines the computational model and how space and time are measured. Sections 4 and 5 are concerned with worst-case space and time behavior, respectively. Section 6 introduces parameter space morphisms which can be used to abstract from unnecessary details of the parameter space. Section 7 handles programming language issues.

Within this paper we will use the following notational conventions:

- When we speak of *recursive procedures*, we mean both *recursive procedures* and *recursive functions.*
- When we speak of *space*, we mean *stack space* and not *heap space.* If dynamic data structures are used for the internal representation of an object, the space allocated from the heap is under control of the object/class manager. On the other hand, the space allocated from the stack originating from the use of recursive procedures cannot be explicitly controlled by the application. This case requires a thorough treatment, which will be performed in this paper.

Throughout this paper we will use two examples to illustrate our theoretical treatment.

Example 1.1. We define

$$a(n) = \begin{cases} 0 & \text{if } n = 0, \\ b(n-1) + 1 & \text{otherwise,} \end{cases}$$

$$b(n) = \begin{cases} 0 & \text{if } n = 0, \\ [a(n-1)]^2 & \text{otherwise.} \end{cases}$$

The first few values of $a(n)$ and $b(n)$ are given in Table 1.

Example 1.2. This example is of little practical interest but it shows which complex indirect recursions can be treated by our method:

$$f(n) = \begin{cases} 1 & \text{if } n = 0. \\ 1 + g(n) + f(n-1) & \text{if } n > 0, \end{cases}$$

$$g(n) = \begin{cases} 1 & \text{if } n = 0. \\ 1 + f(n-1) + \sum_{i=0}^{n-1} g(i) & \text{if } n > 0, \end{cases}$$

Table 2 shows the first few values of $f(n)$ and $g(n)$.

By some manipulations this recurrence relation can be solved and one obtains:

$$f(n) = \frac{7}{4}3^n + \frac{n}{2} - \frac{3}{4}, \quad \text{for } n \geq 0$$
$$g(n) = \frac{7}{2}3^{n-1} - \frac{1}{2}, \quad \text{for } n \geq 1$$

Further examples will be given in the text but those listed above will be our major references.

Remark. In this paper we will use the following notations.

- By $\log N = \log_e N$ we denote the natural logarithm of N.
- By $\mathrm{ld}N$ we denote the binary logarithm of N.
- The greatest integer $n \leq x$ is denoted by $\lfloor x \rfloor$.

2. DEFINITIONS AND PRELIMINARY RESULTS

In general a procedure is an algorithmic description which, given some parameters as input, performs computations to produce its results (output). If such a procedure p uses another procedure p' to compute its results, we say that p calls p'.

If p calls itself and no other procedure is involved, p is called a *direct recursive procedure.* If p calls p' and p' calls p, we call p *indirect recursive.* Of course p' is an indirect recursive procedure too.

In addition, more than two procedures can be involved in such a computation. Each procedure is allowed to call one of the others or itself. More formally we use the notation of the following definition.

Definition 2.1. Let $\mathcal{P} = p^{(1)}, \ldots, p^{(\ell)}$ denote a finite number of *indirect recursive procedures.* $\mathcal{P}$ is called an *indirect recursive procedure system.* By $\mathcal{F}^{(j)}$ we denote the parameter space of $p^{(j)}$. By $\mathcal{F} = \bigcup_{j=1}^{\ell} \mathcal{F}^{(j)}$ we denote the parameter space of $\mathcal{P}$.

Remark. We assume that even if $\mathcal{F}^{(j)}$ and $\mathcal{F}^{(k)}$ $(j \neq k)$ have elements in common, they can be discriminated by the index j and k, respectively.

Definition 2.2. We call an indirect recursive procedure $p^{(j)}$ *well-defined* if for each element of $\mathcal{F}^{(j)}$ the procedure $p^{(j)}$ completes correctly, e.g. does not loop infinitely and does not terminate because of a runtime error (other than those predefined in this paper).

From now on, when we use the term indirect recursive procedure, we mean well-defined indirect recursive procedure. We deal with non-well-defined recursive procedures in Section 7.

TABLE 1
The first few values of $a(n)$ and $b(n)$

n	$a(n)$	$b(n)$
0	0	0
1	1	0
2	1	1
3	2	1
4	2	4
5	5	4
6	5	25
7	26	25
8	26	676
9	677	676

TABLE 2
The first few values of $f(n)$ and $g(n)$

n	$f(n)$	$g(n)$
0	1	1
1	5	3
2	16	10
3	48	31
4	143	94
5	427	283

Given some input $f^{(j)}$ many calls to several $\mathcal{F}^{(k)}$ are necessary to compute the results of $\mathcal{F}^{(j)}(f^{(j)})$. The term "recursion depth" is frequently used in computer science to measure the complexity of recursive procedures. Consider a counter which is incremented each time a recursive call is issued and decremented whenever a procedure is finished. This counter, plotted over the time axis, gives an impression how "complex" the computation is. The maximum number which this counter achieves, is usually called *recursion depth.*

Recursion depth is also very important for our treatment, thus it is formally defined in the rest of this section. In particular, $p^{(j)}$-recursion depth is of great importance. This takes only into account recursive calls to $p^{(j)}$ thereby ignoring the other procedures of the indirect recursive procedure system.

First we formally define sets of necessary parameter values to compute $p^{(j)}(f^{(j)})$ (Defs. 2.3 and 2.4). After that we define a multiset of all parameter values of all recursive procedure calls before a recursive call to $p^{(j)}$ is issued (Def. 2.5).

This enables us to define the $p^{(j)}$-successors of $f^{(j)}$ (Def. 2.6), which is the multiset of all parameter values $\overline{f}^{(j)}$ that are used for recursive calls to $p^{(j)}(\overline{f}^{(j)})$ while computing $p^{(j)}(f^{(j)})$, where we assume that between the call to $p^{(j)}(f^{(j)})$ and the call to $p^{(j)}(\overline{f}^{(j)})$ no other call to $p^{(j)}$ is issued. Several other procedures of the indirect recursive procedure system, however, can be called between the calls to $p^{(j)}(f^{(j)})$ and $p^{(j)}(\overline{f}^{(j)})$, respectively.

The $p^{(j)}$-successors can be used to partition the whole parameter space (Def. 2.8 and Lemma 2.1) in such a way that we can define $p^{(j)}$-recursion depth. It is important to note that our approach is not applicable if the "naive" version of recursion depth is used; $p^{(j)}$-recursion depth is unavoidable. A straight-forward generalization of [BL96] using "naive" recursion depth simply does not work.

For the rest of the paper some form of monotonical properties is extremely important. As a basis we define *monotonical recursive procedures*, a concept which builds on our definition of $p^{(j)}$-recursion depth (Def. 2.10).

Definition 2.3. We define a multiset $\mathcal{R}^{(\mathcal{P})}(f^{(j)}) \subseteq \mathcal{F}$, $f^{(j)} \in \mathcal{F}^{(j)}$ by $f^{(k)} \in \mathcal{R}^{(\mathcal{P})}(f^{(j)})$ iff $p^{(k)}(f^{(k)})$ is directly called in order to compute $p^{(j)}(f^{(j)})$. $\mathcal{R}^{(\mathcal{P})}(f^{(j)})$ is called the set of direct successors of $f^{(j)}$. If no $p \in \mathcal{P}$ is called directly to compute $p^{(j)}(f^{(j)})$, the set $\mathcal{R}^{(\mathcal{P})}(f^{(j)}) = \emptyset$, i.e., it is empty.

Definition 2.4. We define a sequence of multisets $\mathcal{R}_k^{(\mathcal{P})}(f^{(j)})$ by

$$\mathcal{R}_0^{(\mathcal{P})}(f^{(j)}) = \{f^{(j)}\}$$
$$\mathcal{R}_{k+1}^{(\mathcal{P})}(f^{(j)}) = \left\{\overline{f}^{(j)} \,\middle|\, \overline{f}^{(j)} \in \mathcal{R}^{(\mathcal{P})}(g^{(j)}) \text{ where } g^{(j)} \in \mathcal{R}_k^{(\mathcal{P})}(f^{(j)})\right\}$$

and we define the multiset $\mathcal{R}_*^{(\mathcal{P})}(f^{(j)})$ by

$$\mathcal{R}_*^{(\mathcal{P})}(f^{(j)}) = \bigcup_{k\geq 0} \mathcal{R}_k^{(j)}(f^{(j)}).$$

We call $\mathcal{R}_*^{(\mathcal{P})}(f^{(j)})$ the set of *necessary parameter values* to compute $p^{(j)}(f^{(j)})$.

DEFINITION 2.5. We define a sequence of multisets $\mathcal{Q}_i(f^{(j)})$, $f^{(j)} \in \mathcal{F}^{(j)}$ by

$$\mathcal{Q}_0(f^{(j)}) = \{f^{(j)}\}$$
$$\mathcal{Q}_{i+1}(f^{(j)}) = \{g|\ g \in \mathcal{R}^{(\mathcal{P})}(f^{(k)}) \setminus \mathcal{F}^{(j)} \text{ where } f^{(k)} \in \mathcal{Q}_i(f^{(j)})\Big\}$$

and we define the multiset $\mathcal{Q}_*(f^{(j)})$ by

$$\mathcal{Q}_*(f^{(j)}) = \bigcup_{i \geq 0} \mathcal{Q}_i(f^{(j)}).$$

$\mathcal{Q}_*(f^{(j)})$ is the multiset of the parameter values of all recursive procedure calls before a recursive call to $p^{(j)}$ is issued.

Remark. Note that $\mathcal{Q}_*(f^{(j)})$ is a finite multiset because $p^{(j)}$ is well-defined.

DEFINITION 2.6. For some $p^{(j)} \in \mathcal{P}$ we define a multiset $\mathcal{R}^{(j)}(f^{(j)}) \subseteq \mathcal{F}^{(j)}$, $f^{(j)} \in \mathcal{F}^{(j)}$ by $\overline{f}^{(j)} \in \mathcal{R}^{(j)}(f^{(j)})$ iff there exists some $g \in \mathcal{Q}_*(f^{(j)})$ such that $\overline{f}^{(j)} \in \mathcal{R}^{(\mathcal{P})}(g)$.

$\mathcal{R}^{(j)}(f^{(j)})$ is called the set of $p^{(j)}$-successors of $f^{(j)}$.

Remark. Concentrating on $p^{(j)}$, the multiset $\mathcal{R}^{(j)}(f^{(j)})$ contains all parameter values of recursive calls to $p^{(j)}$ which are issued directly by $p^{(j)}$ or indirectly after a recursive call-chain by some other recursive procedure of $\mathcal{P}$.

DEFINITION 2.7. We define a sequence of multisets $\mathcal{R}_k^{(j)}(f^{(j)})$ by

$$\mathcal{R}_0^{(j)}(f^{(j)}) = \{f^{(j)}\}$$
$$\mathcal{R}_{k+1}^{(j)}(f^{(j)}) = \left\{\overline{f}^{(j)} \,\middle|\, \overline{f}^{(j)} \in \mathcal{R}^{(j)}(g^{(j)}) \text{ where } g^{(j)} \in \mathcal{R}_k^{(j)}(f^{(j)})\right\}$$

and we define the multiset $\mathcal{R}_*^{(j)}(f^{(j)})$ by

$$\mathcal{R}_*^{(j)}(f^{(j)}) = \bigcup_{k \geq 0} \mathcal{R}_k^{(j)}(f^{(j)}).$$

We call $\mathcal{R}_*^{(j)}(f^{(j)})$ the set of *necessary $f^{(j)}$-parameter values* to compute $p^{(j)}(f^{(j)})$.

DEFINITION 2.8. We define a sequence of sets $\mathcal{F}_k^{(j)}$ inductively by

1. $\mathcal{F}_0^{(j)}$ contains the values of $\mathcal{F}^{(j)}$ which terminate the $p^{(j)}$-recursion,[†] i.e.,

$$\mathcal{F}_0^{(j)} = \left\{ f^{(j)} \in \mathcal{F}^{(j)} \,\middle|\, \mathcal{R}^{(j)}(f^{(j)}) = \emptyset \right\}$$

2. Let $\mathcal{F}_0^{(j)}, \ldots, \mathcal{F}_k^{(j)}$ be defined. Then we define $\mathcal{F}_{k+1}^{(j)}$ by

$$\mathcal{F}_{k+1}^{(j)} = \left\{ f^{(j)} \in \mathcal{F}^{(j)} \setminus \bigcup_{i=0}^{k} \mathcal{F}_i^{(j)} \,\middle|\, \mathcal{R}^{(j)}(f^{(j)}) \subseteq \bigcup_{i=0}^{k} \mathcal{F}_i^{(j)} \right\}.$$

LEMMA 2.1. *We have* $\bigcup_{k\geq 0} \mathcal{F}_k^{(j)} = \mathcal{F}^{(j)}$.

Proof. By definition we clearly have $\bigcup_{k\geq 0} \mathcal{F}_k^{(j)} \subseteq \mathcal{F}^{(j)}$.

On the other hand assume that there exists some $f^{(j)} \in \mathcal{F}^{(j)}$ for which $f^{(j)} \notin \bigcup_{k\geq 0} \mathcal{F}_k^{(j)}$ holds.

