Theoretical Computer Science 7 (1978) 79–98
© North Holland Publishing Company

SOME APPLICATIONS OF THE McCREIGHT–MEYER ALGORITHM IN ABSTRACT COMPLEXITY THEORY*

P. van EMDE BOAS

Department of Mathematics, University of Amsterdam, Amsterdam, The Netherlands

Communicated by A. Meyer
Received August 1976
Revised November 1977

Abstract. The McCreight–Meyer algorithm is a priority-queue construction from abstract recursion theory which was designed for the proof of the so-called Naming or Honesty theorem. We explain the ideas behind the algorithm, pointing at its behaviour as a "closure operator" and obtaining various known and new results as corollaries of more general assertions.

1. Introduction

The McCreight–Meyer algorithm was designed in order to prove the so-called Naming or Honesty theorem which states that the entire hierarchy of complexity classes with respect to some abstract complexity measure can be renamed uniformly and effectively by a so-called measured transformation. This means that there exists a total recursive function σ such that for each i the programs φ_i and $\varphi_{\sigma(i)}$ compute two names for the same complexity class of programs and such that moreover the predicate $\varphi_{\sigma(i)}(x) = y$ is decidable (whereas the corresponding predicate $\varphi_i(x) = y$ might not be). The result is relevant if related to the Compression theorem and the Gap theorems. The Compression theorem states that a system of complexity classes named by a measured set can be extended uniformly, using functional composition on the names of the classes. The Gap theorems state that such a uniform extension by functional composition or even total effective operators working on the names of classes is not possible for the complete system of classes named by the sequence of all programs. For details the reader is referred to Hartmanis and Hopcroft [5]. Applications of the Naming theorem for the construction of hierarchies over the ordinals are given by Bass and Young [1].

Descriptions of the McCreight–Meyer algorithm have been published in many places; see e.g. McCreight and Meyer [8], Bass and Young [1], Hartmanis and

* This report is registered as ZW 79/77 of the Mathematical Center.

Hopcroft [5], Meyer and Moll [10], or the present author [3]. In the present paper we present a version using a full parallelism instead of the traditional dovetailed computation.

The algorithm uses the old name in order to obtain a test routine which investigates each program infinitely often. Programs belong to the complexity class provided they fail the test only finitely many times. This indicates that as well as renaming complexity classes the algorithm might in general be used for "naming" classes of programs defined using some "almost everywhere" condition. As is well-known, these classes are precisely the classes of programs which have a Σ_2-presentation. However not all Σ_2-classes of programs are complexity classes (the reverse inclusion being trivial). McCreight [8] gives the example of a union of two complexity classes which itself is not a complexity class.

It turns out that the McCreight–Meyer algorithm acts as a closure operator, computing a name for the smallest complexity class containing a given Σ_2-class of programs. If the given class is already a complexity class it renames this class. On the other hand the algorithm can be used to "name" finite intersections and increasing unions, in this way obtaining a strengthening of McCreight's Union theorem.

Next we consider the question whether the algorithm can be adapted to deal with other types of classes of programs defined by complexity of computation. As a first candidate we consider the so-called weak complexity classes, introduced by the present author in order to study the behaviour of the honesty classes [3, 4]. It is known that the Naming theorem fails for these classes, but it turns out that the failing McCreight–Meyer algorithm can be amended (at the price of the "measured-set" property of the sequence of names obtained by the algorithm). This way the "closure operator" properties are preserved.

Finally we introduce the so-called anti-complexity classes consisting of programs having as the name of the class a lower bound on their run-times. By applying an order inversion to the algorithm one obtains after some minor modifications an algorithm computing a name for the smallest anti-complexity class containing a given Σ_2-class of programs. As a corollary we obtain Levin's result that a complexity sequence of a total recursive function has a greatest lower bound [7].

2. Notation and definitions

We use a fixed recursive pairing function $\langle x, y\rangle$ with coordinate inverses π_1 and π_2. Let $((\varphi_i)_{i=0}^{\infty}, (\Phi_i)_{i=0}^{\infty})$ denote a complexity measure, cf. Blum [2]. This means that $(\varphi_i)_{i=0}^{\infty}$ is an *acceptable Gödel numbering* and φ_i and Φ_i have equal domain for each i and the predicate "$\Phi_i(x) = y$" is recursive in i, x and y. The functions φ_i are called *programs* and the functions Φ_i are called *run-times*. All functions are partial recursive unless explicitly stated to be total. The *domain* of a function f is denoted

$\mathscr{D}f$, and we write $f(x)<\infty$ ($f(x)=\infty$) for $x\in\mathscr{D}f$ ($x\notin\mathscr{D}f$). We say that a function f is computed by φ_i or equivalently i is an *index* for f provided $f=\varphi_i$ extensionnally, i.e., $\forall x\ [f(x)=\varphi_i(x)]$. The inequality $f\leq g$ means that $\mathscr{D}g\subseteq\mathscr{D}f$ and $f(x)\leq g(x)$ for all $x\in\mathscr{D}g$.

Throughout this paper programs φ_i will be considered either as syntactical objects, completely determined by their index i, or as representing the functions computed by the program. The intended meaning should always be clear from the context.

We use the quantifier $\forall^\infty$ for "almost everywhere" so $\forall^\infty x[P(x)]$ denotes "$P(x)$ holds for all but finitely many x". The expression $f\propto g$ denotes $\forall^\infty x[f(x)\leq g(x)]$ and is stated as "*g asymptotically bounds f*". By convention $\infty\leq\infty$ holds.

A *transformation of programs σ* is a total recursive function (operating on the indices of programs); examples of transformations are obtained by application of the s-m-n axiom or the Recursion theorem.

If R is a two-variable partial recursive function we say that φ_i is *R-honest* provided $\forall^\infty x\ [\Phi_i(x)\leq R(x,\varphi_i(x))]$, the condition holding vacuously for $x\notin\mathscr{D}\varphi_i$ or $\langle x,\varphi_i(x)\rangle\notin\mathscr{D}R$. A sequence of functions $(\gamma_i)_{i=0}^\infty$ is called a *measured set* provided the predicate "$\gamma_i(x)=y$" is recursive in i, x and y. A transformation σ is *measured* provided $(\varphi_{\sigma(i)})_{i=0}^\infty$ is a measured set. It is known that a measured set consists of R-honest functions for some total R and that conversely for total R the collection of R-honest functions can be presented by a measured set [8, 10].

