Multi-Staged Discrete Loops for Real-Time Systems

Roland Lieger[1] and Johann Blieberger
Department of Automation (183/1)
Technical University of Vienna
Treitlstr. 1/4
1040 Vienna, Austria
rlieger/blieb@auto.tuwien.ac.at

Abstract

In this paper Multi-Staged Discrete Loops *are introduced to narrow the gap between for-loops and general loops. Although multi-staged discrete loops can be used in situations that would otherwise require general loops it is still possible to determine the maximum number of iterations, which is trivial for for-loops but extremely difficult for general loops. Thus multi-staged discrete loops form an excellent framework for determining the worst-case performance of a program.*

1. Introduction

One of the most important properties of a real-time system is, that it must not only supply correct results, but that the computation must be completed within a predefined deadline. It is well known that major progress towards the goal of timeliness can be achieved if the *scheduling problem* is solved properly. As most scheduling algorithms (e.g. [3], [4]) assume that the runtime of a task is known a-priori, the *worst-case performance* of a task plays a crucial role.

Common programming languages support two kinds of loops:

- for-loops: The loop variable is set to all values within a range of integers. Starting with the smallest value, the loop variable is incremented on each iteration of the loop body until it is outside the range. Some languages are more flexible and allow some extensions of this concept, such as starting with the biggest value of the range and decrementing on every iteration or incrementing/decrementing by a fixed step size.
- general-loops: The other loop-statements are of a very general form and are used for implementing loops that can not be handled by for-loops. General loops include while-loops, repeat-loops and loops with exit-statements. (cf. e.g. [2])

Computing the number of iterations for a for-loop is trivial. For example the following for-loop is executed exactly $\lceil N/S \rceil$ times.

```
For I:=1 To N By S Do
  Loop-Body
End;
```

Analyzing general loops is a much more difficult task. In order to avoid the problems connected with estimating the worst-case performance of general loops 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 [6]). Other researchers attempt to do static and dynamic program path analysis using regular expressions (e.g. [5]).

In [1] the concept of *discrete loops* is introduced. Like a for-loop a discrete loop uses a loop variable that is incremented/decremented on every iteration of the loop until it does no longer fall into a given range. With a common for-loop a fixed value (usually one) is added to the loop variable on each pass. Discrete loops permit the use of a wide range of functions for the computation of the next value of the loop-variable. Nevertheless tight bounds on the number of iterations can be computed at compile-time.

One limitation of discrete loops is that the next value of the loop variable can only be based on its current value. It is not possible to refer to past values. This problem is resolved by the use of multi-staged discrete loops, that will be

[1] Supported by the Austrian Science Foundation (FWF) under grant P10188-MAT

introduced in this paper.

2. General Notation

Let **N** denote the natural numbers $\{1, 2, 3, 4, 5, \ldots\}$ and **Z** be the set of integers $\{\ldots - 3, -2, -1, 0, 1, 2, 3, \ldots\}$.

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

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

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 $\mathbf{N} \rightarrow \mathbf{N}$).

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

$(F)(x) = (a_k)$ such that

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

3. Some Interesting Examples

Example 1 *Catalan Numbers*

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

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

Example 2 *Fibonacci Numbers*

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

Computing some Fibonacci Numbers we see that the vector $[a_k]$ *begins with* $[1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots]$.

Example 3 *Factorial Numbers* $k!$

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

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

Example 4

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

Where $\lceil x \rceil$ *denotes the smallest integer* n *with* $n \geq x$. *This formula produces the sequence*
$[a_k] = [1,\ \ 2, 2,\ \ 3, 3, 3,\ \ 4, 4, 4, 4,\ \ 5, \ldots]$.

4. Multi-Staged Discrete Loops

Definition 4.1 (Multi-Staged Discrete Loops) *A* multi-staged discrete loop *(*MSDL*)* $\mathcal{L}$ *is characterized by a value* $N \in \mathbf{N}$ *(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* $\mathcal{S} = [\underline{s} \ldots \overline{s}]$.