Now $\mathcal{R}^{(j)}(f^{(j)})$ contains at least one element, say $\overline{f}^{(j)}$, which is not contained in $\bigcup_{k\geq 0} \mathcal{F}_k^{(j)}$. The same argument applies to $\mathcal{R}^{(j)}(\overline{f}^{(j)})$ and so on. Thus $p^{(j)}$ is not well-defined. Hence $\mathcal{F}^{(j)} \subseteq \bigcup_{k\geq 0} \mathcal{F}_k^{(j)}$. ∎

COROLLARY 2.1. *By definition and by Lemma 2.1 we see that the sequence* $\mathcal{F}_k^{(j)}$ *partitions the set* $\mathcal{F}^{(j)}$*, i.e., for each* $f^{(j)} \in \mathcal{F}^{(j)}$ *holds that there exists exactly one* $k \in \mathbb{N}$ *such that* $f^{(j)} \in \mathcal{F}_k^{(j)}$ *and* $f^{(j)} \notin \mathcal{F}_i^{(j)}$ *for all* $i \neq k$*. Thus the* $\mathcal{F}_k^{(j)}$ *are* equivalence classes.

DEFINITION 2.9. Let $f^{(j)} \in \mathcal{F}^{(j)}$ and let k be such that $f^{(j)} \in \mathcal{F}_k^{(j)}$, then k is called the *$p^{(j)}$-recursion depth* of $p^{(j)}(f^{(j)})$. We write $k = \mathrm{recdep}(f^{(j)})$. For $f^{(j)}, g^{(j)} \in \mathcal{F}^{(j)}$, we write $f^{(j)} \approx g^{(j)}$ iff $\mathrm{recdep}(f^{(j)}) = \mathrm{recdep}(g^{(j)})$.

DEFINITION 2.10. An indirect recursive procedure $p^{(j)}$ is called *monotonical* if for all $f_k^{(j)} \in \mathcal{F}_k^{(j)}$ and for $f_i^{(j)} \in \mathcal{F}_i^{(j)}$, $0 \leq i < k$, we have $f_i^{(j)} \prec f_k^{(j)}$, where "$\prec$" is a suitable binary relation that satisfies for all $f_1^{(j)}, f_2^{(j)}, f_3^{(j)} \in \mathcal{F}^{(j)}$

1. either $f_1^{(j)} \prec f_2^{(j)}$ or $f_2^{(j)} \prec f_1^{(j)}$ or $f_1^{(j)} \approx f_2^{(j)}$ and
2. if $f_1^{(j)} \prec f_2^{(j)}$ and $f_2^{(j)} \prec f_3^{(j)}$, then $f_1^{(j)} \prec f_3^{(j)}$.

We write $f_1^{(j)} \preceq f_2^{(j)}$ if either $f_1^{(j)} \prec f_2^{(j)}$ or $f_1^{(j)} \approx f_2^{(j)}$.

Remark. If $p^{(j)}$ is a monotonical indirect recursive procedure, then $\overline{f}^{(j)} \prec f^{(j)}$ for all $\overline{f}^{(j)} \in \mathcal{R}^{(j)}(f^{(j)})$.

[†]Note that this does not mean that the overall recursion is terminated, rather $p^{(j)}$ can be the root of a recursive call-chain involving some other recursive procedures of $\mathcal{P}$.

EXAMPLE 2.1. In the following we use superscripts $^{(a)}$ and $^{(b)}$ to distinguish between the entities related to procedure a and b. We obtain $\mathcal{F}^{(a)} = \mathbb{N}$, $\mathcal{F}^{(b)} = \mathbb{N}$ and

$$\begin{aligned}
\mathcal{Q}_0(n^{(a)}) &= \{n^{(a)}\},\\
\mathcal{Q}_1(n^{(a)}) &= \{(n^{(a)} - 1)^{(b)}\},\\
\mathcal{Q}_2(n^{(a)}) &= \emptyset,\\
\mathcal{Q}_*(n^{(a)}) &= \{n^{(a)}, (n^{(a)} - 1)^{(b)}\},\\
\mathcal{Q}_0(n^{(b)}) &= \{n^{(b)}\},\\
\mathcal{Q}_1(n^{(b)}) &= \{(n^{(b)} - 1)^{(a)}\},\\
\mathcal{Q}_2(n^{(b)}) &= \emptyset,\\
\mathcal{Q}_*(n^{(b)}) &= \{n^{(b)}, (n^{(b)} - 1)^{(a)}\}
\end{aligned}$$

for $n^{(a)} \geq 1$ and $n^{(b)} \geq 1$ respectively. Furthermore

$$\begin{aligned}
\mathcal{R}^{(a)}(0^{(a)}) &= \mathcal{R}^{(a)}(1^{(a)}) = \emptyset,\\
\mathcal{R}^{(a)}(n^{(a)}) &= \{(n-2)^{(a)}\},\\
\mathcal{R}^{(b)}(0^{(b)}) &= \mathcal{R}^{(b)}(1^{(b)}) = \emptyset,\\
\mathcal{R}^{(b)}(n^{(b)}) &= \{(n-2)^{(b)}\}
\end{aligned}$$

for $n^{(a)} \geq 2$ and $n^{(b)} \geq 2$ and

$$\begin{aligned}
\mathcal{F}_0^{(a)} &= \{0^{(a)}, 1^{(a)}\},\\
\mathcal{F}_k^{(a)} &= \{2k^{(a)}, 2k+1^{(a)}\},\\
\mathcal{F}_0^{(b)} &= \{0^{(b)}, 1^{(b)}\},\\
\mathcal{F}_k^{(b)} &= \{2k^{(b)}, 2k+1^{(b)}\}.
\end{aligned}$$

Thus we have

$$\begin{aligned}
\text{recdep}(n^{(a)}) &= \lfloor n^{(a)}/2 \rfloor \quad \text{and}\\
\text{recdep}(n^{(b)}) &= \lfloor n^{(b)}/2 \rfloor.
\end{aligned}$$

The "$\prec$"-relation for $\mathcal{F}^{(a)}$ and $\mathcal{F}^{(b)}$ is the "$<$"-relation for integers. ■

EXAMPLE 2.2. In the following we use superscripts $^{(f)}$ and $^{(g)}$ to distinguish between the entities related to procedure f and g. We obtain $\mathcal{F}^{(f)} = \mathbb{N}$, $\mathcal{F}^{(g)} = \mathbb{N}$ and

$$\begin{aligned}
\mathcal{R}(n^{(f)}) &= \{n^{(g)}, (n-1)^{(f)}\}\\
\mathcal{R}(n^{(g)}) &= \{(n-1)^{(f)}\} \cup \bigcup_{i^{(g)}=0^{(g)}}^{(n-1)^{(g)}} \{i^{(g)}\}.
\end{aligned}$$

Hence

$$\mathcal{R}_*(n^{(f)}) = \bigcup_{i^{(f)}=0^{(f)}}^{n^{(f)}} \{i^{(f)}[(3^{n-i}+1)/2]\} \cup \bigcup_{i^{(g)}=1^{(g)}}^{n^{(g)}} \{i^{(g)}[3^{n-i}]\} \cup \{0^{(g)}[(3^n-1)/2]\}$$

$$\mathcal{R}_*(n^{(g)}) = \bigcup_{i^{(f)}=0^{(f)}}^{(n-1)^{(f)}} \{i^{(f)}[3^{n-i-1}]\} \cup \{n^{(g)}\} \cup \bigcup_{i^{(g)}=1^{(g)}}^{(n-1)^{(g)}} \{i^{(g)}[2\cdot 3^{n-i-1}]\} \cup \{0^{(g)}[3^{n-1}]\}$$

and

$$\mathcal{Q}_*(n^{(f)}) = \{n^{(f)}\} \cup \{n^{(g)}\} \cup \bigcup_{i^{(g)}=0^{(g)}}^{(n-1)^{(g)}} \{i^{(g)}[2^{n-i-1}]\}$$

$$\mathcal{Q}_*(n^{(g)}) = \{n^{(g)}\} \cup \bigcup_{i^{(f)}=0^{(f)}}^{(n-1)^{(f)}} \{i^{(f)}\},$$

where the numbers within the square brackets indicate how often the corresponding element is contained in the multiset.

Furthermore we derive

$$\mathcal{R}_*^{(f)}(n^{(f)}) = \{n^{(f)}\} \cup \{(n-1)^{(f)}[2]\} \cup \bigcup_{i^{(f)}=0^{(f)}}^{(n-2)^{(f)}} \{i^{(f)}[2^{n-i-2}]\},$$

$$\mathcal{R}_*^{(g)}(n^{(g)}) = \{n^{(g)}\} \cup \bigcup_{i^{(g)}=1^{(g)}}^{(n-1)^{(g)}} \{i^{(g)}[2]\} \cup \{0^{(g)}\}$$

and

$$\mathcal{F}_k^{(f)} = \{k^{(f)}\}$$
$$\mathcal{F}_k^{(g)} = \{k^{(g)}\}.$$

Thus we obtain

$$\text{recdep}(n^{(f)}) = n$$
$$\text{recdep}(n^{(g)}) = n$$

and the "$\prec$"-relation for $\mathcal{F}^{(f)}$ and $\mathcal{F}^{(g)}$ is the "$<$"-relation for integers. ■

3. COMPUTATIONAL MODEL AND SPACE AND TIME EFFORT

The time effort $\mathcal{T}$ of an indirect recursive procedure $p \in \mathcal{P}$ is a recursive function

$$\mathcal{T} : \mathcal{F} \to \mathbb{R}$$

or

$$\mathcal{T} : \mathcal{F} \to \mathbb{N}.$$

If time is measured in integer multiples of say micro-seconds or CPU clock ticks, one can use an integer valued function $\mathcal{T}$ instead of a real valued one.

In a similar way $\mathcal{S}$, the space effort of $\mathcal{P}$, is a recursive function

$$\mathcal{S} : \mathcal{F} \to \mathbb{N},$$

where space is measured in multiples of bits or bytes.

Both functions $\mathcal{T}$ and $\mathcal{S}$ are defined recursively depending on the source code of $\mathcal{P}$. How the recurrence relations for $\mathcal{T}$ and $\mathcal{S}$ are derived from the source code and which statements are allowed in the source code of $\mathcal{P}$, is described in the following subsection.

3.1. Recurrence Relations for $\mathcal{S}$ and $\mathcal{T}$

The source code of an indirect recursive procedure is considered to consist of

- simple segments of linear code, the performance of which is known a priori,
- if-statements,
- loops with known upper bounds of the number of iterations which can be derived at compile time, e.g. for-loops or discrete loops (cf. [Bli94]),[‡] and
- recursive calls.

In terms of a context-free grammar this is stated as follows

$$\begin{array}{rcl}
\text{code}(f) & ::= & \textbf{if } f \in \mathcal{F}_0 \textbf{ then } \text{nonrecursive}(f) \textbf{ else } \text{recursive}(f) \textbf{ end if} \\
\text{recursive}(f) & ::= & \text{seq}(f) \\
\text{seq}(f) & ::= & \text{statement}(f) \ \{\text{statement}(f)\} \\
\text{statement}(f) & ::= & \text{simple}(f) \ \mid \text{compound}(f) \ \mid \text{rproc}(\overline{p}(\overline{f})) \\
\text{compound}(f) & ::= & \text{ifs}(f) \ \mid \text{bloops}(f) \\
\text{ifs}(f) & ::= & \textbf{if } \text{cond}(f) \textbf{ then } \text{seq}(f) \textbf{ else } \text{seq}(f) \textbf{ end if} \\
\text{bloops}(f) & ::= & \textbf{loop } <\mathit{bound}(f)> \text{ seq}(f)
\end{array}$$

The syntax of *nonrecursive*(f) is defined exactly the same way but *rproc*$(\overline{p}(\overline{f}))$ is not part of *statement*(f). By $\overline{p}(\overline{f})$ we denote that procedure $\overline{p} \in \mathcal{P}$ is called with parameters $\overline{f}$.

We use these definitions to derive a recurrence relation for the time effort $\mathcal{T}$:

$$\mathcal{T}(f) = \tau[f \in \mathcal{F}_0] + \mathcal{T}[\mathit{nonrecursive}(f)] \quad \text{if } f \in \mathcal{F}_0,$$
$$\mathcal{T}(f) = \tau[f \in \mathcal{F}_0] + \mathcal{T}[\mathit{recursive}(f)] \quad \text{if } f \notin \mathcal{F}_0,$$

where the first τ-constant comes from the evaluating the condition whether f belongs to the terminating values or not and is known a priori; the nonrecursive term can be computed using the method described below, but without giving rise to a

[‡]This means that the number of iterations does not depend on the result of one or more recursive calls.

recurrence relation, and the recursive term can be determined by

$$\begin{aligned}
&\mathcal{T}[\mathit{recursive}(f)] = \mathcal{T}[\mathit{seq}(f)]\\
&\mathcal{T}[\mathit{seq}(f)] = \sum \mathcal{T}[\mathit{statement}(f)]\\
&\mathcal{T}[\mathit{ifs}(f)] = \mathcal{T}[\mathit{cond}(f)] + \begin{cases} \mathcal{T}[\mathit{seq}_{\mathtt{True}}(f)] & \text{if the condition evaluates to true,}\\ \mathcal{T}[\mathit{seq}_{\mathtt{False}}(f)] & \text{otherwise.}\end{cases}\\
&\mathcal{T}[\mathit{bloops}(f)] = <\!\mathit{bound}(f)\!>\mathcal{T}[\mathit{seq}(f)]\\
&\mathcal{T}[\mathit{simple}(f)] = \tau(\mathit{simple})\\
&\mathcal{T}[\mathit{rproc}(\overline{p}(\overline{f}))] = \mathcal{T}(\overline{f})
\end{aligned}$$

where $\tau(\mathit{simple})$ is known a priori.

