DISCRETE LOOPS AND WORST CASE PERFORMANCE*

JOHANN BLIEBERGER

DEPARTMENT OF AUTOMATION (183/1)
TECHNICAL UNIVERSITY VIENNA
TREITLSTR. 3/4
A-1040 VIENNA
AUSTRIA
EMAIL: BLIEB@AUTO.TUWIEN.AC.AT

ABSTRACT. In this paper so-called *discrete loops* are introduced which narrow the gap between general loops (e.g. while- or repeat-loops) and for-loops. Although discrete loops can be used for applications that would otherwise require general loops, discrete loops are known to complete in any case. Furthermore it is possible to determine the number of iterations of a discrete loop, while this is trivial to do for for-loops and extremely difficult for general loops. Thus discrete loops form an ideal frame-work for determining the worst case timing behavior of a program and they are especially useful in implementing real-time systems and proving such systems correct.

1. Introduction

The most significant difference between real-time systems and other computer systems is that the system behavior must not only be correct but the result of a computation must be available within a predefined deadline. It has turned out that a major progress in order to guarantee the timeliness of real-time systems can only be achieved if the *scheduling problem* is solved accordingly. Most scheduling algorithms assume that the runtime of a task is known a priori (cf. e.g. [1, 2, 3]). Thus the *worst case performance* 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. Ordinary 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-body is iterated until the value of the loop variable is outside the given range.
Some programming languages allow for starting with the largest value and decrementing the loop variable, others allow for defining a fixed step by which the loop variable is incremented or decremented.

*Supported by the Austrian Science Foundation (FWF) under grant P10188-MAT. To appear in *Computer Languages* 20(3), 1995.

Key words and phrases. programming languages, loops, real-time systems, worst case timing analysis.

general loops: The other loop-statement is of a very general form and is considered for implementing those loops that can not be handled by for-loops. These loops include while-loops, repeat-loops, and loops with exit-statements (cf. e.g. [4]).

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

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

is performed exactly N 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.

General loops, however, represent a very difficult task. In order to estimate the worst case performance of general loops many methods and tools have been developed, e.g. [2, 5, 6, 7]. In the following we will discuss some of them:

- In [5] language constructs have been introduced in order to let the programmer integrate knowledge about the actual behavior of algorithms which can not 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 can not 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 [2] the programming language Real-Time Euclid and a corresponding *schedulability analyzer* are described. The estimation of worst case performance 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 [6] 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 performance of programs written in higher-level languages has been introduced in [8]. So-called *schemas* are used to estimate the best and worst case performance of statements of higher-level languages and an extension of Hoare logic (cf. [9]) 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 [8] best and worst case performance is estimated by employing static and dynamic program paths analysis in [7]. 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. [7]) 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 [10]. Real-time concurrent C uses a tool which originally is based on [11]. The code can have loops with user-specified loop bounds.

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. [12]). In any case the user, i.e., the programmer, has to react to such exceptional cases.

In this paper 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.

Remark 1.1. In this paper we will use the following notations.

- By $\log N = \log_e N$ we denote the natural logarithm of N.
- By $\mathrm{ld}\, N$ we denote the binary logarithm of N.
- By $\log_a N = \frac{\log N}{\log a}$ we denote 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 $\lceil x \rceil$.
- By $\Delta f(x) := f(x+1) - f(x)$ we denote the difference operator of finite calculus.

2. Discrete Loops

In this section 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. 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‡. 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 1. In this example the loop-variable `k` will

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

FIGURE 1. A simple example of a discrete loop

assume the values $1, 2, 4, 8, 16, 32, 64, \ldots$ until finally a value greater than `N` 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. In this example the loop-

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

FIGURE 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 can not be achieved by a for-loop, because the value of the loop variable can not 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 can not determine which of the alternatives will be chosen,

‡In Section 5 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.

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 worst-case performance of real-time programs.

By the way, a loop like that in Figure 2 occurs in a not-recursive implementation of *Heapsort* (cf. [13] or [14] for a more readable form in a high-order programming language). Sections 3.2 and 6.2 will be concerned with algorithms that can profit from discrete loops; Heapsort will be treated in detail in Section 3.2.1.

