Real-Time Systems, 11, 115–146 (1996)
© 1996 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Worst-Case Space and Time Complexity of Recursive Procedures*

JOHANN BLIEBERGER blieb@auto.tuwien.ac.at
ROLAND LIEGER rlieger@auto.tuwien.ac.at
Department of Automation, Technical University of Vienna, A-1040 Vienna

Editor:

Abstract. The purpose of this paper is to show that 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, for example Mergesort or recursive tree-traversal algorithms.

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.

Keywords: real-time systems, worst-case performance, recursions, parameter space morphisms

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. (Liu and Layland 1973, Halang and Stoyenko 1991, Mok 1984)). Thus the *worst-case performance* 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 (Blieberger 1994), the second one will be treated in this paper.

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

1. It is not clear, whether a recursive procedure completes or not (cf. e.g. Example 5 below).

* Supported by the Austrian Science Foundation (FWF) under grant P10188-MAT.
A preliminary version of this paper has been presented at the 7th Euromicro Workshop on Real-Time Systems (Blieberger and Lieger 1995).

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. PEARL (cf. (DIN 66 253, 1982)), RT-Euclid (cf. (Kligerman and Stoyenko 1986, Halang and Stoyenko 1991)), Real-Time Concurrent C (cf. (Gehani and Ramamritham 1991)), and the MARS approach (cf. (Kopetz et al. 1989, Puschner and Koza 1989)).

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. (Ada 1995)) and PORTAL (cf. (Businger 1985)). Interestingly, a subset of Ada (cf. (Forsyth 1993)) designed for determining the worst-case timing behavior forbids recursion. 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.

Some other approaches do not address recursion at all (cf. e.g. (Mok et al. 1989, Shaw 1989, Park 1993, Ishikawa et al. 1990)), others propose to replace recursive algorithms by iterative ones or to transform them into non-recursive schemes by applying program transformation rules (cf. e.g. (Puschner and Koza 1989)). 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, even with the use of discrete loops (cf. (Blieberger 1994)).

- 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. (Booch 1991)).

- 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.

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.

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 can not be explicitly controlled by the application. This case requires a delicate treatment, which will be performed in this paper.

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

Example 1. The *Factorial Numbers* $n!$ given by the recursion

$$\mathrm{fac}(n) \;=\; \begin{cases} n \cdot \mathrm{fac}(n-1) & \text{if } n > 0, \\ 1 & \text{if } n = 0. \end{cases}$$ □

Example 2. The *Fibonacci Numbers* $f(n)$ given by the recursion $(n \geq 2)$

$$\begin{aligned} f(0) &= f(1) = 1, \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$ □

Example 3. The *Ackermann Function* $\mathcal{A}(x, y)$ given by

$$\begin{aligned} \mathcal{A}(0, y) &= y + 1, \\ \mathcal{A}(x+1, 0) &= \mathcal{A}(x, 1), \\ \mathcal{A}(x+1, y+1) &= \mathcal{A}(x, \mathcal{A}(x+1, y)) \end{aligned}$$ □

Example 4. A recursive version of *Mergesort*, the source code of which is shown in Figure 1. Note that the Ada source code contains a hidden for-loop, namely at line 18, and a discrete loop starting at line 19. (The syntax and semantics of discrete loops can be found in (Blieberger 1994).) □

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

It is obvious that the first three examples of recursive procedures introduced above will not be used in practical applications. Rather the first two will be implemented without recursion and we suppose the third one will not occur in any practical

```
1.  N: constant integer := ... ;                    -- number of elements to be sorted
2.  subtype index is integer range 1 .. N;
3.  type gen_sort_array is array (index range <>) of ... ;
4.  subtype sort_array is gen_sort_array (index);
5.  sort_arr: sort_array;
6.
7.  procedure merge_sort(from,to: index) is
8.     m: constant integer := (from+to)/2 + 1;
9.     subtype aux_array is gen_sort_array(m..to);
10.    aux: aux_array;
11.    p,q,r: integer;
12. begin
13.    if from = to then
14.       return;
15.    else
16.       merge_sort(from,m-1);
17.       merge_sort(m,to);
18.       aux := sort_arr(m..to);
19.       discrete (p,q,r) := (m-1,aux'last,to)
20.            in reverse (m-1,aux'last,to) .. (from-1,aux'first,from)
21.            new (p,q,r) := (p-1,q,r-1) | (p,q-1,r-1) loop
22.          if p < from or else target(p) < aux(q) then
23.             target(r) := aux(q);
24.             r := r-1;
25.             q := q-1;
26.          else
27.             target(r) := target(p);
28.             r := r-1;
29.             p := p-1;
30.          end if;
31.       end loop;
32.    end if;
33. end merge_sort;
```

Figure 1. Ada Source Code of Mergesort using a Discrete Loop

application. Nevertheless we will use these examples throughout this paper because they are simple enough to illustrate our ideas. Of course this does not mean that our approach can only be applied to simple cases. In fact it is applicable to very complex and general cases as can be seen in the following sections.

Remark 1.1. 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 $\lfloor x \rfloor$.
- The smallest integer $n \geq x$ is denoted by $\lceil x \rceil$.

2. Definitions and Preliminary Results

Definition 2.1. Essential properties of a recursive procedure p are the *parameter space* $\mathcal{F}$, i.e., the set of all possible (tuples of) values of parameters of p, a set $\mathcal{F}_0 \subseteq \mathcal{F}$, the *terminating values* of $\mathcal{F}$, and its code. If p is called with actual parameters $f_0 \in \mathcal{F}_0$, the code being executed must not contain a recursive call of p to itself. If p is called with actual parameters $f \in \mathcal{F} \setminus \mathcal{F}_0$, the code being executed must contain at least one recursive call of p to itself.

Definition 2.2. We call a recursive procedure p *well-defined* if for each element of $\mathcal{F}$ the procedure p 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 recursive procedure, we mean well-defined recursive procedure.

Definition 2.3. We define a set $\mathcal{R}(f) \subseteq \mathcal{F}$, $(f \in \mathcal{F} \setminus \mathcal{F}_0)$ by $\overline{f} \in \mathcal{R}(f)$ iff $p(\overline{f})$ is directly called in order to compute $p(f)$. $\mathcal{R}(f)$ is called the set of *direct successors* of f. If $f \in \mathcal{F}_0$, the set $\mathcal{R}(f) = \emptyset$, i.e., it is empty.

Remark 2.1. We assume that if $\overline{f} \in \mathcal{R}(f)$, it is not essential how often p is called with parameter $\overline{f}$. Note that it can be guaranteed by the runtime system that $p(\overline{f})$ is evaluated only once.