Note that $<\!\mathit{bound}(f)\!>$ may depend on f, e.g. a for-loop with iterations depending on f.

Since we are dealing with stack space only, space is freed whenever a procedure call finishes. Thus for example the stack space used by two successive statements S_1 and S_2 is the maximum of the stack space of each of them. The recurrence relation for the stack space effort $\mathcal{S}$ is given by:

$$\begin{aligned}
&\mathcal{S}(f) = \mathcal{S}(\mathit{decl\_part}(f)) + \max(\sigma[f \in \mathcal{F}_0], \mathcal{S}[\mathit{nonrecursive}(f)]) \quad \text{if } f \in \mathcal{F}_0,\\
&\mathcal{S}(f) = \mathcal{S}(\mathit{decl\_part}(f)) + \max(\sigma[f \in \mathcal{F}_0], \mathcal{S}[\mathit{recursive}(f)]) \quad \text{if } f \notin \mathcal{F}_0,
\end{aligned}$$

where the first σ-constant is known a priori, the nonrecursive term can be computed in a similar way as shown below, but without giving rise to a recurrence relation, and the recursive term is determined by

$$\begin{aligned}
&\mathcal{S}[\mathit{recursive}(f)] = \mathcal{S}[\mathit{seq}(f)]\\
&\mathcal{S}[\mathit{seq}(f)] = \max\left(\mathcal{S}[\mathit{statement}(f)]\right)\\
&\mathcal{S}[\mathit{ifs}(f)] = \begin{cases} \max\left(\mathcal{S}[\mathit{cond}(f)], \mathcal{S}[\mathit{seq}_{\mathtt{True}}(f)]\right) & \text{if the condition evaluates to true,}\\ \max\left(\mathcal{S}[\mathit{cond}(f)], \mathcal{S}[\mathit{seq}_{\mathtt{False}}(f)]\right) & \text{otherwise.}\end{cases}\\
&\mathcal{S}[\mathit{bloops}(f)] = \max(\mathcal{S}[\mathit{seq}(f)])\\
&\mathcal{S}[\mathit{simple}(f)] = \sigma(\mathit{simple})\\
&\mathcal{S}[\mathit{rproc}(\overline{p}(\overline{f}))] = \mathcal{S}(\overline{f})
\end{aligned}$$

where $\sigma(\mathit{simple})$ is known a priori and $\mathcal{S}(\mathit{decl\_part}(f))$ denotes the space effort of the declarative part of the recursive function, e.g. space used by locally declared variables. Note that the space effort of the declarative part may depend on f, since one can declare arrays of a size depending on f for example.

3.2. Monotonical Space and Time Effort

Given a $p^{(j)} \in \mathcal{P}$ and some actual parameters $f^{(j)} \in \mathcal{F}^{(j)}$, $\mathcal{T}(f^{(j)})$ and $\mathcal{S}(f^{(j)})$ can easily be determined at compile time. This can even be done if only upper and lower bounds of $f^{(j)}$ exist, e.g. $l^{(j)} \preceq f^{(j)} \preceq u^{(j)}$, $l^{(j)}u^{(j)} \in \mathcal{F}^{(j)}$, since $\max_{l^{(j)} \preceq f^{(j)} \preceq u^{(j)}} \mathcal{T}(f^{(j)})$ and $\max_{l^{(j)} \preceq f^{(j)} \preceq u^{(j)}} \mathcal{S}(f^{(j)})$ can be computed effectively.

Definition 3.1. If $f_1 \preceq f_2$ implies $\mathcal{S}(f_1) \leq \mathcal{S}(f_2)$ and $\mathcal{T}(f_1) \leq \mathcal{T}(f_2)$, we call the underlying indirect recursive procedure *globally space-monotonical* and *globally time-monotonical*, respectively.

Remark. Note that $f_1 \approx f_2$ implies $\mathcal{S}(f_1) = \mathcal{S}(f_2)$ and $\mathcal{T}(f_1) = \mathcal{T}(f_2)$, respectively.

There are two cases:

1. $\mathcal{S}$ and $\mathcal{T}$ can be shown to be monotonical at compile-time and
2. $\mathcal{S}$ and $\mathcal{T}$ can be solved at compile-time and the (non-recursive) solution can be proved to be monotonical.

In both cases we clearly have:

Theorem 3.1. *If p is globally space or time-monotonical, then*

$$\mathfrak{S}(l,u) := \max_{l \preceq f \preceq u} \mathcal{S}(f) = \max_{g \approx u} \mathcal{S}(g)$$

and

$$\mathfrak{T}(l,u) := \max_{l \preceq f \preceq u} \mathcal{T}(f) = \max_{g \approx u} \mathcal{T}(g),$$

respectively. ■

The difference between case (1.) and (2.) is that in case (2.) Theorem 3.1 can even be applied during runtime, e.g., when generic objects are instantiated (cf. e.g. [Ada95, ES90]), while in case (1.) for real-time applications Theorem 3.1 can only be applied at compile time, because case (1.) requires one or more recursive evaluations of $\mathcal{S}$ or $\mathcal{T}$.

If no proofs are available at compile time that p is globally space or time-monotonical, runtime tests can be performed. Of course this requires some overhead in computing the result of a recursive call.

In the following sections we will define "local" conditions. If these conditions hold, the underlying indirect recursive procedure is called *locally space* or *locally time-monotonical.* It will turn out that if an indirect recursive procedure is locally space (time) monotonical, then it is also globally space (time) monotonical. (It is worth noting that the converse is not true, i.e., if a certain indirect recursive procedure is globally space or time monotonical, it need not be locally space or time monotonical.)

Thus it suffices to prove that a certain indirect recursive procedure is locally space or time-monotonical, before Theorem 3.1 can be applied. This proof often is simpler than proving the corresponding global property.

If the local properties can be proved at compile time, Theorem 3.1 can be applied at compile time. If there is a (non-recursive) solution of $\mathcal{S}$ or $\mathcal{T}$ known and verified at compile time, Theorem 3.1 can also be applied at runtime.

In addition, the local properties can be checked at runtime, such that it is not necessary to have proofs at compile time. In this case an appropriate exception is raised at runtime when the runtime system finds that the local property does not hold in a particular case. Thus timing errors are shifted to runtime errors or in other words timing errors become testable.

The major advantages of local properties are that

- they can easily be proved at compile time and
- they are well-suited for real-time applications.

In the following sections we give several examples of how easy these proofs can be derived. We think that in many cases they can be found by a (smart) compiler. In general, proofs of global properties and solving recurrence relations are more difficult.

4. THE SPACE EFFORT OF INDIRECT RECURSIVE PROCEDURES

In this section we formally define sets of parameter values which are obtained during a call-chain from $f^{(j)}$ to $\overline{f}^{(j)}$ (Def. 4.2). This together with a measure for the stack space used by single procedures (Def. 4.1) allows to define the overall stack space (Defs. 4.3, 4.5, and 4.6).

Introducing a suitable *recursion digraph* and the term *locally space-monotonical* procedure, we can prove our main results on the space effort of indirect recursive procedures.

Definition 4.1. Let $p \in \mathcal{P}$ be an indirect recursive procedure. We define the function $\mathcal{D} : \mathcal{F} \to \mathbb{N}$ such that $\mathcal{D}(f)$ denotes the space being part of the declarative part of p if p is called with parameter f.

Definition 4.2. For all $\overline{f}^{(j)} \in \mathcal{R}^{(j)}(f^{(j)})$ we define the following sequence of sets (not multisets!)

$$\begin{aligned}\overline{\mathcal{O}}_0(f^{(j)},\overline{f}^{(j)}) &= \{(f^{(j)})\},\\ \overline{\mathcal{O}}_i(f^{(j)},\overline{f}^{(j)}) &= \{(\overline{o}_0,\overline{o}_1,\ldots,\overline{o}_{i-1},\overline{o}_i) \mid (\overline{o}_0,\overline{o}_1,\ldots,\overline{o}_{i-1}) \in \overline{\mathcal{O}}_{i-1}(f^{(j)},\overline{f}^{(j)}),\\ &\qquad \overline{o}_i \in \mathcal{R}^{(\mathcal{P})}(\overline{o}_{i-1}) \setminus \mathcal{F}^{(j)} \text{ and } \overline{f}^{(j)} \in \mathcal{R}_*^{(\mathcal{P})}(\overline{o}_i)\}, \text{ for } i \geq 1.\end{aligned}$$

Depending on these sets we define

$$\mathcal{O}_i(f^{(j)},\overline{f}^{(j)}) = \{(o_0,\ldots,o_i) \in \overline{\mathcal{O}}_i(f^{(j)},\overline{f}^{(j)}) \mid \overline{f}^{(j)} \in \mathcal{R}^{(\mathcal{P})}(o_i)\}$$

and

$$\mathcal{O}_*(f^{(j)},\overline{f}^{(j)}) = \bigcup_{i\geq 0} \mathcal{O}_i(f^{(j)},\overline{f}^{(j)})$$

In addition we define

$$\begin{aligned}
\overline{\mathcal{O}}_0(f^{(j)}, \perp) &= \{(f^{(j)})\}, \\
\overline{\mathcal{O}}_i(f^{(j)}, \perp) &= \{(\overline{o}_0, \ldots, \overline{o}_i) \mid (\overline{o}_0, \ldots, \overline{o}_{i-1}) \in \overline{\mathcal{O}}_{i-1}(f^{(j)}, \perp), \\
&\qquad \overline{o}_i \in \mathcal{R}^{(\mathcal{P})}(\overline{o}_{i-1}) \setminus \mathcal{F}^{(j)}\}, \text{ for } i > 1, \\
\mathcal{O}_i(f^{(j)}, \perp) &= \{(o_0, \ldots, o_i) \in \overline{\mathcal{O}}_i(f^{(j)}, \perp) \mid \mathcal{R}^{(\mathcal{P})}(o_i) = \emptyset\},
\end{aligned}$$

and

$$\mathcal{O}_*(f^{(j)}, \perp) = \bigcup_{i \geq 0} \mathcal{O}_i(f^{(j)}, \perp).$$

Remark. $\mathcal{O}_*(f^{(j)}, \overline{f}^{(j)})$ contains all recursive call-chains from $f^{(j)}$ to $\overline{f}^{(j)}$. If there is a call-chain which does not include some $\overline{f}^{(j)} \in \mathcal{F}^{(j)}$, it is in $\mathcal{O}_*(f^{(j)}, \perp)$.

Definition 4.3. We define

$$\Psi(f^{(j)}, g^{(j)}) = \max_{(o_0, o_1, \ldots) \in \mathcal{O}_*(f^{(j)}, g^{(j)})} \sum_{i \geq 1} \mathcal{D}(o_i).$$

for $g^{(j)} \in \mathcal{R}^{(j)}(f^{(j)})$ or $g^{(j)} = \perp$.

Note that the sum is over $i \geq 1$ only, such that $o_0 = f^{(j)}$ is not taken into account.

With these definitions $\mathcal{S}(f^{(j)})$ fulfils

$$\begin{aligned}
\mathcal{S}(f^{(j)}) = \mathcal{D}(f^{(j)}) + \max \Big(&\Psi(f^{(j)}, \perp), \Psi(f^{(j)}, \overline{f}_1^{(j)}) + \mathcal{S}(\overline{f}_1^{(j)}), \ldots, \\
&\Psi(f^{(j)}, \overline{f}_m^{(j)}) + \mathcal{S}(\overline{f}_m^{(j)}) \Big)
\end{aligned}$$

where $\mathcal{R}^{(j)}(f^{(j)}) = \{\overline{f}_1^{(j)}, \ldots \overline{f}_m^{(j)}\}$.

Remark. Evaluating $\mathcal{S}(f)$ for recursive functions increases the height of the stack if the recursive call is part of an expression, because both the recursive call and evaluating the expression use stack space. This, however, can be avoided by introducing temporary variables in the declarative part of the recursive function[§]. These temporary variables are assigned the results of the recursive calls and are used inside the expression instead of the recursive calls themselves.

Definition 4.4. For each $f^{(j)} \in \mathcal{F}^{(j)}$ the *recursion digraph* $\mathcal{G}(f^{(j)})$ is defined by the set of vertices $V = \mathcal{R}_*^{(j)}(f^{(j)})$ and the set of edges $E = \{(g^{(j)}, \overline{g}^{(j)}) \mid g^{(j)}, \overline{g}^{(j)} \in$

[§]Note that this can be done at compile time!

V and $\overline{g}^{(j)} \in \mathcal{R}^{(j)}(g^{(j)})\}$. Each vertex $\overline{g}^{(j)}$ is weighted by $\mathcal{D}(\overline{g}^{(j)})$ and each edge from $g^{(j)}$ to $\overline{g}^{(j)}$ is weighted by $\Psi(g^{(j)}, \overline{g}^{(j)})$.

