DISSERTATION

Bounding the Worst-Case Execution Time of General Loops and Recursion

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften
unter der Leitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Johann Blieberger
Institut für Rechnergestützte Automation (E183-1)

eingereicht an der Technischen Universität Wien
Technisch-Naturwissenschaftl. Fakultät

von

Dipl.-Ing. Roland Lieger
Matr.Nr. 8825560

Goethegasse 39
2340 Mödling

Wien, im Oktober 2002

Kurzfassung

Bounding the Worst-Case Execution Time of General Loops and Recursion

von Dipl.-Ing. Roland Lieger

Im Bereich der Echtzeitsysteme ist es von großer Bedeutung, die *maximale* Laufzeit eines Programmes (worst-case execution time (WCET)) bestimmen zu können (etwa um damit die erforderliche Rechenkapazität abschätzen zu können oder korrekte Schedulingentscheidungen zu treffen).

Solange ein Programm nur aus einer linearen Abfolge von primitiven Befehlen besteht (Zuweisungen, Unterprogrammaufrufen (ohne Rekursion)), ist eine solche Laufzeitbestimmung trivial. Für `For`-Schleifen (mit bekannten Start- und Endwerten) erhält man die (exakte) Laufzeit durch Multiplikation der Laufzeit für einen Schleifendurchlauf mit der Zahl der Durchläufe. Bei Alternativen im Programmablauf (`if-then-else, switch-case`) kann man den maximal erforderlichen Rechenaufwand durch die Laufzeit der aufwendigsten Alternative abschranken. Allgemeine Schleifen (`while, repeat`) lassen sich auf diese Weise aber nicht abschranken, da durch einfache Codeanalyse nicht feststellbar ist, wie oft sie durchlaufen werden. Auch Rekursionen mit ihrer unbekannten Rekursionstiefe/ -breite sind nicht abschrankbar.

Im Echtzeitbereich ist es daher üblich, auf allgemeine Schleifen und Rekursionen zu verzichten und den Programmierer zu nötigen, stattdessen auf `for`-Schleifen auszuweichen, wodurch nur allzu oft die Eleganz und damit die Fehlerfreiheit des Programms leidet.

Aus der theoretischen Informatik ist bekannt, dass eine generelle Abschrankung der Laufzeit allgemeiner Schleifen und Rekursionen wegen des Halteproblems unmöglich ist. In dieser Dissertation wird allerdings gezeigt, dass sich große Klassen praktisch relevanter Fälle allgemeiner Schleifen mit den hier vorgestellten *discreten*-Scheifen behandeln lassen. Ausgehend von (vom Programmierer angegebenen) Bedingungen ist es dabei möglich, automatisch eine Laufzeitabschrankung vorzunehmen und trotzdem (fast) die gesamte Leistungsfähigkeit allgemeiner Schleifen zu nutzen. Weiters wird gezeigt, wie sich das Konzept der Bedinungen auch auf Rekursionen ausdehnen läßt, sodass auch dieses mächtige Konzept für die Echtzeitprogrammierung zur Verfügung steht.

Abstract

Bounding the Worst-Case Execution Time of General Loops and Recursion

by Dipl.-Ing. Roland Lieger

In the field of real-time systems it is essential to know the *worst-case* execution time (WCET) of a program (e.g. for estimating the necessary hardware capacity or for making correct scheduling decisions).

While a program consists only of a linear sequence of primitive commands (assignments, subroutine calls (without recursion)), computing the WCET is trivial. *For*-loops (with known upper and lower bounds) pose no problem either, since their WCET can easily be computed by multiplying the WCET of one pass through the loop body with the number of passes. For conditional statements (*if-then-else, switch-case*) the WCET can be bounded using the WCET of the most demanding alternative. Unfortunatly the WCET of general loops (`while`, `repeat`) cannot be bounded, since it is not possible to determine the (maximal) number of passes through the loop from simple code analysis. Similar reasoning applies to recursions that have undetermined recursion depth/width.

In real-time programming it is therefore common to forbid the usage of general loops and recursion and thus force the programmer to use `for`-loops as only loop construct. Often this forced transformation totally destroys the programs elegance and drastically increases the proneness for errors. It is well known from theoretical computer science that a bound on the WCET of general loops and recursion cannot be found due to the halting-problem. However this thesis will introduce `discrete` loops and will show that a wide class of commonly used general loops can also be easily expressed as `discrete` loops. Using additional conditions (given by the programmer) it is possible for the compiler to compute the WCET of these loops while the ease of use and computational power of general loops is (almost completely) maintained. The idea of adding conditions to loops for the computation of WCET can also be applied to recursion, making this powerful tool available for real-time programming too.

Contents

Chapter 1
INTRODUCTION

1.1 Overview

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. [30, 19, 33]). Thus the *worst-case execution time (WCET)* of a task plays a crucial role.

The most difficult task in estimating the timing behavior of a program is to determine the number of iterations of a certain loop and handling the problems introduced by the use of recursion.

In this thesis we will present methods for the handling of loops in Chapters 2 and 3.

In Chapter 4 we will present a system for handling recursive procedures. In addition to an analysis of the execution time required by a recursive procedure, a detailed examination of the stack space requirements of such a function will be performed.

1.2 Loops

Common programming languages support two different forms of loop-statements:

FOR-LOOPS A loop variable assumes all values of a given integer range. Starting with the smallest value of the range, the loop variable is incremented on each iteration of the loop until its value is outside the given range.

Some programming languages are more flexible and allow for starting with the largest value and decrementing the loop variable, others allow for defining a fixed step size by which the loop variable is incremented or decremented.

GENERAL LOOPS The other loop-statement is of a very general form and is considered for implementing those loops that cannot be handled by for-loops. These loops include while-loops, repeat-loops, and loops with exit-statements (cf. e.g. [2], [14]).

Computing the number of iterations of a for-loop is trivial. For example the loop-body of the loop

```
for i in 1..N by S loop
  -- loop body
end loop;
```

is performed exactly $\lceil N/S \rceil$ times.

Even nested loops do not constitute any problem. For example the innermost body of the loops

```
for i1 in 1..N loop
  for i2 in 1..i1 loop
    for i3 in 1..i2 loop
      ...
        for ir in 1..i{r-1} loop
          -- innermost loop-body
        end loop;
      ...
    end loop;
  end loop;
end loop;
```

is performed exactly

$$\sum_{i_1=1}^{N} \sum_{i_2=1}^{i_1} \sum_{i_3=1}^{i_2} \cdots \sum_{i_r=1}^{i_{r-1}} 1 = \binom{N+r-1}{r}$$

times.

Analyzing general loops is a much more difficult task. In order to avoid the problems connected with estimating the WCET of general loops used in real-time systems, some researchers simply *forbid* general loops and force the programmer into supplying a constant upper bound for the number of iterations. Thus actually transforming the general loop into a for-loop with an additional exit-statement, or they directly require a constant time bound within which the loop has to complete (e.g. [38]). Other researchers attempt to do static and dynamic program path analysis using regular expressions (e.g. [37]).

- In [38] language constructs have been introduced in order to let the programmer integrate knowledge about the actual behavior of algorithms which cannot be expressed using standard programming language features. These constructs are *scopes*, *markers*, and *loop sequences*. Markers are used to define the number of loop iterations if this number cannot be estimated from the program automatically, e.g., if a general loop is used. Nevertheless all loops are forced to have a constant upper bound.

- In [19] the programming language Real-Time Euclid and a corresponding *schedulability analyzer* are described. The estimation of worst-case execution time is facilitated by restricting language constructs, e.g. *constant loop bounds* are required and *recursion* and *dynamic data structures* are forbidden.

- *Partial evaluation* is used in [36] to estimate the execution time of programs at compile time. This is done by use of *compile time variables*, i.e., a variable whose value is definitely known at compile time. Taking advantage of these values, programming language constructs can be simplified thereby speeding up the program in most cases.

  This approach does not need to restrict programming language constructs such as *loops*, *recursion*, or *dynamic storage allocation* as long as compile time known values are involved. It can even solve certain simple problems of concurrent programming and synchronization of concurrent processes at compile time.

- The idea to estimate worst-case execution time of programs written in higher-level languages has been introduced in [41]. So-called *schemas* are used to estimate the best and worst-case execution time of statements of higher-level languages and an extension of Hoare logic (cf. [20]) is employed to prove the timeliness (and correctness) of real-time programs. The method is also able to handle certain real-time language constructs such as delays and time-outs.

  Although Hoare logic is employed, the user has to give constant loop bounds in order to let the compiler determine upper and lower bounds of the number of iterations of a loop.

- Continuing and extending [41] best and worst-case execution time is estimated by employing static and dynamic program paths analysis in [37].

This is done by specifying program paths by *regular expressions.* Since processing this information sometimes requires exponential time, an *interface definition language* is introduced which allows efficient analysis but does not have the expressive power of regular expressions.

The reported examples (cf. [37]) show that tight bounds can be derived using this method. On the other hand, the user must specify upper bounds for general loop statements.

- Determining the execution time of a code segment is also mentioned in [17]. Real-time concurrent C uses a tool which originally is based on [34]. The code can have loops with user-specified loop bounds.

- Real-Time Java [15] provides classes for real-time scheduling. Unfortunatly all these classes expect that the programmer manually provides the worst-case execution time (in nanoseconds). RT-Java then uses this information to produce appropriate real-time schedules. If a scheduled thread still executes, after its deadline has passed, RT-Java executes a `DeadlineMissHandler`, provided by the programmer, for that thread. There is no support for an automatic estimation of suitable bounds of the worst-case execution time.

Summing up, most researchers try to ease the task of estimating the number of general loop iterations by *forbidding* general loops, i.e., by forcing the user to supply constant upper bounds for the number of iterations. Another approach is to let the user specify a time bound within the loop has to complete (cf. e.g. [22]). In any case the user, i.e., the programmer, has to react to such exceptional cases.

In this thesis we will follow a different approach: We will narrow the gap between general loops and for-loops by defining *discrete loops.* These loops are known to complete and are easy to analyze (especially their number of iterations) and capture a large part of applications which otherwise would have been implemented by the use of general loops.

Chapter 2 will introduce the general concept of *discrete loops* and demonstrate their use in many common programming situations. In Chapter 3 *multi-staged discrete loops* (*MSDL*) will be introduced to handle some more tricky cases that cannot be handled by single-staged *discrete loops.*

1.3 Recursion

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 (cf. e.g. Example 5 below).

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. [23, 19]), PEARL (cf. [12]), Real-Time Concurrent C (cf. [17]), and the MARS-approach (cf. [26, 38]).

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. [2]) and PORTAL (cf. [10]). Interestingly, a subset of Ada (cf. [16]) 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.

Other approaches do not address recursion at all (cf. e.g. [34, 41, 37, 22]), others (cf. e.g. [38]) propose to replace recursive algorithms by iterative ones or to transform them into non-recursive schemes by applying program transformation rules. Certainly, if a *simple* iterative version of a recursive algorithm exists and it is also superior in space and time behavior, it should be used instead of a recursive implementation. On the other hand there are the following reasons why recursive algorithms should be implemented by recursive procedures:

- The space and time behavior of transformed programs are by no means easier to investigate than their recursive counterparts, since the stack has to be simulated and because they contain while-loops. In general, the number of iterations of these loops cannot be determined at compile time, even with the use of discrete loops.

- 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. [9]).
- Often recursive algorithms are easier to understand, to implement, to test, and to maintain than non-recursive versions.

The approach presented in Chapter 4 is different in that it does not forbid recursion, but instead constrains 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.

1.4 Notation

In this thesis we will use the following notations.

- $\mathbb{N}$ denotes the set of natural numbers $\{1, 2, 3, \ldots\}$.
- $\mathbb{Z}$ denotes the set of integers $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.
- $\log N = \log_e N$ denotes the natural logarithm of N.
- $\operatorname{ld} N$ denotes the binary logarithm of N.
- $\log_a N = \frac{\log N}{\log a}$ denotes the logarithm to the base a.
- The greatest integer $n \leq x$ is denoted by $\lfloor x \rfloor$.
- The smallest integer $n \geq x$ is denoted by $\lceil x \rceil$.
- $\Delta f(x) := f(x+1) - f(x)$ denotes the difference operator of finite calculus.

Chapter 2
DISCRETE LOOPS

2.1 Discrete Loops

In this Chapter we give an informal introduction to discrete loops, before we perform a theoretical treatment, i.e., an exact definition and some mathematical results.

2.1.1 Introduction to Discrete Loops

In contrast to for-loops, discrete loops allow for a more complex dependency between two successive values of the loop-variable. In fact an arbitrary functional dependency between two successive values of the loop-variable is admissible, but this dependency must be constrained in order to ensure that the loop completes and to determine the number of iterations of the loop. Details of this constraints will follow below.

Like for-loops discrete loops have a loop-variable and an integer range associated with them[1]. The fact that the loop is allowed to range over discrete values, coined the name *discrete loop*. The major difference to for-loops is that the loop-variable is not assigned each of the values of the range. Which values are assigned to the loop-variable, is completely governed by the loop-body. The loop-header, however, contains a list of all those values that can possibly be assigned to the loop-variable during the next iteration. In fact each item of this list of values is a function of the loop-variable.

A simple example is shown in Figure 2.1. In this example the loop-variable `k`

```
discrete k in 1..N new k := 2*k loop
  -- loop body
end loop;
```

Figure 2.1: A simple example of a discrete loop

will assume the values $1, 2, 4, 8, 16, 32, 64, \ldots$ until finally a value greater than `N`

[1]In Section 2.4 a more general form of discrete loops is introduced which does not need a discrete range, but we defer a thorough discussion of these loops until then.

would be reached. Of course the effect of this example can also be achieved by a simple for-loop, where the powers of two are computed within the loop body.

A more complex example is depicted in Figure 2.2. In this example the loop-

```
discrete k in 1..N new k := 2*k | 2*k+1 loop
  -- loop body
end loop;
```

Figure 2.2: A more complex example of a discrete loop

variable `k` can assume the values $1, 2, 4, 9, 18, 37, 75, \ldots$ until finally a value greater than `N` would be reached. But it is also possible that `k` follows the sequence $1, 3, 6, 13, 26, 52, 105, \ldots$. Here the same effect cannot be achieved by a for-loop, because the value of the loop variable cannot be determined exactly before the loop body has been completely elaborated. The reason for this is the *indeterminism* involved in discrete loops.

The term "indeterminism" requires some explanation: Clearly the loop body *determines* exactly which of the given alternatives is chosen, thus one can say that there definitely is no indeterminism involved. On the other hand, from an outside-view of the loop one cannot determine which of the alternatives will be chosen, without having a closer look at the loop body or without exactly knowing which data are processed by the loop. It is this "outside-view" indeterminism we mean here. Furthermore this indeterminism enables us to estimate the number of loop iterations quite accurately without having to know all details of the loop body. Thus discrete loops ease estimating the WCET time of real-time programs.

By the way, a loop like that in Figure 2.2 occurs in a not-recursive implementation of *Heapsort* (cf. [25] or [39] for a more readable form in a high-order programming language). Sections 2.2.2 and 2.5.2 will be concerned with algorithms that can profit from discrete loops; Heapsort will be treated in detail in Section 2.2.2.

There are two main reasons for stating this functional dependency between successive values of the loop-variable in the loop-header:

1. The compiler or, if it cannot be done statically at compile-time, the runtime system should check if the loop-variable does in fact obtain one of the possible values stated in the loop-header. This will evidently ease debugging and shift some runtime errors to compile-time errors. In fact, if the information given in the loop-header is incorrect, this results in a *programming*

error, not in a *timing error*. Of course this programming error could cause a timing error.

2. Under some circumstances, the information in the loop-header will make determining the number of loop iterations feasible.

2.1.2 Theoretical Treatment

Discrete loops can be defined using a range of any discrete type, e.g. an enumeration. In our theoretical treatment, however, we will assume that the range is `1..N` and that the loop-variable starts with $k_1 = s$, where s is the starting value of the loop. This restriction, however, does not inhibit transferring our results to the cases mentioned above. If s is not in the range `1..N`, the loop-body is not executed, rather the control-flow of the program is transferred to the first statement after the loop.