Definition 2.4. We define a sequence of sets $\mathcal{R}_k(f)$ by

$$\begin{aligned} \mathcal{R}_0(f) &= \{f\}, \\ \mathcal{R}_{k+1}(f) &= \mathcal{R}_k(f) \cup \{\overline{f} \mid \overline{f} \in \mathcal{R}(g) \text{ where } g \in \mathcal{R}_k(f)\} \end{aligned}$$

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

$$\mathcal{R}^*(f) = \lim_{k\to\infty} \mathcal{R}_k(f).$$

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

Definition 2.5. We define a sequence of sets $\mathcal{F}_k$ inductively by

1. $\mathcal{F}_0$ is defined as above (cf. Definition 2.1), i.e., $\mathcal{F}_0$ contains the terminating values of $\mathcal{F}$.

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

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

Lemma 2.1 *We have* $\bigcup_{k\geq 0} \mathcal{F}_k = \mathcal{F}$.

Proof: By definition we clearly have $\bigcup_{k\geq 0} \mathcal{F}_k \subseteq \mathcal{F}$.

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

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

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

Definition 2.6. Let $f \in \mathcal{F}$ and let k be such that $f \in \mathcal{F}_k$, then k is called the *recursion depth* of $p(f)$. We write $k = \mathrm{recdep}(f)$. For $f, g \in \mathcal{F}$, we write $f \approx g$ iff $\mathrm{recdep}(f) = \mathrm{recdep}(g)$.

Definition 2.7. A recursive procedure p is called *monotonical* if for all $f_k \in \mathcal{F}_k$ and for $f_i \in \mathcal{F}_i$, $0 \leq i < k$, we have $f_i \prec f_k$, where "$\prec$" is a suitable binary relation that satisfies for all $f_1, f_2, f_3 \in \mathcal{F}$

1. either $f_1 \prec f_2$ or $f_2 \prec f_1$ or $f_1 \approx f_2$ and

2. if $f_1 \prec f_2$ and $f_2 \prec f_3$, then $f_1 \prec f_3$.

We write $f_1 \preceq f_2$ if either $f_1 \prec f_2$ or $f_1 \approx f_2$.

Remark 2.2. Note that a trivial "$\prec$"-relation can always be obtained by defining $f_1 \prec f_2 \Leftrightarrow \mathrm{recdep}(f_1) < \mathrm{recdep}(f_2)$. We will return to this topic in Section 7.

Remark 2.3. If p is a monotonical recursive procedure, then $\overline{f} \prec f$ for all $\overline{f} \in \mathcal{R}(f)$.

Example 1. For the Factorial Numbers we have $\mathcal{F} = N$, $\mathcal{R}(k) = \{k-1\}$, and $\mathcal{F}_0 = \{0\}$, $\mathcal{F}_k = \{k\}$. Furthermore we have $\mathrm{recdep}(k) = k$ and the "$\prec$"-relation for $\mathcal{F}$ is the "$<$"-relation for integers. □

Example 2. For the Fibonacci Numbers we obtain $\mathcal{F} = N$, $\mathcal{R}(k) = \{k-1, k-2\}$, and $\mathcal{F}_0 = \{0, 1\}$, $\mathcal{F}_k = \{k+1\}$. Furthermore we have $\mathrm{recdep}(k) = k-1$, if $k \geq 1$, the "$\prec$"-relation for $\mathcal{F}$ is the "$<$"-relation for integers. □

Example 3. The Ackermann Function gives

$$\begin{aligned} \mathcal{F} &= N^2, \\ \mathcal{R}((x,y)) &= \begin{cases} \emptyset & \text{if } x = 0, \\ \{(x-1, \mathcal{A}(x, y-1)), (x, y-1)\} & \text{if } y \geq 1, \\ \{(x-1, 1)\} & \text{if } y = 0 \end{cases} \end{aligned}$$

and (cf. (Lieger and Blieberger 1994), where proofs of the following facts can be found)

$$\begin{aligned} \mathcal{F}_0 &= \{(0, y) \mid y \in N\}, \\ \mathcal{F}_k &= \{(x, y) \mid \mathcal{A}(x, y) + x - 2 = k, x > 0\}. \end{aligned}$$

Furthermore we have

$$\mathrm{recdep}((0, y)) = 0$$

and for $x > 0$

$$\mathrm{recdep}((x, y)) = \mathcal{A}(x, y) + x - 2,$$

the "$\prec$"-relation for $\mathcal{F}$ is

$$(x_1, y_1) \prec (x_2, y_2) \Leftrightarrow \mathcal{A}(x_1, y_1) + x_1 < \mathcal{A}(x_2, y_2) + x_2$$

if $x_1, x_2 > 0$. □

Example 4. For Mergesort we derive

$$\begin{aligned} \mathcal{F} &= N^2, \\ \mathcal{R}((x,y)) &= \left\{ \left(x, \left\lfloor \frac{x+y}{2} \right\rfloor \right), \left(\left\lfloor \frac{x+y}{2} \right\rfloor + 1, y \right) \right\}, \end{aligned}$$

and

$$\begin{aligned}\mathcal{F}_0 &= \{(x,x) \mid x \in N\},\\ \mathcal{F}_k &= \{(x,y) \mid 2^{k-1} \le (y-x) < 2^k\}.\end{aligned}$$

Furthermore we have

$$\mathrm{recdep}((x,y)) = \lceil \mathrm{ld}\ (y-x+1) \rceil,$$

the "$\prec$"-relation for $\mathcal{F}$ is given by

$$(x_1,y_1) \prec (x_2,y_2) \Leftrightarrow y_1 - x_1 < y_2 - x_2,$$

where "$<$" denotes the "$<$"-relation of integer numbers. □

Example 5. An interesting example of a recursive function is the "wondrous" function (cf. (Hofstadter 1979)). This function is not known to be well-defined, but we will study it anyway since it has interesting properties. It is defined by

$$d(n) = \begin{cases} d(n/2) & \text{if } n \equiv 0(2) \text{ and} \\ d(3n+1) & \text{if } n \equiv 1(2). \end{cases}$$

It has been conjectured that finally the "wondrous" function finds itself repeating the three numbers 4, 2, and 1 infinitely, irrespective of the initial value $n \in N$. This, however, has not been proved.

Now defining a (possibly not well-defined) recursive procedure by

$$\begin{aligned}\mathcal{F} &= N,\\ \mathcal{F}_0 &= \{1,2,4\}, \text{ and}\\ \mathcal{R}(k) &= \begin{cases} k/2 & \text{if } k \equiv 0(2) \\ 3k+1 & \text{if } k \equiv 1(2), \end{cases}\end{aligned}$$

we obtain

$$\begin{aligned}\mathcal{F}_1 &= \{8\},\ \mathcal{F}_2 = \{16\},\ \mathcal{F}_3 = \{5,32\},\\ \mathcal{F}_4 &= \{10,64\},\ \mathcal{F}_5 = \{3,20,21,128\}, \ldots .\end{aligned}$$

It is not obvious how $\mathrm{recdep}(n)$ and a suitable "$\prec$"-relation can be expressed by a simple formula. □

3. Computational Model and Space and Time Effort

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

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

or

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

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 p, is a recursive function

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

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 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 p, is described in the following subsection.