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 can not 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.

2.2. 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. 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 2.2. 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. Note that $k_\nu \in \mathbb{N}$ for all $\nu \in \mathbb{N}$.

Definition 2.3. 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 2.4. 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 2.5. Let a discrete loop be characterized by N and the iteration functions $f_i(x)$, $1 \leq i \leq e$. Let the initial value of the loop variable be s. For $1 \leq x \leq N$ associating to each function f_i a function $\overline{f}_i$ by

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

we define the corresponding *loop digraph* $\mathcal{L}$ 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. *A discrete loop completes if and only if the corresponding loop digraph is acyclic.* □

Each acyclic digraph can be *topologically sorted* (cf. [15]), i.e., we can find a mapping $\mathrm{ord} : V \to \{0, 1, \ldots, N, N+1\}$ such that for all edges $(v, w) \in E$ we have $\mathrm{ord}(v) < \mathrm{ord}(w)$. 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.

3. Monotonical Discrete Loops

Definition 3.1. A sequence (k_ν) is called *strictly monotonically increasing* if $k_{\nu+1} > k_\nu$ for all $\nu \geq 1$. It is called *strictly monotonically decreasing* if $k_{\nu+1} < k_\nu$ for all $\nu \geq 1$.

Definition 3.2. 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 3.1. *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 3.2. *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 3.2. □

Lemma 3.3. *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 3.2. □

3.1. 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 [4].

```
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 3.1. 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 3.2. 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.

3.2. Some Examples of Monotonical Discrete Loops.

3.2.1. *Heapsort.* An Ada-like implementation of Heapsort using a monotonical discrete loop is shown in Figure 3. Referring to the code shown in Figure 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

$$\begin{aligned} h_1^{(\min)} &= k \\ h_{\nu+1}^{(\min)} &= 2h_\nu^{(\min)} \end{aligned} \tag{3.1}$$

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 4.)

Solving (3.1) we arrive at

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

We want to determine the value of ω such that

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

```
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 3. Implementation of Heapsort using a Discrete Loop

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. [16, problem 3.34])

$$\sum_{k=1}^{n} \lceil \operatorname{ld} k \rceil = n \lceil \operatorname{ld} n \rceil - 2^{\lceil \operatorname{ld} n \rceil} + 1 \tag{3.2}$$

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

$$\sum_{k=1}^{n} \lfloor \operatorname{ld} k \rfloor = n \lceil \operatorname{ld} n \rceil - n - 2^{\lceil \operatorname{ld} n \rceil} + \lfloor \operatorname{ld} n \rfloor + 2. \tag{3.3}$$

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. [16, 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

$$\begin{aligned} F \leq & \lfloor N/2 \rfloor \left(\lceil \operatorname{ld} N \rceil - \lfloor \operatorname{ld} N \rfloor + 2 \right) + 2^{\lfloor \operatorname{ld} N \rfloor} - \lfloor \operatorname{ld} N \rfloor - 1 \leq \\ & 3 \lfloor N/2 \rfloor + 2^{\operatorname{ld} N} - \lfloor \operatorname{ld} N \rfloor - 1 \leq \left\lfloor \frac{5}{2} N \right\rfloor - \lfloor \operatorname{ld} N \rfloor - 1. \end{aligned}$$

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

$$L \leq \sum_{t=2}^{N} \lfloor \operatorname{ld} t \rfloor .$$

Using (3.3), L can be estimated by

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

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 4 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. [13]) and even the average timing behavior (cf. [14]) 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 Project WOOP which is carried out at the *Department of Automation* at the *Technical University of Vienna*.

3.2.2. *Other Examples.* Other examples showing the advantages of discrete loops over general while or repeat-loops include the bottom-up version of *Mergesort* (cf. [17]), *Euclid's algorithm*, and the solution of *Josephus' Problem* (cf. [16]).

4. 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.