Definition 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*

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

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

Definition 4.3 (History Depth) *Let* $\mathcal{D}(f^{(k)}) = \max(j :$ a_{k+1-j} *is accessed to compute* $a_{k+1} = f^{(k)}([a_k])$. *If there exists a* $C \in \mathbf{N}$ *such that* $\mathcal{D}(f^{(k)}) < C$ *for all* $k \in \mathbf{N}$, *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 4.4 (Monotonic Functions, Sequences, Loops) *We call* $f^{(k)}$ *a* monotonic *function if* $[a_k] \leq [b_k] \Rightarrow f^{(k)}([a_k]) \leq f^{(k)}([b_k])$. *It is called a* strictly monotonic *function if* $[a_k] < [b_k] \Rightarrow f^{(k)}([a_k]) < f^{(k)}([b_k])$

and strictly d-monotonic *if it is monotonic and* $[a_k] <_d [b_k] \Rightarrow f^{(k)}([a_k]) < f^{(k)}([b_k])$*. 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* $f^{(k)}$ *is monotonic for all* $k \in \mathbf{N}$*. 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.

Definition 4.5 (Length of a Path) *Let* $[a_k]$ *be a path* $\mathcal{P}$ *of a strictly monotonic 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* $len(\mathcal{P}) = l$*. Since we require strict monotonicity such a* l *will always exist.*

Theorem 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.*

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

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 4.6 (Complete, Partial and Bounded History) *Depending on* $\mathcal{D}(\mathcal{L})$ *various forms of* MSDL *can be distinguished. If* $\mathcal{D}(\mathcal{L}) = 1$ *(e.g. Example 3 - Factorial Numbers) we have a (plain, single staged) discrete loop. This case as been studied extensively in [1], and will not be treated in this paper. If* $1 < \mathcal{D}(\mathcal{L}) < \infty$ *(e.g. Example 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 1 - Catalan Numbers) than we say* $\mathcal{L}$ *uses the* complete history. *Otherwise (e.g. Example 4) we speak of* partial histories.

5. Iteration Bounds for *MSDL*

Definition 5.1 (Min./Max. Successor and 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 select $\underline{i}([a_k])$ *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 5.1 *Let* $\mathcal{L}$ *be a monotonic multi-stage discrete loop, then we know for any path* $\mathcal{P} \in \mathcal{K}$ *that*

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

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

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

Theorem 5.1 *Let* $\mathcal{L}$ *be a monotonic* MSDL *and let* $f^{(k)}_i([a_k]) > a_k$ *for all* $1 \leq i \leq e$*,* $k \in \mathbf{N}$ *then*

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

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

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 $k \in \mathbf{N}$. As the a_k are elements of **N** there are only a finite number of elements (exactly: N) in the range $[1 \ldots N]$ the loop must complete after at most N iterations.

2. By definition of $\underline{l} := \text{len}(\overline{\mathcal{P}})$ (cf. Def. 4.5) it is obvious that $\overline{a_{\underline{l}-1}} \leq N < \overline{a_{\underline{l}}}$. In Lemma 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 the the above case we have $N < \underline{a_{\overline{l}}} \leq a_{\overline{l}}$ and thus $\text{len}(\mathcal{P}) \leq \overline{l}$. $\square$

Theorem 5.1 requires that $f^{(k)}([a_k]) > a_k$. Sometimes an alternative condition is easier to prove.

Theorem 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:

- $a_k = 1$: $[a_k] = (a_1, a_2, \ldots, a_{k-1}, 1), a_j \geq 1$ 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 $f^{(k)}([a_k]) > 1 = a_k$.

- $a_k > 1$: 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 $f^{(k)}([a_k]) > a_k$. $\square$

Remark 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 $a_k \geq Fib(k)$, 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 $a_k \geq 2^{k-2} + 1$.

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 $\overline{l}$. As discussed in Theorem 5.1 $\overline{l}$ 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 $\overline{l}$ is an upper bound for the number of iterations of the *MSDL*.