Remark. Let $\mathcal{M}$ denote the path from $f^{(j)}$ to some $f_0^{(j)} \in \mathcal{F}_0^{(j)}$, $f_0^{(j)} \in \mathcal{R}_*^{(j)}(f^{(j)})$ with maximum weight $W(f^{(j)}) = \sum_{g^{(j)}} \mathcal{D}(g^{(j)}) + \sum_{e=(g^{(j)},\overline{g}^{(j)})} \Psi(g^{(j)}, \overline{g}^{(j)})$, where $g^{(j)}$ runs through all vertices of $\mathcal{M}$ and e runs through all edges of $\mathcal{M}$. Then $W(f^{(j)})$ is equal to $\mathcal{S}(f^{(j)})$.

Remark. Using $\mathcal{G}(f^{(j)})$, for fixed $f^{(j)}$ the quantity $\mathcal{S}(f^{(j)})$ can be computed off-line at compile time in $O(\|V\| + \|E\|)$ time (cf. e.g. [Meh84a]).

Definition 4.5. We define $\mathcal{B}(f^{(j)})$ by

$$\mathcal{B}(f^{(j)}) = \max_{\overline{f}^{(j)} \in \mathcal{R}^{(j)}(f^{(j)})} \mathcal{D}(\overline{f}^{(j)}) + \Psi(f^{(j)}, \overline{f}^{(j)}).$$

Remark. In terms of the recursion digraph $\mathcal{G}(f^{(j)})$, $\mathcal{B}(f^{(j)})$ is the maximum of weights of all edges leaving vertex $f^{(j)}$ plus the weight of the successor.

Definition 4.6. Let $p^{(j)}$ be a monotonical indirect recursive procedure. We define $\mathcal{N}^{(j)} : \mathcal{F}^{(j)} \to \mathcal{F}^{(j)}$ to be a function such that $\mathcal{N}^{(j)}(f^{(j)}) = f_{\max}^{(j)}$, where $f_{\max}^{(j)}$ is such that $f_{\max}^{(j)} \in \mathcal{R}^{(j)}(f^{(j)})$, $\mathcal{D}(f_{\max}^{(j)}) + \Psi(f^{(j)}, f_{\max}^{(j)}) = \mathcal{B}(f^{(j)})$, i.e., it is maximized, and $\operatorname{recdep}(f_{\max}^{(j)}) = \operatorname{recdep}(f^{(j)}) - 1$.

Remark. This means that $\mathcal{N}^{(j)}(f^{(j)})$ is that successor of $f^{(j)}$ in the recursion digraph $\mathcal{G}(f^{(j)})$ which needs maximum space.

Remark. Note that $\mathcal{N}^{(j)}$ may be not defined for some $f^{(j)} \in \mathcal{F}^{(j)}$, e.g. compare Example 1.2 below.

Definition 4.7. We call a monotonical indirect recursive procedure $p^{(j)}$ *locally space-monotonical* if $\mathcal{N}^{(j)}(f^{(j)})$ is defined for all $f^{(j)} \in \mathcal{F}^{(j)}$ and if for all $f^{(j)} \in \mathcal{F}^{(j)}$, $f_1^{(j)} \prec f_2^{(j)}$ implies $\mathcal{B}(f_1^{(j)}) \le \mathcal{B}(f_2^{(j)})$ and, if $f_1^{(j)} \approx f_2^{(j)}$ and $\mathcal{B}(f_1^{(j)}) \le \mathcal{B}(f_2^{(j)})$ implies $\mathcal{B}(\mathcal{N}^{(j)}(f_1^{(j)})) \le \mathcal{B}(\mathcal{N}^{(j)}(f_2^{(j)}))$.

THEOREM 4.1. *If $p^{(j)}$ is a locally space-monotonical indirect recursive procedure, then we have for all $f^{(j)}$*

$$\mathcal{S}(f^{(j)}) = \sigma_0 + \sum_{0 \le k < \mathrm{recdep}(f^{(j)})} \mathcal{B}(\mathcal{N}^{(j)(k)}(f^{(j)})),$$

where $\mathcal{N}^{(j)(k)}$ is the kth iterate of $\mathcal{N}^{(j)}$ ($\mathcal{N}^{(j)(k+1)}(f^{(j)}) = \mathcal{N}^{(j)}(\mathcal{N}^{(j)(k)}(f^{(j)}))$ for $k \ge 0$) and for simplicity $\mathcal{N}^{(j)(0)}(f^{(j)}) = f^{(j)}$.

Proof. Theorem 4.1 is proved if we can show that in $\mathcal{G}(f^{(j)})$ no path $\mathcal{M}'$ exists such that $W(\mathcal{M}') > W(\mathcal{M})$.

Assume on the contrary that $\mathcal{M}'$ exists. This means we must have a situation like that depicted in Figure 1. The path along $(f^{(j)}, \ldots, v_0, v_1, \ldots, v_r, w, \ldots, f_0^{(j)})$, $f_0^{(j)} \in \mathcal{F}_0^{(j)}$ is identical to $\mathcal{M}$. The path along $(f^{(j)}, \ldots, v_0, x_1, \ldots, x_s, w, \ldots, \overline{f}_0^{(j)})$, $\overline{f}_0^{(j)} \in \mathcal{F}_0^{(j)}$ is denoted by $\mathcal{M}'$.

By definition we have $\mathcal{B}(v_1) \ge \mathcal{B}(x_1)$. Thus

$$\mathcal{B}(\mathcal{N}^{(j)}(v_1)) = \mathcal{B}(v_2) \ge \mathcal{B}(\mathcal{N}^{(j)}(x_1)) \ge \mathcal{B}(x_2).$$

Continuing this procedure, we get $\mathcal{B}(v_3) \ge \mathcal{B}(x_3)$, and so on.

Because of Definition 4.6 we must have $r \ge s$ since $\mathrm{recdep}(v_i) = \mathrm{recdep}(v_{i+1}) + 1$. Hence we obviously have a contradiction. ■

The following lemma is needed in order to prove our main result on the space effort of recursive procedures, which is given in Theorem 4.2.

LEMMA 4.1. *If $p^{(j)}$ is locally space-monotonical and $f_1^{(j)} \prec f_2^{(j)}$, $f_1^{(j)}, f_2^{(j)} \in \mathcal{F}^{(j)}$, then*

$$\mathcal{S}(f_1^{(j)}) \le \mathcal{S}(f_2^{(j)}).$$

Proof. Clearly we have for all $f^{(j)}$ and for all $0 \le k < \mathrm{recdep}(f_1^{(j)})$

$$\mathcal{N}^{(j)(k)}(f^{(j)}, f_1^{(j)}) \prec \mathcal{N}^{(j)(k)}(f^{(j)}, f_2^{(j)}).$$

Hence we also have

$$\mathcal{B}(f^{(j)}, \mathcal{N}^{(j)(k)}(f^{(j)}, f_1^{(j)})) \le \mathcal{B}(f^{(j)}, \mathcal{N}^{(j)(k)}(f^{(j)}, f_2^{(j)}))$$

for all $0 \le k < \mathrm{recdep}(f_1^{(j)})$.

Thus we obtain

$$\mathcal{S}(f_1^{(j)}) \le \mathcal{S}(f_2^{(j)})$$

and the lemma is proved. ■

THEOREM 4.2. *If $p^{(j)}$ is locally space-monotonical, then*

$$\mathfrak{S}(l^{(j)}, u^{(j)}) = \max_{l^{(j)} \preceq f^{(j)} \preceq u^{(j)}} \mathcal{S}(f^{(j)}) = \max_{g^{(j)} \approx u^{(j)}} \mathcal{S}(g^{(j)}).$$

Proof. By virtue of Lemma 4.1,

$$\mathcal{S}(f^{(j)}) \le \mathcal{S}(u^{(j)}) \quad \text{for all } l^{(j)} \preceq f^{(j)} \prec u^{(j)}.$$

It remains to take into account all $g^{(j)} \approx u^{(j)}$. Thus the theorem is proved. ■

Remark. Note that this section is a non-trivial generalization of the corresponding section in [BL96]. Specifically $\mathcal{B}$ simplifies to $\mathcal{D}$ for direct recursion.

Moreover, Theorem 4.1 correctly mirrors programming languages that allow block-statements which contain local declarations (cf. e.g. [Ada95]).

Example 4.1. For this example we will assume that $\mathcal{D}(0^{(a)}) = \mathcal{D}(0^{(b)}) = \sigma_0$ and $\mathcal{D}(n^{(a)}) = \mathcal{D}(n^{(b)}) = \sigma_1$ for $n^{(a)} \ge 1$ and $n^{(b)} \ge 1$ respectively. We clearly have

$$\begin{aligned} \mathcal{O}_*(1^{(a)}, \perp) &= \{(1^{(a)}, 0^{(b)})\} \\ \mathcal{O}_*(n^{(a)}, (n-2)^{(a)}) &= \{(n^{(a)}, (n^{(a)} - 1)^{(b)})\} \\ \mathcal{O}_*(1^{(b)}, \perp) &= \{(1^{(b)}, 0^{(a)})\} \\ \mathcal{O}_*(n^{(b)}, (n-2)^{(b)}) &= \{(n^{(b)}, (n^{(b)} - 1)^{(a)})\} \end{aligned}$$

Hence

$$\begin{aligned} \Psi(1^{(a)}, \perp) &= \sigma_0 \\ \Psi(n^{(a)}, (n-2)^{(a)}) &= \sigma_1 \\ \Psi(1^{(b)}, \perp) &= \sigma_0 \\ \Psi(n^{(b)}, (n-2)^{(b)}) &= \sigma_1 \end{aligned}$$

and

$$\begin{aligned} \mathcal{B}(n^{(a)}) &= \sigma_1 + \Psi(n^{(a)}, (n-2)^{(a)}) = 2\sigma_1 \\ \mathcal{B}(n^{(b)}) &= 2\sigma_1. \end{aligned}$$

Thus both $a(n)$ and $b(n)$ are locally space-monotonical procedures.

However we can show more. Since

$$\begin{aligned} \mathcal{N}^{(a)}(n^{(a)}) &= n^{(a)} - 2 \quad \text{and} \\ \mathcal{N}^{(b)}(n^{(b)}) &= n^{(b)} - 2 \quad \text{for } n \ge 2, \end{aligned}$$

we obtain

$$\mathcal{S}(n^{(a)}) = \mathcal{S}(n^{(b)}) = \sigma_0 + n \cdot \sigma_1. \quad ■$$

Example 4.2. We assume that $\mathcal{D}(n^{(f)}) = \sigma_1^{(f)}$ and $\mathcal{D}(n^{(g)}) = \sigma_1^{(g)}$ for $n^{(f)} \ge 1$ and $n^{(g)} \ge 1$, respectively, and $\mathcal{D}(0^{(f)}) = \sigma_0^{(f)}$, $\mathcal{D}(0^{(g)}) = \sigma_0^{(g)}$. We clearly have for some $a_i > 0$

$$\begin{aligned}
\mathcal{O}_*(n^{(f)}, \overline{n}^{(f)}) &= \{(n^{(f)}, n^{(g)} - 1, n^{(g)} - a_1, n^{(g)} - (a_1 + a_2), \dots, \overline{n}^{(g)} + 1)\},\\
\mathcal{O}_*(n^{(g)}, \overline{n}^{(g)}) &= \{(n^{(g)}, n^{(f)} - 1, n^{(f)} - 2, \dots, \overline{n}^{(f)} + 1)\},\\
\mathcal{O}_*(n^{(f)}, \bot) &= \{(n^{(f)}, (n - a_1)^{(g)}, (n - (a_1 + a_2))^{(g)}, \dots, 0^{(g)})\}, \text{ and}\\
\mathcal{O}_*(n^{(g)}, \bot) &= \{(n^{(g)}, (n - 1)^{(f)}, (n - 2)^{(f)}, \dots, 0^{(f)})\} \quad \text{for all } n^{(g)} \geq 1.
\end{aligned}$$

Thus

$$\begin{aligned}
\Psi(n^{(f)}, \overline{n}^{(f)}) &= \max_{(o_0, o_1, \dots) \in \mathcal{O}_*(n^{(f)}, \overline{n}^{(f)})} \sum_{i \geq 1} \mathcal{D}(o_i)\\
&= \sigma_1^{(g)}(n - \overline{n}),\\
\Psi(n^{(g)}, \overline{n}^{(g)}) &= \sigma_1^{(f)}(n - \overline{n} - 1),\\
\Psi(n^{(f)}, \bot) &= n \cdot \sigma_1^{(g)} + \sigma_0^{(g)},\\
\Psi(n^{(g)}, \bot) &= (n - 1) \cdot \sigma_1^{(f)} + \sigma_0^{(f)}.
\end{aligned}$$

and we obtain

$$\begin{aligned}
\mathcal{B}(n^{(f)}) &= \max_{\overline{n}^{(f)} \in \mathcal{R}^{(f)}(n^{(f)})} \mathcal{D}(\overline{n}^{(f)}) + \Psi(n^{(f)}, \overline{n}^{(f)})\\
&= \max_{\overline{n}^{(f)} \in \mathcal{R}^{(f)}(n^{(f)})} \mathcal{D}(\overline{n}^{(f)}) + \sigma_1^{(g)}(n - \overline{n})\\
&= \max(\sigma_0^{(f)} + \sigma_1^{(g)} \cdot n, \sigma_1^{(f)} + \sigma_1^{(g)} \cdot (n - 1))\\
&= \sigma_1^{(g)} \cdot (n - 1) + \max(\sigma_0^{(f)} + \sigma_1^{(g)}, \sigma_1^{(f)}),\\
\mathcal{B}(n^{(g)}) &= \max_{\overline{n}^{(g)} \in \mathcal{R}^{(g)}(n^{(g)})} \mathcal{D}(\overline{n}^{(g)}) + \Psi(n^{(g)}, \overline{n}^{(g)})\\
&= \max_{\overline{n}^{(g)} \in \mathcal{R}^{(g)}(n^{(g)})} \mathcal{D}(\overline{n}^{(g)}) + \sigma_1^{(f)}(n - \overline{n} - 1)\\
&= \max(\sigma_0^{(g)} + \sigma_1^{(f)} \cdot (n - 1), \sigma_1^{(g)} + \sigma_1^{(f)} \cdot (n - 2))\\
&= \sigma_1^{(f)} \cdot (n - 2) + \max(\sigma_0^{(g)} + \sigma_1^{(f)}, \sigma_1^{(g)}).
\end{aligned}$$

However, since recdep $\big(\mathcal{N}^{(f)}(n^{(f)})\big) = n - 1$ is fulfilled only for $\mathcal{N}^{(f)}(n^{(f)}) = n^{(f)} - 1$ and $\mathcal{B}(n^{(f)})$ is maximized for $0^{(f)}$, $\mathcal{N}^{(f)}$ is not defined for all $n^{(f)}$. The same applies to $\mathcal{N}^{(g)}$. Hence neither f nor g are locally space-monotonical.