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

The source code of a 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. (Blieberger 1994)),[1] and
- recursive calls to the procedure itself.

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

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

The syntax of $\mathit{nonrecursive}(f)$ is defined exactly the same way but $\mathit{rproc}(f \to \overline{f})$ is not part of $\mathit{statement}(f)$. By $f \to \overline{f}$ we denote that the parameters $\overline{f}$ are used for the recursive call.

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

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

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

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

where

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

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.

The recurrence relation for the space effort $\mathcal{S}$ is given by:

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

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

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

where

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

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 some actual parameter $f \in \mathcal{F}$, $\mathcal{T}(f)$ and $\mathcal{S}(f)$ can easily be determined at compile time. This can even be done if only upper and lower bounds of f exist, e.g. $l \preceq f \preceq u, l, u \in \mathcal{F}$, since $\max_{l \preceq f \preceq u} \mathcal{T}(f)$ and $\max_{l \preceq f \preceq u} \mathcal{S}(f)$ 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 recursive procedure *globally space-monotonical* and *globally time-monotonical*, respectively.

Remark 3.1. Note that for such procedures $f_1 \approx f_2$ 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*

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

and

$$\overline{\mathcal{T}}(l,u) = \max_{l \preceq f \preceq u} \mathcal{T}(f) = \max_{g \approx u} \mathcal{T}(g). \qquad \blacksquare$$

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. (Ada 1995, Ellis and Stroustrup 1990)), 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 to p.

In the following sections we will define "local" conditions. If these conditions hold, the underlying recursive procedure is called *locally space* or *locally time-monotonical*. It will turn out that if a 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 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 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. Rather 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 Recursive Procedures

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

The general form of $\mathcal{S}(f)$ simplifies to

$$\begin{aligned} \mathcal{S}(f) &= \sigma_0' \quad \text{if } f \in \mathcal{F}_0 \\ \mathcal{S}(f) &= \mathcal{D}(f) + \max\left(\sigma_{\max}, \mathcal{S}(\overline{f}_1), \ldots, \mathcal{S}(\overline{f}_m)\right) \quad \text{if } f \notin \mathcal{F}_0, \end{aligned}$$

where $\mathcal{R}(f) = \{\overline{f}_1, \ldots, \overline{f}_m\}$. Since the $\sigma_{\max}$-term is present in all $\mathcal{S}(f)$ provided that $f \notin \mathcal{F}_0$, we obtain

$$\begin{aligned} \mathcal{S}(f) &= \sigma_0 \quad \text{if } f \in \mathcal{F}_0 \\ \mathcal{S}(f) &= \mathcal{D}(f) + \max\left(\mathcal{S}(\overline{f}_1), \ldots, \mathcal{S}(\overline{f}_m)\right) \quad \text{if } f \notin \mathcal{F}_0, \end{aligned}$$

where $\sigma_0 = \max(\sigma_0', \sigma_{\max})$. Note that this does not change the value of $\mathcal{S}(f)$ if $f \in \mathcal{F} \setminus \mathcal{F}_0$.

Remark 4.1. Evaluating $\mathcal{S}(f)$ for recursive functions increases the height of the stack if the recursive call is part of an expression, but this can be avoided by

introducing temporary variables in the declarative part of the recursive function. (Note that this can be done at compile time!)

Definition 4.2. For each $f \in \mathcal{F}$ the *recursion digraph* $\mathcal{G}(f)$ is defined by the set of vertices $V = \mathcal{R}^*(f)$ and the set of edges $E = \{(g, \overline{g}) \mid g, \overline{g} \in V \text{ and } \overline{g} \in \mathcal{R}(g)\}$. Each vertex g is weighted by $\mathcal{D}(g)$.

Remark 4.2. Let $\mathcal{M}$ denote the path from f to some $f_0 \in \mathcal{F}_0$, $f_0 \in \mathcal{R}^*(f)$ with maximum weight $W(f) = \sum_g \mathcal{D}(g)$, where g runs through all vertices on $\mathcal{M}$. Then $W(f)$ is equal to $\mathcal{S}(f)$.

Remark 4.3. Using $\mathcal{G}(f)$, the quantity $\mathcal{S}(f)$ can be computed off-line at compile time in $O(\|V\| + \|E\|)$ time (cf. e.g. (Mehlhorn 1984a)).

Definition 4.3. Let p be a monotonical recursive procedure. We define $\mathcal{N} : \mathcal{F} \to \mathcal{F}$ to be a function such that $\mathcal{N}(f) = f_{\max}$, where $f_{\max}$ is such that $\mathcal{D}(f_{\max}) = \max_{\overline{f} \in \mathcal{R}(f)} \mathcal{D}(\overline{f})$ and $\mathrm{recdep}(f_{\max}) = \mathrm{recdep}(f) - 1$.

Definition 4.4. We call a monotonical recursive procedure p *locally space-monotonical* if $f_1 \prec f_2$ implies $\mathcal{D}(f_1) \leq \mathcal{D}(f_2)$ and, if $f_1 \approx f_2$ and $\mathcal{D}(f_1) \leq \mathcal{D}(f_2)$ implies $\mathcal{D}(\mathcal{N}(f_1)) \leq \mathcal{D}(\mathcal{N}(f_2))$.

Remark 4.4. If $\mathcal{D}(f)$ is constant, then the underlying recursive procedure is locally space-monotonical.

Theorem 4.1 *If p is a locally space-monotonical recursive procedure, then*

$$\mathcal{S}(f) = \sigma_0 + \sum_{0 \leq k < recdep(f)} \mathcal{D}(\mathcal{N}^{(k)}(f)),$$

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

Proof: Theorem 4.1 is proved if we can show that in $\mathcal{G}(f)$ 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 2. The path along $(f, \ldots, v_0, v_1, \ldots, v_r, w, \ldots, f_0)$, $f_0 \in \mathcal{F}_0$ is identical to $\mathcal{M}$. The path along $(f, \ldots, v_0, x_1, \ldots, x_s, w, \ldots, \overline{f}_0)$, $\overline{f}_0 \in \mathcal{F}_0$ is denoted by $\mathcal{M}'$.