A *Σ_2-class of programs* is a set of programs $\mathscr{X}=\{\varphi_i\mid i\in A\}$ where A is a Σ_2-class in the usual sense; consequently the set A can be presented as $\{i\mid\forall^\infty x\ [B(i,x)]\}$ where B is a total recursive predicate called the *discriminator* of $\mathscr{X}$. These classes of programs correspond to what has been called *index-sets* elsewhere.

For partial recursive functions t we consider the following classes of programs determined by complexity of computation:

$F(t)=\{\varphi_i\mid\Phi_i\propto t\}$	*the complexity class*
$F_w(t)=\{\varphi_i\mid\forall^\infty x\ [\Phi_i(x)\leq t(x)\text{ or }\varphi_i(x)=\infty]\}$	*the weak complexity class*
$A(t)=\{\varphi_i\mid\forall^\infty x\ [x\in\mathscr{D}t\Rightarrow\Phi_i(x)\geq t(x)]\}$	*the anti-complexity class*

The function t is called a *name* for these classes. Each of the above classes can be shown to be a Σ_2-class of programs. Note that the above classes may be defined for non-recursive names t as well, but in the latter case the resulting classes no longer are Σ_2-classes of programs.

Complexity classes have been studied from the start of abstract complexity theory, cf. [1, 2, 5, 8, 11]. Weak complexity classes were introduced by the present author for the analysis of the behaviour of honesty classes, cf. [3 and 4]. The anti-complexity classes are introduced here for the first time in order to obtain as a corollary a recent result by Levin on the greatest lower bound of complexity sequences [7].

As announced in the introduction we present our algorithms using parallelism instead of the traditional dovetailing. This is done in order to illustrate the modular decomposition of the McCreight–Meyer algorithm and the large degree of freedom one has in obtaining a sequential algorithm by combining the asynchronous parts in some synchrone order [10]. Clearly, the use of parallelism requires some method of protecting oneself against undesired misusing of shared data. We will use the concept of critical sections, during which a process has private access to the shared data. Consequently we are obliged to ensure that no process can diverge within a critical section. This fact being verified the remaining necessary condition for our parallel program to be correct is that no process is prevented from becoming active by the other processes, and freedom of deadlocks. In our algorithms we will indicate the critical sections used, taking the absence of deadlocks and lockouts for granted.

Note moreover that by using parallelism we obtain non-deterministic programs as well. Again the motivation is that the non-determinism illustrates the freedom one has for obtaining correct sequential implementations. Also the use of parallelism might yield non-recursive functions as a result of parallel non-deterministic computations. We shall assume however that the parallelism is sufficiently regulated to obtain nothing but recursive functions.

3. The classical McCreight–Meyer algorithm

The *McCreight–Meyer algorithm* (called the MCM-algorithm hereafter) was introduced in order to prove the following theorem [8]:

Theorem 3.1. *There exists a measured transformation of programs σ satisfying $\forall i\, [F(\varphi_i) = F(\varphi_{\sigma(i)})]$.*

In the algorithm the old name φ_i is used in order to obtain a discriminator for $F(\varphi_i)$, which during a dovetailed computation executes an infinite sequence of tests on all programs φ_j. Depending on the outcome of these tests these programs are manipulated on a priority queue as being "good" or "bad"; in the meantime a new name $\varphi_{\sigma(i)}$ is computed in such a way that "good" programs are respected and "bad" programs are punished. A "bad" program which is punished becomes "good" whereas a "good" program becomes "bad" after failing a test by the discriminator. Each change of status induces loss of priority, thus ensuring that the stable "good" programs will be respected in the limit. The resulting names $(\varphi_{\sigma(i)})_{i=0}^{\infty}$ form a measured set because of the fact that they are computed by a well-behaved "least-number operation" on some ever changing condition.

The tests against the old name φ_i are of the following type:

$$B_i(j, x) = \textbf{if } \Phi_i(\pi_1 x) = \pi_2 x \textbf{ then } \Phi_j(\pi_1 x) \leq \varphi_i(\pi_1 x) \textbf{ else true fi}$$

This shows that for each i the predicate B_i is total and consequently $\varphi_j \in F(\varphi_i)$ iff $\forall^\infty x\ [B_i(j, x)]$. Hence $F(\varphi_i)$ is nothing but the Σ_2-class of programs discriminated by B_i. It makes sense therefore to consider the behavior of the MCM algorithm which results from replacing the discriminator B_i by some arbitrary discriminator B. The resulting algorithm is described in this section using the framework of full parallelism (leaving the transformation to a dovetailed computation to the underlying operating system).

Let B be some total recursive function in two variables. The *McCreight–Meyer algorithm based on B* consists of three processes operating on a shared data structure, which we call the *priority queue*. The priority queue consists of a linear list of items, composed from an integral value *index* and a boolean value *colour*, where for didactical reasons the values **true** and **false** are denoted by **white** and **black**. We say that item A *has a higher priority than* B if A precedes B in the linear list. An item A is *displaced* by moving it to the tail of the list and reversing its colour simultaneously.

The first process is the so-called *governor*. Its task is to introduce all indices into the priority queue. The second process is called the *incriminator*. It is the unique process which has access to the discriminator B. Its task is to execute tests on the white items currently on the priority queue displacing those white items which fail a test.

The third process *computation* turns out to consist of an infinite sequence of processes running in parallel, called *comp*(y). The process comp(y) tries to compute a value for the new name t at argument y, and after succeeding to provide such a value, it tests all black items on the priority queue against it, displacing (i.e., punishing) those black indices which fail this test. The processes comp(y) are the unique processes having access to the decision procedure for the predicate $\Phi_i(x) \leq z$. Computation of the value is performed by the procedure *searchval*. Together with this later procedure comp(y) reflects the structure of the complexity classes; MCM-algorithms for weak complexity classes and anti-complexity classes are obtained by modifying the procedures searchval, and comp.

In Fig. 1 we illustrate the modular decomposition explained above.