4.1. Lower and Upper Bounds.

Definition 4.1. 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 [15] 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(f̄_i(k)) := max{c(f̄_i(k)), c(k)+1};
  end loop;
end loop;
```

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

```
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(f̄_i(k)) := min{c(f̄_i(k)), c(k)+1};
  end loop;
end loop;
```

Remark 4.1. 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 4.1 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 4.2. 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 4.1. *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 4.1 is easily proved by induction. □

Lemma 4.2. *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

$$f(x+k) - f(x) \geq k \tag{4.1}$$

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 (4.1) 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 4.3. *If $y > x$ and $\frac{f(y)-f(x)}{y-x} \geq 1$, then $f(y) > f(x)$.* □

Theorem 4.1. *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_ν^{max}) is equal to $\mathfrak{l}$, and*
(3) *the length of (k_ν^{min}) is equal to $\mathfrak{u}$.*

Proof. Case (1) follows immediately from Lemma 4.1.

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 4.

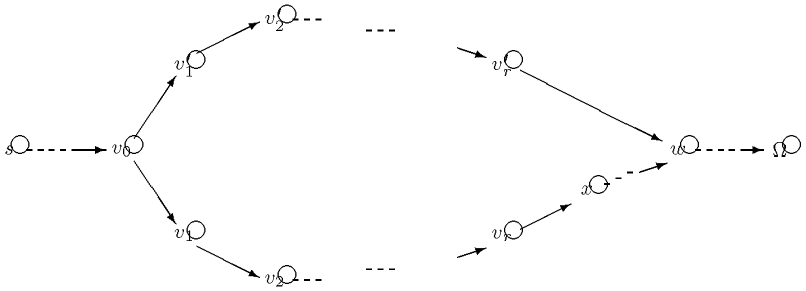

FIGURE 4. Paths in a Loop Digraph

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 $(s, \ldots, v_0, v_1', \ldots, v_r', w, \ldots, N+1 = \Omega)$, $v_1' \neq v_1, \ldots, v_r' \neq v_r$ can not 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 4.3.

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

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

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 4.1 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.

4.2. 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 4.2. *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 4.3. *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*

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

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 4.4. *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*

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

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 4.3 to this we have proved Theorem 4.4. □

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

Integrating the results of Theorems 4.2, 4.3, 4.4, 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 worst case timing behavior of a real-time program and for a real-time scheduler.

4.3. 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 can not be presented in a simpler form. It simplifies, however, if more special cases are considered.

4.3.1. *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

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

4.3.2. *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 $\omega_j(N) = \lfloor \mathrm{ld}\, N + 1 \rfloor$. Hence the number of executions of the innermost loop body is bounded above by

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

which by (3.3) is equal to

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

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

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

5. 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 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 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 6.1.

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.

```
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 5. Implementation of Binary Search using a Discrete Loop

5.1. 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

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

depending on which sub-interval contains the element being sought. If the sought element is equal to $\left\lfloor \frac{u_\nu + l_\nu}{2} \right\rfloor$, 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

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

Thus

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

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

$$\begin{aligned} U_1 &= N, \\ U_{\nu+1} &= \lfloor U_\nu/2 \rfloor. \end{aligned} \tag{5.1}$$

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 (5.1) by applying standard techniques (cf. e.g. [16]) 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.

6. Discrete Loops with a Remainder Function

Definition 6.1. 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 6.1. Definition 6.1 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 6.2. A loop sequence of remaining items is called *monotonical* if $r_{\nu+1} < r_\nu$.

Definition 6.3. A discrete loop with a remainder function is called *monotonical* if all its loop sequences $(r_\nu) \in \mathcal{R}$ are monotonical.

Lemma 6.1. *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. □

6.1. 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 [4]. In fact we add to the syntax definition of Section 3.1.

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 6.2. 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 6.3. 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.

6.2. Some Examples of Monotonical Discrete Loops with Remainder Functions. One illustrative example, *binary search*, has already been discussed in Section 5.1, but one syntactical remark is necessary: Line 14 of Figure 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.