By definition we have $\mathcal{D}(v_1) \geq \mathcal{D}(x_1)$. Thus

$$\mathcal{D}(\mathcal{N}(v_1)) = \mathcal{D}(v_2) \geq \mathcal{D}(\mathcal{N}(x_1)) \geq \mathcal{D}(x_2).$$

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

Because of Definition 4.3 we must have $r \geq s$ since $\mathrm{recdep}(v_i) = \mathrm{recdep}(v_{i+1}) + 1$. Hence we obviously have a contradiction. ■

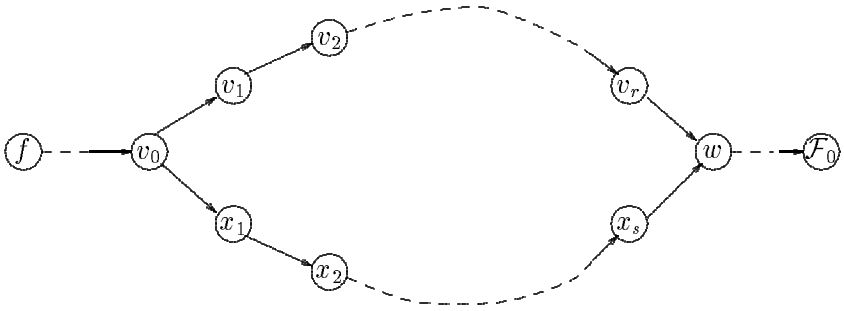

Figure 2. Paths in a Recursion Digraph

Corollary 4.1 *If $\mathcal{D}(f) = d$, $d \in N$ constant for all $f \in \mathcal{F}$,*

$$\mathcal{S}(f) = \sigma_0 + d \cdot recdep(f).$$

Remark 4.5. Theorem 4.1 and Corollary 4.1 show the intuitively clear connection between the recursion depth (the height of the stack) and the space complexity of recursive procedures.

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 is locally space-monotonical and $f_1 \prec f_2$, $f_1, f_2 \in \mathcal{F}$, then*

$$\mathcal{S}(f_1) \leq \mathcal{S}(f_2).$$

Proof: Clearly we have for all $0 \leq k < \text{recdep}(f_1)$

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

Hence we also have

$$\mathcal{D}(\mathcal{N}^{(k)}(f_1)) \leq \mathcal{D}(\mathcal{N}^{(k)}(f_2))$$

for all $0 \leq k < \text{recdep}(f_1)$.
Thus we obtain

$$\mathcal{S}(f_1) \leq \mathcal{S}(f_2)$$

and the lemma is proved. ■

Theorem 4.2 *If p is locally space-monotonical, then*

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

Proof: By virtue of Lemma 4.1,

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

It remains to take into account all $g \approx u$. Thus the theorem is proved. ■

Remark 4.6. Lemma 4.1 does even hold if

$$\mathcal{R}^*(f_1) \cap \mathcal{R}^*(f_2) = \emptyset.$$

The same applies to Theorem 4.2, i.e., it even holds if

$$\bigcap_{l \preceq f \preceq u} \mathcal{R}^*(f) = \emptyset.$$

In the following examples the constants σ_0, σ_d, and $\tilde{\sigma}$ are derived from the (source) code of the recursive procedures.

Example 1. For the Factorial Numbers we get $\mathcal{D}(n) = \sigma_d$, constant. Thus they are locally space-monotonical (cf. Remark 4.4) and we can even show that

$$\begin{aligned} \mathcal{S}(0) &= \sigma_0, \\ \mathcal{S}(n) &= \sigma_d + \mathcal{S}(n-1). \end{aligned}$$

Mentioning $\mathrm{recdep}(n) = n$ and $\mathcal{N}(n) = n - 1$ we derive

$$\mathcal{S}(n) = \sigma_0 + \sum_{k=0}^{n-1} \sigma_d = \sigma_0 + n \cdot \sigma_d.$$ □

Example 2. For the Fibonacci Numbers we obtain $\mathcal{D}(n) = \sigma_d$, constant. Thus they are locally space-monotonical (cf. Remark 4.4) and we can even show that

$$\begin{aligned} \mathcal{S}(0) &= \mathcal{S}(1) = \sigma_0, \\ \mathcal{S}(n) &= \sigma_d + \mathcal{S}(n-1). \end{aligned}$$

Mentioning $\mathrm{recdep}(n) = n - 1$ and $\mathcal{N}(n) = n - 1$ we derive for $n \geq 1$

$$\mathcal{S}(n) = \sigma_0 + \sum_{k=0}^{n-2} \sigma_d = \sigma_0 + (n-1) \cdot \sigma_d.$$ □

Example 3. Since $\mathcal{D}((x,y)) = \sigma_d$, constant, the Ackermann Function is locally space-monotonical (cf. Remark 4.4). In addition, since

$$\mathcal{N}((x,y)) = (x-1, \mathcal{A}(x,y-1)),$$

we get for the Ackermann Function (cf. (Lieger and Blieberger 1994))

$$\begin{aligned} \mathcal{S}((0,y)) &= \sigma_0, \\ \mathcal{S}((x,y)) &= \sigma_0 + \sigma_d \cdot (\mathcal{A}(x,y) + x - 2). \end{aligned}$$ □

Example 4. Mergesort is treated a little inexactly. An exact treatment is possible by use of parameter space morphisms which are introduced in Section 6.

Writing $n = y - x + 1$ we get $\mathcal{D}(n) = \sigma_d + \lfloor n/2 \rfloor \tilde{\sigma}$. Thus Mergesort is locally space-monotonical.

But we can also determine the exact behavior of Mergesort. We obtain

$$\begin{aligned} \mathcal{S}((x,x)) &= \sigma_0, \\ \mathcal{S}((x,y)) &= \sigma_d + \left(y - \left\lceil \frac{x+y}{2} \right\rceil \right) \tilde{\sigma} + \mathcal{S}\left(\left(x, \left\lfloor \frac{x+y}{2} \right\rfloor \right) \right). \end{aligned}$$

because

$$\mathcal{N}((x,y)) = (x, \lceil (x+y)/2 \rceil).$$

Since $\mathcal{S}(x,y)$ does only depend on the length of the array under consideration, we write again $n = y - x + 1$ and obtain

$$\begin{aligned} \mathcal{S}(1) &= \sigma_0, \\ \mathcal{S}(n) &= \sigma_d + \lfloor n/2 \rfloor \tilde{\sigma} + \mathcal{S}(\lceil n/2 \rceil). \end{aligned}$$

This can be solved and we finally get

$$\mathcal{S}(n) = \sigma_0 + \lceil \mathrm{ld}\ n \rceil \sigma_d + (n-1)\tilde{\sigma}.$$ □

5. The Time Effort of Recursive Procedures

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

$$\mathcal{T}(f) = \tau(f) + \sum_{\overline{f} \in \mathcal{R}(f)} \mathcal{T}(\overline{f}).$$

Definition 5.1. For all $f_1, f_2 \in \mathcal{F}$ we write $f_1 \sqsubseteq f_2$ (or equivalently $f_2 \sqsupseteq f_1$) if $f_1 \preceq f_2$ and $\tau(f_1) \leq \tau(f_2)$.

Definition 5.2. Let $f_1, f_2 \in \mathcal{F}$, $\mathcal{R}(f_i) = \{f_{i,1}, \ldots, f_{i,m_i}\}$, $i = 1, 2$, such that $f_{i,1} \sqsupseteq f_{i,2} \sqsupseteq \cdots \sqsupseteq f_{i,m_i-1} \sqsupseteq f_{i,m_i}$, $i = 1, 2$.

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

Remark 5.1. If for all $f_1, f_2 \in \mathcal{F}$ $f_1 \prec f_2$ implies $\tau(f_1) \leq \tau(f_2)$ and if $\|\mathcal{R}(f)\| \leq 1$ for all $f \in \mathcal{F}$, then the underlying recursive procedure is locally time-monotonical.