The crucial procedure within the MCM-algorithm is the procedure searchval. Its task consists of computing a value z for $t(y)$ such that there exists a black item with index j for which $\Phi_j(y) > z$ such that for all white items $(i, \textit{white})$ which have higher priority we have $\Phi_i(y) \leq z$; moreover the black item used this way should have the highest possible priority. Below we present a description of searchval; its behaviour is illustrated in Fig. 2.

```
Procedure searchval;
Input an argument $y \in \mathbf{N}$;
Output a value for $t(y) \in \mathbf{N}$;
```

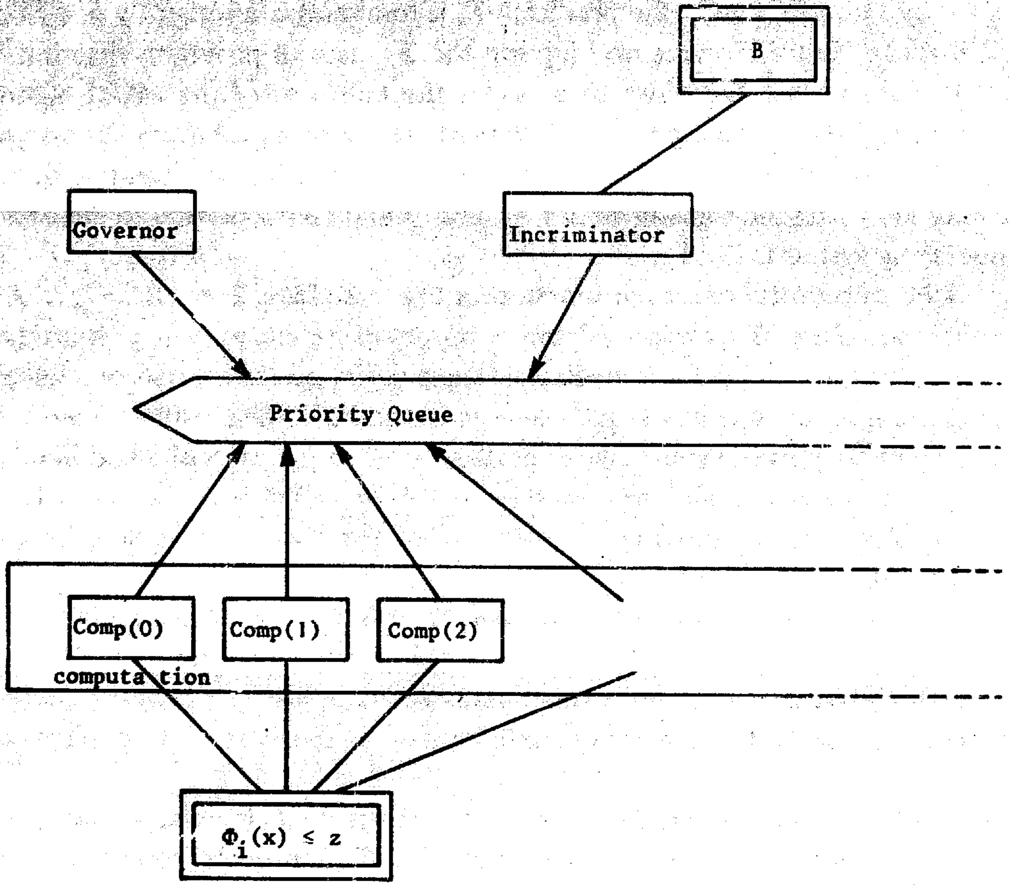

Fig. 1. Modular decomposition of the MCM-algorithm.

```
Initialization start with attempted value val = 0 at the beginning of the priority
    queue;
loop if the next item on the queue is (i, white) stepwise increase val until
    $\Phi_i(y) \le$ val, and proceed to the next item on the queue; if the next item on
    the queue is (i, black) then terminate with output val if $\Phi_i(y) >$ val and
    proceed to the next item on the queue otherwise; if there is no more item on
    the queue the computation is set to diverge;
end searchval
```

The critical sections in searchval are the test for whether there is a next element on the queue and the acquisition of this element. The computation of searchval is illustrated in Fig. 2. The list on the bottom illustrates the priority queue, a shaded second square indicating a black item. A dot with an upward arrow indicates a white run-time which has to be respected by increasing val above it, whereas a downward arrow shows a black run-time which is violated by attempting to define val below it. Computation starts in the lower left corner and terminates successfully at the dot next to the exclamation mark.

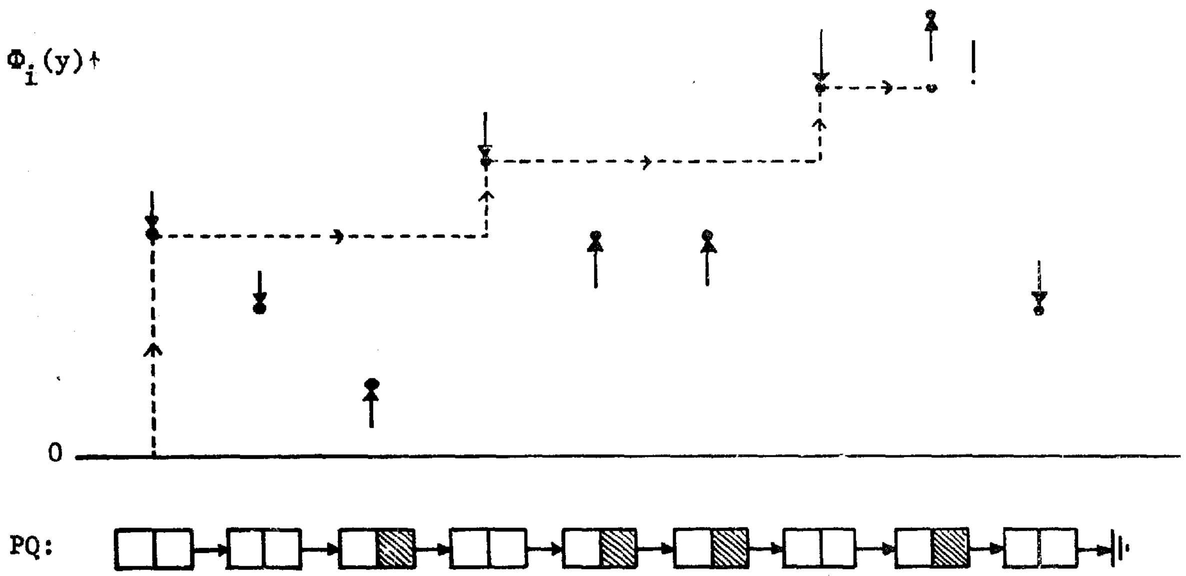

Fig. 2. The computation of searchval.