Definition 2.1.1 (Discrete Loop). A *discrete loop* is characterized by $N \in \mathbb{N}$ and a finite number of functions $f_i : \mathbb{N} \to \mathbb{N}$, $1 \leq i \leq e$.

Definition 2.1.2 (Iteration Sequence). An *iteration sequence* (k_ν) is defined by the recurrence relation

$$\begin{aligned} k_1 &:= s, \quad s \in [1, N] \\ k_{\nu+1} &:= f_i(k_\nu) \end{aligned}$$

for some i. The set of all possible iteration sequences is denoted by $\mathcal{K} = \{(k_\nu)\}$.

Remark 2.1.1. Note that $k_\nu \in \mathbb{N}$ for all $\nu \in \mathbb{N}$.

Definition 2.1.3 (Completing Iteration Sequences). An iteration sequence (k_ν) is said to *complete* if $1 \leq k_\nu \leq N$ for all $\nu \leq \omega$ but $k_{\omega+1} < 1$ or $k_{\omega+1} > N$ for some $\omega \in \mathbb{N}$. The number ω is denoted by $\operatorname{len} k_\nu$ and called the length of (k_ν). It corresponds to the number of iterations of the discrete loop if the loop variable iterates through (k_ν).

Definition 2.1.4 (Completing Discrete Loops). A discrete loop is called a *completing discrete loop* if all $(k_\nu) \in \mathcal{K}$ are completing sequences for all N and for all $s \in [1, N]$.

Definition 2.1.5 (Loop Digraphs). Let a discrete loop be characterized by N and the iteration functions $f_i(x)$, $1 \le i \le e$. Let the initial value of the loop variable be s. For $1 \le x \le N$ associating to each function f_i a function $\overline{f}_i$ by

$$\overline{f}_i(x) = \begin{cases} 0, & \text{if } f_i(x) < 1, \\ f_i(x), & \text{if } 1 \le f_i(x) \le N, \text{ and} \\ N+1, & \text{if } f_i(x) > N, \end{cases}$$

we define the corresponding *loop digraph* $\mathcal{G}$ by the set of vertices

$$V = \{0, 1, \ldots, N, N+1\}$$

and the set of edges E, where E is defined by

$$(v, w) \in E \quad \Leftrightarrow \quad w = \overline{f}_i(v)$$

for some $i \in [1, e]$.

With these definitions the following lemma is trivially true.

Lemma 2.1.1. A discrete loop completes if the corresponding loop digraph is acyclic. $\square$

Each acyclic digraph can be *topologically sorted* (cf. [31]), i.e., we can find a mapping $\operatorname{ord} : V \to \{0, 1, \ldots, N, N+1\}$ such that for all edges $(v, w) \in E$ we have $\operatorname{ord}(v) < \operatorname{ord}(w)$. Since we are only interested in completing discrete loops, we restrict ourselves to discrete loops that result in topologically sorted loop digraphs. This is certainly the case if $f_i(x) > x$ or if $f_i(x) < x$ for all $x \in \mathbb{N}$ and for all $i \in [1, e]$. The next section is devoted to such loops.

2.2 Monotonical Discrete Loops

Definition 2.2.1 (Monotonic Iteration Sequences). A sequence (k_ν) is called *strictly monotonically increasing* if $k_{\nu+1} > k_\nu$ for all $\nu \ge 1$. It is called *strictly monotonically decreasing* if $k_{\nu+1} < k_\nu$ for all $\nu \ge 1$.

Definition 2.2.2 (Monotonic Loops). A discrete loop is called a *monotonically increasing discrete loop* if all $(k_\nu) \in \mathcal{K}$ are strictly monotonically increasing sequences. It is called a *monotonically decreasing discrete loop* if all $(k_\nu) \in \mathcal{K}$ are strictly monotonically decreasing sequences. A discrete loop is called a *monotonical discrete loop* if it is either monotonically increasing or monotonically decreasing.

Lemma 2.2.1. A monotonical discrete loop is completing.

Proof. If all (k_ν) are strictly monotonically increasing, there certainly must exist some $\omega \geq 1$ such that $k_\omega \leq N < k_{\omega+1}$. Thus the loop completes.

On the other hand, if all (k_ν) are strictly monotonically decreasing, there certainly must exist some $\omega \geq 1$ such that $k_\omega \geq 1 > k_{\omega+1}$. Thus the loop completes in this case too. □

Lemma 2.2.2. Let a monotonically increasing discrete loop be characterized by N and the functions f_i. Then all functions f_i fulfill

$$f_i(x) > x$$

for all $x \in [1, N]$.

Proof. If there would exist some f_d such that $f_d(x) \leq x$, there would exist an iteration sequence (k_ν) such that $k_{\nu+1} = f_d(k_\nu) \leq k_\nu$ which contradicts Definition 2.2.2. □

Lemma 2.2.3. Let a monotonically decreasing discrete loop be characterized by N and the functions f_i. Then all functions f_i fulfill

$$f_i(x) < x$$

for all $x \in [1, N]$.

Proof. If there would exist some f_j such that $f_j(x) \geq x$, there would exist an iteration sequence (k_ν) such that $k_{\nu+1} = f_j(k_\nu) \geq k_\nu$ which contradicts Definition 2.2.2. □

2.2.1 Syntactical and Semantical Issues of Monotonical Discrete Loops

Although the syntax of discrete loops is certainly important, we consider the semantical issues more important. In order to be able to demonstrate the advantages of discrete loops over conventional loops, however, we define an Ada-like syntax which will be used in the following examples. But it is important to note that an appropriate syntax can be defined for other languages too.

The syntax of a monotonical discrete loop is given by a notation similar to that in [2].

```
loop_statement ::=
    [loop_simple_name:]
        [iteration_scheme] **loop**
            sequence_of_statements
        **end loop** [loop_simple_name];

iteration_scheme ::= **while** condition
    | **for** for_loop_parameter_specification
    | **discrete** discrete_loop_parameter_specification

for_loop_parameter_specification ::=
    identifier **in** [**reverse**] discrete_range

discrete_loop_parameter_specification ::=
    identifier **:=** initial_value **in** [**reverse**] discrete_range
        **new** identifier **:=** list_of_iteration_functions

list_of_iteration_functions ::=
    iteration_function { | iteration_function }

iteration_function ::= expression
```

For a loop with a **discrete** iteration scheme, the loop parameter specification is the declaration of the *loop variable* with the given identifier. The loop variable is an object whose type is the base type of the discrete range. The initial value of the loop variable is given by initial_value. The optional keyword **reverse** defines the loop to be monotonically decreasing; if it is missing the loop is considered to be monotonically increasing. Within the sequence of statements the loop variable behaves like any other variable, i.e., it can be used on both sides of an assignment statement for example.

Before the sequence of statements is executed, the list of iteration functions is evaluated. This results in a list of *possible successive values*. It is also checked whether all of these values are greater than the value of the loop variable if the keyword **reverse** is missing, or whether they are smaller than the value of the loop variable if **reverse** is present. If one of these checks fails, the exception **monotonic_error** is raised.

After the sequence of statements has been executed, it is checked whether the value of the loop variable is contained in the list of possible successive values. If this check fails, the exception **successor_error** is raised.

If the value of the loop variable is still within the discrete range stated in the loop header, the loop is iterated (at least) once more. If it is not within the range, the loop completes.

Remark 2.2.1. The semantics of monotonical discrete loops ensure that such a loop will always complete, either because the value of the loop variable is outside the given discrete range or because one of the above checks fail, i.e., one of the exceptions **monotonic_error** or **successor_error** is raised.

Remark 2.2.2. A corresponding compiler is free to perform as many checks as it likes in order to inhibit one of the runtime exceptions **monotonic_error** and **successor_error**. This can be done by ensuring that the iteration functions are monotonical functions and by performing data-flow analysis to make sure that **successor_error** will never be raised. Thus a lot of runtime checks can be avoided.

Moreover the compiler might even detect the number of iterations of the loop, which is a valuable result for real-time applications. Clearly the number of iterations depends on the initial value of the loop variable, on the discrete range (especially the number of elements in the range), and on the iteration functions.

2.2.2 Some Examples of Monotonical Discrete Loops

Heapsort

An Ada-like implementation of Heapsort using a monotonical discrete loop is shown in Figure 2.3. Referring to the code shown in Figure 2.3, we easily see that the number of iterations of the discrete loop in procedure `siftdown` is bounded above by the length of $(h_\nu^{(\min)})$, where $(h_\nu^{(\min)})$ fulfills the recurrence relation

$$\begin{aligned} h_1^{(\min)} &= k \\ h_{\nu+1}^{(\min)} &= 2h_\nu^{(\min)} \end{aligned} \tag{2.1}$$

since the length of any loop sequence containing two successive elements that fulfill $k_{\nu+1} = 2k_\nu + 1$ will be smaller than that of $(h_\nu^{(\min)})$. (How lower and upper bounds of the number of iterations of discrete loops can be estimated, is investigated in detail in Section 2.3.)

Solving (2.1) we arrive at

$$h_\nu^{(\min)} = k2^{\nu-1}.$$

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

procedure heapsort(
    arr:    in out sort_array) is

  N: index := arr'length;
  t: index;

  procedure siftdown(N,k:index) is
    j: index;
    v: integer;
  begin
    v := arr(k);
    discrete h := k in 1..N/2 new h := 2*h | 2*h+1 loop
      j := 2*h;
      if j<N and then arr(j)<arr(j+1) then
        j := j+1;
      end if;
      if v >= arr(j) then
        arr(h) := v;
        exit;
      end if;
      arr(h) := arr(j);
      h := j;
    end loop;
  end siftdown;

begin  -- heapsort
  for k in reverse 1..N/2 loop
    siftdown(N,k);
  end loop;
  for M in reverse 2..N loop
    t := arr(1);
    arr(1) := arr(M);
    arr(M) := t;
    siftdown(M,1);
  end loop;
end heapsort;
```

Figure 2.3: Implementation of Heapsort using a Discrete Loop

We want to determine the value of ω such that

$$h_{\omega}^{(\min)} \leq N/2 \leq h_{\omega+1}^{(\min)},$$

thus taking logarithms we obtain

$$\omega = \lfloor \operatorname{ld} N - \operatorname{ld} k \rfloor \leq \lceil \operatorname{ld} N \rceil - \lfloor \operatorname{ld} k \rfloor$$

for the number of iterations of the discrete loop in procedure `siftdown`.

The number F of iterations of the first for-loop in the main procedure is bounded above by

$$F \leq \sum_{k=1}^{\lfloor N/2 \rfloor} \lceil \operatorname{ld} N \rceil - \lfloor \operatorname{ld} k \rfloor = \lfloor N/2 \rfloor \lceil \operatorname{ld} N \rceil - \sum_{k=1}^{\lfloor N/2 \rfloor} \lfloor \operatorname{ld} k \rfloor.$$

Using (cf. [18, problem 3.34])

$$\sum_{k=1}^{n} \lceil \operatorname{ld} k \rceil = n \lceil \operatorname{ld} n \rceil - 2^{\lceil \operatorname{ld} n \rceil} + 1 \tag{2.2}$$

and noticing that

$$\lceil \operatorname{ld} k \rceil - \lfloor \operatorname{ld} k \rfloor = \begin{cases} 0 & \text{if } k \text{ is a power of 2, and} \\ 1 & \text{otherwise,} \end{cases}$$

we obtain

$$\sum_{k=1}^{n} \lfloor \operatorname{ld} k \rfloor = n \lceil \operatorname{ld} n \rceil - n - 2^{\lceil \operatorname{ld} n \rceil} + \lfloor \operatorname{ld} n \rfloor + 2. \tag{2.3}$$

Hence

$$-\sum_{k=1}^{\lfloor N/2 \rfloor} \lfloor \operatorname{ld} k \rfloor = -\lfloor N/2 \rfloor \lceil \operatorname{ld} \lfloor N/2 \rfloor \rceil + \lfloor N/2 \rfloor + 2^{\lceil \operatorname{ld} \lfloor N/2 \rfloor \rceil} - \lfloor \operatorname{ld} \lfloor N/2 \rfloor \rfloor - 2.$$

Furthermore we have (cf. [18, problem 3.19])

$$\lceil \operatorname{ld} \lfloor N/2 \rfloor \rceil \geq \lfloor \operatorname{ld} \lfloor N/2 \rfloor \rfloor = \lfloor \operatorname{ld}(N/2) \rfloor = \lfloor \operatorname{ld} N \rfloor - 1$$

and

$$\lceil \operatorname{ld} \lfloor N/2 \rfloor \rceil \leq \lfloor \operatorname{ld} \lfloor N/2 \rfloor \rfloor + 1 = \lfloor \operatorname{ld} N \rfloor.$$

Thus

$$F \le \lfloor N/2 \rfloor \left(\lceil \mathrm{ld}\, N \rceil - \lfloor \mathrm{ld}\, N \rfloor + 2\right) + 2^{\lfloor \mathrm{ld}\, N \rfloor} - \lfloor \mathrm{ld}\, N \rfloor - 1 \le$$
$$3\lfloor N/2 \rfloor + 2^{\mathrm{ld}\, N} - \lfloor \mathrm{ld}\, N \rfloor - 1 \le \left\lfloor \frac{5}{2} N \right\rfloor - \lfloor \mathrm{ld}\, N \rfloor - 1.$$

The number L of iterations of the second for-loop in the main procedure is bounded above by

$$L \le \sum_{t=2}^{N} \lfloor \mathrm{ld}\, t \rfloor.$$

Using (2.3), L can be estimated by

$$L \le N\lceil \mathrm{ld}\, N \rceil - N - 2^{\lceil \mathrm{ld}\, N \rceil} + \lfloor \mathrm{ld}\, N \rfloor + 2.$$

In a very similar way a lower bound for the number of iterations can be found.

These computations are very easy and we think that they can also be performed by an automated tool during compile-time. Section 2.3 contains some theoretical foundations in order to ease the task of these compile-time computations.

Concluding we would like to remark that the purpose of this section was not to show how to analyze Heapsort. In fact, the worst-case timing behavior (cf. [25]) and even the average timing behavior (cf. [39]) of Heapsort are well understood. The purpose of this section was to show that monotonical discrete loops can ease the task of worst-case timing analysis of algorithms significantly. Sometimes the analysis is so easy that it can be performed by an automated tool. The development of such a tool is part of an ongoing project which is carried out at the *Department of Computer Aided Automation* at the *Technical University of Vienna.*

Other Examples

Other examples showing the advantages of discrete loops over general while or repeat-loops include the bottom-up version of *Mergesort* (cf. [40]), *Euclid's algorithm*, and the solution of *Josephus' Problem* (cf. [18]).

2.3 The Number of Iterations of a Monotonical Discrete Loop

Because of the indeterminism involved in the definition of discrete loops, the number of iterations of such a loop cannot be determined exactly. We can, however, find lower and upper bounds for the number of iterations. Corresponding theoretical results are given in the following subsection.

2.3.1 Lower and Upper Bounds

Definition 2.3.1 (Length of Sequences). Let $\omega(\mathcal{K})$ denote the multi-set of the length of all sequences $(k_\nu) \in \mathcal{K}$ of a monotonical discrete loop and let

$$\mathfrak{l} = \min \omega(\mathcal{K}) \quad \text{and} \quad \mathfrak{u} = \max \omega(\mathcal{K})$$

denote the lower and upper bound of the length of the sequences. These represent lower and upper bounds for the number of iterations of the discrete loop too.

In the rest of this section we will only be concerned with montonically increasing discrete loops. Of course the following treatment can easily be modified in order to deal with monotonically decreasing discrete loops. Besides the loop digraph corresponding to a certain loop is very important in this section to prove properties of discrete loops. Note, however, that the vertex 0 can be avoided since the underlying loop is monotonically increasing.

In order to calculate $\mathfrak{u}$ we can use an algorithm given in [31] which determines the longest path in topologically sorted digraphs. The path is supposed to start at node s.

```
for k in 1..N+1 loop
  c(k) := -∞
end loop;
c(s) := 0;
for k in s..N loop
  for i in 1..e loop
    c($\overline{f}_i(k)$) := max{c($\overline{f}_i(k)$), c(k)+1};
  end loop;
end loop;
```

A similar procedure can be used to determine the shortest path in $\mathcal{G}$.