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

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

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

  For $j \geq i$ we derive
  $$\mathcal{T}(f_1) = \tau(f_1) + \sum_{\overline{f}_1 \in \mathcal{R}(f_1)} \mathcal{T}(\overline{f}_1) \quad \text{and} \tag{1}$$
  $$\mathcal{T}(f_2) = \tau(f_2) + \sum_{\overline{f}_2 \in \mathcal{R}(f_2)} \mathcal{T}(\overline{f}_2). \tag{2}$$
  By induction hypothesis the sum in (1) is smaller than or equal to the sum in (2). Since $\tau(f_1) \leq \tau(f_2)$, we get
  $$\mathcal{T}(f_1) \leq \mathcal{T}(f_2).$$ ■

Remark 5.2. If we have $f_1 \sqsubseteq f_2$ and $f_2 \sqsubseteq f_1$, we conclude that $f_1 \approx f_2$ and $\tau(f_1) = \tau(f_2)$. By Lemma 5.1 this implies $\mathcal{T}(f_1) = \mathcal{T}(f_2)$.

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 is locally time-monotonical, then*

$$\overline{\mathcal{T}}(l, u) = \max_{l \preceq f \preceq u} \mathcal{T}(f) = \max_{g \approx u} \mathcal{T}(g).$$ ■

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

Example 1. Because of Remark 5.1 the Factorial Numbers are locally time-monotonical.

In addition, we get

$$\begin{aligned} \mathcal{T}(0) &= \tau_0, \\ \mathcal{T}(n) &= \tau_d + \mathcal{T}(n-1). \end{aligned}$$

Mentioning $\mathrm{recdep}(n) = n$ we derive

$$\mathcal{T}(n) = \tau_0 + \sum_{k=0}^{n-1} \tau_d = \tau_0 + n \cdot \tau_d. \qquad \square$$

Example 2. It is easy to see that the Fibonacci Numbers are locally time-monotonical.

In addition, we derive

$$\begin{aligned} \mathcal{T}(0) &= \mathcal{T}(1) = \tau_0, \\ \mathcal{T}(n) &= \tau_d + \mathcal{T}(n-1) + \mathcal{T}(n-2). \end{aligned}$$

Thus for $n \geq 2$

$$\mathcal{T}(n) = f(n)\tau_0 + (f(n) - 1)\tau_d,$$

where $f(n)$ denotes the nth Fibonacci Number. $\square$

Example 3. It turns out that the Ackermann Function is *not* locally and *not* globally time-monotonical (cf. (Lieger and Blieberger 1994)). The following gives a simple counter-example.

Let $(x_1, y_1) = (1, 13)$ and $(x_2, y_2) = (3, 1)$. Because of $\mathrm{recdep}((1,13)) = 14 = \mathrm{recdep}((3,1))$ and (for all reasonable implementations) $\tau(1,13) = \tau(3,1)$ we find that $(1,13) \approx (3,1)$ and $(1,13) \sqsubseteq (3,1)$ as well as (for reasons of symmetry) $(1,13) \sqsupseteq (3,1)$ (cf. Remark 5.2).

Now $\mathcal{R}((1,13)) = \{(1,12),(0,14)\}$ and $\mathcal{R}((3,1)) = \{(3,0),(2,5)\}$.

As expected $\mathrm{recdep}(1,12) = \mathrm{recdep}(2,5) = 13 = 14 - 1$, $\tau(1,12) = \tau(2,5)$ and therefore $(1,12) \sqsubseteq (2,5)$ and $(1,12) \sqsupseteq (2,5)$.

Unfortunately $\mathrm{recdep}((0,14)) = 1 \neq 6 = \mathrm{recdep}((3,0))$ and thus $(0,14) \not\sqsupseteq (3,0)$, which contradicts Remark 5.2 and Remark 3.1. $\square$

Example 4. Writing $n = y - x + 1$, we have $\tau(n) = \tau_1 + n\tau_2$. Clearly, if $n_1 < n_2$, then $\tau(n_1) < \tau(n_2)$. This together with the fact that the length of the subarrays is $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ shows that Mergesort is locally time-monotonical.

In addition, we are able to show that

$$\begin{aligned} \mathcal{T}(1) &= \tau_0 \\ \mathcal{T}(n) &\leq \tau_1 + n\tau_2 + \mathcal{T}(\lfloor n/2 \rfloor) + \mathcal{T}(\lceil n/2 \rceil). \end{aligned}$$

The "$\leq$" originates from the fact that we can only find an upper bound for the number of iterations of the discrete loop from line 19 to 31 in Figure 1. The above recurrence relation can be solved and we finally get

$$\mathcal{T}(n) \leq n\tau_0 + (n-1)\tau_1 + \left(n - 2^{\lceil \mathrm{ld}\, n \rceil} + n\lceil \mathrm{ld}\, n \rceil\right)\tau_2. \qquad \square$$

6. Parameter Space Morphisms

The theoretical results of the previous sections are impressive as they are valid for recursive procedures with very general parameter spaces. 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.

Commonly, data structures are analyzed by informally introducing some sort of *complexity measure* (cf. (Vitter and Flajolet 1990)) or *size* (cf. (Mehlhorn 1984b, Aho et al. 1974)) of the data structure. We prefer a more formal approach.

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

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

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

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

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

Remark 6.1. Note that $\|\mathcal{M}(f)\| \geq 1$, but $\mathrm{recdep}(g_1) = \mathrm{recdep}(g_2)$ if $g_1 \in \mathcal{M}(f)$ and $g_2 \in \mathcal{M}(f)$.

Remark 6.2. Note that $\mathrm{recdep}_{\mathcal{H}}$ implies a (trivial) "$\prec$"-relation upon $\mathcal{F}'$, namely

$$f' \prec g' \quad \Leftrightarrow \quad \mathrm{recdep}_{\mathcal{H}}(f') < \mathrm{recdep}_{\mathcal{H}}(g') \tag{3}$$

for $f', g' \in \mathcal{F}'$. We will assume in the following that a "$\prec$"-relation exists which is consistent with equation (3) and denote it by "$\prec_{\mathcal{H}}$".

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

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

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

$$\mathcal{S}_\mathcal{H}(f') = \max_{f'=\mathcal{H}(g)} \mathcal{S}(g) \quad \text{and}$$

$$\mathcal{T}_\mathcal{H}(f') = \max_{f'=\mathcal{H}(g)} \mathcal{T}(g)$$

where $f' \in \mathcal{F}'$ and $g \in \mathcal{F}$.