The reason why Example 1.2 is not locally space-monotonical is very similar to the reason why Quicksort is not locally time-monotonical (cf. [BL96]). Such cases occur if the maximum space is not encountered on the path containing the parameter value(s) with recursion depth decremented by one, which means that one cannot decide "locally" on which path the maximum space effort can be expected.

Nevertheless the recurrence relations

$$\begin{aligned}\mathcal{S}(n^{(f)}) &= \sigma_1^{(f)} + \max(\sigma_1^{(g)}(n-1) + \mathcal{S}(0^{(f)}), \dots, \\ &\qquad \sigma_1^{(g)}(n-k-1) + \mathcal{S}(k^{(f)}), \dots), \\ \mathcal{S}(n^{(g)}) &= \sigma_1^{(g)} + \max(\sigma_1^{(f)}(n-1) + \mathcal{S}(0^{(g)}), \dots, \\ &\qquad \sigma_1^{(f)}(n-k-1) + \mathcal{S}(k^{(g)}), \dots),\end{aligned}$$

can be solved directly to obtain the linear space behavior of f and g. One gets

$$\begin{aligned}\mathcal{S}(n^{(f)}) &= n \cdot \sigma_1^{(f)} + n \cdot \sigma_1^{(g)} + \max(\sigma_0^{(f)}, \sigma_0^{(g)}), \\ \mathcal{S}(n^{(g)}) &= (n-1) \cdot \sigma_1^{(f)} + n \cdot \sigma_1^{(g)} + \max(\sigma_0^{(f)}, \sigma_0^{(g)}). \quad \blacksquare\end{aligned}$$

5. THE TIME EFFORT OF INDIRECT RECURSIVE PROCEDURES

Denoting by $\tau(f^{(j)})$, $f^{(j)} \in \mathcal{F}^{(j)}$ the time used to perform $p^{(j)}(f^{(j)})$ without taking into account the (recursive) calls to some $p \in \mathcal{P}$, we have

$$\mathcal{T}(f^{(j)}) = \tau(f^{(j)}) + \sum_{g \in \mathcal{R}^{(\mathcal{P})}(f^{(j)})} \mathcal{T}(g). \tag{1}$$

Definition 5.1. For $f^{(j)} \in \mathcal{F}^{(j)}$ we define

$$\Upsilon(f^{(j)}) = \sum_{g \in \mathcal{Q}_*(f^{(j)})} \tau(g).$$

By Definitions 2.5 and 5.1, equation (1) can be written

$$\mathcal{T}(f^{(j)}) = \Upsilon(f^{(j)}) + \sum_{g^{(j)} \in \mathcal{R}^{(j)}(f^{(j)})} \mathcal{T}(g^{(j)}). \tag{2}$$

Definition 5.2. For all $f_1^{(j)}, f_2^{(j)} \in \mathcal{F}^{(j)}$ we write $f_1^{(j)} \sqsubseteq f_2^{(j)}$ (or equivalently $f_2^{(j)} \sqsupseteq f_1^{(j)}$) if $f_1^{(j)} \preceq f_2^{(j)}$ and $\Upsilon(f_1^{(j)}) \leq \Upsilon(f_2^{(j)})$.

Definition 5.3. Let $f_1^{(j)}, f_2^{(j)} \in \mathcal{F}^{(j)}$, $\mathcal{R}^{(j)}(f_i^{(j)}) = \{f_{i,1}^{(j)}, \dots, f_{i,m_i}^{(j)}\}$, $i = 1,2$, such that $f_{i,1}^{(j)} \sqsupseteq f_{i,2}^{(j)} \sqsupseteq \dots \sqsupseteq f_{i,m_i-1}^{(j)} \sqsupseteq f_{i,m_i}^{(j)}$, $i = 1,2$.

If for all $f_1^{(j)} \sqsubseteq f_2^{(j)}$, we have $m_1 \leq m_2$ and $f_{1,r}^{(j)} \sqsubseteq f_{2,r}^{(j)}$, $r = 1, \dots, m_1$, then the underlying indirect recursive procedure is called *locally time-monotonical.*

Lemma 5.1. *If a monotonical indirect recursive procedure $p^{(j)}$ is locally time-monotonical, then $f_1^{(j)} \sqsubseteq f_2^{(j)}$ implies $\mathcal{T}(f_1^{(j)}) \leq \mathcal{T}(f_2^{(j)})$.*

Proof. Let $f_1^{(j)} \in \mathcal{F}_i^{(j)}$ and $f_2^{(j)} \in \mathcal{F}_k^{(j)}$, $i \leq k$. We prove the theorem by double induction on the recursion depth.

- At first let $i = 0$. We prove by induction on k that our claim is correct.
  - If $k = 0$, we have
  $$\mathcal{T}(f_1^{(j)}) = \Upsilon(f_1^{(j)}) \leq \Upsilon(f_2^{(j)}) = \mathcal{T}(f_2^{(j)}).$$
  - If $k > 0$, we obtain
  $$\begin{aligned}\mathcal{T}(f_1^{(j)}) &= \Upsilon(f_1^{(j)}) \leq \Upsilon(f_2^{(j)}) \\ &\leq \Upsilon(f_2^{(j)}) + \sum_{\overline{f}_2^{(j)} \in \mathcal{R}^{(j)}(f_2^{(j)})} \mathcal{T}(\overline{f}_2^{(j)}) = \mathcal{T}(f_2^{(j)}).\end{aligned}$$
- Next we consider $i > 0$.

For $k \geq i$ we derive

$$\mathcal{T}(f_1^{(j)}) = \Upsilon(f_1^{(j)}) + \sum_{\overline{f}_1^{(j)} \in \mathcal{R}^{(j)}(f_1^{(j)})} \mathcal{T}(\overline{f}_1^{(j)}) \quad \text{and} \tag{3}$$

$$\mathcal{T}(f_2^{(j)}) = \Upsilon(f_2^{(j)}) + \sum_{\overline{f}_2^{(j)} \in \mathcal{R}^{(j)}(f_2^{(j)})} \mathcal{T}(\overline{f}_2^{(j)}). \tag{4}$$

By induction hypothesis the sum in (3) is smaller than or equal to the sum in (4). Since $\Upsilon(f_1^{(j)}) \leq \Upsilon(f_2^{(j)})$, we get

$$\mathcal{T}(f_1^{(j)}) \leq \mathcal{T}(f_2^{(j)}).$$

Hence the lemma is proved. ■

Lemma 5.1 enables us to find upper and lower bounds of the timing behavior if a range of parameter values is given.

THEOREM 5.1. *If $p^{(j)}$ is locally time-monotonical, then*

$$\mathfrak{T}(l^{(j)}, u^{(j)}) = \max_{l^{(j)} \preceq f^{(j)} \preceq u^{(j)}} \mathcal{T}(f^{(j)}) = \max_{g^{(j)} \approx u^{(j)}} \mathcal{T}(g^{(j)}). \quad ■$$

Remark. Note that this section is a non-trivial generalization of the corresponding section in [BL96]. Specifically Υ simplifies to τ for direct recursion.

In the following examples the constants τ_0, τ_1, τ_2, τ_3, and τ_4 are derived from the (source) code of the recursive procedures.

EXAMPLE 5.1. We assume that $a(n)$ and $b(n)$ take τ_1 and τ_2 computation time if $n > 0$, and both take τ_0 if $n = 0$. Thus we obtain

$$\begin{aligned}\Upsilon(n^{(a)}) &= \tau_1 + \tau_2 \quad \text{and}\\ \Upsilon(n^{(b)}) &= \tau_2 + \tau_1.\end{aligned}$$

Hence both $a(n)$ and $b(n)$ are locally time monotonical procedures. However we can derive more. We have

$$\begin{aligned}\mathcal{T}(0^{(a)}) &= \tau_0\\ \mathcal{T}(1^{(a)}) &= \tau_2 + \tau_0\\ \mathcal{T}(n^{(a)}) &= \tau_1 + \tau_2 + \mathcal{T}(n^{(a)} - 2)\end{aligned}$$

and

$$\begin{aligned}\mathcal{T}(0^{(b)}) &= \tau_0\\ \mathcal{T}(1^{(b)}) &= \tau_1 + \tau_0\\ \mathcal{T}(n^{(b)}) &= \tau_1 + \tau_2 + \mathcal{T}(n^{(b)} - 2).\end{aligned}$$

Hence we get

$$\begin{aligned}\mathcal{T}(n^{(a)}) &= (\tau_1 + \tau_2)\lfloor n^{(a)}/2\rfloor + \tau_0 + \tau_2(n^{(a)} \bmod 2) \quad \text{and}\\ \mathcal{T}(n^{(b)}) &= (\tau_1 + \tau_2)\lfloor n^{(b)}/2\rfloor + \tau_0 + \tau_1(n^{(b)} \bmod 2). \quad \blacksquare\end{aligned}$$

EXAMPLE 5.2. We assume

$$\begin{aligned}\tau(0^{(f)}) &= \tau_0,\\ \tau(n^{(f)}) &= \tau_1 \text{ for } n \geq 1,\\ \tau(0^{(g)}) &= \tau_2,\\ \tau(n^{(g)}) &= \tau_3 \cdot n + \tau_4 \text{ for } n \geq 1.\end{aligned}$$

Thus we obtain

$$\begin{aligned}\Upsilon(n^{(f)}) &= \tau_1 + \tau_3 \cdot n + \tau_4 + \sum_{i=0}^{n-1} (\tau_3 i + \tau_4) 2^{n-i-1},\\ \Upsilon(n^{(g)}) &= \tau_3 + n \cdot \tau_4 + \sum_{i=1}^{n-1} \tau_1 + \tau_0\end{aligned}$$

The sums in both formulas can be simplified easily. One obtains

$$\begin{aligned}\Upsilon(n^{(f)}) &= \tau_1 + 2^n \cdot \tau_4 + (2^n - 1) \cdot \tau_3,\\ \Upsilon(n^{(g)}) &= \tau_3 + n \cdot \tau_4 + (n-1) \cdot \tau_1 + \tau_0.\end{aligned}$$

It is easy to see that both Υ-functions are monotonically increasing. Thus f and g are locally time-monotonical.

The exact timing behavior can be derived by solving appropriate recurrence relations. These computations are left to the reader. ■

6. PARAMETER SPACE MORPHISMS

The theoretical results of the previous sections are impressive in that they are valid for recursive procedures with very general parameter space. For many applications, however, only a small "part" of the parameter space is responsible for the space and time behavior of the recursive procedure. In this section we are concerned with the problem how to "abstract" from unnecessary details of the parameter space.

For example consider some recursively traversed tree-structure. Here the parameter space is the set of all possible trees. For determining the worst-case behavior, however, it often suffices to know how many nodes are contained in the traversed tree.

Commonly, data structures are analyzed by informally introducing some sort of *complexity measure* (cf. [VF90]) or *size* (cf. [Meh84c, AHU74]) of the data structure. We prefer a more formal approach.

Definition 6.1. A *parameter space morphism* is a mapping $\mathcal{H}: \mathcal{F}^{(j)} \to \mathcal{F}'^{(j)}$ such that for all $f^{(j)} \in \mathcal{F}^{(j)}$ the set

$$\mathcal{M}(f^{(j)}) = \max\{g^{(j)} : \mathcal{H}(f^{(j)}) = \mathcal{H}(g^{(j)})\},$$

where the elements of the max-term are ordered by the "$\prec$"-relation of $\mathcal{F}^{(j)}$, and the target recursion depth

$$\mathrm{recdep}_{\mathcal{H}}(f'^{(j)}) := \mathrm{recdep}(g^{(j)}) \quad \text{where } g^{(j)} \in \mathcal{M}(f^{(j)}) \text{ and } f'^{(j)} = \mathcal{H}(f^{(j)}),$$

are well-defined and $\mathrm{recdep}_{\mathcal{H}}(f'^{(j)}) < \infty$ for all $f'^{(j)} \in \mathcal{F}'^{(j)}$.