```
for k in 1..N+1 loop
  c(k) := +∞
end loop;
c(s) := 0;
for k in s..N loop
  for i in 1..e loop
    c($\overline{f}_i(k)$) := min{c($\overline{f}_i(k)$), c(k)+1};
  end loop;
end loop;
```

Remark 2.3.1. The shortest and longest path, i.e., the final value of $c(N+1)$, computed by the algorithms above, correspond to $\mathfrak{l}$ and $\mathfrak{u}$, respectively.

Summing up, we have found algorithms that compute lower and upper bounds of the number of iterations of monotonical discrete loops in time $O(Ne)$. The following Theorem 2.3.1 will show that under certain conditions $\mathfrak{u}$ and $\mathfrak{l}$ can be determined much easier. Before that we need some further definitions and lemmas.

Definition 2.3.2 (Smallest and Largest Sequences). Let a monotonically increasing discrete loop be given by the number N and the iteration functions $f_i(x)$. Then we denote by

$$k_{\nu+1}^{(\min)} = \min_i f_i(k_\nu^{(\min)}) \quad \text{and by} \quad k_{\nu+1}^{(\max)} = \max_i f_i(k_\nu^{(\max)})$$

the sequences that always assume the smallest and largest possible values, respectively.

Lemma 2.3.1. If for all i, $f_i(1) > 1$ and $\Delta f_i(x) \geq 1$ for all $x \in \mathbb{N}$, then $f_i(x) > x$ for all $x \in \mathbb{N}$, i.e., the corresponding discrete loop completes.

Proof. Lemma 2.3.1 is easily proved by induction. □

Lemma 2.3.2. We have $\Delta f(x) \geq 1$ for all $x \in \mathbb{N}$ if and only if $\frac{f(y)-f(x)}{y-x} \geq 1$ for all $x, y \in \mathbb{N}$, $x \neq y$.

Proof. Setting $y = x+1$ clearly implies one part of the proof.

To prove the other part we will in fact show that

$$f(x+k) - f(x) \geq k \tag{2.4}$$

for all $k \geq 1, k \in \mathbb{N}$. We prove this by induction.

Setting $k = 1$ gives the starting point of the induction. Assuming that (2.4) is correct, we have to show that it is correct in the case $k+1$, too. But we have

$$f(x+k+1) - f(x) = (f(x+k+1) - f(x+k)) + (f(x+k) - f(x)) \geq 1 + k.$$

Thus we have proved the lemma. □

The following lemma is trivially true.

Lemma 2.3.3. If $y > x$ and $\frac{f(y)-f(x)}{y-x} \geq 1$, then $f(y) > f(x)$. □

Theorem 2.3.1. If for all $1 \leq i \leq e$ $f_i(1) > 1$ and $\Delta f_i(x) \geq 1$ for all $x \in \mathbb{N}$, then

1. the corresponding discrete loop completes,
2. the length of $(k_\nu^{\max})$ is equal to $\mathfrak{l}$, and
3. the length of $(k_\nu^{\min})$ is equal to $\mathfrak{u}$.

Proof. Case (1) follows immediately from Lemma 2.3.1.

We will only prove case (2), the proof of (3) is very similar.

First we define a path along $(k_\nu^{(\max)})$, i.e., given a node v of this path we choose the next node of the path to be $\max_i\{f_i(v)\}$.

Now assume that there exists a shorter path from node s to node $\Omega = N+1$, i.e., we must have a situation like that depicted in Figure 2.4.

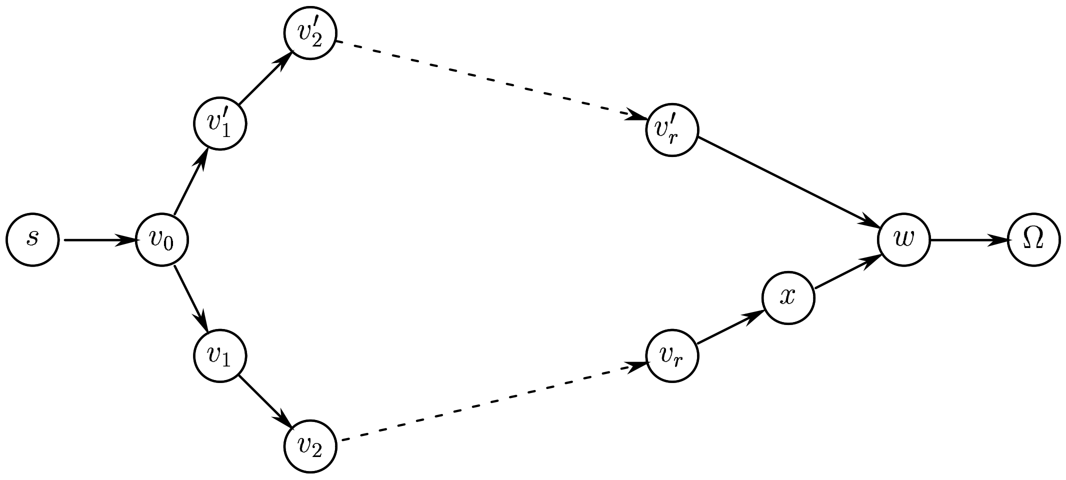

Figure 2.4: Paths in a Loop Digraph

The "lower" path $(s,\ldots,v_0,v_1,v_2,\ldots,v_r,x,\ldots,w,\ldots,N+1=\Omega)$ is the path along $(k_\nu^{(\max)})$ and we want to show that a shorter path like the "upper" path $(s,\ldots,v_0,v_1',\ldots,v_r',w,\ldots,N+1=\Omega)$, $v_1' \neq v_1,\ldots,v_r' \neq v_r$ cannot exist. Denoting by $f_{\max}(x) = \max_i\{f_i(x)\}$, we clearly have

$$v_1 = f_{\max}(v_0) > v_1' = f_{i_1}(v_0).$$

Furthermore

$$v_2 = f_{\max}(v_1) \geq f_{i_2}(v_1) > f_{i_2}(v_1') = v_2'$$

because of Lemma 2.3.3.

Continuing this procedure we finally arrive at $v_r > v_r'$ and $x > w$, which contradicts the fact that $\mathcal{G}$ is topologically sorted, i.e., $f_i(x) > x$ for all $x \in \mathbb{N}$.

Thus, no shorter path exists than that along $(k_\nu^{(\max)})$. $\square$

If $f_{\min}(x) = \min_i\{f_i(x)\}$ and $f_{\max}(x) = \max_i\{f_i(x)\}$ can be determined independently of x, Theorem 2.3.1 enables us to restrict our interest to two single functions in estimating lower and upper bounds of the number of iterations of a discrete loop.

2.3.2 Some Results on Special Iteration Functions

In this subsection we prove some theorems which cover many important cases. We study monotonically increasing discrete loops which are characterized by $N \in \mathbb{N}$ and the iteration functions $f_i(x)$ and we assume that $f(x) = f_{\min}(x)$ can be determined independently of x. The initial value of the loop variable is assumed to be $k_1 = 1$, but our results can easily be generalized.

Theorem 2.3.2. If $f(x) = \lceil \alpha x + \beta \rceil$, $\alpha > 1$, $\beta \geq 0$, then the length of the corresponding loop sequence is bounded above by

$$\left\lfloor \log_\alpha \left(\frac{N(\alpha - 1) + \beta}{\alpha + \beta - 1} \right) + 1 \right\rfloor .$$

Proof. We clearly have

$$f(x) = \lceil \alpha x + \beta \rceil \geq \alpha x + \beta.$$

Thus

$$k_\nu \geq \alpha^{\nu-1} + \frac{\alpha^{\nu-1} - 1}{\alpha - 1}\beta = \alpha^{\nu-1} \left(\frac{\alpha + \beta - 1}{\alpha - 1} \right) - \frac{\beta}{\alpha - 1}.$$

To estimate $\operatorname{len} k_\nu$ we must have

$$\alpha^{\nu-1} \left(\frac{\alpha + \beta - 1}{\alpha - 1} \right) - \frac{\beta}{\alpha - 1} > N$$

which is equivalent to

$$\alpha^{\nu-1} > \frac{N(\alpha - 1)}{\alpha + \beta - 1} + \frac{\beta}{\alpha + \beta - 1}.$$

Taking logarithms we have proved the theorem. □

Theorem 2.3.3. If $f(x) = \lceil \alpha x^\gamma + \beta \rceil$, $\alpha > 1$, $\beta \geq 0$, $\gamma > 1$, then the length of the corresponding loop sequence is bounded above by

$$\lfloor \log_\gamma ((\gamma - 1) \log_\alpha N + 1) + 1 \rfloor .$$

Proof. We clearly have

$$f(x) = \lceil \alpha x^\gamma + \beta \rceil \geq \alpha x^\gamma + \beta.$$

Thus $k_\nu \geq l_\nu$ where

$$\begin{aligned} l_1 &= 1, \\ l_{\nu+1} &= \alpha l_\nu^\gamma + \beta = \alpha l_\nu^\gamma \left(1 + \frac{\beta}{\alpha l_\nu^\gamma}\right). \end{aligned}$$

Taking logarithms and setting $m_\nu = \log l_\nu$ we obtain

$$\begin{aligned} m_1 &= 0, \\ m_{\nu+1} &= \gamma m_\nu + \log \alpha + \log \left(1 + \frac{\beta}{\alpha l_\nu^\gamma}\right). \end{aligned}$$

Since $\log\left(1 + \frac{\beta}{\alpha l_\nu^\gamma}\right) \geq 0$, we have $m_\nu \geq n_\nu$ where

$$\begin{aligned} n_1 &= 0, \\ n_{\nu+1} &= \gamma n_\nu + \log \alpha. \end{aligned}$$

Hence

$$n_\nu = \frac{\gamma^{\nu-1} - 1}{\gamma - 1} \log \alpha$$

and to estimate $\operatorname{len} k_\nu$ we must have

$$k_\nu \geq \alpha^{\frac{\gamma^{\nu-1}-1}{\gamma-1}} > N.$$

Thus

$$\gamma^{\nu-1} > (\gamma - 1)\log_\alpha N + 1$$

and taking logarithms once more, we have proved the theorem. □

Theorem 2.3.4. If $f(x) = \lceil q(x) + \beta \rceil$, where $\beta \geq 0$ and $q(x) = \sum_i \alpha_i x^{\gamma_i}$, $\alpha_i > 1$, $\gamma_i > 1$, then the length of the corresponding loop sequence is bounded above by

$$\left\lfloor \log_{\gamma_m} \big((\gamma_m - 1)\log_{\alpha_m} N + 1\big) + 1 \right\rfloor,$$

where the index m is defined such that $\gamma_m = \max_i \gamma_i$.

Proof. We clearly have

$$f(x) \geq \alpha_m x^{\gamma_m} + \beta.$$

Applying Theorem 2.3.3 to this we have proved Theorem 2.3.4. □

By similar methods lower bounds for the number of iterations of monotonically increasing discrete loops can be derived.

Integrating the results of Theorems 2.3.2, 2.3.3, 2.3.4, and similar theorems into a compiler, the number of iterations of discrete loops can often be estimated at compile time, thus producing valuable information for a tool estimating the WCET of a real-time program and for a real-time scheduler.

2.3.3 Nested Monotonical Discrete Loops

In this subsection we consider the number of iterations of *nested monotonical discrete loops*, i.e., the question, how often the innermost loop-body of nested monotonical discrete loops is executed.

In the following we denote by $\omega(N)$ the upper bound of the number of iterations of a monotonical discrete loop with the associated range $1..N$. With this notation an upper bound for the number of iterations of r nested monotonical discrete loops can be estimated by

$$\sum_{i_1=1}^{\omega_1(N)} \sum_{i_2=1}^{\omega_2(k_{i_1})} \sum_{i_3=1}^{\omega_3(k_{i_2})} \cdots \sum_{i_r=1}^{\omega_r(k_{i_{r-1}})} 1$$

This general formula cannot be presented in a simpler form. It simplifies, however, if more special cases are considered.

Nested Identical Monotonical Discrete Loops

If all involved discrete loops are the same or if $f_{\min}(x) = \min_i\{f_i(x)\}$ can be determined independently of x for all involved discrete loops and if all these functions are the same, we clearly have

$$\omega_t(k_{i_{t-1}}) = \omega(k_{i_{t-1}}) = i_{t-1}.$$

Thus

$$\sum_{i_r=1}^{\omega(k_{i_{r-1}})} 1 = \omega(k_{i_{r-1}}) = i_{r-1}$$

and by induction

$$\sum_{i_1=1}^{\omega(N)} \sum_{i_2=1}^{\omega(k_{i_1})} \sum_{i_3=1}^{\omega(k_{i_2})} \cdots \sum_{i_r=1}^{\omega(k_{i_{r-1}})} 1 = \binom{\omega(N)+r-1}{r}.$$

Some Simple Examples

Next we study some examples involving a for-loop and a simple discrete loop.

```
for i in 1..N loop
  discrete j := 1 in 1..i new j := 2*j loop
    -- innermost loop body
  end loop;
end loop;
```

Here we have $\omega_i(N) = N$ and $\omega_j(N) = \lfloor \operatorname{ld} N + 1 \rfloor$. Hence the number of executions of the innermost loop body is bounded above by

$$\sum_{i=1}^{N} \sum_{j=1}^{\lfloor \operatorname{ld} i+1 \rfloor} 1 = \sum_{i=1}^{N} \lfloor \operatorname{ld} i + 1 \rfloor$$

which by (2.3) is equal to

$$N \lceil \operatorname{ld} N \rceil - 2^{\lceil \operatorname{ld} N \rceil} + \lfloor \operatorname{ld} N \rfloor + 2.$$

Exchanging the loops of the previous example, we get:

```
discrete i := 1 in 1..N new i := 2*i loop
  for j in 1..i loop
    -- innermost loop body
  end loop;
end loop;
```

Here we have $\omega_i(N) = \lfloor \operatorname{ld} N + 1 \rfloor$ and $\omega_j(N) = N$. Hence the number of executions of the innermost loop body is bounded above by

$$\sum_{i=1}^{\lfloor \operatorname{ld} N+1 \rfloor} \sum_{j=1}^{2^{i-1}} 1 = \sum_{i=1}^{\lfloor \operatorname{ld} N+1 \rfloor} 2^{i-1} = 2^{\lfloor \operatorname{ld} N+1 \rfloor} - 1 \leq 2N - 1.$$

2.4 Non-Monotonical Discrete Loops

Although monotonical discrete loops are interesting for their own, many applications rely on discrete loops which are not monotonical. One example is *binary search*, where the corresponding loop sequences are non-monotonical, but the number of iterations is bounded above. An Ada-like implementation using a discrete loop is shown in Figure 2.5. Note that we have omitted the range of the discrete loop since it does not make sense in this connection and that we have