From Fig. 2 and the explanation of searchval, one concludes that searchval indeed stops at the highest priority black item having a run-time exceeding the run-times of the preceding white items (where by the run-time of an item we understand the run-time of the program with the same index at the underlying argument y). Consequently termination of searchval is certain whenever such a black item exists. The procedure searchval may fail either by diverging to infinity at a fixed white item with infinite run-time, or by exhausting the priority queue. In both cases the value of the new name will be undefined at y, and the process comp(y) will fail to terminate. In fact we might as well have closed the priority queue by a symbolic white item (**nil**) with infinite run-time, in this way preventing exhaustion of the priority queue. Note however that searchval will never diverge within a critical section.

As a consequence of the parallelism in the MCM-algorithm it is moreover thinkable that the queue as indicated in Fig. 2 is transformed during a computation of searchval by activities of other processes. Such transformations always displace items to the tail of the queue and in this way they may prevent eligible black items from being selected. This does however not affect the validity of the proofs given in the sequel, the argument being that a certain black item will be displaced now unless it is displaced by some other process before, in which case nothing remains to be proven.

The three modules and the MCM-algorithm are described below:

```
process governor;
  initialization set $i$ to 0;
  loop create item ($i$, white) and append it at the end of the priority queue; next
        increment $i$ and wait for a while
end governor;
```

The critical section in the governor is the append operation. The governor successively introduces all indices in the priority queue, but in doing so it should yield to the remaining processes time for accessing the queue as well. Clearly the governor will never diverge within a critical section.

```
**process** incriminator;
  **initialization** set $x$ to 0;
  **loop** scan the priority queue for items ($i$, **white**) for which $B(i,x) = $ false displacing these items; next increment $x$ and wait for a while
**end** incriminator;
```

The critical section in the incriminator is the complete scan over the priority queue. Since at each moment during the computation the queue is finite, and since B is total the incriminator will never diverge within a critical section. Again the incriminator should leave time to the other processes for proceeding and accessing the queue.

```
**process** comp($y$);
compute testval by performing the call searchval($y$); next set $t(y) = $ testval and scan
the complete priority queue for items ($i$, **black**) for which $\Phi_i(y) \simeq$ testval, displacing
these items; after the scan terminate
**end** comp($y$);
```

The critical sections in comp(y) are the critical sections in searchval(y) and the entire scan over the priority queue. Again, the priority queue always being finite, and searchval already being shown to be correct in this regard, the process comp(y) will never diverge within a critical section.

process computation consists of running all processes comp(y) for $y = 0, 1, 2, \ldots$ in parallel.

The MCM-algorithm consists of running the three processes governor, incriminator and computation in parallel.

This completes our description of the MCM-algorithm. Its behaviour is analysed by considering the three possible behaviours of an item during the infinite computation. An item is called *unstable* provided it is displaced infinitely often, and it is called *stable* otherwise. A stable item is called *white stable* if it is a white item after its final displacement and *black stable* otherwise. The class of programs discriminated by B is denoted by $\mathscr{X}$ and the new name computed by the MCM-algorithm is denoted by t.

Case 1: Item A is unstable.

Let i denote the index of A. Since A is displaced infinitely often by the incriminator there are infinitely many values of x such that $B(i, x) =$ **false**. Consequently $\varphi_i \notin \mathscr{X}$. On the other hand A is displaced infinitely often by some comp(y) process. Since, for a given y, comp(y) will displace an item at most a single time this means that $\Phi_i(y) > t(y)$ for infinitely many arguments y. This shows that $\varphi_i \notin F(t)$.

Case 2: Item A is white stable.

Let i be the index of A. Since A becomes white stable it is displaced by the incriminator only finitely many times. Therefore $\varphi_i \in \mathscr{X}$. After having been displaced for the last time, it can occur only finitely many times that within a process comp(y) the procedure searchval stops at a black item preceding A (since such a black item is displaced subsequently). Consequently the value of val in searchval will be increased up to at least $\Phi_i(y)$ for almost all arguments y, and hence $\Phi_i(y) \leq t(y)$ almost everywhere. This shows that $\varphi_i \in F(t)$.

Case 3: Item A is black stable.

Let i be the index of A. Since A becomes black stable there are at most finitely many processes comp(y) displacing A and consequently $\Phi_i(y) \leq t(y)$ almost everywhere. On the other hand once having been displaced for the last time it will occur only finitely many times within a process comp(y) that the procedure searchval stops at a black item preceding A. Consequently for almost all y item A will be considered in searchval and be passed over for having a run-time $\Phi_i(y)$ not exceeding the current value of val, (i.e. not exceeding the maximum of the run-times of the preceding white items, all of which have (for almost all y) already become stable), or else searchval never gets past some preceding white item with infinite run-time, as val increases indefinitely. This shows that the run-time Φ_i is asymptotically bounded by the maximum of the run-times of a fixed finite set of programs contained in $\mathscr{X}$.

Gathering all evidence collected thus far we obtain:

Lemma 3.2. *If $\varphi_i \in F(t)$ then there exists a finite collection*

$$\{\varphi_{i_1}, \ldots, \varphi_{i_k}\} \subseteq \mathscr{X} \text{ such that } \forall^\infty x\ [\Phi_i(x) \leq \max_{l \leq k} \{\Phi_{i_l}(x)\}].$$

Theorem 3.3. *Let B be a total recursive discriminator for the Σ_2-class $\mathscr{X}$. Then the* MCM-*algorithm based on B computes a name t such that the class $F(t)$ satisfies $F(t) = \bigcap\{F(u) \mid \mathscr{X} \subseteq F(u)\}$, where u ranges over the class of all partial recursive functions; moreover $\mathscr{X} \subseteq F(t)$.*

The "measured set" property results from:

Corollary 3.4. *Let $(B_i)_{i=0}^\infty$ be a uniform recursive sequence of discriminators and let the* MCM-*algorithm based on B_i compute the name t_i. Then the sequence $(t_i)_{i=0}^\infty$ is a measured set.*

Proofs. Consider the MCM-algorithm based on B as described above. If $\varphi_i \in F(t)$ then by the analysis above the item A with index i is white or black stable. If we have case 2 (white stable) then Lemma 3.2 is satisfied trivially since $\Phi_i \leq \Phi_i$ and $\varphi_i \in \mathscr{X}$. If A is black stable then the analysis of case 3 above shows Lemma 3.2 holds. This proves Lemma 3.2.