Remark. Note that $\|\mathcal{M}(f^{(j)})\| \geq 1$, but $\mathrm{recdep}(g_1^{(j)}) = \mathrm{recdep}(g_2^{(j)})$ if $g_1^{(j)} \in \mathcal{M}(f^{(j)})$ and $g_2^{(j)} \in \mathcal{M}(f^{(j)})$.

Remark. Note that $\mathrm{recdep}_{\mathcal{H}}$ implies a (trivial) "$\prec$"-relation upon $\mathcal{F}'^{(j)}$, namely

$$f'^{(j)} \prec g'^{(j)} \quad \Leftrightarrow \quad \mathrm{recdep}_{\mathcal{H}}(f'^{(j)}) < \mathrm{recdep}_{\mathcal{H}}(g'^{(j)}) \tag{5}$$

for $f'^{(j)}, g'^{(j)} \in \mathcal{F}'^{(j)}$. We will assume in the following that a "$\prec$"-relation exists which is consistent with equation (5) and denote it by "$\prec_{\mathcal{H}}$".

Definition 6.2. In the following we will frequently apply $\mathcal{H}$ to subsets of $\mathcal{F}^{(j)}$. Let $\mathcal{G}^{(j)} \subseteq \mathcal{F}^{(j)}$ denote such a subset. Then we write $\mathcal{H}(\mathcal{G}^{(j)})$ to denote the multiset $\mathcal{G}'^{(j)} = \mathcal{H}(\mathcal{G}^{(j)}) = \{\mathcal{H}(g^{(j)}) \mid g^{(j)} \in \mathcal{G}^{(j)}\}$.

In order to estimate space and timing properties of recursive procedures, we define how space and time will be measured in $\mathcal{F}'^{(j)}$.

DEFINITION 6.3. The functions $\mathcal{S}_{\mathcal{H}}$ and $\mathcal{T}_{\mathcal{H}}$ are defined in the following way:

$$\mathcal{S}_{\mathcal{H}}(f'^{(j)}) = \max_{g^{(j)}:f'^{(j)}=\mathcal{H}(g^{(j)})} \mathcal{S}(g^{(j)}) \quad \text{and}$$
$$\mathcal{T}_{\mathcal{H}}(f'^{(j)}) = \max_{g^{(j)}:f'^{(j)}=\mathcal{H}(g^{(j)})} \mathcal{T}(g^{(j)})$$

where $f'^{(j)} \in \mathcal{F}'^{(j)}$ and $g^{(j)} \in \mathcal{F}^{(j)}$.

DEFINITION 6.4. If $f'^{(j)}_1 \preceq_{\mathcal{H}} f'^{(j)}_2$ implies $\mathcal{S}_{\mathcal{H}}(f'^{(j)}_1) \leq \mathcal{S}_{\mathcal{H}}(f'^{(j)}_2)$ and $\mathcal{T}_{\mathcal{H}}(f'^{(j)}_1) \leq \mathcal{T}_{\mathcal{H}}(f'^{(j)}_2)$, we call the underlying recursive procedure *globally $\mathcal{H}$-space-monotonical* and *globally $\mathcal{H}$-time-monotonical*, respectively.

DEFINITION 6.5. In addition, we need the following definitions:

$$\mathcal{D}_{\mathcal{H}}(f'^{(j)}) = \max_{f'^{(j)}=\mathcal{H}(g^{(j)})} \mathcal{D}(g^{(j)}) \tag{6}$$
$$\mathcal{B}_{\mathcal{H}}(f'^{(j)}) = \max_{f'^{(j)}=\mathcal{H}(g^{(j)})} \mathcal{B}(g^{(j)}) \tag{7}$$
$$\Upsilon_{\mathcal{H}}(f'^{(j)}) = \max_{f'^{(j)}=\mathcal{H}(g^{(j)})} \Upsilon(g^{(j)}) \tag{8}$$
$$\mathcal{R}^{(j)}_{\mathcal{H}}(f'^{(j)}) = \bigcup_{f'^{(j)}=\mathcal{H}(g^{(j)})} \left\{\mathcal{H}(\mathcal{R}^{(j)}(g^{(j)}))\right\} \tag{9}$$
$$\mathcal{N}^{(j)}_{\mathcal{H}}(f'^{(j)}) = f'^{(j)}_{\max}\,, \tag{10}$$

where

$$\Gamma(f'^{(j)}) = \{g'^{(j)} \mid \max_{\overline{f}'^{(j)}\in\overline{\mathcal{R}}'^{(j)},\ \overline{\mathcal{R}}'^{(j)}\in\mathcal{R}^{(j)}_{\mathcal{H}}(f'^{(j)})} \operatorname{recdep}_{\mathcal{H}}\overline{f}'^{(j)} = \operatorname{recdep}_{\mathcal{H}} g'^{(j)}\},$$

$f'^{(j)}_{\max} \in \Gamma(f'^{(j)})$, and

$$\mathcal{B}_{\mathcal{H}}(f'^{(j)}_{\max}) = \max_{g'^{(j)}\in\Gamma(f'^{(j)})} \mathcal{B}(g'^{(j)}).$$

Let

$$\mathcal{O}_{\mathcal{H}}(f'^{(j)},\overline{f}'^{(j)}) = \{(o_0,\ldots,o_i) \mid f'^{(j)} = \mathcal{H}(o_0),$$
$$o_{\ell+1} \in \mathcal{R}^{(j)}(o_\ell),\ \text{for } 0 \leq \ell < i,\ \overline{f}'^{(j)} = \mathcal{H}(o_i)\}$$

and

$$\mathcal{O}_{\mathcal{H}}(f'^{(j)},\emptyset) = \{(o_0,\ldots,o_i) \mid f'^{(j)} = \mathcal{H}(o_0),$$
$$o_{\ell+1} \in \mathcal{R}^{(j)}(o_\ell),\ \text{for } 0 \leq \ell < i,\ \mathcal{H}(o_i) \in \mathcal{F}'^{(k)}_0, \text{for some } k \neq j\},$$

then we define

$$\Psi_{\mathcal{H}}(f'^{(j)}, \overline{f}'^{(j)}) = \max_{(o_0,\dots,o_i)\in\mathcal{O}_{\mathcal{H}}(f'^{(j)},\overline{f}'^{(j)})} \sum_{\ell=0}^{i-1} \Psi(o_\ell, o_{\ell+1})$$

for $\overline{f}'^{(j)} \in \mathcal{R}^{(j)}_{\mathcal{H}}(f'^{(j)})$ or $\overline{f}'^{(j)} = \emptyset$.

Remark. Note that $\mathcal{H}(\mathcal{R}^{(j)}(g^{(j)}))$ is a multiset and $\mathcal{R}^{(j)}_{\mathcal{H}}(f'^{(j)})$ is a set of multisets.

DEFINITION 6.6. A recursive procedure p is called $\mathcal{H}$-monotonical if for all $g'^{(j)} \in \mathcal{R}'^{(j)}$ and for all $\mathcal{R}'^{(j)} \in \mathcal{R}^{(j)}_{\mathcal{H}}(f'^{(j)})$ it holds that $g'^{(j)} \prec_{\mathcal{H}} f'^{(j)}$.

With these definitions it is easy to prove the following results.

LEMMA 6.1. *If p is $\mathcal{H}$-monotonical, the following relation holds:*

$$\mathcal{T}_{\mathcal{H}}(f'^{(j)}) \le \Upsilon_{\mathcal{H}}(f'^{(j)}) + \max_{\mathcal{R}'^{(j)}\in\mathcal{R}^{(j)}_{\mathcal{H}}(f'^{(j)})} \sum_{\overline{g}'^{(j)}\in\mathcal{R}'^{(j)}} \mathcal{T}_{\mathcal{H}}(\overline{g}'^{(j)})$$

Proof. By definition

$$\mathcal{T}_{\mathcal{H}}(f'^{(j)}) = \max_{f'^{(j)}=\mathcal{H}(g^{(j)})} \left(\Upsilon(g^{(j)}) + \sum_{\overline{g}^{(j)}\in\mathcal{R}^{(j)}(g^{(j)})} \mathcal{T}(\overline{g}^{(j)}) \right)$$

which can be estimated by

$$\begin{aligned}
&\le \Upsilon_{\mathcal{H}}(f'^{(j)}) + \max_{f'^{(j)}=\mathcal{H}(g^{(j)})} \sum_{\overline{g}^{(j)}\in\mathcal{R}^{(j)}(g^{(j)})} \mathcal{T}(\overline{g}^{(j)}) \\
&\le \Upsilon_{\mathcal{H}}(f'^{(j)}) + \max_{f'^{(j)}=\mathcal{H}(g^{(j)})} \sum_{\overline{g}'^{(j)}\in\mathcal{H}(\mathcal{R}^{(j)}(g^{(j)}))} \max_{\overline{g}'^{(j)}=\mathcal{H}(k^{(j)})} \mathcal{T}(k^{(j)}) \\
&= \Upsilon_{\mathcal{H}}(f'^{(j)}) + \max_{f'^{(j)}=\mathcal{H}(g^{(j)})} \sum_{\overline{g}'^{(j)}\in\mathcal{H}(\mathcal{R}^{(j)}(g^{(j)}))} \mathcal{T}_{\mathcal{H}}(\overline{g}'^{(j)}) \\
&= \Upsilon_{\mathcal{H}}(f'^{(j)}) + \max_{\mathcal{R}'^{(j)}\in\mathcal{R}^{(j)}_{\mathcal{H}}(f'^{(j)})} \sum_{\overline{g}'^{(j)}\in\mathcal{R}'^{(j)}} \mathcal{T}_{\mathcal{H}}(\overline{g}'^{(j)}).
\end{aligned}$$

Thus the lemma is proved. ■

LEMMA 6.2. *If p is $\mathcal{H}$-monotonical, the following relation holds:*

$$\begin{aligned}
&\mathcal{S}_{\mathcal{H}}(f'^{(j)}) \le \mathcal{D}_{\mathcal{H}}(f'^{(j)}) \\
&\quad + \max\left(\Psi_{\mathcal{H}}(f'^{(j)}, \emptyset), \max_{\mathcal{R}'^{(j)}\in\mathcal{R}^{(j)}_{\mathcal{H}}(f'^{(j)})} \max_{\overline{g}'^{(j)}\in\mathcal{R}'^{(j)}} \Psi_{\mathcal{H}}(f'^{(j)}, \overline{g}'^{(j)}) + \mathcal{S}_{\mathcal{H}}(\overline{g}'^{(j)}) \right)
\end{aligned}$$

Proof. The proof is suppressed since it is very similar to the proof of Lemma 6.1. ■

DEFINITION 6.7. A $\mathcal{H}$-monotonical recursive procedure p is called *locally $\mathcal{H}$-space-monotonical* if ${f'}_1^{(j)} \prec_{\mathcal{H}} {f'}_2^{(j)}$ implies $\mathcal{B}_{\mathcal{H}}({f'}_1^{(j)}) \leq \mathcal{B}_{\mathcal{H}}({f'}_2^{(j)})$, ${f'}_1^{(j)} \preceq_{\mathcal{H}} {f'}_2^{(j)}$ implies $\mathcal{N}_{\mathcal{H}}^{(j)}({f'}_1^{(j)}) \preceq_{\mathcal{H}} \mathcal{N}_{\mathcal{H}}^{(j)}({f'}_2^{(j)})$, and, if ${f'}_1^{(j)} \approx {f'}_2^{(j)}$ and $\mathcal{B}_{\mathcal{H}}({f'}_1^{(j)}) \leq \mathcal{B}_{\mathcal{H}}({f'}_2^{(j)})$ implies $\mathcal{B}_{\mathcal{H}}(\mathcal{N}_{\mathcal{H}}^{(j)}({f'}_1^{(j)})) \leq \mathcal{B}_{\mathcal{H}}(\mathcal{N}_{\mathcal{H}}^{(j)}({f'}_2^{(j)}))$.

DEFINITION 6.8. For all ${f'}_1^{(j)}, {f'}_2^{(j)} \in \mathcal{F}'^{(j)}$ we write ${f'}_1^{(j)} \sqsubseteq_{\mathcal{H}} {f'}_2^{(j)}$ (or equivalently ${f'}_2^{(j)} \sqsupseteq_{\mathcal{H}} {f'}_1^{(j)}$) if ${f'}_1^{(j)} \preceq_{\mathcal{H}} {f'}_2^{(j)}$ and $\Upsilon_{\mathcal{H}}({f'}_1^{(j)}) \leq \Upsilon_{\mathcal{H}}({f'}_2^{(j)})$.