```
N: constant positive := ??; -- number of elements
subtype index is positive range 1 .. N;
type sort_array is array(index) of integer;

function binary_search(
    item:   in integer;
    arr:    in sort_array)
  return index is
l: index := arr'first;
u: index := arr'last;
m: index;
  -- sucessful search only
begin
  discrete (l,u) new (l,u) := (l,(l+u)/2-1) | ((l+u)/2+1,u) loop
    m := (l+u)/2;
    if item < arr(m) then
      u := m-1;
    elsif item > arr(m) then
      l := m+1;
    else
      return m;
    end if;
  end loop;
end binary_search;
```

Figure 2.5: Implementation of Binary Search using a Discrete Loop

used a non-Ada-like notation for two-dimensional vectors for the loop variable `(l,u)`. Correct syntactical and semantical considerations of the kind of loop we are discussing in this section are postponed until Section 2.5.1.

In studying binary search we will investigate how we can generalize monotonical discrete loops such that we still can guarantee that the loop completes but the corresponding loop sequence is not monotonical.

2.4.1 Binary Search

The essential property of binary search is a sequence of intervals which become smaller and smaller with each iteration of the loop. The starting interval is $i_1 = [l_1, u_1] = [1, N]$ and with each iteration the interval $i_\nu = [l_\nu, u_\nu]$ is changed according to

$$i_{\nu+1} = [l_{\nu+1}, u_{\nu+1}] = \begin{cases} [l_\nu, \lfloor \frac{u_\nu + l_\nu}{2} - 1 \rfloor] & \text{or} \\ [\lfloor \frac{u_\nu + l_\nu}{2} + 1 \rfloor, u_\nu] & \end{cases}$$

depending on which sub-interval contains the element being sought. If the sought element is equal to $\lfloor \frac{u_\nu + l_\nu}{2} \rfloor$, the algorithm terminates. In the worst case this is true if the interval contains just one element, i.e., if $l_\omega = u_\omega$ for some $\omega \geq 1$.

On one hand this shows a clear relationship to discrete loops, e.g. the indeterminism and the recurrently defined loop variable, on the other hand the corresponding loop sequences are non-monotonical in general. But a closer inspection shows that there is a monotonical sequence hidden in the algorithm, namely the length of the intervals.

We can even determine an upper bound for the number of loop iterations. Let $\ell_\nu = u_\nu - l_\nu + 1$ denote the length of the interval i_ν. Then

$$\ell_{\nu+1} = \max \left\{ \left\lfloor \frac{u_\nu + l_\nu}{2} \right\rfloor - l_\nu, u_\nu - \left\lfloor \frac{u_\nu + l_\nu}{2} \right\rfloor \right\}.$$

We have

$$\left\lfloor \frac{u_\nu + l_\nu}{2} \right\rfloor - l_\nu \leq \frac{u_\nu + l_\nu}{2} - l_\nu = \frac{u_\nu - l_\nu}{2}$$

and

$$u_\nu - \left\lfloor \frac{u_\nu + l_\nu}{2} \right\rfloor \leq u_\nu - \frac{u_\nu + l_\nu - 1}{2} = \frac{u_\nu - l_\nu + 1}{2}.$$

Thus

$$\ell_{\nu+1} \leq \frac{u_\nu - l_\nu + 1}{2} = \frac{\ell_\nu}{2}.$$

Mentioning $\ell_{\nu+1} \in \mathbb{N}$, we must have $\ell_{\nu+1} \leq \lfloor \ell_\nu/2 \rfloor$ and the length of the interval ℓ_ν is bounded above by U_ν which satisfies the recurrence relation

$$\begin{aligned} U_1 &= N, \\ U_{\nu+1} &= \lfloor U_\nu/2 \rfloor. \end{aligned} \tag{2.5}$$

Hence the number of iterations performed by binary search is bounded above by ω which is defined by $\omega = \min\{\nu : U_\nu = 0\}$.

Solving equation (2.5) by applying standard techniques (cf. e.g. [18]) we obtain

$$U_\nu = \left\lfloor \frac{U_1}{2^{\nu-1}} \right\rfloor.$$

Thus $U_\nu = 0$ if $2^{\nu-1} > N$, i.e., if $\nu > \operatorname{ld} N + 1$. Hence the number of iterations performed by binary search is bounded above by

$$\omega = \lfloor \operatorname{ld} N + 2 \rfloor.$$

The ideas we have seen in studying binary search, can be generalized to a new kind of discrete loop which is treated in the following section.

2.5 Discrete Loops with a Remainder Function

Definition 2.5.1 (Loop Sequence with Remaining Items). In contrast to the previous sections we now define a *loop sequence of remaining items* to be the sequence of *the number of data items* that remain to be processed during the remaining iterations of the loop. Such a loop sequence is denoted by (r_ν) and the set of all loop sequences by $\mathcal{R} = \{(r_\nu)\}$. A corresponding discrete loop is called a *discrete loop with a remainder function.*

Remark 2.5.1. Definition 2.5.1 is justified by the fact that normally each iteration of a loop excludes a certain number of data items from future processing (within the same loop statement). Thus the sequence of the number of the remaining items is responsible for the overall number of loop iterations. This situation is typical for *divide and conquer* algorithms. In our example of binary search the number of the remaining items is equal to the length of the remaining interval.

Definition 2.5.2 (Monotonical Loop Sequence). A loop sequence of remaining items is called *monotonical* if $r_{\nu+1} < r_\nu$.

Definition 2.5.3 (Monotonical Discrete Loop). A discrete loop with a remainder function is called *monotonical* if all its loop sequences $(r_\nu) \in \mathcal{R}$ are monotonical.

Lemma 2.5.1. A monotonical discrete loop with a remainder function is completing.

Proof. Since a monotonically decreasing discrete function will become smaller than 1 in finitely many steps, the corresponding loop will complete. □

2.5.1 Syntactical and Semantical Issues of Discrete Loops with Remainder Functions

The syntax of a discrete loop with a remainder function is again given by a notation similar to that in [2]. In fact we add to the syntax definition of Section 2.2.1.

```
loop_statement ::=
   [loop_simple_name:]
      [iteration_scheme] loop
         sequence_of_statements
      end loop [loop_simple_name];

iteration_scheme ::= while condition
   | for for_loop_parameter_specification
   | discrete discrete_loop_parameter_specification

for_loop_parameter_specification ::=
   identifier in [reverse] discrete_range

discrete_loop_parameter_specification ::=
   monotonical_discrete_loop_parameter_specification |
   discrete_loop_with_remainder_function_parameter_specification

monotonical_discrete_loop_parameter_specification ::=
   identifier := initial_value in [reverse] discrete_range
      new identifier := list_of_iteration_functions

discrete_loop_with_remainder_function_parameter_specification ::=
   [identifier := initial_value
```

new identifier **:=** list_of_iteration_functions]
with *rem*_identifier **:=** initial_value **new** remainder_function

list_of_iteration_functions ::=
  iteration_function { | iteration_function }

iteration_function ::= expression

remainder_function ::=
  *rem*_identifier **=** expression |
  *rem*_identifier **<=** expression [**and** *rem*_identifier **>=** expression]

For a discrete loop with a remainder function, the corresponding loop parameter specification is the optional declaration of the *loop variable* with the given identifier. The loop variable is an object whose type is the base type of result type of the iteration functions, which must be the same for all iteration functions. The initial value of the loop variable is given by initial_value. Within the sequence of statements the loop variable behaves like any other variable,i.e., it can be used on both sides of an assignment statement for example.

After the keyword **with** the *remainder loop variable* is declared by the given identifier (*rem*_identifier). Its type must be a subtype of **natural** in the cases (1) and (2) below or an interval between two **natural** numbers in the case (3). Its initial value is given by initial_value. The remainder function itself may have three different forms:

1. If the remainder function can be determined exactly, it is given by an equation.

2. If only an upper bound of the remainder function is available, it is given by an inequality (<=).

3. If in addition to (2) a lower bound of the remainder function is known, it can be given by an optional inequality (>=). The second inequality must be separated from the first one by the keyword **and**.

The base type of the expressions defining the remainder function or its bounds must be **natural**.

In case (1) the remainder loop variable behaves like a constant within the sequence of statements. In cases (2) and (3) the remainder loop variable behaves like any other variable within the sequence of statements. If the value of the remainder loop variable is changed during the execution of the statements, we call the original value *previous value* and the new value *current value.*

Before the sequence of statements is executed, the list of iteration functions is evaluated if a loop variable is given. This results in a list of *possible successive values.*

After the sequence of statements has been executed, it is checked whether the value of the loop variable is contained in the list of possible successive values. If this check fails, the exception **successor_error** is raised.

After the sequence of statements has been executed, the remainder function or its bounds (depending on which are given by the programmer) are evaluated. In case (1) the new value of the remainder loop variable is set to the value calculated by the remainder function if it is smaller than the previous value, otherwise the exception **monotonic_error** is raised.

In case (2) the new value of the remainder loop variable is set to the value calculated by the remainder function if the previous value of the remainder loop variable is equal to its current value and if the calculated value is smaller than the current value, otherwise the exception **monotonic_error** is raised. If the previous and the current value differ, the remainder loop variable is set to the current value if it is smaller than or equal to the calculated value, which in turn must be smaller than the previous value. If this is not true, the exception **monotonic_error** is raised.

In case (3) the new value of the remainder loop variable is set to the value calculated by the remainder function if the current value is equal to the previous value and if the calculated interval is contained strictly in the previous one. If the current value and previous value differ, the new value is set to the current value if the current interval is contained (not necessarily strictly) in the calculated interval, which in turn must be contained strictly in the previous interval. Otherwise the exception **monotonic_error** is raised. This exception is raised too if the interval does not contain at least one element.

If in cases (1) and (2) the value of the remainder loop variable is zero or if in case (3) the upper bound is zero, the exception **loop_error** is raised, otherwise the loop is continued.

The regular way to complete a discrete loop with a remainder function is to use an *exit* statement, before the remainder loop variable is equal to zero.

Remark 2.5.2. The semantics of discrete loops with remainder functions ensure that such a loop will always complete, either if the loop is terminated by an *exit* statement or because one of the above check fails, i.e., one of the exceptions **monotonic_error**, **successor_error**, or **loop_error** is raised.

Remark 2.5.3. A corresponding compiler is free to perform as many checks as it likes in order to inhibit one of the runtime exceptions **monotonic_error**, **successor_error**, and **loop_error**. This can be done by ensuring that the remainder function or its bounds are monotonical, by performing data-flow analysis to make sure that **successor_error** will never be raised, or by ensuring that the loop will complete before the remainder loop variable is equal to zero. Thus a lot of runtime checks can be avoided.

Moreover the compiler might even detect bounds of the number of iterations of the loop, which is a valuable result for real-time applications.

2.5.2 Some Examples of Monotonical Discrete Loops with Remainder Functions

One illustrative example, *binary search*, has already been discussed in Section 2.4.1, but one syntactical remark is necessary: Line 14 of Figure 2.5 must be replaced with

```
14a  discrete (l,u) new (l,u) := (l,(l+u)/2-1) | ((l+u)/2+1,u)
14b    with i := u-l+1 new i <= i/2 loop
```

Some more runtime checks can be achieved if we insert

```
22a  i := u-l+1;
```

between lines 22 and 23.

In the following we will give further examples of discrete loops with remainder functions.

Traversing Binary Trees

Discrete loops with remainder functions are especially well-suited for algorithms designed to traverse binary trees. A template showing such applications is given in Figure 2.6. In this figure `root` denotes a pointer to the root of the tree, `height` denotes the maximum height of the tree, and `node_pointer` is a pointer to a node of the tree. The actual value of `height` depends on which kind of tree is used, e.g. standard binary trees or AVL-trees.

```
discrete node_pointer := root
    new node_pointer := node_pointer.left | node_pointer.right
  with h := height
    new h := h-1 loop

  -- loop body:
  --   Here the node pointed at by node_pointer is processed
  --   and node_pointer is either set to the left or right
  --   successor.
  --   The loop is completed if node_pointer = null;

end loop;
```

Figure 2.6: Template for Traversing Binary Trees

Weight-Balanced Trees

So-called *weight-balanced trees* have been introduced in [35] and are treated in detail in [32] and in [8].

Definition 2.5.4. We define:

1. Let T be a binary tree with left subtree T_ℓ and right subtree T_r. Then

$$\rho(T) = |T_\ell|/|T| = 1 - |T_r|/|T|$$

   is called the root balance of T. Here $|T|$ denotes the number of leaves of tree T.

2. Tree T is of bounded balance α if for every subtree T' of T:

$$\alpha \leq \rho(T') \leq 1 - \alpha$$

3. BB[α] is the set of all trees of bounded balance α.

If the parameter α satisfies $1/4 < \alpha \leq 1 - \sqrt{2}/2$, the operations *Access*, *Insert*, *Delete*, *Min*, and *Deletemin* take time $O(\log N)$ in BB[α]-trees. Here N is the number of leaves in the BB[α]-tree. Some of the above operations can move the root balance of some nodes on the path of search outside the permissible range $[\alpha, 1-\alpha]$. This can be "repaired" by *single* and *double rotations* (for details see [32] and [8]).

BB[α]-trees are binary trees with bounded height. In fact it is proved in [32] that

$$\mathrm{height}(T) \leq \frac{\mathrm{ld}\, N - 1}{-\,\mathrm{ld}(1-\alpha)} + 1,$$

where N is the number of leaves in the BB[α]-tree T.

A template for the above operations is shown in Figure 2.7, where `floor(x)` is

```
discrete node_pointer := root
    new node_pointer := node_pointer.left | node_pointer.right
  with h := floor(ld(N)/(-ld(1-alpha)))+1
    new h := h-1 loop

  -- loop body

end loop;
```

Figure 2.7: A Template for Operations on BB[α]-trees

supposed to implement $\lfloor x \rfloor$. Since the notion of height defined in [32] is not well-suited for direct application of discrete loops, the remainder function in Figure 2.7 has been slightly modified.

A semantically equivalent template for traversing BB[α]-trees is shown in Figure 2.8. The remainder function of Figure 2.8 has the advantage that it does not

```
discrete node_pointer := root
    new node_pointer := node_pointer.left | node_pointer.right
  with r := N -- N = number of leaves of tree
    new r := floor((1-alpha)*r) loop