To prove Theorem 3.3 we consider some program φ_i. If $\varphi_i \notin F(t)$ then by the above analysis its corresponding item is unstable and consequently $\varphi_i \notin \mathscr{X}$. This shows $\mathscr{X} \subseteq F(t)$. Next assume $\varphi_i \in F(t)$. By Lemma 3.2 there exists a finite set $\{\varphi_{i_1}, \ldots, \varphi_{i_k}\} \subseteq \mathscr{X}$ such that $\forall^\infty x[\Phi_i(x) \leq \max_{l \leq k}\{\Phi_{i_l}(x)\}]$. If $\mathscr{X} \subseteq F(u)$ then $\Phi_{i_l} \propto u$ for $l = 1, \ldots, k$ therefore $\lambda x[\max_{l \leq k}\{\Phi_{i_l}(x)\}] \propto u$ also. Since by Lemma 3.2 $\Phi_i \propto \lambda x\,[\max_{l \leq k}\{\Phi_{i_l}(x)\}]$ we derive $\Phi_i \propto u$ and therefore $\Phi_i \in F(u)$. This shows $F(t) \subseteq \bigcap\{F(u) \mid \mathscr{X} \subseteq F(u)\}$. The inclusion $\bigcap\{F(u) \mid \mathscr{X} \subseteq F(u)\} \subseteq F(t)$ is trivial since $\mathscr{X} \subseteq F(t)$ and since the function t is itself recursive. This proves Theorem 3.3.

There remains to prove Corollary 3.4. If $(B_i)_{i=0}^\infty$ is uniform sequence of discriminators we can design an MCM-algorithm based on the sequence $(B_i)_{i=0}^\infty$ using the value of i as an additional input parameter. This MCM-algorithm can be transformed into a decision procedure for $t_i(y) = z$ as follows:

To decide $t_i(y) = z$ the MCM-algorithm based on B_i is initiated and simulated up to the time where either comp(y) has halted or has increased the value of val in searchval(y) above z. In the first case output whether the result of searchval(y) equals z, and output **false** otherwise. This completes the proof. □

Theorem 3.3 shows that the class $F(t)$ can be described independently from the name t and that $F(t)$ is completely determined by the given Σ_2-class $\mathscr{X}$. We denote the class $F(t)$ in the sequel by $\bar{\mathscr{X}}$. Clearly if the collection $\mathscr{X}$ is already a complexity class we have $\mathscr{X} = \bar{\mathscr{X}}$; in particular $\bar{\bar{\mathscr{X}}} = \bar{\mathscr{X}}$. As a consequence Theorem 3.1 is seen to be a direct consequence of Theorem 3.3 and Corollary 3.4. It is also clear from Theorem 3.3 that for Σ_2-classes $\mathscr{X}$ and $\mathscr{Y}$ we have $\mathscr{X} \subseteq \mathscr{Y} \Rightarrow \bar{\mathscr{X}} \subseteq \bar{\mathscr{Y}}$; hence the operator $\mathscr{X} \to \bar{\mathscr{X}}$ acts as a closure operator. Theorem 3.3 also yields for Σ_2-classes $\overline{\mathscr{X} \cap \mathscr{Y}} \subseteq \bar{\mathscr{X}} \cap \bar{\mathscr{Y}}$. From this one obtains:

Corollary 3.5. *The intersection of two complexity classes $F(t) \cap F(u)$ is again a complexity class.*

Note that although $F(t) \cap F(u) = F(v)$ where $v = \lambda x\,[\min(t(x), u(x))]$ this yields no proof for the above corollary since the function v may be non-recursive in case t or u are partial.

Our next application deals with unions of an effective increasing sequence of complexity classes. By considering items of a slightly different format we can discriminate such a union, and by the fact that the sequence is increasing the class obtained using the MCM-algorithm turns out to be the union of the sequence itself.

Theorem 3.6. *Let $(t_i)_{i=0}^{\infty}$ be a uniform recursive sequence of partial recursive functions such that $F(t_i) \subseteq F(t_{i+1})$ for all i. Then $\bigcup_{i=0}^{\infty} F(t_i)$ is again a complexity class.*

For total names $(t_i)_{i=0}^{\infty}$ satisfying $t_i \leq t_{i+1}$ this result has already been proved by McCreight [8].

Proof. We consider an MCM-algorithm manipulating items indexed by pairs $\langle i, j\rangle$ consisting of a program-index i and an index j of a function in the sequence $(t_j)_{j=0}^{\infty}$. These pairs correspond to new programs $\varphi_{i,j}$ obtained by replacing each program φ_i by an infinite sequence $(\varphi_{i,j})_{j=0}^{\infty}$ all computing the same function and having the same run-time. Consequently in the process computation of the MCM-algorithm these copies behave equally. We can obtain however an Incriminator which treats these copies differently. Since for each index j the class $F(t_j)$ is a Σ_2-class a discriminator B_j for $F(t_j)$ can be obtained. Moreover the sequence $(t_j)_{j=0}^{\infty}$ being uniform the sequence $(B_j)_{j=0}^{\infty}$ is uniform as well. We now define the discriminator B by $B(\langle i, j\rangle, x) = B_j(i, x)$. The class $\mathscr{X}$ discriminated by B then becomes $\mathscr{X} = \{\varphi_{i,j} \mid \varphi_i \in F(t_j)\}$.

Let t_{inf} be the name computed by the MCM-algorithm based on B. If $\varphi_i \in F(t_j)$ then $\varphi_{i,j} \in \mathscr{X}$ and consequently the item with index $\langle i, j\rangle$ must stabilize and consequently $\Phi_{i,j} = \Phi_i \propto t_{\text{inf}}$. This shows that $\bigcup_{j=0}^{\infty} F(t_j) \subseteq F(t_{\text{inf}})$.