Definition 6.4. If $f'_1 \preceq_\mathcal{H} f'_2$ implies $\mathcal{S}_\mathcal{H}(f'_1) \leq \mathcal{S}_\mathcal{H}(f'_2)$ and $\mathcal{T}_\mathcal{H}(f'_1) \leq \mathcal{T}_\mathcal{H}(f'_2)$, 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:

$$\begin{aligned}
\mathcal{D}_\mathcal{H}(f') &= \max_{f'=\mathcal{H}(g)} \mathcal{D}(g) \\
\tau_\mathcal{H}(f') &= \max_{f'=\mathcal{H}(g)} \tau(g) \\
\mathcal{R}_\mathcal{H}(f') &= \bigcup_{f'=\mathcal{H}(g)} \{\mathcal{H}(\mathcal{R}(g))\} \\
\mathcal{N}_\mathcal{H}(f') &= f'_{\max} ,
\end{aligned}$$

where

$$\Gamma(f') = \{g' \mid_{\overline{f}' \in \overline{\mathcal{R}}'} , \max_{\overline{\mathcal{R}}' \in \mathcal{R}_\mathcal{H}(f')} \text{recdep}_\mathcal{H} \overline{f}' = \text{recdep}_\mathcal{H} g'\},$$

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

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

Remark 6.3. Note that $\mathcal{H}(\mathcal{R}(g))$ is a multiset and $\mathcal{R}_\mathcal{H}(f')$ is a set of multisets.

Definition 6.3. A recursive procedure p is called $\mathcal{H}$-monotonical if for all $g' \in \mathcal{R}'$ and for all $\mathcal{R}' \in \mathcal{R}_\mathcal{H}(f')$ it holds that $g' \prec_\mathcal{H} f'$.

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

Lemma 6.1 *If p is $\mathcal{H}$-monontonical, the following relation holds:*

$$\mathcal{T}_{\mathcal{H}}(f') \leq \tau_{\mathcal{H}}(f') + \max_{\mathcal{R}' \in \mathcal{R}_{\mathcal{H}}(f')} \sum_{\overline{g}' \in \mathcal{R}'} \mathcal{T}_{\mathcal{H}}(\overline{g}')$$

Proof: By definition

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

which can be estimated by

$$\begin{aligned}
\mathcal{T}_{\mathcal{H}}(f') &\leq \tau_{\mathcal{H}}(f') + \max_{f'=\mathcal{H}(g)} \sum_{\overline{g} \in \mathcal{R}(g)} \mathcal{T}(\overline{g}) \\
&\leq \tau_{\mathcal{H}}(f') + \max_{f'=\mathcal{H}(g)} \sum_{\overline{g}' \in \mathcal{H}(\mathcal{R}(g))} \max_{\overline{g}'=\mathcal{H}(k)} \mathcal{T}(k) \\
&= \tau_{\mathcal{H}}(f') + \max_{f'=\mathcal{H}(g)} \sum_{\overline{g}' \in \mathcal{H}(\mathcal{R}(g))} \mathcal{T}_{\mathcal{H}}(\overline{g}') \\
&= \tau_{\mathcal{H}}(f') + \max_{\mathcal{R}' \in \mathcal{R}_{\mathcal{H}}(f')} \sum_{\overline{g}' \in \mathcal{R}'} \mathcal{T}_{\mathcal{H}}(\overline{g}'). \qquad \blacksquare
\end{aligned}$$

Lemma 6.2 *If p is $\mathcal{H}$-monontonical, the following relation holds:*

$$\mathcal{S}_{\mathcal{H}}(f') \leq \mathcal{D}_{\mathcal{H}}(f') + \max_{\mathcal{R}' \in \mathcal{R}_{\mathcal{H}}(f')} \max_{\overline{g}' \in \mathcal{R}'} \mathcal{S}_{\mathcal{H}}(\overline{g}')$$

Proof: The proof is suppressed since it is very similar to the proof of Lemma 6.1. $\blacksquare$

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

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

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

If for all $\overline{f}'_1 \sqsubseteq_{\mathcal{H}} \overline{f}'_2$, we have $m_{j_1,1} \leq m_{j_2,2}$ and $\overline{f}'_{j_1,1,r} \sqsubseteq_{\mathcal{H}} \overline{f}'_{j_2,2,r}$, $r = 1, \ldots, m_{j_1,1}$, for all j_1, j_2 such that $\mathcal{R}_{j_i}(f'_i) \in \mathcal{R}_{\mathcal{H}}(f'_i)$, 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*, is studied in the following.

Example 6. We start by showing that Quicksort[2] (without a parameter space morphism) is *not* locally and *not* globally time-monotonical. We assume that the time spent for arrays of length one and zero is equal to τ_0 and that the local time spent for comparing the elements of an array of length n is equal to $(n-1)\tau_1 + \tau_2$.

In the following we set up two permutations π_1 and π_2 of integer numbers. The recursion depth of Quicksort applied to both of them is the same (equal to 6). The length of π_1 is 14 and the length of π_2 is 13, but Quicksort uses more (overall) time to sort π_2 than it needs to sort π_1.

$\pi_1 = [8, 3, 1, 2, 6, 5, 7, 4, 9, 10, 11, 12, 13, 14]$ is transferred by Quicksort in the following way (underlined elements are placed at their final position)

$$\begin{aligned}
\pi_1 &\to [4, 3, 1, 2, 6, 5, 7, \underline{8}, 9, 10, 11, 12, 13, 14] \\
&\to [2, 3, 1, \underline{4}, 6, 5, 7, \underline{8}, \underline{9}, 10, 11, 12, 13, 14] \\
&\to [1, \underline{2}, 3, \underline{4}, 5, \underline{6}, 7, \underline{8}, \underline{9}, \underline{10}, 11, 12, 13, 14] \\
&\to [\underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \underline{6}, \underline{7}, \underline{8}, \underline{9}, \underline{10}, \underline{11}, 12, 13, 14] \\
&\to [\underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \underline{6}, \underline{7}, \underline{8}, \underline{9}, \underline{10}, \underline{11}, \underline{12}, 13, 14] \\
&\to [\underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \underline{6}, \underline{7}, \underline{8}, \underline{9}, \underline{10}, \underline{11}, \underline{12}, \underline{13}, 14] \\
&\to [\underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \underline{6}, \underline{7}, \underline{8}, \underline{9}, \underline{10}, \underline{11}, \underline{12}, \underline{13}, \underline{14}]
\end{aligned}$$

This results in $\mathcal{T}(\pi_1) = 10\tau_0 + 38\tau_1 + 9\tau_2$.

