Real-Time Recursive Procedures

J. Blieberger

Department of Automation
Technical University Vienna
Vienna, AUSTRIA, A-1040

R. Lieger

Department of Automation
Technical University Vienna
Vienna, AUSTRIA, A-1040

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.*

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 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. [8, 5, 10]). 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 [1], the second one will be treated 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. [6, 5]), PEARL (cf. [3]), Real-Time Concurrent C (cf. [4]), and the MARS-approach (cf. [7, 11]).

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 and/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. Moreover the conditions are simple enough such that many important recursive algorithms can be implemented, for example *Mergesort* or recursive tree-traversal algorithms.

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

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

$$\mathrm{fac}(0) = 1, \mathrm{fac}(n) = n \cdot \mathrm{fac}(n-1) \quad \text{if } n > 0.$$

Example 2. A recursive version of *Mergesort*, the source code of which is shown in Figure 1.

2 Some Definitions

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.

From now on, when we use the term recursive procedure, we mean well-defined recursive procedure. Note that it cannot be decided whether a recursive procedure is well-defined or not.

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

Supported by the Austrian Science Foundation (FWF) under grant P10188-MAT.

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

$$\mathcal{R}_0(f) = \{f\}$$

$$\mathcal{R}_{k+1}(f) = \mathcal{R}_k(f) \cup \{\overline{f} \mid \overline{f} \in \mathcal{R}(g) \text{ where } g \in \mathcal{R}_k(f)\}$$

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 $p(f)$.

Note that $\mathcal{R}_1(f) = \mathcal{R}(f)$ from Definition 2.3.

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

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

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. 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. Denoting by (x, y) the interval of the array to be sorted by Mergesort, we derive

$$\mathcal{F} = N^2,$$

$$\mathcal{R}((x,y)) = \left\{ \left(x, \left\lfloor \frac{x+y}{2} \right\rfloor\right), \left(\left\lfloor \frac{x+y}{2} \right\rfloor + 1, y\right) \right\},$$

and

$$\mathcal{F}_0 = \{(x,x) \mid x \in N\},$$

$$\mathcal{F}_k = \{(x,y) \mid 2^{k-1} \leq (y-x) < 2^k\}.$$

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.

3 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 microseconds 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. [1]),* and
- recursive calls to the procedure itself.

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

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

The syntax of *nonrecursive*(f) is defined exactly the same way but *rproc*$(f \to \overline{f})$ is not part of *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 the evaluating the condition whether f belongs to the terminating values or not and is known a priori; the second one

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

can be computed using the method described below, but without giving rise to a recurrence relation.

For the recursive part we obtain

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

where

$$\mathcal{T}[recursive(f)] = \mathcal{T}[seq(f)],$$

$$\mathcal{T}[seq(f)] = \sum \mathcal{T}[statement(f)],$$

$$\mathcal{T}[ifs(f)] = \mathcal{T}[cond(f)] + \mathcal{T}[\text{then\_or\_else}]$$

where

$$\mathcal{T}[\text{then\_or\_else}] = \mathcal{T}[seq_{\texttt{True}}(f)]$$

if the condition evaluates to true,

$$\mathcal{T}[\text{then\_or\_else}] = \mathcal{T}[seq_{\texttt{False}}(f)]$$

otherwise,

$$\mathcal{T}[bloops(f)] = <bound(f)> \mathcal{T}[seq(f)],$$

$$\mathcal{T}[simple(f)] = \tau(simple),$$

$$\mathcal{T}[rproc(f \to \overline{f})] = \mathcal{T}(\overline{f})$$

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

Note that $<bound(f)>$ 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:

$$\mathcal{S}(f) = \mathcal{S}(decl\_part(f)) +$$

$$\max(\sigma[f \in \mathcal{F}_0], \sigma[nonrecursive(f)]) \quad \text{if } f \in \mathcal{F}_0,$$

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.

For the recursive part we get

$$\mathcal{S}(f) = \mathcal{S}(decl\_part(f)) +$$

$$\max(\sigma[f \in \mathcal{F}_0], \sigma[recursive(f)]) \quad \text{if } f \notin \mathcal{F}_0,$$

where

$$\mathcal{S}[recursive(f)] = \mathcal{S}[seq(f)],$$

$$\mathcal{S}[seq(f)] = \max\left(\mathcal{S}[statement(f)]\right),$$

$$\mathcal{S}[ifs(f)] = \max\left(\mathcal{S}[cond(f)], \mathcal{S}[seq_{\texttt{True}}(f)]\right)$$

if the condition evaluates to true,

$$\mathcal{S}[ifs(f)] = \max\left(\mathcal{S}[cond(f)], \mathcal{S}[seq_{\texttt{False}}(f)]\right)$$

otherwise,

$$\mathcal{S}[bloops(f)] = \max(\mathcal{S}[seq(f)]),$$

$$\mathcal{S}[simple(f)] = \sigma(simple),$$

$$\mathcal{S}[rproc(f \to \overline{f})] = \mathcal{S}(\overline{f})$$

where $\sigma(simple)$ is known a priori and $\mathcal{S}(decl\_part(f))$ 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 parameters $f \in \mathcal{F}$, $\mathcal{T}(f)$ 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 at compile time.