Conversely assume that $\varphi_i \in F(t_{\text{inf}})$. Select an arbitrary index k and consider the item with index $\langle i, k\rangle$. Since $\Phi_{i,k} = \Phi_i \propto t_{\text{inf}}$ this item must stabilize. If it becomes white stable then $\varphi_{i,k} \in \mathscr{X}$ and consequently

$$\varphi_i \in F(t_j) \subseteq \bigcup_{j=0}^{\infty} F(t_j).$$

If it becomes black stable then let $\{\langle i_1, k_1\rangle, \ldots, \langle i_s, k_s\rangle\}$ be the finite collection of white stable items preceding $\langle i, k\rangle$, and let $k_0 = \max_{j \leq s} k_j$. Then $\varphi_{i_j} \in F(t_{k_0})$ for $j \leq s$. Since the run-time $\Phi_i = \Phi_{i,k}$ is bounded by the pointwise maximum of the run-times $\Phi_{i_1}, \ldots, \Phi_{i_s}$ which in its turn is bounded by t_{k_0} we conclude that $\varphi_i \in F(t_{k_0}) \subseteq \bigcup_{j=0}^{\infty} F(t_j)$. This shows that $F(t_{\text{inf}}) \subseteq \bigcup_{j=0}^{\infty} F(t_j)$.

Having shown both inclusions we conclude that

$$F(t_{\text{inf}}) = \bigcup_{j=0}^{\infty} F(t_j). \quad \square$$

For a final application we first weaken our assumption that the discriminator B of the MCM-algorithm is recursive. Let B be an arbitrary total predicate. Then the function computed by a MCM-algorithm based on B no longer will be recursive but it will be recursive relative to B. However, Theorem 3.3 remains valid provided we let u range over the partial recursive functions relative B, and analogous relativized versions of Corollary 3.5 and Theorem 3.6 hold as well.

Corollary 3.7. *The intersection of an infinite sequence of complexity classes $\bigcap_{i=0}^{\infty} F(t_i)$ is a complexity class with a not necessarily recursive name.*

For uniform sequences of recursive functions this result has been obtained already by Robertson [11]. It is proved by using a (non-recursive) discriminator for the infinite intersection $\mathscr{X} = \bigcap_{i=0}^{\infty} F(t_i)$. Clearly

$$\mathscr{X} = \bar{\mathscr{X}}$$

since

$$\bigcap_{i=0}^{\infty} F(t_i) \supseteq \bigcap \left\{ F(u) \,\middle|\, \bigcap_{i=0}^{\infty} F(t_i) \subseteq F(u) \right\}.$$

4. A McCreight–Meyer algorithm for weak classes

The weak complexity classes were introduced in order to study the so-called honesty classes; cf. [3, 4] for example. If t is a recursive function and if we define $T(x, y) = t(x)$ then it turns out that the T-honest programs are exactly the programs contained in $F_w(t)$. Some existing theorems for complexity classes remain valid for weak classes and honesty classes as well, the Naming theorem being a surprising exception. In fact one has:

Theorem 4.1. *For every measured transformation σ there exists an index e such that $F_w(\varphi_e) \neq F_w(\varphi_{\sigma(e)})$.*

The proof can be found in [4].

It turns out, however, that by sacrificing the "measuredness" of the resulting new names, a McCreight–Meyer algorithm for weak classes, showing some of the "closure operator" properties discussed in the preceding section, can be designed. In particular some weaker generalizations of Theorem 3.3, Corollary 3.5 and Theorem 3.6 are given in Theorem 4.3, Corollary 4.4 and Corollary 4.5 respectively.

The new algorithm is obtained by modifying the procedures searchval and comp. The two problems we have to solve are:

(1) If searchval is to use a black item it has to make sure that its run-time is finite. This is solved as follows: instead of using a black item at the moment when its run-time is found to exceed the current value of val, this black item is only marked as a candidate and the computation proceeds onwards, increasing val occasionally, until one of the candidate black items turns out to have a run-time equal to val + 1; this later black item will be used.

(2) If searchval is attempting to respect a white item by increasing val to its run-time, this will never succeed if this run-time is infinite; however, an infinite run-time does not need to be respected at all!

It is the second problem which causes the troubles reflected in Theorem 4.1. Once a white item with an infinite run-time at all arguments gets itself installed in the front of the priority queue almost all subsequent computations of searchval will diverge.

In order to amend this failure we provide the algorithm with a so-called *wizard* predicting the divergence of "good" programs.

Definition 4.2. A *wizard* is a recursive predicate b such that for a given value x the predicate $\lambda i[b(i, x)]$ either is total or is undefined for all i (depending on x). The set $\{x \mid \lambda i[b(i, x)]$ is total$\}$ is called the *reach* of b and is denoted $\mathscr{D}'b$. If $\mathscr{X}$ is a Σ_2-class of programs and $Z \subseteq \mathbf{N}$ then we say that the wizard b is *justified for* $\mathscr{X}$ *on* Z provided

(1) $\mathscr{D}'b = Z$ and

(2) $\forall i\, \forall^{\infty} x\, [\varphi_i \in \mathscr{X}$ *and* $x \in Z \Rightarrow (\Phi_i(x) < \infty$ *iff* $b(i, x))]$.

The wizard has the function of predicting (within a reasonable tolerance) whether certain computations converge or not.

The procedure *weaksearchval* described below uses the wizard for disregarding white items which are suspected of diverging, increasing the value of val until one of the candidate black items is found to have a finite run-time.

```
Procedure weaksearchval;
  Input an argument $y \in \mathbf{N}$;
  Output a value for $t(y) \in \mathbf{N}$;
  Initialization: compute $b(0, y)$ in order to determine whether $y \in \mathscr{D}'b$; if this
      computation diverges weaksearchval diverges as well. Start with attempted
      valued val = 0 at the beginning of the priority queue.
      initialize the list of candidates as being empty.
  loop if the next item on the queue is $(i$, white$)$ with $b(i, y) =$ true stepwise increase
      val until $\Phi_i(y) \leq$ val, each time inspecting all indices $j$ on the list of candi-
      dates to see whether $\Phi_j(y) =$ val$+1$ and upon finding such an index $j$
      terminating with output val.
      once $\Phi_i(y) \leq$ val has been achieved proceed to the next item on the priority
      queue.
      if the next item on the queue is $(i$, black$)$ with $\Phi_i(y) >$ val insert index $i$ in the
      list of candidates and proceed to the next item on the priority queue.
      in all other circumstances proceed to the next item on the priority queue.
        If no more items are present stepwise increase val, inspecting the list of
      candidates as explained above.
end weak searchval;
```

The critical sections in weaksearchval are the test for whether there is an element on the queue and the acquisition of this element, and as before weaksearchval will never diverge within a critical section. Its computation is illustrated in Fig. 3, which has to be interpreted analogously to Fig. 2. At the bottom of Fig. 3 we have indicated the priority queue, the predictions of the wizard b on the behaviour of the white items and the list of canditates created during computation of weaksearchval.