On the other hand $\pi_2 = [7, 2, 3, 4, 5, 6, 1, 8, 9, 10, 11, 12, 13]$ is sorted by Quicksort in the following way

$$\begin{aligned}
\pi_2 &\to [1, 2, 3, 4, 5, 6, \underline{7}, 8, 9, 10, 11, 12, 13] \\
&\to [\underline{1}, 2, 3, 4, 5, 6, \underline{7}, \underline{8}, 9, 10, 11, 12, 13] \\
&\to [\underline{1}, \underline{2}, 3, 4, 5, 6, \underline{7}, \underline{8}, \underline{9}, 10, 11, 12, 13] \\
&\to [\underline{1}, \underline{2}, \underline{3}, 4, 5, 6, \underline{7}, \underline{8}, \underline{9}, \underline{10}, 11, 12, 13] \\
&\to [\underline{1}, \underline{2}, \underline{3}, \underline{4}, 5, 6, \underline{7}, \underline{8}, \underline{9}, \underline{10}, \underline{11}, 12, 13] \\
&\to [\underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, 6, \underline{7}, \underline{8}, \underline{9}, \underline{10}, \underline{11}, \underline{12}, 13] \\
&\to [\underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \underline{6}, \underline{7}, \underline{8}, \underline{9}, \underline{10}, \underline{11}, \underline{12}, \underline{13}]
\end{aligned}$$

Here we get $\mathcal{T}(\pi_2) = 12\tau_0 + 42\tau_1 + 11\tau_2$, which proves that Quicksort is not locally and not globally time-monotonical, because $\pi_1 \approx \pi_2$ would imply $\mathcal{T}(\pi_1) = \mathcal{T}(\pi_2)$ (cf. Remark 5.2 and Remark 3.1).

Now, mapping input arrays of Quicksort to their length by $\mathcal{H}(f) = \mathrm{size}(f) = n$, we obtain a parameter space morphism. It is easy to see that $\mathrm{recdep}_{\mathcal{H}}(n) = n - 1$,

$$\mathcal{R}_{\mathcal{H}}(n) = \bigcup_{1 \le i \le n} \{\{i-1, n-i\}\},$$

and Quicksort is $\mathcal{H}$-monotonical.

Clearly, we have $\tau_{\mathcal{H}}(n) = (n-1)\tau_1 + \tau_2$. In order to see that Quicksort is not locally $\mathcal{H}$-time-monotonical, consider $n_1 = 5$ and $n_2 = 6$. Obviously $n_1 \prec_{\mathcal{H}} n_2$, but the direct successors of n_1 include $(2, 2)$ and those of n_2 include $(4, 1)$. As expected $2 \prec_{\mathcal{H}} 4$, but $2 \not\prec_{\mathcal{H}} 1$.

Nevertheless, strengthening Lemma 6.1, the following recurrence relation is valid:

$$\mathcal{T}_{\mathcal{H}}(n) = (n-1)\tau_1 + \tau_2 + \max_{1 \le i \le n} \left(\mathcal{T}_{\mathcal{H}}(i-1) + \mathcal{T}_{\mathcal{H}}(n-i)\right).$$

Mentioning $\mathcal{T}_{\mathcal{H}}(0) = \mathcal{T}_{\mathcal{H}}(1) = \tau_0$, this relation can be solved and we finally obtain for all $n \ge 0$

$$\mathcal{T}_{\mathcal{H}}(n) = \frac{n(n-1)}{2}\tau_1 + n\tau_2 + (n+1)\tau_0,$$

which shows that Quicksort is globally $\mathcal{H}$-time-monotonical. □

Example 6 shows that a recursive procedure p which is not globally time-monotonical, can be globally $\mathcal{H}$-time-monotonical for some suitable morphism $\mathcal{H}$. Interestingly, we loose information on the timing behavior by applying $\mathcal{H}$ (notice the max-terms in various definitions), but we gain monotonicity, i.e., we get coarser, but more well-behaved estimates.

Finally, we would like to note that in most cases a morphism $\mathcal{H} : \mathcal{F} \to N$ will be used. This can be supported by the following arguments:

- Parameter space morphisms are useful only if $\mathcal{D}_{\mathcal{H}}$ and $\tau_{\mathcal{H}}$ (cf. Definition 6.5) can be found easily. In most cases this can be obtained if already $\mathcal{D}$ and τ do depend on some $f' \in \mathcal{F}'$ and not on some $f \in \mathcal{F}$. Thus we are left with determining how the function $\mathcal{D}$ and τ will look like.

- The function $\mathcal{D}$ 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 $\mathcal{D}$ to be a polynomial function from N to N.

- The function τ will usually depend on the number of iterations of the loops within the code of the underlying recursive procedure. Again, we expect τ to be a function from N to N (or 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. (Blieberger 1994)), respectively.

Summing up, usually $\mathcal{D}$ and τ are functions from N to N (or R). Thus one can suspect that a morphism from $\mathcal{F}$ to 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 $\mathcal{F}$ by $f_1 \lhd f_2$ exists such that for all $f_1, f_2 \in \mathcal{F}$, $f_1 \lhd f_2$ $(f_1 \neq f_2)$ implies

1. $f_1 \preceq f_2$,
2. the underlying recursive procedure is locally space-monotonical, and
3. the underlying recursive procedure is locally time-monotonical,

we call $\mathcal{F}$ *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. Note that for Mergesort the "$\prec$"-relation is a valid "$\lhd$"-relation too (cf. end of Section 2).

We are able to show the following theorems.

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

$$\overline{\mathcal{S}}(l,u) = \max_{l \trianglelefteq f \trianglelefteq u} \mathcal{S}(f) = \mathcal{S}(u).$$

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

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

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

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

$$\overline{\mathcal{T}}(l,u) = \max_{l \trianglelefteq f \trianglelefteq u} \mathcal{T}(f) = \mathcal{T}(u).$$

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

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

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

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

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

7.1. The recursion depth

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

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

$$f_1 \prec f_2, \quad f_2 \prec f_1, \quad \text{or} \quad f_1 \approx f_2$$

for all $f_1, f_2 \in \mathcal{F}$.

If no "$\lhd$"-relation exists, the recursion depth must be bounded by a programmer supplied constant `R`. 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 `U`.

Since it is extremely difficult to verify the function `recdep` supplied by the programmer at compile time[3], the correctness of `recdep` is checked at runtime. Note that it is this check that enforces the well-definedness of the recursive procedure. To be more specific, the following conditions must be met:

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

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 $\mathcal{D}(f)$ is constant or if there is a simple connection between $\mathcal{D}(f)$ and `recdep`(f), 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`: $N \to \mathcal{F}$, which given some $k = \mathrm{recdep}(f)$ returns $\overline{f}$ such that $f \approx \overline{f}$ and $\mathcal{S}(\overline{f}) = \max_{\overline{f} \approx g} \mathcal{S}(g)$.

At runtime for each $f \in \mathcal{F}$, it is checked whether $\mathcal{S}(f) \leq \mathcal{S}(u_k)$ where $k =$ `recdep`(f) and $u_k =$ `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.

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 $\mathcal{F}$ to 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 $f \in \mathcal{F}$ and not for $f' \in \mathcal{F}'$ (i.e. for $f' \in N$).

7.4. Checking Time Properties

If there is a simple connection between $\tau(f)$ and $\mathrm{recdep}(f)$ and if $\|\mathcal{R}(f)\| \leq 1$, 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`: $N \to \mathcal{F}$, which given some $k = \mathrm{recdep}(f)$ returns $\overline{f}$ such that $f \approx \overline{f}$ and $\mathcal{T}(\overline{f}) = \max_{\overline{f} \approx g} \mathcal{T}(g)$.