6.2.1. *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 6. In this figure `root` denotes a pointer to the

```
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 6. Template for Traversing Binary Trees

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.

6.2.2. *Weight-Balanced Trees.* So-called *weight-balanced trees* have been introduced in [18] and are treated in detail in [19] and in [20].

Definition 6.4. 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$[\alpha]$ 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$[\alpha]$-trees. Here N is the number of leaves in the BB$[\alpha]$-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 [19] and [20]).

BB$[\alpha]$-trees are binary trees with bounded height. In fact it is proved in [19] that

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

where N is the number of leaves in the BB$[\alpha]$-tree T.

A template for the above operations is shown in Figure 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 7. A Template for Operations on BB$[\alpha]$-trees

supposed to implement $\lfloor x \rfloor$. Since the notion of height defined in [19] is not well-suited for direct application of discrete loops, the remainder function in Figure 7 has been slightly modified.

A semantically equivalent template for traversing BB$[\alpha]$-trees is shown in Figure 8. The remainder function of Figure 8 has the advantage that it does not 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.

```
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 8. Another Template for Operations on BB[α]-trees

6.3. The Number of Iterations of a Monotonical Discrete Loop with a Remainder Function. A special case has already been discussed in Section 5.1, but these computations can be generalized.

Theorem 6.1. *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

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

Thus

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

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

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

Taking logarithms the theorem is proved. □

7. 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 7.1. *If the number of iterations of a general loop can be determined by an integer-valued* computable function [21] 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 7.1. Obviously, in practical applications the remainder function in the proof of Theorem 7.1 will not always be the best choice with regard to software engineering, but it is the purpose of Theorem 7.1 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 7.2. 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 7.1 may loop forever in computing $r_1 := \Phi$.

8. Conclusion

In this paper 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 timing behavior 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

$$r_i = U,$$
$$r_{\nu+1} = r_\nu - 1.$$

Obviously this is semantically equivalent to the approaches described in the introduction (cf. [2, 5, 7, 8, 10]): 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

$$\begin{aligned} r_i &= T, \\ r_{\nu+1} &= r_\nu - \mathrm{time(loop\_body)}, \end{aligned} \tag{8.1}$$

where time(loop_body) is the time that passed since (8.1) 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.

Acknowledgment. The author would like to thank Roland Lieger for his valuable suggestions which resulted in several improvements of the manuscript.

References

1. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
2. W. A. Halang and A. D. Stoyenko. *Constructing predictable real time systems.* Kluwer Academic Publishers, Boston, 1991.
3. A. K. Mok. The design of real-time programming systems based on process models. In *Proceedings of the IEEE Real Time Systems Symposium*, pages 5–16, Austin, Texas, 1984. IEEE Press.
4. ANSI/MIL-STD 1815 A. *Reference manual for the Ada programming language*, 1983.
5. P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.
6. V. Nirkhe and W. Pugh. A partial evaluator for the Maruti hard real-time system. *The Journal of Real-Time Systems*, 5:13–30, 1993.
7. C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *The Journal of Real-Time Systems*, 5:31–62, 1993.
8. A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.
9. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12:576–580, 1969.
10. N. Gehani and K. Ramamritham. Real-time Concurrent C: A language for programming dynamic real-time systems. *The Journal of Real-Time Systems*, 3:377–405, 1991.
11. A. 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.
12. Y. Ishikawa, H. Tokuda, and C. W. Mercer. Object-oriented real-time language design: Constructs for timing constraints. In *ECOOP/OOPSLA '90 Proceedings*, pages 289–298, October 1990.
13. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, Mass., 1973.
14. R. Schaffer and R. Sedgewick. The analysis of heapsort. *Journal of Algorithms*, 15:76–100, 1993.
15. K. Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms.* Springer-Verlag, Berlin, 1984.
16. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading, MA, 1989.
17. R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, second edition, 1988.
18. I. Nievergelt and E. Reingold. Binary search trees of bounded balance. *SIAM Journal of Computing*, 2(1):33–43, 1973.
19. K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms.* Springer-Verlag, Berlin, 1984.
20. N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980.
21. M. Davis. *Computatbility and Unsolvability.* Dover, New York, N.Y., 1982.