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.

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

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

and

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

respectively.

In the following sections we will replace these global properties by local ones, which are well-suited for being checked at compile time and, if they can be proved to hold, imply that the global properties hold too.

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

$$\mathcal{S}(f) = \sigma_0' \quad \text{if } f \in \mathcal{F}_0$$

$$\mathcal{S}(f) = \mathcal{D}(f) + \max\left(\sigma_{\max}, \mathcal{S}(\overline{f}_1), \ldots, \mathcal{S}(\overline{f}_m)\right)$$

if $f \notin \mathcal{F}_0$, 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

$$\mathcal{S}(f) = \sigma_0 \quad \text{if } f \in \mathcal{F}_0$$

$$\mathcal{S}(f) = \mathcal{D}(f) + \max\left(\mathcal{S}(\overline{f}_1), \ldots, \mathcal{S}(\overline{f}_m)\right)$$

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

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 4.1. 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 4.2. 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. [9]).

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

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.

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 $0 \leq k < \mathrm{recdep}(f_1)$

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

Hence we also have

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

for all $0 \leq k < \mathrm{recdep}(f_1)$.

Thus we obtain

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

and the lemma is proved.

This lemma enables us to find upper and lower bounds of the space behavior if a range of parameter values is given.

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

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

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.

Corollary 1. If $\mathcal{D}(f)$ is constant for all $f \in \mathcal{F}$, the underlying recursive procedure clearly is locally space-monotonical.

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

$$\mathcal{S}(0) = \sigma_0, \qquad \mathcal{S}(n) = \sigma_d + \mathcal{S}(n-1).$$

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. 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

$$\mathcal{S}((x,x)) = \sigma_0,$$

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

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

$$\mathcal{S}(1) = \sigma_0,$$

$$\mathcal{S}(n) = \sigma_d + \lfloor n/2 \rfloor \tilde{\sigma} + \mathcal{S}(\lceil n/2 \rceil).$$

This can be solved and we finally get

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

Note that it is very easy to verify that our examples are locally space-monotonical, but difficult to derive the exact worst-case space behavior. If a certain recursive procedure p is locally space-monotonical and an upper bound of the recursion depth is known, an upper bound of the space to be used by p can be found by Theorem 4.2 *at compile time* with little effort.

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 $f_{i,1} \sqsupseteq f_{i,2} \sqsupseteq \ldots \sqsupseteq f_{i,m_i-1} \sqsupseteq f_{i,m_i}$, $i = 1, 2$.

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.*

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
  $$\mathcal{T}(f_1) = \tau(f_1) + \sum_{\overline{f}_1 \in \mathcal{R}(f_1)} \mathcal{T}(\overline{f}_1)$$
  and
  $$\mathcal{T}(f_2) = \tau(f_2) + \sum_{\overline{f}_2 \in \mathcal{R}(f_2)} \mathcal{T}(\overline{f}_2).$$
  By induction hypothesis the sum in the first equation is smaller than or equal to the sum in the second one. Since $\tau(f_1) \leq \tau(f_2)$, we get
  $$\mathcal{T}(f_1) \leq \mathcal{T}(f_2).$$

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 is locally time-monotonical, then*

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

Proof. The proof of Theorem 5.1 is very similar to that of Theorem 4.2. Thus it is left to the reader.

Corollary 1. If $\|\mathcal{R}(f)\| \leq 1$ and $\tau(f)$ is constant for all $f \in \mathcal{F}$, the underlying recursive procedure is locally time-monotonical.

Example 1. It is easy to see that the Factorial Numbers are locally time-monotonical.

In addition, we get

$$\mathcal{T}(0) = \tau_0, \; \mathcal{T}(n) = \tau_d + \mathcal{T}(n-1)$$

and derive

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

Example 2. 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

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

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 20 to 35 in Figure 1. The above recurrence relation can be solved and we finally get

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

Again, showing that our examples are locally time-monotonical is very easy, while deriving exact worst-case timing estimates is more difficult. In addition, Theorem 5.1 can be used to find an upper bound of the time to be used, if the recursion depth of the underlying recursive procedure is bounded.

6 Compile Time vs. Runtime Checking

In the previous sections we have set up conditions which guarantee real-time behavior of recursive procedures in the following sense: If a recursive procedure is locally space-monotonical and locally time-monotonical and if the recursion depth of the procedure is bounded, then the worst-case space and time behavior can be determined at compile time.

For the compiler this means that it must not only be able to prove certain properties of the recursive procedure, but also that it must determine the recursion depth of the recursive procedure. Since it is undecidable to derive this knowledge from the code of the recursive procedure, the programmer has to provide a function `recdep` that given a certain parameter of the recursive procedure computes its recursion depth. Of course this function must not be recursive.