DEFINITION 6.9. Let p be a $\mathcal{H}$-monotonical recursive procedure and let ${f'}_1^{(j)}, {f'}_2^{(j)} \in \mathcal{F}'^{(j)}$, $\mathcal{R}_{j_i}^{(j)}({f'}_i^{(j)}) \in \mathcal{R}_{\mathcal{H}}^{(j)}({f'}_i^{(j)})$, $\mathcal{R}_{j_i}^{(j)}({f'}_i^{(j)}) = \{\overline{f}'^{(j)}_{j_i,i,1}, \ldots, \overline{f}'^{(j)}_{j_i,i,m_i}\}$, $i = 1, 2$, such that $\overline{f}'^{(j)}_{j_i,i,1} \sqsupseteq \overline{f}'^{(j)}_{j_i,i,2} \sqsupseteq \ldots \sqsupseteq \overline{f}'^{(j)}_{j_i,i,m_i-1} \sqsupseteq \overline{f}'^{(j)}_{j_i,i,m_i}$, $i = 1, 2$.

If for all $\overline{f}'^{(j)}_1 \sqsubseteq \overline{f}'^{(j)}_2$, we have $m_{j_1,1} \leq m_{j_2,2}$ and $\overline{f}'^{(j)}_{j_1,1,r} \sqsubseteq \overline{f}'^{(j)}_{j_2,2,r}$, $r = 1, \ldots, m_{j_1,1}$, for all j_1, j_2 such that $\mathcal{R}_{j_i}^{(j)}({f'}_i^{(j)}) \in \mathcal{R}_{\mathcal{H}}^{(j)}({f'}_i^{(j)})$, then p is called *locally $\mathcal{H}$-time-monotonical.*

By slightly modifying the proofs of Theorem 4.1 and Lemmas 4.1 and 5.1, $\mathcal{H}$-versions of Theorems 4.2 and 5.1 can easily be proved.

It is worth noting that a globally ($\mathcal{H}$-)time-monotonical recursive procedure does not need to be locally ($\mathcal{H}$-)time-monotonical. A prominent example, *Quicksort*, has been studied in [BL96].

Finally, we would like to repeat (cf. [BL96]) that in most cases a morphism $\mathcal{H} : \mathcal{F}^{(j)} \to \mathbb{N}$ will be used. This can be supported by the following arguments:

- Parameter space morphisms are useful only if $\mathcal{B}_{\mathcal{H}}$ and $\Upsilon_{\mathcal{H}}$ (cf. Definition 6.5) can be found easily. In most cases this can be obtained if already $\mathcal{B}$ and Υ do depend on some $f'^{(j)} \in \mathcal{F}'^{(j)}$ and not on some $f^{(j)} \in \mathcal{F}^{(j)}$. Thus we are left with determining how the functions $\mathcal{B}$ and Υ will look.
- The function $\mathcal{B}$ will usually depend on the size of locally declared objects. Typical "sizes" originate in the length of arrays or the size of two-dimensional arrays, and so on. Hence we can expect $\mathcal{B}$ to be a polynomial function from $\mathbb{N}$ to $\mathbb{N}$.
- The function Υ will usually depend on the number of iterations of the loops within the code of the underlying recursive procedure. Again, we expect Υ to be a function from $\mathbb{N}$ to $\mathbb{N}$ (or $\mathbb{R}$) since the number of iterations can usually be expressed in terms of n^k and $(\mathrm{ld} n)^k$ for for-loops and discrete loops (cf. [Bli94]), respectively.

Summing up, usually $\mathcal{B}$ and Υ are functions from $\mathbb{N}$ to $\mathbb{N}$ (or $\mathbb{R}$). Thus one can suspect that a morphism from $\mathcal{F}^{(j)}$ to $\mathbb{N}$ will be helpful in determining the space and time behavior.

7. PROGRAMMING LANGUAGE ISSUES

Before we discuss details of how (real-time) programming languages are influenced by our previous results, we restate Theorems 4.2 and 5.1 in a way more suitable to programming language issues.

DEFINITION 7.1. If an additional ordering on $\mathcal{F}^{(j)}$ by $f_1^{(j)} \lhd f_2^{(j)}$ exists such that for all $f_1^{(j)}, f_2^{(j)} \in \mathcal{F}^{(j)}$, $f_1^{(j)} \lhd f_2^{(j)}$ $(f_1^{(j)} \neq f_2^{(j)})$ implies

1. $f_1^{(j)} \preceq f_2^{(j)}$,
2. the underlying recursive procedure is locally space-monotonical w.r.t. $\lhd$, and
3. the underlying recursive procedure is locally time-monotonical w.r.t. $\lhd$,

we call $\mathcal{F}^{(j)}$ *totally ordered.*

The advantage of the "$\lhd$"-relation is that it can be used to compare elements with the same recursion depth in a useful manner. We are able to show the following theorems.

THEOREM 7.1. *If the parameter space of a recursive procedure is totally ordered, then*

$$\mathfrak{S}(l^{(j)}, u^{(j)}) = \max_{l^{(j)} \unlhd f^{(j)} \unlhd u^{(j)}} \mathcal{S}(f^{(j)}) = \mathcal{S}(u^{(j)}).$$

Proof. In conjunction with Theorem 4.2 it remains to show that

$$\max_{g^{(j)} \approx u^{(j)}} \mathcal{S}(g^{(j)}) = \mathcal{S}(u^{(j)}).$$

Because of Definition 7.1, however, we have $\mathcal{D}(g^{(j)}) \leq \mathcal{D}(u^{(j)})$ for all $g^{(j)} \lhd u^{(j)}$. A slight modification of Lemma 4.1 shows that in this case $\mathcal{S}(g^{(j)}) \leq \mathcal{S}(u^{(j)})$ too. Thus the theorem is proved. ■

THEOREM 7.2. *If the parameter space of a recursive procedure is totally ordered, then*

$$\mathfrak{T}(l^{(j)}, u^{(j)}) = \max_{l^{(j)} \unlhd f^{(j)} \unlhd u^{(j)}} \mathcal{T}(f^{(j)}) = \mathcal{T}(u^{(j)}).$$

Proof. In conjunction with Theorem 5.1 it remains to show that

$$\max_{g^{(j)} \approx u^{(j)}} \mathcal{T}(g^{(j)}) = \mathcal{T}(u^{(j)}).$$

Because of Definition 7.1, however, we have $\tau(g^{(j)}) \leq \tau(u^{(j)})$ for all $g^{(j)} \lhd u^{(j)}$. A slight modification of Lemma 5.1 shows that in this case $\mathcal{T}(g^{(j)}) \leq \mathcal{T}(u^{(j)})$ too. Thus the theorem is proved. ■

Obviously $\mathcal{H}$-versions of these theorems can also be proved.

If $\mathcal{F}^{(j)}$ is totally ordered, we assume that there exists a programming language defined function `pred`, which given some $f^{(j)} \in \mathcal{F}^{(j)}$ computes `pred`$(f^{(j)})$ such that `pred`$(f^{(j)}) \lhd f^{(j)}$ and there is no $g^{(j)} \in \mathcal{F}^{(j)}$ such that `pred`$(f^{(j)}) \lhd g^{(j)} \lhd f^{(j)}$.

7.1. The recursion depth

Let p be a locally time- and space-monotonical recursive procedure system with parameter space $\mathcal{F}$. In order to perform a time and space analysis of p, the programmer has to supply for all $p^{(j)}$ *non-recursive* functions without while loops `recdep`: $\mathcal{F}^{(j)} \to \mathbb{N}$ that for all $f^{(j)} \in \mathcal{F}^{(j)}$ compute recdep$(f^{(j)})$.

This implies that we can decide effectively (at runtime) whether

$$f_1^{(j)} \prec f_2^{(j)}, \quad f_2^{(j)} \prec f_1^{(j)}, \quad \text{or} \quad f_1^{(j)} \approx f_2^{(j)}$$

for all $f_1^{(j)}, f_2^{(j)} \in \mathcal{F}^{(j)}$.

If no "$\lhd$"-relation exists, the recursion depth must be bounded by a programmer supplied constant $\mathtt{R}^{(j)}$. If a "$\lhd$"-relation exists, a bound of the recursion depth can be derived from a programmer supplied upper bound of the parameter values, say $\mathtt{U}^{(j)}$.

Since it is extremely difficult to verify some function `recdep` supplied by the programmer at compile time[¶], the correctness of `recdep` is checked at runtime. Note that it is these checks that enforce the well-definedness of the recursive procedure system. To be more specific, the following conditions must be met:

1. `recdep`$(f^{(j)})$ can be computed for each $f^{(j)} \in \mathcal{F}^{(j)}$ without a runtime error
2. for all $\overline{f}^{(j)} \in \mathcal{R}^{(j)}(f^{(j)})$, `recdep`$(\overline{f}^{(j)}) <$ `recdep`$(\overline{f}^{(j)})$
3. if no parameter space morphism is used, at least one $\overline{f}^{(j)} \in \mathcal{R}^{(j)}(f^{(j)})$ has to exist such that recdep$(\overline{f}^{(j)}) =$ recdep$(f^{(j)}) - 1$
4. for all $f^{(j)} \in \mathcal{F}^{(j)}$, `recdep`$(f^{(j)}) \leq \mathtt{R}^{(j)}$

All these conditions can be checked at runtime with little effort. If one of them is violated the exception `recursion_depth_error` is raised.

7.2. Checking Space Properties

If $\mathcal{B}(f^{(j)})$ is constant or if there is a simple connection between $\mathcal{B}(f^{(j)})$ and `recdep`$(f^{(j)})$, the compiler can derive that the underlying recursive procedure is locally space-monotonical. Thus no runtime checks are necessary.

Checking of global space properties without a "$\lhd$"-relation

In this case the programmer must supply a function `maxspacearg`: $\mathbb{N} \to \mathcal{F}^{(j)}$, which given some $k =$ recdep$(f^{(j)})$ returns $\overline{f}^{(j)}$ such that $f^{(j)} \approx \overline{f}^{(j)}$ and $\mathcal{S}(\overline{f}^{(j)}) = \max_{\overline{f}^{(j)} \approx g^{(j)}} \mathcal{S}(g^{(j)})$.

At runtime for each $f^{(j)} \in \mathcal{F}^{(j)}$, it is checked whether $\mathcal{S}(f^{(j)}) \leq \mathcal{S}(u_k^{(j)})$ where $k =$ `recdep`$(f^{(j)})$ and $u_k^{(j)} =$ `maxspacearg`(k). If this condition is violated, the exception `space_monotonic_error` is raised.

Checking of local space properties with help of a "$\lhd$"-relation

Here we can perform an exhaustive enumeration of all parameter values with help of the function `pred` at compile time. For each pair of these values it can be checked whether Definition 7.1 is valid.

[¶]In fact it is undecidable, whether two given Turing machines accept the same language.

Hence we do not need any runtime checks except testing the recursion depth in order to guarantee the upper bound of the space behavior (cf. Theorem 7.1).

7.3. Space behavior and morphisms

Everything is still valid if we take into account parameter space morphisms. The only exception is that we can perform an exhaustive enumeration of all parameter values with help of a "$\lhd$"-relation only if the morphism is a function from $\mathcal{F}^{(j)}$ to $\mathbb{N}$. This, however, as already noted at the end of Section 6, covers most important cases.

It is, however, crucial in this context to perform checks of local properties since global properties can only be checked for $f^{(j)} \in \mathcal{F}^{(j)}$ and not for $f'^{(j)} \in \mathcal{F}'^{(j)}$ (i.e. for $f'^{(j)} \in \mathbb{N}$).

7.4. Checking Time Properties

If there is a simple connection between $\Upsilon(f^{(j)})$ and recdep$(f^{(j)})$ and if

$$\|\mathcal{R}^{(j)}(f^{(j)})\| \leq 1,$$

it can be derived at compile time that the underlying recursive procedure is locally time-monotonical. Thus no runtime checks are necessary.

Checking of global time properties without a "$\lhd$"-relation

In this case the programmer must supply a function `maxtimearg`: $\mathbb{N} \to \mathcal{F}^{(j)}$, which given some $k =$ recdep$(f^{(j)})$ returns $\overline{f}^{(j)}$ such that $f^{(j)} \approx \overline{f}^{(j)}$ and $\mathcal{T}(\overline{f}^{(j)}) = \max_{\overline{f}^{(j)} \approx g^{(j)}} \mathcal{T}(g^{(j)})$.

At runtime for each $f^{(j)} \in \mathcal{F}^{(j)}$, it is checked whether $\mathcal{T}(f^{(j)}) \leq \mathcal{T}(u_k^{(j)})$ where $k =$ `recdep`$(f^{(j)})$ and $u_k^{(j)} =$ `maxtimearg`(k). If this condition is violated, the exception `time_monotonic_error` is raised.

Checking of local time properties with help of a "$\lhd$"-relation

Here we can perform an exhaustive enumeration of all parameter values with help of the function `pred` at compile time. For each pair of these values it can be checked whether Definition 7.1 is valid.

Hence we do not need any runtime checks except testing the recursion depth in order to guarantee the upper bound of the space behavior (cf. Theorem 7.1).

7.5. Time behavior and morphisms

Here the same arguments are valid as in Section 7.3.

EXAMPLE 7.1. *Two-dimensional trees* are dynamic, adaptable data structures that are very similar to binary trees but divide up a geometric space in a manner convenient for use in range searching and other problems. The idea is to build binary search trees with points in the nodes, using y and x coordinates of the points as keys in a strictly alternating sequence.