  -- loop body

end loop;
```

Figure 2.8: Another Template for Operations on BB[α]-trees

need logarithms since it works with the number of leaves instead of the height of the tree. In addition it does require less mathematical skill from the programmer.

2.5.3 The Number of Iterations of a Monotonical Discrete Loop with a Remainder Function

A special case has already been discussed in Section 2.4.1, but these computations can be generalized.

Theorem 2.5.1. If a loop sequence of remaining items fulfills

$$\begin{aligned} r_1 &= N, \\ r_{\nu+1} &= \lfloor r_\nu / \mu \rfloor, \end{aligned}$$

where $\mu > 1$, then $\operatorname{len} r_\nu$ is bounded above by

$$\lfloor \log_\mu N + 2 \rfloor.$$

Proof. We clearly have

$$\lfloor r_\nu / \mu \rfloor \le r_\nu / \mu.$$

Thus

$$r_\nu \le \frac{N}{\mu^{\nu-1}}$$

and to estimate the length of (r_ν) we must have

$$N < \mu^{\nu-1}.$$

Taking logarithms the theorem is proved. □

2.6 Computational Power of Discrete Loops with a Remainder Function

In this section we prove that the computational power of discrete loops with remainder functions is considerably great if we restrict our interest to applications which do not loop forever.

Theorem 2.6.1. If the number of iterations of a general loop can be determined by an integer-valued *computable function* [11] Φ, a discrete loop with a remainder function can be used to achieve the same effect.

Proof. We define the remainder function of the discrete loop by

$$\begin{aligned} r_1 &:= \Phi, \quad \text{i.e., the number of iterations of the general loop} \\ r_{\nu+1} &:= r_\nu - 1. \end{aligned}$$

Clearly, after Φ iterations, $r_\Phi = 0$ and thus the loop completes. □

Remark 2.6.1. Obviously, in practical applications the remainder function in the proof of Theorem 2.6.1 will not always be the best choice with regard to software engineering, but it is the purpose of Theorem 2.6.1 to show the computational power of discrete loops with remainder functions and not to set up a style-guide for discrete loops with remainder functions.

Remark 2.6.2. If, on the other hand, the number of iterations of a general loop can only be determined by a *partially computable function*, the procedure in the proof of Theorem 2.6.1 may loop forever in computing $r_1 := \Phi$.

2.7 Summary

In this Chapter we have introduced discrete loops which narrow the gap between general loops and for-loops. Since they are well-suited for determining the number of iterations, they form an ideal frame-work for estimating the worst-case execution time of real-time programs.

It remains to compare discrete loops with recent approaches in the domain of real-time systems. Some of these approaches have already been mentioned in the introduction.

1. Assume that the only thing that is known is $U \in \mathbb{N}$, an upper bound for the number of iterations of a general loop. Then we define the remainder function of a discrete loop by

$$\begin{aligned} r_i &= U, \\ r_{\nu+1} &= r_\nu - 1. \end{aligned}$$

   Obviously this is semantically equivalent to the approaches described in the introduction (cf. [19, 38, 37, 41, 17]): If the upper bound U is exceeded, the exception **loop_error** is raised, which must be caught by an appropriate exception handler in order to treat this exceptional case.

2. An upper bound for the amount of time T the loop uses can be given by

$$\begin{aligned} r_i &= T, \\ r_{\nu+1} &= r_\nu - \text{time(loop\_body)}, \end{aligned} \tag{2.6}$$

   where time(loop_body) is the time that passed since (2.6) has been elaborated the last time.

> Hence the loop completes if the upper bound T has been exceeded. But an unpredictable amount of time may pass, until this fact is recognized, if the process executing the loop has been set into a waiting state by the scheduler.

Thus we have shown that discrete loops can simulate all important recent concepts that have been invented to handle general loops in the domain of real-time systems. It is, however, more important that we have demonstrated in the previous sections, how to use discrete loops in applications and how easy the timing behavior of discrete loops can be analyzed.

Chapter 3

MULTI-STAGED DISCRETE LOOPS

3.1 Multi-Staged Discrete Loops

In the previous Chapter we have introduced the concept of *dicrete loops.*

Like a for-loop a discrete loop uses a loop variable that is adjusted on every iteration of the loop until if does no longer fall into a given range. Unlike for-loops, where the adjustment of the loop-variable always consists of adding (or subtracting) a fixed amount (usually one), discrete loops allow a wide range of functions for the adjustment of the loop variable. One limitation that remains is that only the previous value of the loop variable can be used to compute the next value. In this Chapter we will extend the concept of discrete loops to include *multi-staged discrete loops* (*MSDL*) that permit the use of all previous values of the loops variable for the computation of the next value. Nevertheless tight bounds on the number of iterations can be computed at compile-time.

3.2 Additional Notation

A n-dimensional vector $(a_1, a_2, \ldots, a_n)$ of natural numbers is written as $[a_n]$. A constant n-dimensional vector $(c, c, \ldots, c)$ is written as $[c]_n$. We define some binary relations on n-dimensional vectors

$$\begin{aligned}
[a_n] = [b_n] &\Longleftrightarrow a_k = b_k \quad \text{for all } 1 \leq k \leq n, \\
[a_n] \leq [b_n] &\Longleftrightarrow a_k \leq b_k \quad \text{for all } 1 \leq k \leq n, \\
[a_n] < [b_n] &\Longleftrightarrow [a_n] \leq [b_n] \text{ and } [a_n] \neq [b_n].
\end{aligned}$$

Note that we are only comparing vectors of equal length. Note further that there are vectors $[a_n]$, $[b_n]$ such that neither $[a_n] \leq [b_n]$ nor $[a_n] \geq [b_n]$ (e.g. $[a_2] = (2, 5)$, $[b_2] = (3, 4)$).

We will see that it can be useful to base the comparison of two vectors on their trailing ends. Thus we define for all $1 \leq d \leq n$

$$\begin{aligned} [a_n] =_d [b_n] &\iff a_k = b_k \quad \text{for all } n-(d-1) \leq k \leq n, \\ [a_n] \leq_d [b_n] &\iff a_k \leq b_k \quad \text{for all } n-(d-1) \leq k \leq n, \\ [a_n] <_d [b_n] &\iff [a_n] \leq_d [b_n] \text{ and } [a_n] \neq_d [b_n]. \end{aligned}$$

For $d = n$ these definitions are equivalent to the original relations on vectors. Obviously $[a_k] = [b_k] \Rightarrow [a_k] =_d [b_k]$ and $[a_k] \leq [b_k] \Rightarrow [a_k] \leq_d [b_k]$.

Let (a_k) denote a sequence of natural numbers (i.e. a function $\mathbb{N} \to \mathbb{N}$).

We write $f^{(k)} : \mathbb{N}^k \to \mathbb{N}$ for a k-dimensional function and $(F) = (f^{(1)}, f^{(2)}, \ldots)$ for a sequence of functions.

Furthermore we denote $(F)(x) = (a_k)$ such that

$$\begin{aligned} a_1 &= x \\ a_{k+1} &= f^{(k)}([a_k]) \qquad \text{for all } k+1 > 1. \end{aligned}$$

3.3 Some Interesting Examples

Example 3.3.1. Catalan Numbers

$$\begin{aligned} a_1 &= 1 \\ a_k &= 2a_{k-1} + \sum_{i=1}^{k-2} a_i a_{k-1-i} \qquad \text{for all } k > 1 \end{aligned}$$

Evaluating for a few elements of $[a_k]$ we get $[1, 2, 5, 14, 42, 132, 429, 1430, \ldots]$.

Example 3.3.2. Fibonacci Numbers

$$\begin{aligned} a_1 &= 1 \\ a_2 &= f_1([a_1]) = 2 \\ a_k &= a_{k-1} + a_{k-2} \qquad \text{for all } k > 2 \end{aligned}$$

The vector $[a_k]$ of Fibonacci Numbers begins with $[1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots]$.

Example 3.3.3. Factorial Numbers $k!$

$$\begin{aligned} a_1 &= 1 \\ a_k &= k \cdot a_{k-1} \qquad \text{for all } k > 1 \end{aligned}$$

Thus the first few elements of $[a_k]$ are $[1, 2, 6, 24, 120, 720, 5040, 40320, \ldots]$.

Example 3.3.4.

$$\begin{aligned} a_1 &= 1 \\ a_k &= \left\lceil \frac{1}{a_{k-1}}(1 + \sum_{t=1}^{a_{k-1}} a_{k-t}) \right\rceil \qquad \text{for all } k > 1 \end{aligned}$$

This formula produces the sequence $[a_k] = [1,\ \ 2,2,\ \ 3,3,3,\ \ 4,4,4,4,\ \ 5,\ldots]$.

3.4 Theoretical Treatment

Definition 3.4.1 (Multi-Staged Discrete Loops). A *multi-staged discrete loop* (*MSDL*) $\mathcal{L}$ is characterized by a value $N \in \mathbb{N}$ (producing a range $[1 \ldots N]$) and a finite number of sequences of successor functions (F_i), $1 \leq i \leq e$. Furthermore we require a set of starting values $\mathcal{S} \subseteq \mathbb{N}$. In addition we write $\underline{s} = \min_{s \in \mathcal{S}} s$ and $\overline{s} = \max_{s \in \mathcal{S}} s$.

Definition 3.4.2 (Paths in *MSDL*). Let $\mathcal{L}$ be a *MSDL*. A *path* $\mathcal{P}$ through $\mathcal{L}$ is defined by a starting value $s \in \mathcal{S}$ and a sequence of successor functions $(f_{i_1}^{(1)}, f_{i_2}^{(2)}, \ldots)$ where $1 \leq i_j \leq e$ for all $j \geq 1$. The vector $[a_k]$ of traveled places along the path $\mathcal{P}$ is therefore

$$\begin{aligned} a_1 &= s && \text{the starting value } s \in \mathcal{S} \\ a_{k+1} &= f_{i_k}^{(k)}([a_k]) && \text{for some } i_k : 1 \leq i_k \leq e, \text{ for all } k+1 > 1. \end{aligned}$$

Let $\mathcal{K}$ be the set of all possible paths $\mathcal{P}$.

Definition 3.4.3 (Length of a Path). Let $[a_k]$ be a path $\mathcal{P}$ of a multi-staged discrete loop, and l be the smallest value with $a_l > N$, then we call l the *length* of $\mathcal{P}$ and write $\operatorname{len}(\mathcal{P}) = l$.

Definition 3.4.4 (History Depth). Let $\mathcal{D}(f^{(k)}) = \max\big(j : \ a_{k+1-j} \text{ is accessed to compute } a_{k+1} = f^{(k)}([a_k])\big)$. If there exists a $C \in \mathbb{N}$ such that $\mathcal{D}(f^{(k)}) < C$ for all $k \in \mathbb{N}$, then $\mathcal{D}(F) = \max_{k \geq 1} \mathcal{D}(f^{(k)})$ is called the *history depth* of (F). Otherwise we define $\mathcal{D}(F) = \infty$.

The *history depth* of a *MSDL* $\mathcal{L}$ is defined as $\mathcal{D}(\mathcal{L}) = \max_{1 \leq i \leq e} \mathcal{D}(F_i)$.

Definition 3.4.5 (Monotonic Functions, Sequences, and Loops). We call $f^{(k)}$ a *monotonic* function if for all $[a_k] \leq [b_k]$, we have $f^{(k)}([a_k]) \leq f^{(k)}([b_k])$. It is called a *strictly monotonic* function if $[a_k] < [b_k]$ implies $f^{(k)}([a_k]) < f^{(k)}([b_k])$

and *strictly d-monotonic* if it is monotonic and $[a_k] <_d [b_k]$ implies $f^{(k)}([a_k]) < f^{(k)}([b_k])$. Note that strict monotonicity is a special case of strict d-monotonicity (i.e. $d = k$).

Similarly $(F) = (f^{(1)}, f^{(2)}, f^{(3)}, \ldots)$ is called a *monotonic* sequence of functions if all $f^{(k)}$, $k \geq 1$ are monotonic. A *MSDL* is called *monotonic* if all (F_i), $1 \leq i \leq e$ are monotonic.

An analogous definition is used for *strictly (d-)monotonic MSDL.*

Theorem 3.4.1. When checking whether a function $f^{(k)}$ is monotonic or not, it is sufficient to look at the monotonicity in the last $d = \mathcal{D}(f^{(k)})$ elements of the argument, i.e.

$$\big([a_k] \leq [b_k] \Rightarrow f^{(k)}([a_k]) \leq f^{(k)}([b_k])\big) \Longleftrightarrow \big([a_k] \leq_d [b_k] \Rightarrow f^{(k)}([a_k]) \leq f^{(k)}([b_k])\big)$$
$$\text{for any } d \geq \mathcal{D}(f^{(k)}).$$

Proof. If $d = k$ then $[a_k] \leq_d [b_k] \Leftrightarrow [a_k] \leq [b_k]$ and the theorem is trivial. Thus the case $k > d \geq \mathcal{D}(f^{(k)})$ remains to be proved:

$\Leftarrow$: As $[a_k] \leq [b_k] \Rightarrow [a_k] \leq_d [b_k]$ and $[a_k] \leq_d [b_k] \Rightarrow f^{(k)}([a_k]) \leq f^{(k)}([b_k])$ we have $[a_k] \leq [b_k] \Rightarrow f^{(k)}([a_k]) \leq f^{(k)}([b_k])$.

$\Rightarrow$: Let $[a_k] \leq_d [b_k]$ where $[a_k] = (a_1, \ldots, a_{k-d}, a_{k-(d-1)}, \ldots, a_{k-1}, a_k)$ and $[b_k] = (b_1, \ldots, b_{k-d}, b_{k-(d-1)}, \ldots, b_{k-1}, b_k)$.
We now construct a $[b'_k] = (a_1, \ldots, a_{k-d}, b_{k-(d-1)}, \ldots, b_{k-1}, b_k)$. Obviously we have $[a_k] \leq_d [b'_k]$ and as the first $k - d$ elements of $[a_k]$ and $[b'_k]$ are the same also $[a_k] \leq [b'_k]$. By definition of $\mathcal{D}(f^{(k)})$ we know that $f^{(k)}([b_k]) = f^{(k)}([b'_k])$. All together we now have $[a_k] \leq_d [b_k] \Rightarrow [a_k] \leq [b'_k]$, $[a_k] \leq [b'_k] \Rightarrow f^{(k)}([a_k]) \leq f^{(k)}([b'_k])$ as well as $f^{(k)}([b'_k]) = f^{(k)}([b_k])$ and therefore $[a_k] \leq_d [b_k] \Rightarrow f^{(k)}([a_k]) \leq f^{(k)}([b_k])$. □

Definition 3.4.6 (Complete, Partial and Bounded History). Depending on $\mathcal{D}(\mathcal{L})$ various forms of *MSDL*s can be distinguished. If $\mathcal{D}(\mathcal{L}) = 1$ (e.g. Example 3.3.3 - Factorial Numbers) we have a (plain, single staged) discrete loop. This case as been studied extensively in the previous Chapter and will not be treated here. If $1 < \mathcal{D}(\mathcal{L}) < \infty$ (e.g. Example 3.3.2 - Fibonacci Numbers) we call $\mathcal{L}$ a *MSDL* with *bounded history.*

Otherwise $\mathcal{D}(\mathcal{L}) = \infty$. If $\mathcal{D}(f^{(k)}) = k$ for all k (e.g. Example 3.3.1 - Catalan Numbers) then we say $\mathcal{L}$ uses the *complete history.* Otherwise (e.g. Example 3.3.4) we speak of *partial histories.*

3.5 Iteration Bounds for *MSDL*

Definition 3.5.1 (Min./Max. Successor and Min./Max. Path). Let $\mathcal{L}$ be a monotonic *MSDL* and $\bar{i}([a_k])$ be defined by $f^{(k)}_{\bar{i}([a_k])}([a_k]) = \max_{1\leq i \leq e} f^{(k)}_i([a_k])$. Then $f^{(k)}_{\bar{i}([a_k])}([a_k])$ is called the *maximum successor* and $f^{(k)}_{\bar{i}([a_k])}$ one of the *maximum successor functions* of $[a_k]$. This leads to the definition of a *maximum path* $\overline{\mathcal{P}}$ along $[\overline{a_k}]$

$$\begin{aligned}\overline{a_1} &= \overline{s}\\ \overline{a_{k+1}} &= f^{(k)}_{\bar{i}([\overline{a_k}])}([\overline{a_k}]).\end{aligned}$$

Also let $\underline{i}([a_k])$ be such that $f^{(k)}_{\underline{i}([a_k])}([a_k]) = \min_{1\leq i\leq e} f^{(k)}_i([a_k])$. Then $f^{(k)}_{\underline{i}([a_k])}([a_k])$ is called the *minimum successor* and $f^{(k)}_{\underline{i}([a_k])}$ a *minimum successor function* of $[a_k]$. The *minimum path* $\underline{\mathcal{P}}$ along $[\underline{a_k}]$ is defined by

$$\begin{aligned}\underline{a_1} &= \underline{s}\\ \underline{a_{k+1}} &= f^{(k)}_{\underline{i}([\underline{a_k}])}([\underline{a_k}]).\end{aligned}$$

Lemma 3.5.1. Let $\mathcal{L}$ be a monotonic multi-staged discrete loop, then we know for any path $\mathcal{P} \in \mathcal{K}$ that

$$\underline{\mathcal{P}} \leq \mathcal{P} \leq \overline{\mathcal{P}} \qquad \text{or equivalently} \qquad [\underline{a_k}] \leq [a_k] \leq [\overline{a_k}].$$

Proof. We only show $\mathcal{P} \leq \overline{\mathcal{P}}$. The other half of the proof is analogous.

- $k = 1$: Obviously we have $[a_1] = (a_1) = (s) \leq (\overline{s}) = (\overline{a_1}) = [\overline{a_1}]$
- $k > 1$: By induction we know that $[a_{k-1}] \leq [\overline{a_{k-1}}]$. Now

$$\begin{aligned}a_k &= f^{(k-1)}_j([a_{k-1}]) \qquad \text{for some value } j : 1 \leq j \leq e\\ &\leq f^{(k-1)}_j([\overline{a_{k-1}}])\\ &\leq \max_{1\leq i\leq e} f^{(k-1)}_i([\overline{a_{k-1}}]) = \overline{a_k}. \quad \square\end{aligned}$$

Theorem 3.5.1. Let $\mathcal{L}$ be a monotonic *MSDL* and let $f^{(k)}_i([a_k]) > a_k$ for all $1 \leq i \leq e, k \in \mathbb{N}$ then

1. the multi-staged discrete loop completes,
2. $\underline{l} := \text{len}(\overline{\mathcal{P}}) = \min_{\mathcal{P}\in\mathcal{K}} \text{len}(\mathcal{P})$,

3. $\bar{l} := \text{len}(\underline{\mathcal{P}}) = \max_{\mathcal{P}\in\mathcal{K}} \text{len}(\mathcal{P})$.

In other words the longest path can be found by always following the minimum successor and vice versa.

Proof. We will elaborate only the proof of the first two properties, and just hint the idea to the proof of (3), which is quite similar to that of (2).

1. By requirement we have $a_{k+1} = f_i^{(k)}[a_k] > a_k$ for all $k \in \mathbb{N}$. As the a_k are elements of $\mathbb{N}$ there are only a finite number of elements (exactly: N) in the range $[1 \ldots N]$. Thus the loop must complete after at most N iterations.

2. By definition of $\underline{l} := \text{len}(\overline{\mathcal{P}})$ (cf. Def. 3.4.3) it is obvious that $\overline{a_{\underline{l}-1}} \leq N < \overline{a_{\underline{l}}}$. In Lemma 3.5.1 we have shown $\mathcal{P} \leq \overline{\mathcal{P}}$ for all $\mathcal{P} \in \mathcal{K}$. Thus $a_{\underline{l}-1} \leq \overline{a_{\underline{l}-1}} \leq N$ and hence $\text{len}(\mathcal{P}) > \underline{l} - 1$, i.e., $\text{len}(\mathcal{P}) \geq \underline{l}$ for all $\mathcal{P} \in \mathcal{K}$. Thus $\min_{\mathcal{P}\in\mathcal{K}} \text{len}(\mathcal{P}) \geq \underline{l}$. As $\overline{\mathcal{P}} \in \mathcal{K}$ we also know that $\min_{\mathcal{P}\in\mathcal{K}} \text{len}(\mathcal{P}) \leq \text{len}(\overline{\mathcal{P}}) = \underline{l}$. Thus $\underline{l} \geq \min_{\mathcal{P}\in\mathcal{K}} \text{len}(\mathcal{P}) \geq \underline{l}$ or $\underline{l} = \min_{\mathcal{P}\in\mathcal{K}} \text{len}(\mathcal{P})$.

3. Similar to the above case we have $N < \underline{a_{\bar{l}}} \leq a_{\bar{l}}$ and thus $\text{len}(\mathcal{P}) \leq \bar{l}$. □

Theorem 3.5.1 requires that $f^{(k)}([a_k]) > a_k$. Sometimes an alternative condition is easier to prove.

Theorem 3.5.2. If (F) is strictly d-monotonic for some $d \geq 1$ and $f^{(k)}([1]_k) > 1$ for all k, then $f^{(k)}([a_k]) > a_k$.

Proof. Induction on a_k:

- If $a_k = 1$, we have $[a_k] = (a_1, a_2, \ldots, a_{k-1}, 1), a_j \geq 1$ for all $1 \leq j < k$. Thus $[a_k] \geq [1]_k$. As (F) is strictly d-monotonic $f^{(k)}([a_k]) \geq f^{(k)}([1]_k)$ and $f^{(k)}([1]_k) > 1$ we have $f^{(k)}([a_k]) > 1 = a_k$.

- If $a_k > 1$, we assume that we have already shown $f^{(k)}([a'_k]) > a'_k$ for all $[a'_k] = (a_1, a_2, \ldots, a_{k-1}, a'_k), 1 \leq a'_k < a_k$. By definition we know that $[a_k] >_1 [a'_k]$ ($\Rightarrow [a_k] >_d [a'_k]$ and therefore $f^{(k)}([a_k]) > f^{(k)}([a'_k])$ i.e. $f^{(k)}([a_k]) \geq f^{(k)}([a'_k]) + 1 > a'_k + 1$. Using the special case $a'_k = a_k - 1$ it is obvious that $f^{(k)}([a_k]) > a_k$. □

Remark 3.5.1. Note that the above condition not only provides $f^{(k)}([a_k]) > a_k$ but also $f^{(k)}([a_k]) > \sum_{j=k-(d-1)}^{k}(a_j - 1) + 1 = \sum_{j=k-(d-1)}^{k} a_j - d + 1$. For $d = 1$ this is equivalent to the original $f^{(k)}([a_k]) > a_k$ or $a_{k+1} \geq a_k + 1$. Mentioning that $a_1 \geq 1$ it is obvious that $a_k \geq k$. For $d = 2$ this sum expands to $f^{(k)}([a_k]) > a_k + a_{k-1} - 1$

or $a_{k+1} \geq a_k + a_{k-1}$. Using the smallest possible values $a_1 = 1$ and $a_2 = 2$ it is easy to show that $a_k \geq \mathrm{Fib}(k)$, the kth Fibonacci Number. In the extreme case of $d = k$ (i.e., (F) is strictly monotonic) we have $f^{(k)}([a_k]) \geq \sum_{j=1}^{k}(a_j - 1) + 2$. Together with $a_1 \geq 1$ this means that $a_k \geq 2^{k-2} + 1$.

3.6 Number of Iterations of a *MSDL*

Note that $\underline{\mathcal{P}}$ is independent of any run-time parameters. Thus it can effectively be constructed during compile-time making it simple to obtain $\bar{l}$. As discussed in Theorem 3.5.1, $\bar{l}$ is the upper bound for the length of any path through the *MSDL*. Obviously the length of a path through a *MSDL* is the same as the number of iterations it takes to complete the loop. Thus $\bar{l}$ is an upper bound for the number of iterations of the *MSDL*.

Using $\bar{l}$, it is easy to compute maximum amounts of processing time required to execute the loop.

An upper bound for the time required to process the loop is simply obtained by multiplying $\bar{l}$ with an upper bound for the time required to execute a single pass through the loop body.

Obviously it is possible, that the loop body itself does contain loops. Note that this does not create any additional problem, because the same concepts that were used on the outer loop can also be used on the inner loops. As there has to be an innermost loop, this recursion is bound to end.

The stack space required to hold the vector $[a_k]$ is proportional to k. The requirements of other variables, e.g. temporary internal variables to store intermediate results can easily be computed at compile time. Thus the worst case stack usage is the sum of the space needed for auxiliary variables plus $\bar{l}$ times the space needed for a natural.

Frequently $\mathcal{D}(F) \ll \bar{l}$ making it inefficient to store the entire history. Major reductions of space consumptions can be obtained by keeping only the last $\mathcal{D}(F)$ elements (e.g. in a cyclic list).

3.7 Iteration Functions and Programming Language Features

3.7.1 Monotonically increasing *MSDL*s

In this section we investigate which programming language features can be used for implementing iteration functions that are consistent with the conditions of Theorems 3.5.1 and 3.5.2.

At first, the loop variable is considered to be an array of "suitable" size within the code of the iteration functions. By "suitable" we mean that if the jth value of the loop variable a is computed, the ith value, $0 < i < j$, of a can be accessed by writing `a(i)`.

Generally speaking, there are several possible ways how iteration functions can be specified by the programmer:

1. If the *MSDL* uses only a bounded history, the iteration functions can be given by a simple expression, e.g. by `a(i) = a(i-1) + a(i-2)`, provided that the initial values are specified too.

   In this case the condition of Theorems 3.5.1 and 3.5.2 are easy to verify even at compile time.

2. If a complete or a partial history is used in order to define a *MSDL*, more sophisticated programming language features must be used to specify the iteration functions. Here we can discriminate two cases:

   (a) Either we introduce new language features for implementing sums, convolutions, and so on, or

   (b) we choose a more general approach by constraining conventional language features.

   The last case will be treated in the rest of this section.

To guarantee that the conditions of Theorems 3.5.1 and 3.5.2 are met, the syntax of the programming language must be constrained suitably. In the following we list necessary constraints which meet these conditions:

1. All expressions used to compute the next value of the loop variable must only contain "positive" operators, namely `"+"`, `"*"`, and `"**"`, i.e., the operators for adding and multiplying two operands and the operator for raising one operand to the power of the second, where the latter must be greater than 1.

   The expressions involved in the computation of the next value of the loop variable can easily be determined by data-flow analysis.

2. The expressions mentioned in (1) are also allowed to contain the `"-"`-operator if only constant values are subtracted and if the sum of the constant values encountered during the computation is not larger than the number of "recent" values of the loop variable used.

3. If- and case-statements can be used in the code of iteration functions.

4. The same applies to for-loops.

5. Since we are primarily interested in real-time systems, general while-loops must not be used in the code of iteration functions.

6. Instead discrete loops (see Chapter 2) or *MSDL*s can be used.

7. Even the use of appropriately constrained recursions (see Chapter 4) is allowed to implement iteration functions.

Constraints (1) and (2) are consistent with Theorem 3.5.2. Constraints (3) to (6) are obvious. Note that Constraint (6) does not constitute problems although it introduces some form of "recursiveness". The same applies to Constraint (7) provided that the code of the recursive procedure or function adheres to constraints (1) to (7). In addition, note that all constraints can be checked at compile time.

Notice that these constraints allow to implement Example 3.3.1, but disallow Example 3.3.4.

The syntactical and semantical issues given above can easily be incorporated into a suitable (real-time) programming language. Thus a programmer can be "forced" to implement only correct *MSDL*s.

Of course the constraints listed above are sufficient to meet the conditions of Theorems 3.5.1 and 3.5.2. Other, even weaker, constraints are possible, and can be supplied by the designers of real-time programming languages. These constraints, however, must be consistent with Theorems 3.5.1 and 3.5.2 and it should be possible to check them at compile time.

3.7.2 Monotonically decreasing *MSDL*s

Until now only monotonically increasing *MSDL*s have been considered. In this section we will be concerned with monotonically decreasing *MSDL*s.

Clearly we can discriminate between monotonically increasing and decreasing iteration functions and iteration functions which are neither monotonically increasing nor decreasing. Although the set of monotonically increasing and that of monotonically decreasing iteration functions are disjoint, their complement sets are not. Thus we have to find a characterization for monotonically decreasing iteration functions different from that given in Subsection 3.7.1.

It, however, turns out that monotonically decreasing iteration functions are not so easy to characterize than their increasing counterparts, because the constraints given in 3.7.1 cannot simply be "inverted" to obtain constraints valid for monotonically decreasing iteration functions. For example allowing `"-"` or `"/"`-operators and forbidding the `"+"` and `"*"`-operators does not result in constraints that produce monotonically decreasing iteration functions.

Defining a monotonically decreasing iteration function d by $\lfloor 1/i \rfloor$ where i denotes a monotonically increasing iteration function also does not work, because in this case d can only assume the values 0 and 1.

We choose a different approach.

Definition 3.7.1 (Monotonically Decreasing Iteration Function). Let i denote a monotonically increasing iteration function, let the range of the corresponding loop be $1..N$, and let the length of the path induced by i be denoted by ℓ. Let i_j denote the jth value of the loop variable. Then we define a monotonically decreasing iteration function d by

$$d_j = N - i_{\ell-j+1} + 1.$$

Remark 3.7.1. Definition 3.7.1 defines a one-to-one correspondence between monotonically increasing and monotonically decreasing iteration functions.

In the following subsection we give an example of how the syntax and semantics of *MSDL*s can be defined.

3.7.3 Syntax and semantics of *MSDL*s

We define an Ada-like syntax, but it is important to note that an appropriate syntax can be defined for other languages too.

The syntax of a monotonic *MSDL* is given by a notation similar to that in [2].