Using $\overline{l}$, 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 $\overline{l}$ 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 does itself 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 $\overline{l}$ times the space needed for a natural.

Frequently $\mathcal{D}(F) \ll \overline{l}$ 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).

7. Examples of useful iteration functions

Only a very limited set of iteration functions is actually used in actual programming practice.

Probably the most frequently used type of iteration function is

$$f^{(k)}([a_k]) = \begin{cases} B^{(k)} & \text{for } k < \mathcal{D}(F) \\ \sum_{j=0}^{\mathcal{D}(F)-1} c_j^{(k)} a_{k-j} + C^{(k)} & \text{otherwise} \end{cases}$$

where $c_j^{(k)} \in \mathbf{N}$, $B^{(k)} \in \mathbf{N}$, $B^{(k+1)} > B^{(k)}$, and $C^{(k)} \in \mathbf{Z}$, $C^{(k)} > \mathcal{D}(F) + 1 - \sum_{j=0}^{\mathcal{D}(F)-1} c_j^{(k)} \cdot (\mathcal{D}(F) - j)$. Example 2 (Fibonacci Numbers) uses this type of recursion function.

It is not necessary that the values of $c_j^{(k)}$ be the same in all $f^{(k)}$ independent of k (even so this is the most common situation). For instance Example 3 (Factorial Numbers) uses a new value for $c_0^{(k)}$ on every iteration.

Positive integral polynomials offer a wide field of possible iteration functions.

$$f^{(k)}([a_k]) = \begin{cases} B^{(k)} & \text{for } k < \mathcal{D}(F) \\ \sum_{i=0}^{m} c_i \prod_{j=0}^{\mathcal{D}(F)-1} a_{k-j}^{d_{i,j}} + C & \text{otherwise} \end{cases}$$

where $C, B^{(k)} \in \mathbf{N}$ constant, $B^{(k+1)} > B^{(k)}$, $c_i, d_{i,j} \in \mathbf{N}$ independent of k.

Probably the most frequently used type of (non-linear) polynomials are convolutions, as they are used in Example 1 (Catalan Numbers).

$$f^{(k)}([a_k]) = \begin{cases} B^{(k)} & \text{for } k < \mathcal{D}(F) \\ \sum_{j=0}^{\mathcal{D}(F)-1} c_j a_{k-j} a_{k-(\mathcal{D}(F)-j-1)} + C & \\ & \text{otherwise} \end{cases}$$

where $B^{(k)} \in \mathbf{N}$ constant, $B^{(k+1)} > B^{(k)}$, $c_j \in \mathbf{N}$ independent of k and $C \in \mathbf{Z}$ constant, $C > \mathcal{D}(F) + 1 - \sum_{j=0}^{\mathcal{D}(F)-1} c_j \cdot j \cdot (\mathcal{D}(F) - j)$.

All the above types of iteration functions are guaranteed to satisfy the prerequisites of Theorem 5.1.

Of course, this brief list of function types can not cover all possibilities, but it gives a good overview of those types that are frequently encountered in every-day use.

It is easy to extend the syntax of any common programming language to accommodate *MSDL*s. If only the iteration functions of the above type are permitted then the compiler can easily check at compile time whether the code written by the user contains valid functions. If the syntax of the programming language ensures that the program can use only the given iteration functions to modify the loop variable (i.e. side effects of non-loop-related statements that change the loop variable are prohibited) then the upper bound for the number of iterations can be guaranteed without incurring any runtime overhead.

8. Conclusion

In this paper 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 on 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 [1] 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.

Nevertheless some work remains to be done in the future. The following lists a few items:

- Additional types of useful iteration functions can be included.
- We can not yet handle loops that only eventually increase the loop variable, but do not do so on every pass through the loop body (e.g. Example 4).
- The compile time computations should be done automatically. Implementing the necessary tools is part of Project WOOP currently being performed at the Technical University of Vienna.

References

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

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

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

[4] 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.

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

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