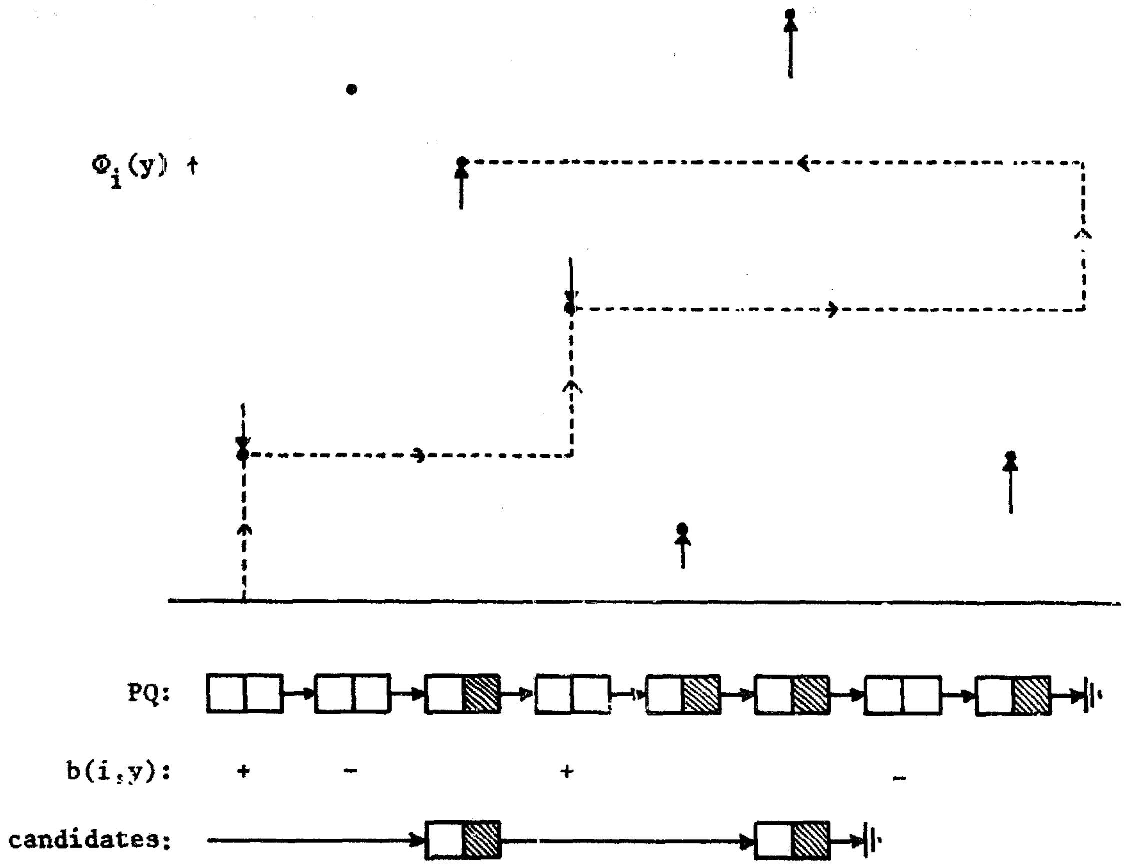

Fig. 3. The computation of weaksearchval.

After having described weaksearchval the weak MCM-algorithm is obtained as follows. We take the processes governor and incriminator from the MCM-algorithm of the preceding section. In the process computation we replace all processes comp(y) by processes weakcomp(y) described below. The weak MCM-algorithm consist of running the three processes in parallel.

```
process weakcomp(y);
  initialization: compute testval by performing the call weaksearchval(y); next set
      $f(y)$ = testval and scan the complete priority queue for items (i, black) for
      which $\Phi_i(y) >$ testval, gathering these indices on a list of candidates.
```

```
loop for $k = 1, 2, \ldots$ while the list of candidates is not empty test whether there
    exist an index $i$ on the list of candidates for which $\Phi_i(y) = \text{testval} + k$;
    displacing these corresponding items (provided they are still black) and
    removing them from the list of candidates.
      If the loop terminates by exhaustion of the list of candidates weakcomp($y$)
    terminates.
end weakcomp.
```

The critical sections in weakcomp are the scan over priority queue during the initialization, and the displacements of items during the loop, (together with the critical sections in weaksearchval). Again weakcomp will never diverge within a critical section.

Note that a single call of weakcomp(y) will displace a single item only once. Moreover, if weaksearchval has delivered a value there exists at least a single black item which will be displaced in the sequel of the computation of weakcomp (unless this item is displaced by a call of weakcomp(z) running in parallel).

We analyse the behaviour of the weak MCM-algorithm described above. Assume that the Σ_2-class of programs $\mathscr{X}$ is discriminated by B and let the wizard b be justified for $\mathscr{X}$ on $Z \subseteq \mathbf{N}$. The test at the beginning of weaksearchval(y) makes sure that $y \in \mathscr{D}'b$. Therefore we have $\mathscr{D}t \subseteq \mathscr{D}'b = Z$. On the other hand, knowing that $y \in \mathscr{D}'b$ we conclude that weaksearchval(y) terminates provided there exists a black item whose run-time at y is finite and exceeds the run-times of the preceding white items whose indices i satisfy $b(i, y) = \textbf{true}$. Actually weaksearchval(y) will locate the lowest run-time of a black item satisfying the above condition and weakcomp(y) will make sure that all black items satisfying this condition will be displaced.

Inspecting the behaviour of an item during the computation we again consider three cases:

Case 1: Item A is unstable.

If i denotes the index of A then as before $\varphi_i \notin \mathscr{X}$ and $t(y) < \Phi_i(y) < \infty$ for infinitely many $y \in Z$. Thus $\varphi_i \notin F_w(t)$.

Case 2: Item A is white stable.