```
loop_statement ::=
    [loop_simple_name:]
        [iteration_scheme] loop
            sequence_of_statements
        end loop [loop_simple_name];

iteration_scheme ::= while condition
    | for for_loop_parameter_specification
```

| **discrete** discrete_loop_parameter_specification

for_loop_parameter_specification ::=
identifier **in** [**reverse**] discrete_range

discrete_loop_parameter_specification ::=
identifier **:=** initial_value **in** [**reverse**] discrete_range
new identifier **:=** list_of_iteration_functions

list_of_iteration_functions ::=
iteration_function { | iteration_function }

iteration_function ::= expression

For a loop with a **discrete** iteration scheme, the loop parameter specification is the declaration of the *loop variable* with the given identifier. The loop variable is an object whose type is the base type of the discrete range. The initial value of the loop variable is given by initial_value. The optional keyword **reverse** defines the loop to be monotonically decreasing; if it is missing the loop is considered to be monotonically increasing. Within the sequence of statements the loop variable behaves like any other variable, i.e., it can be used on both sides of an assignment statement for example.

Before the sequence of statements is executed, the list of iteration functions is evaluated. This results in a list of *possible successive values*. It is also checked whether all of these values are greater than the value of the loop variable if the keyword **reverse** is missing, or whether they are smaller than the value of the loop variable if **reverse** is present. If one of these checks fails, the exception **monotonic_error** is raised.

After the sequence of statements has been executed, it is checked whether the value of the loop variable is contained in the list of possible successive values. If this check fails, the exception **successor_error** is raised.

If the value of the loop variable is still within the discrete range stated in the loop header, the loop is iterated (at least) once more. If it is not within the range, the loop completes.

Remark 3.7.2. The semantics of monotonic *MSDL*s ensure that such a loop will always complete, either because the value of the loop variable is outside the given discrete range or because one of the above checks fail, i.e., one of the exceptions **monotonic_error** or **successor_error** is raised.

Remark 3.7.3. A corresponding compiler is free to perform as many checks as it likes in order to inhibit one of the runtime exceptions **monotonic_error** and **successor_error**. This can be done by ensuring that the iteration functions are monotonic functions and by performing a data-flow analysis to make sure that **successor_error** will never be raised. Thus a lot of runtime checks can be avoided.

The syntax and semantics of the iteration functions can be defined analogous to our investigation in Subsection 3.7.1 in order to guarantee syntactically correct, monotonically increasing iteration functions. An exact treatment is left to the reader.

If the keyword **reverse** is present, the next value of the loop variable may be computed by the method shown in Subsection 3.7.2.

Moreover the compiler might even detect the number of iterations of the loop, which is a valuable result for real-time applications. Clearly the number of iterations depends on the initial value of the loop variable, on the discrete range (especially the number of elements in the range), and on the iteration functions.

Remark 3.7.4. The syntax of single-staged discrete loops (cf. [3]) and *MSDL*s differ only in that in the latter case iteration functions refer to recent values of the loop variable in terms of array elements.

3.8 *MSDL*s with Remainder Functions

Definition 3.8.1 (Loop Sequence of Remaining Items). In contrast to the previous sections we now define a *loop sequence of remaining items* to be the sequence of the number of data items that remain to be processed during the remaining iterations of the loop. Such a loop sequence is denoted by (r_k) and the set of all loop sequences by $\mathcal{R} = \{(r_k)\}$. A corresponding *MSDL* is called a MSDL *with a remainder function.*

Remark 3.8.1. Definition 3.8.1 is justified by the fact that normally each iteration of a loop excludes a certain number of data items from future processing (within the same loop statement). Thus the sequence of the number of the remaining items is responsible for the overall number of loop iterations. This situation is typical for *divide and conquer* algorithms.

Definition 3.8.2 (Monotonic Loop Sequence). A loop sequence of remaining items is called *monotonic* if $r_{k+1} < r_k$.

Definition 3.8.3 (Monotonic Remainder Function). A *MSDL* with a remainder function is called *monotonic* if all its loop sequences $(r_k) \in \mathcal{R}$ are monotonic.

Lemma 3.8.1. A monotonic *MSDL* with a remainder function is completing.

Proof. Since a monotonically decreasing discrete function will become smaller than 1 in finitely many steps, the corresponding loop will complete. □

3.8.1 Syntactical and semantical issues of *MSDL*s with remainder functions

The syntax of a *MSDL* with a remainder function is again given by a notation similar to that in [2]. In fact we add to the syntax definition of Section 3.7.3.