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

Checking of local time properties with help of a "$\triangleleft$"-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 4. In our discussion of Mergesort the reader will discover that morphisms have been used several times. We leave it to the reader to perform an exact treatment. □

Example 7. *Balanced trees* are interesting since operations defined upon them can easily be implemented by recursion and their recursion depth is usually bounded above by $O(\mathrm{ld}\ n)$, where n denotes the number of nodes in the tree. We study BB[α]-trees in some detail (cf. (Mehlhorn 1984b, Blum and Mehlhorn 1980, Nievergelt and Reingold 1973)). In Figure 3 part of the specification of a BB[α]-tree package is given. Figure 4 shows all additional functions necessary for a recursive implementation of the procedure `insert` using a morphism.

In the following let denote $\mathcal{E}$ the set of elements stored in the BB[α]-tree and let denote $\mathcal{B}_\alpha$ the set of all BB[α]-trees. Then the function `insert` is a mapping `insert`: $\mathcal{B}_\alpha \times \mathcal{E} \to \mathcal{B}_\alpha$ and the `current_size` of the tree can be considered a function `current_size`: $\mathcal{B}_\alpha \to N$.

Let $B \in \mathcal{B}_\alpha$ and $E \in \mathcal{E}$. Then $\mathcal{H}(B) = \texttt{current\_size}(B) = n$ implies

$$\texttt{current\_size}(\texttt{insert}(B, E)) = \begin{cases} n+1, & \text{if } E \notin B, \text{ and} \\ n, & \text{if } E \in B. \end{cases}$$

In the following we will assume that only the first case is encountered.

Obviously the set $\mathcal{M}$ exists and the recursion depth is found to be

$$\mathrm{recdep}_{\mathcal{H}}(n) = 1 + \left\lfloor \frac{\log(n+1) - 1}{\log(1/(1-\alpha))} \right\rfloor .$$

In addition, we have

1. $\mathcal{D}_{\mathcal{H}}(n) = \sigma_1$,
2. $\tau_{\mathcal{H}}(n) = \tau_1$,
3. $\mathcal{R}_{\mathcal{H}}(n) = \{\{i\} \mid \lceil \alpha n \rceil \le i \le \lfloor (1-\alpha)n \rfloor\}$, and

```
1.  generic
2.
3.     size: natural;
4.     alpha: float;
5.     type element is private;
6.     with function "<"(left,right:element) return boolean is <>;
7.
8.  package BB_alpha_tree is
9.
10.    type tree is limited private;
11.
12.    procedure insert(an: element; into: tree);
13.
14.    -- other operations suppressed
15.
16. private
17.    type tree is
18.       record
19.          current_size: natural;          -- the current number of nodes in the tree
20.          -- other stuff representing the tree structure suppressed
21.       end record;
22. end BB_alpha_tree;
```

Figure 3. Ada Code of Specification of BB[α]-tree (Fragment)

```
1.  package body BB_alpha_tree is
2.
3.     subtype node_number is natural range 0 .. size;
4.
5.     recursive procedure insert(an: element; into: tree)
6.
7.        with function morphism(t: tree)
8.              return node_number is
9.        begin
10.         return t.current_size;
11.       end morphism;
12.
13.       with function recdep(current_size: node_number)
14.             return natural is
15.       begin
16.         return floor(1.0+(ld(current_size+1)-1.0)/ld(1.0/(1.0-alpha)));
17.       end recdep;
18.
19.    is
20.    begin
21.       -- recursive implementation of insert
22.    end insert;
23. end BB_alpha_tree;
```

Figure 4. Recursive Implementation of BB[α]-tree (Fragment)

4. $\mathcal{N}_{\mathcal{H}}(n) = \lfloor (1-\alpha)n \rfloor$.

Clearly `insert` is $\mathcal{H}$-monotonical. Thus it is also locally $\mathcal{H}$-space-monotonical (cf. Remark 4.4) and locally $\mathcal{H}$-time-monotonical (cf. Remark 5.1).

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

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 (Puschner and Koza 1989), 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 (Puschner and Koza 1989) excludes recursions.

In (Park 1993) such information on data influencing execution time can be incorporated into the program by means of program path analysis, but (Park 1993) 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.

Nevertheless a lot of work needs to be done in the future. The following lists a few items.

- Indirect recursive procedures remain to be studied.

- Other models of space behavior can be imagined. In this paper we assume that $\mathcal{D}(f)$ is constant during the execution of p without taking into account recursion. Some programming languages permit block-statements which can contain local declarations (cf. e.g. Ada (Ada 1995)). If a recursive procedure contains such a block-statement, $\mathcal{D}(f)$ may increase and decrease during the execution of p. (Reimplementing Mergesort using such block-statements, approximately halfs $\mathcal{S}(n)$.)

- The compile time proofs mentioned in Sections 3 and 7 should be done automatically. As already mentioned this is one of the goals of Project WOOP.

The purpose of this paper is to show that recursion can and should be used in real-time applications. We think that this goal is reached and there are no more reasons to exclude recursive procedures from real-time programming languages.

Notes

1. This means that the number of iterations does not depend on the result of one or more recursive calls.
2. An implementation of the well-known Quicksort algorithm can be found in any good book on algorithms and data structures (cf. e.g. (Knuth 1973, Mehlhorn 1984b, Sedgewick 1988)).
3. In fact it is undecidable, whether two given Turing machines accept the same language.

References

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

Johann Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.

Johann Blieberger and Roland Lieger. Real-time recursive procedures. In *Proceedings of the 7th EUROMICRO Workshop on Real-Time Systems*, pages 229–235, Odense, 1995.

Norbert Blum and Kurt Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980.

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

Arnold Businger. *PORTAL Language Description*, volume 198 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1985.

DIN 66253, Teil 2, Beuth Verlag, Berlin. *Programmiersprache PEARL, Full PEARL*, 1982.

Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

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

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

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

Douglas R. Hofstadter. *Gödel, Escher, Bach - an Eternal Golden Braid.* Basic Books, New York, 1979.

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

ISO/IEC 8652. *Ada Reference manual*, 1995.

Eugene Kligerman and Alexander D. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, 12(9):941–949, 1986.

Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, Mass., 1973.

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*, pages 25–40, 1989.

Roland Lieger and Johann Blieberger. The Ackermann-function effort in space and time. Technical Report 183/1-48, Department of Automation, Technical University Vienna, 1994.

C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms.* Springer-Verlag, Berlin, 1984.

Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms.* Springer-Verlag, Berlin, 1984.

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

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

I. Nievergelt and E.M. Reingold. Binary search trees of bounded balance. *SIAM Journal of Computing*, 2(1):33–43, 1973.

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

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

Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, second edition, 1988.

Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

Jeffrey S. Vitter and Phillipe Flajolet. Average-case analysis of algorithms and data structures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 431–524. North-Holland, 1990.

Received Date
Accepted Date
Final Manuscript Date