But now it is undecidable to verify the function `recdep` at compile time. Thus `recdep` is checked at runtime. Notice also that this is the only way to check well-definedness of a recursive procedure, which is undecidable too.

To be more specific, the following conditions are checked:

1. `recdep(f)` can be computed for each $f \in \mathcal{F}$ without a runtime error

2. for all $\overline{f} \in \mathcal{R}(f)$, $\texttt{recdep}(\overline{f}) < \texttt{recdep}(f)$

3. at least one $\overline{f} \in \mathcal{R}(f)$ has to exist such that $\mathrm{recdep}(\overline{f}) = \mathrm{recdep}(f) - 1$

4. for all $f \in \mathcal{F}$, $\texttt{recdep}(f) \leq \texttt{R}$

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

If the compiler cannot prove the properties mentioned above, additional runtime checks become necessary to guarantee that all space and time requirements are met. Details can be found in [2].

7 Implementing Mergesort

A recursive implementation of Mergesort using our real-time recursions is given in Figure 3. Function `ceiling(x)` is supposed to implement $\lceil x \rceil$ and `ld(x)` denotes the binary logarithm.

Note that the programmer's task is extremely easy. All necessary proofs and checks can be performed by the compiler.

The development of such a compiler is part of Project WOOP which is carried out at the *Department of Automation* at the *Technical University of Vienna*.

8 Conclusion

In this paper we have demonstrated how recursive procedures can be constrained in order to use them in real-time applications without harm.

We have set up conditions which easily can be checked at compile time. Thus our approach is well-suited for real-time applications.

In our forth-coming paper [2] we will give more examples of our approach. In addition, we develop a method how one can abstract from "unnecessary" details of the algorithm during estimating space and time properties, and we give prerequisites for real-time programming languages which incorporate our approach.

Finally we would like to mention that our approach can be applied successfully to many important recursive algorithms, e.g. many sorting algorithms and recursive tree traversal algorithms such as weight-balanced trees, AVL trees, and so on (cf. [2]).

References

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

[2] J. Blieberger and R. Lieger. Worst-case space and time complexity of recursive procedures. (to appear), 1994.

[3] DIN 66 253, Teil 2, Beuth Verlag, Berlin. *Programmiersprache PEARL, Full PEARL*, 1982.

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

[5] W. A. Halang and A. D. Stoyenko. *Constructing predictable real time systems*. Kluwer Academic Publishers, Boston, 1991.

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

[7] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40, 1989.

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

[9] K. Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms*. Springer-Verlag, Berlin, 1984.

[10] A. 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.

[11] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.

```
N: constant integer := ... ;
-- number of elements to be sorted
subtype index is integer range 1 .. N;
type gen_sort_array is array
  (index range <>) of ... ;
subtype sort_array is gen_sort_array (index);
sort_arr: sort_array;

procedure merge_sort(from,to: index) is
  m: constant integer := (from+to)/2 + 1;
  subtype aux_array is gen_sort_array(m..to);
  aux: aux_array;
  p,q,r: integer;
begin
  if from = to then
    return;
  end if;
  merge_sort(from,m-1);
  merge_sort(m,to);
  aux := sort_arr(m..to);
  discrete (p,q,r) := (m-1,aux'last,to)
      in reverse (m-1,aux'last,to) ..
  (from-1,aux'first,from)
      new (p,q,r) := (p-1,q,r-1) |
  (p,q-1,r-1) loop
    if p < from or else
target(p) < aux(q) then
      target(r) := aux(q);
      r := r-1;
      q := q-1;
    else
      target(r) := target(p);
      r := r-1;
      p := p-1;
    end if;
  end loop;
end merge_sort;
```

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

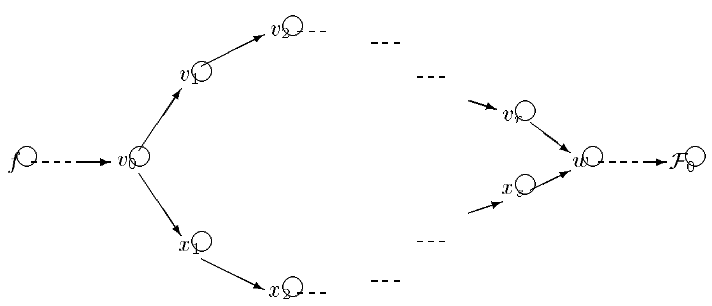

Figure 2: Paths in a Recursion Digraph

```
N: constant integer := ... ;
  -- number of elements to be sorted
subtype index is integer range 1 .. N;
type gen_sort_array is array
    (index range <>) of ... ;
subtype sort_array is gen_sort_array (index);
sort_arr: sort_array;

recursive procedure merge_sort(
from,to: index) is

  with function recdep(from,to: index)
    return natural is
  begin
    return ceiling(ld(to-from+1));
  end;

  ...

begin

  ...

end merge_sort;
```

Figure 3: Recursive implementation of Mergesort using real-time recursion