```
loop_statement ::=
    [loop_simple_name:]
        [iteration_scheme] loop
            sequence_of_statements
        end loop [loop_simple_name];

iteration_scheme ::= while condition
    | for for_loop_parameter_specification
    | discrete discrete_loop_parameter_specification

for_loop_parameter_specification ::=
    identifier in [reverse] discrete_range

discrete_loop_parameter_specification ::=
    monotonical_discrete_loop_parameter_specification |
    discrete_loop_with_remainder_function_parameter_specification

monotonical_discrete_loop_parameter_specification ::=
    identifier := initial_value in [reverse] discrete_range
        new identifier := list_of_iteration_functions
```

discrete_loop_with_remainder_function_parameter_specification ::=
  [identifier **:=** initial_value
    new identifier **:=** list_of_iteration_functions]
    with *rem*_identifier **:=** initial_value **new** remainder_function

list_of_iteration_functions ::=
  iteration_function { | iteration_function }

iteration_function ::= expression

remainder_function ::=
  *rem*_identifier **=** expression |
  *rem*_identifier **<=** expression [**and** *rem*_identifier **>=** expression]

For a *MSDL* with a remainder function, the corresponding loop parameter specification is the optional declaration of the *loop variable* with the given identifier. The loop variable is an object whose type is the base type of the result type of the iteration functions, which must be the same for all iteration functions. The initial value of the loop variable is given by initial_value. Within the sequence of statements the loop variable behaves like any other variable, i.e., it can be used on both sides of an assignment statement for example.

After the keyword **with** the *remainder loop variable* is declared by the given identifier (*rem*_identifier). Its type must be a subtype of **natural** and its initial value is given by initial_value. The remainder function itself may have three different forms:

1. If the remainder function can be determined exactly, it is given by an equation.

2. If only an upper bound of the remainder function is available, it is given by an inequality (<=).

3. If in addition to (2) a lower bound of the remainder function is known, it can be given by an optional inequality (>=), too. The second inequality must be separated from the first one by the keyword **and**.

The base type of the expressions defining the remainder function or its bounds must be **natural**.

Before the sequence of statements is executed, the list of iteration functions is evaluated if a loop variable is given. This results in a list of *possible successive values.*

After the sequence of statements has been executed, it is checked whether the value of the loop variable is contained in the list of possible successive values. If this check fails, the exception **successor_error** is raised.

After the sequence of statements has been executed, the remainder function or its bounds (depending on which are given by the programmer) are evaluated. If the new value is smaller than that of the remainder loop variable, the new value is assigned to the remainder loop variable. Otherwise the exception **monotonic_error** is raised. If the value of the remainder loop variable is zero, the exception **loop_error** is raised, otherwise the loop is continued. The regular way to complete a *MSDL* with a remainder function is to use an *exit* statement, before the remainder loop variable is equal to zero.

Remark 3.8.2. The semantics of *MSDL*s with remainder functions ensure that such a loop will always complete, either if the loop is terminated by an *exit* statement or because one of the above check fails, i.e., one of the exceptions **monotonic_error**, **successor_error**, or **loop_error** is raised.

Remark 3.8.3. A corresponding compiler is free to perform as many checks as it likes in order to inhibit one of the runtime exceptions **monotonic_error**, **successor_error**, and **loop_error**. This can be done by ensuring that the remainder function or its bounds are monotonic, by performing a data-flow analysis to make sure that **successor_error** will never be raised, or by ensuring that the loop will complete before the remainder loop variable is equal to zero. Thus a lot of runtime checks can be avoided.

The compiler might also ensure that the remainder function is monotonically decreasing by enforcing the conditions of Subsections 3.7.1 and 3.7.2.

Moreover the compiler might even detect bounds of the number of iterations of the loop, which is a valuable result for real-time applications. Clearly the number of iterations depends on the initial value of the remainder loop variable and on the remainder functions or its bounds.

3.8.2 Examples for *MSDL*s with remainder functions

Consider a tree structure with Fibonacci-like degrees, i.e.,

1. the root of the tree has degree 2,

2. the sons of the root have degree 3,

3. the nodes at level 3 have degree 5, and

4. in general the nodes at level $\ell \geq 3$ have degree f_ℓ, where

$$\begin{aligned} &f_1 = 2,\ f_2 = 3, \\ &f_\ell = f_{\ell-1} + f_{\ell-2}. \end{aligned}$$

Assume that this tree structure is balanced, i.e., all subtrees rooted at a certain level have the same number of nodes. Now, we would like to implement an algorithm for searching in such a balanced tree structure. Let the overall number of nodes be N.

The tree is traversed top down. Thus, after doing the necessary computations at level ℓ, there are at most

$$\left\lceil N / \prod_{i=1}^{\ell} f_i \right\rceil$$

to consider further on.

Since $f_\ell = r_\ell / r_{\ell+1}$, we obtain by simple properties of the Fibonacci Numbers the following recurrence relation for r_ℓ, the number of remaining items,

$$\begin{aligned} &r_1 = N,\ r_2 = N/2,\ r_3 = N/6, \\ &r_\ell = r_{\ell-1} / \left(\frac{r_{\ell-2}}{r_{\ell-1}} + \frac{r_{\ell-3}}{r_{\ell-2}} \right). \end{aligned}$$

This formula allows for computing the number of remaining items without explicit usage of the Fibonacci Numbers.

In the following we estimate the number of iterations. Mentioning (cf. e.g. [24, 18])

$$f_i \sim \frac{1}{\sqrt{5}} \phi^{i+2}$$

where $\phi = (1 + \sqrt{5})/2$, we easily find for the number of iterations ω

$$N = \prod_{i=1}^{\omega} f_i \sim \prod_{i=1}^{\omega} \frac{1}{\sqrt{5}} \phi^{i+2} = 5^{\omega/2} \phi^{(\omega+3)(\omega+2)/2-3}.$$

Hence we conclude that the number of iterations fulfils

$$\omega \sim \sqrt{2 \log_\phi N}.$$

A more accurate treatment is left to the reader.

3.9 Summary

In this Chapter we have introduced multi-staged discrete loops and demonstrated how they help to bridge the gap between for-loops and general loops. Since *MSDL*s are well suited for determining bounds of the number of iterations they form an excellent frame-work for estimating the worst-case execution time of a real-time program.

Obviously discrete loops are a special case of *MSDL*s where $\mathcal{D}(\mathcal{L}) = 1$. As shown in the previous Chapter, for-loops and loops with a bound for the maximum runtime can also be expressed in terms of discrete loops and therefore also in terms of *MSDL*s, proving *MSDL* a very powerful, yet simple tool.

Chapter 4

RECURSION

4.1 Recursion

In this Chapter we will show how bounds on both execution time and stack usage of recursive procedures can be determined at compiletime or checked at runtime, if certain constraints are met. 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.

4.1.1 Additional Notation

In this Chapter we will use the following notational conventions:

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

4.1.2 Examples

Throughout this Chapter we will use 4 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 4.1. Note that the Ada source code contains a hidden for-loop, namely at line 17, and a discrete loop starting at line 18.

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 application. Nevertheless we will use these examples throughout this Chapter 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.

4.2 Definitions and Preliminary Results

Definition 4.2.1 (Parameter Space, Terminating Values). 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 4.2.2 (Well-Defined Procedure). 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 Chapter).

```
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 4.1: Ada Source Code of Mergesort using a Discrete Loop

From now on, when we use the term recursive procedure, we mean well-defined recursive procedure.

Definition 4.2.3 (Direct Successor). 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 4.2.1. 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 4.2.4 (Successors / Necessary Parameter Values). 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 $p(f)$.

Note that $\mathcal{R}_1(f) = \mathcal{R}(f)$ from Definition 4.2.3.

Definition 4.2.5 (Parameter Classes by Recursion Depth). We define a sequence of sets $\mathcal{F}_k$ inductively by

1. $\mathcal{F}_0$ is defined as above (cf. Definition 4.2.1), 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 4.2.1. 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 4.2.1. By definition and by Lemma 4.2.1 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 $k \in \mathbb{N}$ 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 4.2.6 (Recursion Depth). 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 $k = \operatorname{recdep}(f)$. For $f, g \in \mathcal{F}$, we write $f \approx g$ iff $\operatorname{recdep}(f) = \operatorname{recdep}(g)$.

Definition 4.2.7 (Monotonical Recursive Procedures). 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 4.2.2. Note that a trivial "$\prec$"-relation can always be obtained by defining $f_1 \prec f_2 \Leftrightarrow \operatorname{recdep}(f_1) < \operatorname{recdep}(f_2)$. We will return to this topic in Section 4.8.

Remark 4.2.3. 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 $\mathcal{F} = \mathbb{N}$, $\mathcal{R}(k) = \{k-1\}$, and $\mathcal{F}_0 = \{0\}$, $\mathcal{F}_k = \{k\}$. Furthermore we have $\operatorname{recdep}(k) = k$ and the "$\prec$"-relation for $\mathcal{F}$ is the "$<$"-relation for integers. □

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

Example 3. The Ackermann Function gives

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

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

and (cf. [28], where proofs of the following facts can be found)

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

Furthermore we have

$$\begin{aligned} &\text{recdep}((0,y)) = 0 \text{ and for } x > 0 \\ &\text{recdep}((x,y)) = \mathcal{A}(x,y) + x - 2, \end{aligned}$$

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

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

and

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

Furthermore we have

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

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 is the "wondrous" function (cf. [21]). 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 $n \in \mathbb{N}$. This, however, has not been proved.

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

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

we obtain

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

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

4.3 Computational Model and Space and Time Effort

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

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

or

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

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

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

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

where space is measured in multiples of bits or bytes.

Both functions $\mathcal{T}$ and $\mathcal{S}$ are defined recursively depending on the source code of 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.

4.3.1 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. Chapters 2 and 3),[1] and
- recursive calls to the procedure itself.

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

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

The syntax of *nonrecursive*(f) is defined exactly the same way but *rproc*$(f \to \overline{f})$ 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 can be computed using the method described below, but without giving rise to a recurrence relation,

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

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

where

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

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

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

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.

4.3.2 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 effectively.

Definition 4.3.1. 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 4.3.1. Note that $f_1 \approx f_2$ implies $\mathcal{S}(f_1) = \mathcal{S}(f_2)$ and $\mathcal{T}(f_1) = \mathcal{T}(f_2)$, respectively.

There are two cases:

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

In both cases we clearly have:

Theorem 4.3.1. If p is globally space or time-monotonical, then

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

and

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

respectively. □

The difference between case (1) and (2) is that in case (2) Theorem 4.3.1 can even be applied during runtime, e.g., when generic objects are instantiated (cf. e.g. [2, 13]), while in case (1) for real-time applications Theorem 4.3.1 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 respectively)

Thus it suffices to prove that a certain recursive procedure is locally space or time-monotonical, before Theorem 4.3.1 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 4.3.1 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 4.3.1 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.

4.4 The Space Effort of Recursive Procedures

Definition 4.4.1 (Declarative Stack Space). Let p be a recursive procedure. We define the function $\mathcal{D} : \mathcal{F} \to \mathbb{N}$ such that $\mathcal{D}(f)$ denotes the space being part of the declarative part of p if p is called with parameter f.

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

$$\begin{aligned}
&\mathcal{S}(f) = \sigma_0 \quad \text{if } f \in \mathcal{F}_0 \\
&\mathcal{S}(f) = \mathcal{D}(f) + \max\left(\mathcal{S}(\overline{f}_1), \ldots, \mathcal{S}(\overline{f}_m)\right) \quad \text{if } f \notin \mathcal{F}_0, \qquad (4.1)
\end{aligned}$$

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 4.4.1. 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 4.4.2 (Recursion Digraph). 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 4.4.2. 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 4.4.3. 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. [31]).

Definition 4.4.3 (Successor with maximal remaining Recursion Depth and maximal Stack Requirement). 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 $\operatorname{recdep}(f_{\max}) = \operatorname{recdep}(f) - 1$.

Definition 4.4.4 (Locally Space-Monotonical Procedure). 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 4.4.4. If $\mathcal{D}(f)$ is constant, then the underlying recursive procedure is locally space-monotonical.

Theorem 4.4.1. If p is a locally space-monotonical recursive procedure, then

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

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

Proof. Theorem 4.4.1 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 4.2. The path along $(f, \ldots, v_0, v_1, \ldots, v_r, w, \ldots, f_0)$,

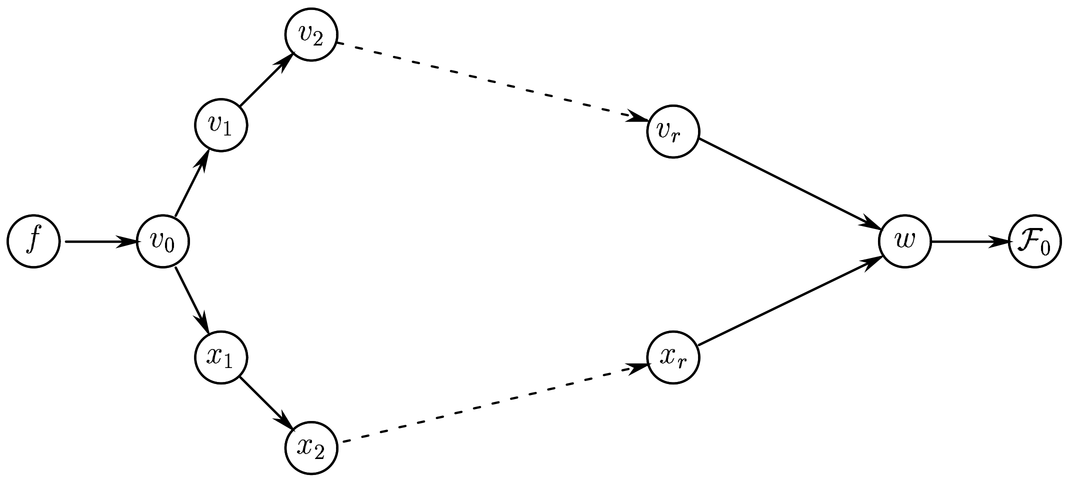

Figure 4.2: Paths in a Recursion Digraph

$f_0 \in \mathcal{F}_0$ 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 Def. 4.4.3 we must have $r \geq s$ since $\operatorname{recdep}(v_i) = \operatorname{recdep}(v_{i+1}) + 1$. Hence we obviously have a contradiction. □

Corollary 4.4.1. If $\mathcal{D}(f) = d$, $d \in \mathbb{N}$ constant for all $f \in \mathcal{F}$,

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

Remark 4.4.5. Theorem 4.4.1 and Corollary 4.4.1 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 4.4.2.

Lemma 4.4.1. 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 $0 \leq k < \operatorname{recdep}(f_1)$

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

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 4.4.2. If p is locally space-monotonical, then

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

Proof. By virtue of Lemma 4.4.1,

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

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

Remark 4.4.6. Lemma 4.4.1 does even hold if

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

The same applies to Theorem 4.4.2, 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 4.4.4) 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 $\text{recdep}(n) = n$ and $\mathcal{N}(n) = n - 1$ we derive

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

Example 2. For the Fibonacci Numbers we obtain $\mathcal{D}(n) = \sigma_d$, constant. Thus they are locally space-monotonical (cf. Remark 4.4.4) 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 $\operatorname{recdep}(n) = n - 1$ and $\mathcal{N}(n) = n - 1$ we derive for $n \geq 1$

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

Example 3. Since $\mathcal{D}((x, y)) = \sigma_d$, constant, the Ackermann Function is locally space-monotonical (cf. Remark 4.4.4). In addition, since

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

we get for the Ackermann Function (cf. [28])

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

Example 4. Mergesort is treated a little inexactly. An exact treatment is possible by use of parameter space morphisms which are introduced in Section 4.7.

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

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

4.5 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 4.5.1 (Ordering Operator for local Time). 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 4.5.2 (Locally Time-Monotonical Procedure). 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.*

Remark 4.5.1. 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 4.5.1. 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

$$\mathcal{T}(f_1) = \tau(f_1) + \sum_{\overline{f}_1 \in \mathcal{R}(f_1)} \mathcal{T}(\overline{f}_1) \qquad \text{and} \tag{4.2}$$

$$\mathcal{T}(f_2) = \tau(f_2) + \sum_{\overline{f}_2 \in \mathcal{R}(f_2)} \mathcal{T}(\overline{f}_2). \tag{4.3}$$

  By induction hypothesis the sum in (4.2) is smaller than or equal to the sum in (4.3). Since $\tau(f_1) \leq \tau(f_2)$, we get

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

Hence the lemma is proved. □

Remark 4.5.2. 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 4.5.1 this implies $\mathcal{T}(f_1) = \mathcal{T}(f_2)$.

Lemma 4.5.1 enables us to find upper and lower bounds of the timing behavior if a range of parameter values is given.

Theorem 4.5.1. If p is locally time-monotonical, then

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

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 4.5.1 the Factorial Numbers are locally time-monotonical.

In addition, we get

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

Mentioning recdep$(n) = n$ we derive

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

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

In addition, we derive

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

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. 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 4.5.2).

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 4.5.2 and Remark 4.3.1. $\square$

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

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

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 18 to 30 in Figure 4.1. The above recurrence relation can be solved and we finally get

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

4.6 Discussion of Examples

In this section we summarize and comment our Examples 1 to 4.

Example 1. Our computations show that the Factorial Numbers can be computed in $O(n)$ space and time. Using a simple for-loop, however, they can be computed in $O(n)$ time and $O(1)$ space.

Thus we conclude that the Factorial Numbers should not be calculated with help of recursion. The purpose of this example was to show how recursive procedures can be analyzed (automatically) and not to rephrase common places.

Example 2. Here our results show that the Fibonacci Numbers can be calculated in $O(n)$ space and $O(f(n))$ time. Again this is a rather academic example since $f(n)$ can be computed in $O(1)$ time and space via

$$f(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} + \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}.$$

Example 3. The results of [28] show that the Ackermann Function $\mathcal{A}(x,y)$ can be evaluated in $O(\mathcal{A}(x,y))$ space and in $\Omega(\mathcal{A}(x,y))$ time. Both results are not very encouraging.