The same algorithm is used to insert points into two-dimensional trees as in normal binary search trees, but at the root we use the y coordinate (if the point to be inserted has a smaller y coordinate than the point at the root, go left; otherwise

FIG. 1. Paths in a Recursion Digraph

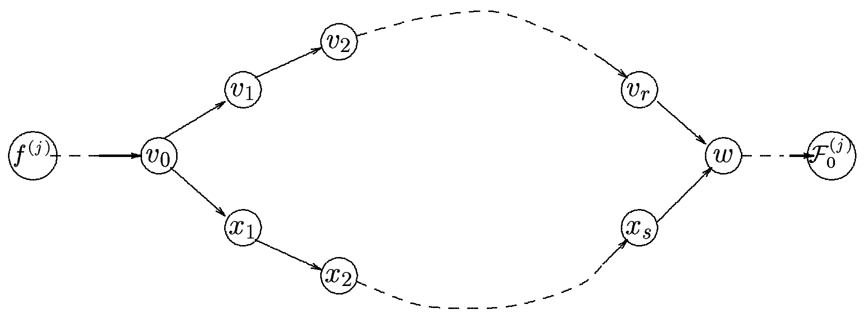

FIG. 2. Specification of the Color Operation for Two-Dimensional Trees

```
package Two_Dim_Tree is

  type two_dim_tree is limited private;

  function Color(
      x,y: integer;                                   -- the coordinates
      into: two_dim_tree)
    return integer;
  – retrieves color of point (x,y)
  – assumes that point is present in the tree

  – other operations suppressed

private

  type two_dim_tree_t;
  type two_dim_tree is access two_dim_tree_t;

  type two_dim_tree_t is
    record
      x,y: integer;
      n: integer;                         -- number of nodes in this subtree
      color: integer;               -- number describing the color of the point
      left,right: two_dim_tree;
    end record;

end Two_Dim_Tree;
```

FIG. 3. Recursive Implementation of the Color Operation for Two-Dimensional Trees (fragment)

```
package body Two_Dim_Tree is

  recursive function Color_y(
      x,y: integer;
      into: two_dim_tree)
    return integer;

  recursive function Color_x(
      x,y: integer;
      into: two_dim_tree)
    return integer;

  function Color(
      x,y: integer;                                   -- the coordinates
      into: two_dim_tree)
    return integer
  is
  begin
    return Color_y(x,y,into);
  end Color;

  recursive function Color_y(
      x,y: integer;
      into: two_dim_tree)
    return integer

    with function morphism(t: two_dim_tree)
      return integer is
    begin
      return t.n;
    end morphism;

    with function recdep(current_size: integer)
      return natural is
    begin
      return floor(C*log(current_size)**2);
    end recdep;

  is
  begin
    if y<into.y then
      return Color_x(x,y,into.left);
    else
      return Color_x(x,y,into.right);
    end if;
  end Color_y;
```

FIG. 3. Recursive Implementation of the Color Operation for Two-Dimensional Trees (continued)

```
  recursive function Color_x(
      x,y: integer;
      into: two_dim_tree)
    return integer

    with function morphism(t: two_dim_tree)
      return integer is
    begin
      return t.n;
    end morphism;

    with function recdep(current_size: integer)
      return natural is
    begin
      return floor(C*log(current_size)**2);
    end recdep;

  is
  begin
    if x=into.x and then y=into.y then
       return into.color;
    elsif x<into.x then
      return Color_y(x,y,into.left);
    else
      return Color_y(x,y,into.right);
    end if;
  end Color_x;

  – other operations suppressed

end Two_Dim_Tree;
```

go right), then at the next level we use the x coordinate, then at the next level the y coordinate, etc., alternating until an external node is encountered.

By use of *dynamization* (cf. [Meh84b]) two-dimensional trees (or trees of higher dimension) can be "balanced" such that the worst-case timing behavior for an insert and other critical operations is $O((\log n)^2)$.

Obviously the operations for two-dimensional trees can be implemented by two indirect recursive procedures. Figure 2 shows the specification of a procedure for retrieving the color of a point. In Figure 3 the recursive implementation of this operation is depicted. For our space and time analysis we assume that the tree is balanced, i.e., the recursion depth is bounded by $\lfloor 2C \log^2 n \rfloor$ where n denotes the current size of the tree and C is some constant. The current size of the tree is used as morphism $\mathcal{H}$.

Thus the formulas for `recdep` follow immediately. In addition, we have

$$\begin{aligned}
\mathcal{D}_{\mathcal{H}}(n^{(x)}) &= \sigma_1^{(x)}, \\
\mathcal{D}_{\mathcal{H}}(n^{(y)}) &= \sigma_1^{(y)}, \\
\mathcal{B}_{\mathcal{H}}(n^{(x)}) &= \sigma_1^{(y)} + \sigma_1^{(x)}, \\
\mathcal{B}_{\mathcal{H}}(n^{(y)}) &= \sigma_1^{(x)} + \sigma_1^{(y)}, \\
\Upsilon_{\mathcal{H}}(n^{(x)}) &= \tau_1^{(x)} + \tau_1^{(y)}, \\
\Upsilon_{\mathcal{H}}(n^{(y)}) &= \tau_1^{(y)} + \tau_1^{(x)}
\end{aligned}$$

where we have used superscripts $^{(x)}$ and $^{(y)}$ to denote entities related to `Color_x` and `Color_y` respectively.

Clearly `Color_x` and `Color_y` are $\mathcal{H}$-monotonical. Thus they are also locally $\mathcal{H}$-space-monotonical and locally $\mathcal{H}$-time-monotonical.

The required function `pred` is given by the predefined function `integer'PRED`. Thus compile time checks of local space and time properties can be performed with help of `pred`. The functions `recdep` in conjunction with `morphism` are checked during runtime.

Since `Color` calls `Color_y` and `Color_x` is not called by any other procedure than `Color_y`, we can restrict our analysis to `Color_y`. The space and time behavior can be estimated by

$$\mathcal{S}_{\mathcal{H}}(n^{(y)}) = \lfloor 2C \log^2 n \rfloor (\sigma_1^{(x)} + \sigma_1^{(y)})$$

and

$$\mathcal{T}_{\mathcal{H}}(n^{(y)}) = \lfloor 2C \log^2 n \rfloor (\tau_1^{(x)} + \tau_1^{(y)}),$$

respectively. ■

8. CONCLUSION

Note that Theorems 4.2 and 5.1 are valid although we do *not* study *static* bounds of space and time behavior. This is in strict contrast to [PK89], where the execution time of code blocks is estimated statically without taking into account that the execution time may depend on certain parameters (or global data). Anyway, the MARS approach [PK89] excludes recursions.

In [Par93] such information on data influencing execution time can be incorporated into the program by means of program path analysis, but [Par93] does not address recursion at all.

Our results are impressive in that they assume very general parameter spaces, and are very useful together with parameter space morphisms. These morphisms allow for concentrating on the essential properties of the recursive procedure while estimating time and space behavior.

Note that this paper generalizes [BL96]. The results of [BL96] are strictly contained in the results of this paper as special cases. This paper, however, requires a much more delicate analysis.

- Defining the very important concept of *recursion depth* is much more complex than for direct recursion.
- The results on space and time effort are much harder to derive.
- The morphisms of Section 6 are more complex than in [BL96]
- Theorem 4.1 correctly mirrors programming languages that allow block-statements which contain local declarations (cf. e.g. [Ada95]). This was considered a useful generalization of direct recursions too (cf. [BL96]).

Summing up this paper and [BL96], there are no more reasons to exclude recursive procedures from real-time programming languages. However some research has to be done in order to extract the necessary information out of the source code of a recursive procedure system. Symbolic evaluation described in [CHT79, Plo80, Sch96, Bli00, FS97] can be used to build automated tools for handling the analysis described in this paper. We are currently implementing a tool based on symbolic evaluation for symbolic use/def analysis ([BB98]), which is planned to be extended in this direction.

Acknowledgements

The author would like to thank the WOOP-team and especially Roland Lieger for many helpful discussions.

REFERENCES

Ada95. ISO/IEC 8652, *Ada reference manual*, 1995.

AHU74. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, MA, 1974.

Bar97. John Barnes, *High Integrity Ada - The SPARK Approach*, Addison-Wesley, Harlow, England, 1997.

BB98. Johann Blieberger and Bernd Burgstaller, *Symbolic reaching definitions analysis of Ada programs*, Proceedings of Ada-Europe'98 (Uppsala, Sweden), June 1998, pp. 238–250.

BL96. Johann Blieberger and Roland Lieger, *Worst-case space and time complexity of recursive procedures*, Real-Time Systems **11** (1996), no. 2, 115–144.

Bli94. Johann Blieberger, *Discrete loops and worst case performance*, Computer Languages **20** (1994), no. 3, 193–212.

Bli00. Johann Blieberger, *Data-flow frameworks for worst-case execution time analysis*, Real-Time Systems (2000), (to appear).

Boo91. Grady Booch, *Object-oriented design with applications*, Benjamin/Cummings, Redwood City, CA, 1991.

Bus85. Arnold Businger, *PORTAL language description*, Lecture Notes in Computer Science, vol. 198, Springer Verlag, Berlin, 1985.

CHT79. T. E. Cheatham, H. Holloway, and J. A. Townley, *Symbolic evaluation and the analysis of programs*, IEEE Trans. Software Eng. **5** (1979), no. 4, 402–317.

DIN82. DIN 66 253, Teil 2, Beuth Verlag, Berlin, *Programmiersprache PEARL, Full PEARL*, 1982.

ES90. Margaret A. Ellis and Bjarne Stroustrup, *The annotated C++ reference manual*, Addison-Wesley, Reading, MA, 1990.

For93. Charles Forsyth, *Using the worst-case execution analyser*, Tech. report, York Software Engineering Ltd., University of York: Task 8, Volume D Deliverable on ESTEC contract 9198/90/NL/SF, May 1993.

FS97. Thomas Fahringer and Bernhard Scholz, *Symbolic Evaluation for Parallelizing Compilers*, Proc. of the ACM International Conference on Supercomputing, July 1997.

GR91. Narain Gehani and Krithi Ramamritham, *Real-time Concurrent C: A language for programming dynamic real-time systems*, The Journal of Real-Time Systems **3** (1991), 377–405.

HS91. Wolfgang A. Halang and Alexander D. Stoyenko, *Constructing predictable real time systems*, Kluwer Academic Publishers, Boston, 1991.

ITM90. Yutaka Ishikawa, Hideyuki Tokuda, and Clifford W. Mercer, *Object-oriented real-time language design: Constructs for timing constraints*, ECOOP/OOPSLA '90 Proceedings, October 1990, pp. 289–298.

KDK$^+$89. Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger, *Distributed fault-tolerant real-time systems: The Mars approach*, IEEE Micro (1989), 25–40.

KS86. Eugene Kligerman and Alexander D. Stoyenko, *Real-time Euclid: A language for reliable real-time systems*, IEEE Transactions on Software Engineering **12** (1986), no. 9, 941–949.

LL73. C.L. Liu and J.W. Layland, *Scheduling algorithms for multiprogramming in a hard real-time environment*, Journal of the ACM **20** (1973), no. 1, 46–61.

MACT89. Aloysius K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat, *Evaluating tight execution time bounds of programs by annotations*, Proc. IEEE Workshop on Real-Time Operating Systems and Software, 1989, pp. 74–80.

Meh84a. Kurt Mehlhorn, *Graph algorithms and NP-completeness*, Data Structures and Algorithms, vol. 2, Springer-Verlag, Berlin, 1984.

Meh84b. Kurt Mehlhorn, *Multi-dimensional searching and computational geometry*, Data Structures and Algorithms, vol. 3, Springer-Verlag, Berlin, 1984.

Meh84c. Kurt Mehlhorn, *Sorting and searching*, Data Structures and Algorithms, vol. 1, Springer-Verlag, Berlin, 1984.

Mok84. Aloysius K. Mok, *The design of real-time programming systems based on process models*, Proceedings of the IEEE Real Time Systems Symposium (Austin, Texas), IEEE Press, 1984, pp. 5–16.

Nil96. Kelvin Nilsen, *Java for real-time*, Real-Time Systems **11** (1996), no. 2, 197–205.

Par93. Chang Yun Park, *Predicting program execution times by analyzing static and dynamic program paths*, The Journal of Real-Time Systems **5** (1993), 31–62.

PK89. Peter Puschner and Christian Koza, *Calculating the maximum execution time of real-time programs*, The Journal of Real-Time Systems **1** (1989), 159–176.

Plo80. Erhard Ploedereder, *A semantic model for the analysis and verification of programs in general, higher-level languages*, Ph.D. thesis, Division of Applied Sciences, Harvard University, 1980.

Sch96. Bernhard Scholz, *Symbolische Verifikation von Echtzeitprogrammen*, Diploma thesis, TU Vienna, Dept. of Automation, 1996.

Sha89. Alan C. Shaw, *Reasoning about time in higher-level language software*, IEEE Transactions on Software Engineering **15** (1989), no. 7, 875–889.

VF90. Jeffrey S. Vitter and Phillipe Flajolet, *Average-case analysis of algorithms and data structures*, Handbook of Theoretical Computer Science (Jan van Leeuwen, ed.), vol. A: Algorithms and Complexity, North-Holland, 1990, pp. 431–524.