Example 4. Mergesort performs in $O(n \log n)$ time and needs $O(\log n)$ stack space. It is a very useful algorithm for real-time applications and can be analyzed easily by the devices developed in this Chapter.

4.7 Parameter Space Morphisms

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

Commonly, data structures are analyzed by informally introducing some sort of *complexity measure* (cf. [42]) or *size* (cf. [32, 1]) of the data structure. We prefer a more formal approach.

Definition 4.7.1 (Parameter Space Morphism). 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

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

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

Remark 4.7.1. Note that $\|\mathcal{M}(f)\| \geq 1$, but $\operatorname{recdep}(g_1) = \operatorname{recdep}(g_2)$ if $g_1 \in \mathcal{M}(f)$ and $g_2 \in \mathcal{M}(f)$.

Remark 4.7.2. Note that $\operatorname{recdep}_{\mathcal{H}}$ implies a (trivial) "$\prec$"-relation upon $\mathcal{F}'$, namely

$$f' \prec g' \quad \Leftrightarrow \quad \operatorname{recdep}_{\mathcal{H}}(f') < \operatorname{recdep}_{\mathcal{H}}(g') \tag{4.4}$$

for $f', g' \in \mathcal{F}'$. We will assume in the following that a "$\prec$"-relation exists which is consistent with equation (4.4) and denote it by "$\prec_{\mathcal{H}}$".

Definition 4.7.2. 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 4.7.3. 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 4.7.4 (Global $\mathcal{H}$-Space/Time Monotonical Procedure). 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 4.7.5. In addition, we need the following definitions:

$$\mathcal{D}_{\mathcal{H}}(f') = \max_{f'=\mathcal{H}(g)} \mathcal{D}(g) \tag{4.5}$$

$$\tau_{\mathcal{H}}(f') = \max_{f'=\mathcal{H}(g)} \tau(g) \tag{4.6}$$

$$\mathcal{R}_{\mathcal{H}}(f') = \bigcup_{f'=\mathcal{H}(g)} \{\mathcal{H}(\mathcal{R}(g))\} \tag{4.7}$$

$$\mathbb{N}_{\mathcal{H}}(f') = f'_{\max} , \tag{4.8}$$

where

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

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

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

Remark 4.7.3. Note that $\mathcal{H}(\mathcal{R}(g))$ is a multiset and $\mathcal{R}_{\mathcal{H}}(f')$ is a set of multisets.

Definition 4.7.6 ($\mathcal{H}$-Monotonical Procedure). 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 4.7.1. 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

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

Thus the lemma is proved. □

Lemma 4.7.2. 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 4.7.1. □

Definition 4.7.7 (Locally $\mathcal{H}$-Space-Monotonical Proc.). 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 $\mathbb{N}_{\mathcal{H}}(f'_1) \preceq_{\mathcal{H}} \mathbb{N}_{\mathcal{H}}(f'_2)$, and, if $f'_1 \approx f'_2$ and $\mathcal{D}_{\mathcal{H}}(f'_1) \leq \mathcal{D}_{\mathcal{H}}(f'_2)$ implies $\mathcal{D}_{\mathcal{H}}(\mathbb{N}_{\mathcal{H}}(f'_1)) \leq \mathcal{D}_{\mathcal{H}}(\mathbb{N}_{\mathcal{H}}(f'_2))$.

Definition 4.7.8. 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 4.7.9 (Locally $\mathcal{H}$-Time-Monotonical Procedure). 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 $\overline{f}'_{j_i,i,1} \sqsupseteq \overline{f}'_{j_i,i,2} \sqsupseteq \ldots \sqsupseteq \overline{f}'_{j_i,i,m_i}$, $i = 1, 2$.

If for all $\overline{f}'_1 \sqsubseteq \overline{f}'_2$, we have $m_{j_1,1} \leq m_{j_2,2}$ and $\overline{f}'_{j_1,1,r} \sqsubseteq \overline{f}'_{j_2,2,r}$, $r = 1, \ldots, m_{j_1,1}$, 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 4.4.1 and Lemmas 4.4.1 and 4.5.1, $\mathcal{H}$-versions of Theorems 4.4.2 and 4.5.1 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

[2]An implementation of the well-known Quicksort algorithm can be found in any good book on algorithms and data structures (cf. e.g. [25, 32, 40]).

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 4.5.2 and Remark 4.3.1).

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

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

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 4.7.1, 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 $n \geq 0$

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

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 $\mathcal{H}$ (consider 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 $\mathcal{H} : \mathcal{F} \to \mathbb{N}$ 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. Def. 4.7.5) 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 $\mathbb{N}$ to $\mathbb{N}$.

- 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 $\mathbb{N}$ to $\mathbb{N}$ (or $\mathbb{R}$) since the number of iterations can usually be expressed in terms of n^k and $(\operatorname{ld} n)^k$ for for-loops and discrete loops (cf. Chapters 2 and 3), respectively.

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

4.8 Programming Language Issues

Before we discuss details of how (real-time) programming languages are influenced by our previous results, we restate Theorems 4.4.2 and 4.5.1 in a way more suitable to programming language issues.

Definition 4.8.1 (Totally Ordered Procedure). 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 4.2). We are able to show the following theorems.

Theorem 4.8.1. If the parameter space of a recursive procedure is totally ordered, then

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

Proof. In conjunction with Theorem 4.4.2 it remains to show that

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

Because of Definition 4.8.1, however, we have $\mathcal{D}(g) \leq \mathcal{D}(u)$ for all $g \lhd u$. A slight modification of Lemma 4.4.1 shows that in this case $\mathcal{S}(g) \leq \mathcal{S}(u)$ too. Thus the theorem is proved. □

Theorem 4.8.2. If the parameter space of a recursive procedure is totally ordered, then

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

Proof. In conjunction with Theorem 4.5.1 it remains to show that

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

Because of Definition 4.8.1, however, we have $\tau(g) \leq \tau(u)$ for all $g \lhd u$. A slight modification of Lemma 4.5.1 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 $\texttt{pred}(f)$ such that $\texttt{pred}(f) \lhd f$ and there is no $g \in \mathcal{F}$ such that $\texttt{pred}(f) \lhd g \lhd f$.

4.8.1 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`: $\mathcal{F} \to \mathbb{N}$ that for all $f \in \mathcal{F}$ computes $\mathrm{recdep}(f)$.

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 $\overline{f} \in \mathcal{R}(f)$, $\texttt{recdep}(\overline{f}) < \texttt{recdep}(f)$
3. if no parameter space morphism is used, 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.

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

4.8.2 Checking Space Properties

If $\mathcal{D}(f)$ is constant or if there is a simple connection between $\mathcal{D}(f)$ and $\texttt{recdep}(f)$, 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`: $\mathbb{N} \to \mathcal{F}$, which given some $k = \text{recdep}(f)$ 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 $k = \texttt{recdep}(f)$ and $u_k = \texttt{maxspacearg}(k)$. 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 4.8.1 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 4.8.1).

4.8.3 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 $\mathbb{N}$. This, however, as already noted at the end of Section 4.7, 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 $f' \in \mathbb{N}$).

4.8.4 Checking Time Properties

If there is a simple connection between $\tau(f)$ and 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`: $\mathbb{N} \to \mathcal{F}$, 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 "$\lhd$"-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 4.8.1 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 4.8.1).

4.8.5 Time behavior and morphisms

Here the same arguments are valid as in Section 4.8.3.

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 (cf. [32, 8, 35]) in some detail. In Figure 4.3 part of the specification of a BB[α]-tree package is given. Figure 4.4 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`: $\mathcal{B}_\alpha \to \mathbb{N}$.

Let $B \in \mathcal{B}_\alpha$ and $E \in \mathcal{E}$. Then $\mathcal{H}(B) =$ `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.

```
generic

  size:  natural;
  alpha: float range 0.25 .. 0.2928932;
  type element is private;
  with function "<"(left,right:element) return boolean is <>;

package BB_alpha_tree is

  type tree is limited private;

  procedure insert(an: element; into: tree);

  -- other operations suppressed

private
  type tree is
    record
      current_size: natural; -- the current number of nodes in the tree
      -- other stuff representing the tree structure suppressed
    end record;
end BB_alpha_tree;
```

Figure 4.3: Ada Code of Specification of BB[α]-tree (Fragment)

```
package body BB_alpha_tree is

  subtype node_number is natural range 0 .. size;

  recursive procedure insert(an: element; into: tree)

    with function morphism(t: tree)
        return node_number is
    begin
      return t.current_size;
    end morphism;

    with function recdep(current_size: node_number)
        return natural is
    begin
      return floor(1.0+(ld(current_size+1)-1.0)/ld(1.0/(1.0-alpha)));
    end recdep;

  is
  begin
    -- recursive implementation of insert
  end insert;
end BB_alpha_tree;
```

Figure 4.4: Recursive Implementation of BB[α]-tree (Fragment)

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. $\mathcal{R}_{\mathcal{H}}(n) = \{\{i\} \mid \lceil \alpha n \rceil \leq i \leq \lfloor (1-\alpha)n \rfloor\}$, and
4. $\mathbb{N}_{\mathcal{H}}(n) = \lfloor (1-\alpha)n \rfloor$.

Clearly `insert` is $\mathcal{H}$-monotonical. Thus it is also locally $\mathcal{H}$-space-monotonical (cf. Remark 4.4.4) and locally $\mathcal{H}$-time-monotonical (cf. Remark 4.5.1).

The required function `pred` is given by the function `node_number'PRED`, which is predefined in Ada. 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. □

4.9 Summary

Note that Theorems 4.4.2 and 4.5.1 are valid although we do *not* study *static* bounds of space and time behavior. This is in strict contrast to [38], 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 [38] excludes recursions.

In [37] such information on data influencing execution time can be incorporated into the program by means of program path analysis, but [37] 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.

Chapter 5
CONCLUSION

In this thesis *discrete loops* have been introduced. They are much more flexible than for-loops and can be used instead of general loops in many common real-world algorithms. Discrete loops are easy to use for a programmer and while it is almost impossible to bound the worst-case execution time of a general loop, the time required by a discrete loop can be computed automatically. Usually this can be done by an apropriate tool at compile time. In those cases where this is not possible the compiler can at least automatically create code, that checks proper behaviour of the program at runtime. Thus timing errors are transformed to logical errors, that are much easier to debug.

Extending the concept of discrete loops towards multi-staged discrete loops, it becomes possible to express even more algorithms. Even so *MSDL*s are more flexible than (single-staged) discrete loops, they are still analysable by a compiler.

Furthermore is has been shown how the worst-case execution time of recursive procedures can be bounded by adding a few constraints that are frequently fulfilled by real-world algorithms. In addition to execution time, recursive algorithms require an analysis of their stack space requirements to be safely used in real-time systems. Both space and time analysis can be frequently done using automated tools.

We have seen how discrete loops and recursion can be bounded in both execution time and stack space usage making them safe for use in real-time systems.

Chapter 6

LIST OF SYMBOLS AND OPERATORS

$\mathbb{N}$	set of natural numbers $\{1, 2, 3, \ldots\}$
$\mathbb{N}_0$	set of natural numbers with zero $\{0, 1, 2, 3, \ldots\}$
$\mathbb{Z}$	set of integer numbers $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
$\mathbb{R}$	set of real numbers
$\log_a x$	logarithm of x to the base a
$\log x := \log_e x$	natural logarithm of x
$\operatorname{ld} x := \log_2 x$	binary logarithm of x)
$\lfloor x \rfloor$	the greatest integers $n \leq x$
$\lceil x \rceil$	the smallest integers $n \geq x$
$\Delta f(x)$	the difference operator of finite calculus $(f(x+1) - f(x))$
$f_i : \mathbb{N} \to \mathbb{N}$	successor funciton of a discrete loop
(k_ν)	iteration sequence of a discrete loop
$\mathcal{K} := \{(k_\nu)\}$	set of all possible iteration sequences
$\mathcal{G}$	loop digraph of a discrete loop or recursive function
$\omega = \operatorname{len} k_\nu$	length of an iteration sequence
$\omega(\mathcal{K})$	multi-set of the length of all iteration sequences
$\mathfrak{l} := \min \omega(\mathcal{K})$	lower bound of the length of an iteration sequence
$\mathfrak{u} := \max \omega(\mathcal{K})$	upper bound of the length of an iteration sequence
(r_ν)	loop sequence of a discrete loop with remainder function
$\mathcal{R} := \{(r_\nu)\}$	set of all loop sequences for discrete loops with remainder functions

$\mathcal{L}$	Multistaged Discrete Loop (*MSDL*)
$\mathcal{S}$	Set of starting values
$\mathcal{P}$	Path in a *MSDL*
$[a_n]$	n-dimensional vector of numbers $(a_1, a_2, a_3, \ldots, a_n)$
$[c]_n$	constant n-dimensional vector $(c, c, c, \ldots, c)$
$\overline{\mathcal{P}}$	maximum path of a *MSDL*
$\underline{\mathcal{P}}$	minimum path of a *MSDL*
$\mathcal{D}(f)$	history depth of a (f) - Number of recent elements needed to compute the next element
$l := \text{len}(\mathcal{P})$	length of a path
$\mathcal{F}$	parameter space of a recursive function
$\mathcal{F}_0$	terminating values of $\mathcal{F}$
$\mathcal{F}_k$	values of $\mathcal{F}$ that terminate at recursion level k
$\mathcal{R}(f)$	set of direct succesors
$\mathcal{R}_k(f)$	set of successors of degree k
$\mathcal{D}$	stack space used for the declarative part of a procedure
σ	local stack requirement of a procedure (without effort for recursion)
$\mathcal{S}$	(stack) space requirement of a recursive function
$\mathfrak{S}(l, u)$	worst case stack space requirement
τ	local time requirement of a procedure (without effort for recursion)
$\mathcal{T}$	time requirement of a recursive function
$\mathfrak{T}(l, u)$	worst case execution time

References

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

[2] ANSI/MIL-STD 1815 A. *Reference manual for the Ada programming language*, 1983.

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

[4] Johann Blieberger. Loops for safety-critical applications. In *Proceedings of Safecomp '95*, pages 269–282, Belgirate, Italy, 1995.

[5] Johann Blieberger. Real-time properties of indirect recursive procedures. *Information and Computation*, pages 156–182, 2001.

[6] Johann Blieberger and Roland Lieger. Worst-case space and time complexity of recursive procedures. *Real-Time Systems*, 11:193–212, 1994.

[7] Johann Blieberger and Roland Lieger. Using discrete loops for easy comprehension of algorithms. In *Proceedings of the Workshop on Automation and Control Engineering in Higher Education*, pages 125–135, Vienna, Austria, 1995.

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

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

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

[11] Martin Davis. *Computatbility and Unsolvability.* Dover, New York, N.Y., 1982.

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

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

[14] Alice E. Fischer and Frances S. Grodzinsky. *The Anatomy of Programming Languages.* Pretience Hall, Englewood Cliff, New Jersey 07632, 1993.

[15] Real-Time for Java Expert Group (`http://www.rtj.org`). *The Real-Time Specification for Java.* Addison-Wesley, 2000.

[16] 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.

[17] 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.

[18] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading, MA, 1989.

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

[20] Charles Anthony Richard Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12:576–580, 1969.

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

[22] 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.

[23] 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.

[24] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, Reading, Mass., second edition, 1973.

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

[26] 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.

[27] Leslie Lamport. *LATEX- User's Guide and Reference Manual.* Addison Wesley, 1994.

[28] 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.

[29] Roland Lieger and Johann Blieberger. Multi-staged discrete loops for real-time systems. In *Proceedings of the 8th EUROMICRO Workshop on Real-Time Systems*, l'Aquila, Italy, 1996.

[30] 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.

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

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

[33] 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.

[34] 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.

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

[36] Vivek Nirkhe and William Pugh. A partial evaluator for the Maruti hard real-time system. *The Journal of Real-Time Systems*, 5:13–30, 1993.

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

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

[39] Russel Schaffer and Robert Sedgewick. The analysis of heapsort. *Journal of Algorithms*, 15:76–100, 1993.

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

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

[42] 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.