If i denotes the index of A then as before $\varphi_i \in \mathscr{X}$. This implies that for almost all $x \in Z$, $b(i, x)$ *iff* $\varphi_i(x) < \infty$. Consequently for almost all $x \in Z$ weaksearchval(x) will either disregard item A or it will succeed in making val greater than $\Phi_i(x)$. Consequently the computation of weaksearchval is prevented from passing item A by divergence of $\Phi_i(x)$ for at most finitely many arguments. On the other hand, if weaksearchval terminates before passing A, a higher priority item will be displaced. As a consequence we see that for almost all arguments $x \in Z$ for which $\varphi_i(x) < \infty$ one has $\Phi_i(x) \leq t(x)$. Since moreover $\mathscr{D}t \subseteq Z$ this suffices to show that $\varphi_i \in F_w(t)$.

Case 3: **Item A is black stable.**

Let i denote the index of A. Since it is discovered only a finite number of times that $t(y) < \Phi_i(y) < \infty$ one has $\varphi_i \in F_w(t)$. After item A has been displaced for the last time, there are at most finitely many arguments z where weaksearchval(z) is blocked or terminates before accessing item A. For the remaining arguments $x \in Z$ either $\varphi_i(x)$ diverges or $\Phi_i(x)$ is bounded by the maximum of the run-times of the white items preceding A which are not disregarded by the wizard. Since the wizard is justified, with finitely many exceptions this maximum is finite. As in the proof of Theorem 3.3 we conclude that the program φ_i is contained within $F_w(u)$ for each u such that $\mathscr{D}u \subset Z$ and $\mathscr{X} \subseteq F_w(u)$.

Hence we have shown:

Theorem 4.3. *If b is justified for $\mathscr{X}$ on Z and if $\mathscr{X}$ is discriminated by B then the weak* MCM-*algorithm based on B and b, computes a name t satisfying*:

$$\mathscr{D}t \subseteq Z \quad \text{and} \quad F_w(t) = \bigcap\{F_w(u) \mid \mathscr{X} \subseteq F_w(u), \mathscr{D}u \subseteq Z\};$$

moreover

$$\mathscr{X} \subseteq F_w(t).$$

As applications we mention:

Corollary 4.4. *The intersection of two weak complexity classes with recursive names is again a weak complexity class with a recursive name.*

Corollary 4.5. *The increasing union of weak complexity classes with a uniform recursive sequence of total names is again a weak complexity class, with a recursive name.*

Proofs. It suffices to provide a wizard which is justified on a suitable domain.

For Corollary 4.4. let $\varphi_i = u$ and $\varphi_j = v$ be the (partial) names for two weak classes. Take $Z = \mathscr{D}u \cup \mathscr{D}v$ and let $w(x) =$ **if** $\Phi_i(x) \leq \Phi_j(x)$ **then** $u(x)$ **else** $v(x)$ **fi**, i.e. $w(x)$ is the first of the pair of values $u(x)$, $v(x)$ which is computed when the two are computed in parallel. Finally let $b(k, x) = (\Phi_k(x) \leq w(x))$, i.e. $b(k, x)$ diverges if $w(x)$ diverges. Then clearly $Z = \mathscr{D}'b = \mathscr{D}w = \mathscr{D}u \cup \mathscr{D}v$ and b is justified for $F_w(u) \cap F_w(v)$ on Z. The weak McCreight–Meyer algorithm based upon some discriminator for the Σ_2-class $F_w(u) \cap F_w(v)$ and b as defined above now computes a name t satisfying

$$F_w(u) \cap F_w(v) \subseteq F_w(t) = \bigcap\{F_w(v') \mid F_w(u) \cap F_w(v) \subseteq F_w(v'), \mathscr{D}v' \subseteq Z\}$$
$$\subseteq F_w(u) \cap F_w(v).$$

Hence

$$F_w(t) = F_w(u) \cap F_w(v).$$

For Corollary 4.5 let $(t_i)_{i=0}^{\infty}$ be the sequence of total names and let $\mathscr{X} = \bigcup_{i=0}^{\infty} F_w(t_i)$. As before in the proof of Theorem 3.6 we consider items whose indices are pairs i consisting of a program index $\pi_1 i$ and an index of a name $\pi_2 i$ in the sequence. The run-time of an index pair i is the run-time of its program $\Phi_{\pi_1 i}$. Let $u(x) = \max_{j \leq x} \{t_j(x)\}$. As wizard for $\mathscr{X}$ we take the predicate $b(i, x) = \Phi_{\pi_1 i}(x) \leq u(x))$. Clearly this wizard is justified for $\mathscr{X}$ on $\mathbf{N}$. The rest of the proof is left to the reader. □

Since honesty classes form a special type of weak classes these corollaries hold for honesty classes as well; cf. [3].

5. A McCreight–Meyer algorithm for anti-complexity classes

Since the anti-complexity classes are defined by reversing the order $\leq$ on the integers, it would suffice to perform the same order reversal within the part of the MCM-algorithm which reflects the structure of the classes considered, i.e. the processes comp(i). This would yield however a "largest-number" computation in searchval, a computation proceeding downwards from infinity to zero. This clearly is unfeasible. In order to choose a place to start the downward computations we need some additional information; this information is called an *á priori upper bound* (abbreviated apupb). We say that the total function h is an *á priori upper bound for the class $\mathscr{X}$* provided there exists a finite set of programs $\{\varphi_{i_1}, \ldots, \varphi_{i_k}\} \subseteq \mathscr{X}$ such that $\forall^{\infty} x\ [h(x) \geq \min_{l \leq k} [\Phi_{i_l}(x)]]$.

Clearly each run-time of a total program in $\mathscr{X}$ is an apupb for $\mathscr{X}$. Below we present a description of the procedure *antisearchval* and its calling routine *anti-comp*. Replacing comp by anticomp in the MCM-algorithm in Section 3 one obtains the *anti-McCreight–Meyer algorithm based upon the discriminator B and the á priori upper bound h*. The computation of antisearchval is illustrated in Fig. 4.

```
procedure antisearchval;
  Input an argument $y \in \mathbf{N}$;
  Output a value for $t(y) \in \mathbf{N}$;
  Initialization: Set val to $h(y)$ and start at the beginning of the priority queue.
  loop: if the next item on the queue is ($i$, white) stepwise decrease val until
      $\Phi_i(y) \geq$ val, and proceed to the next item on the queue; if the next item on
      the queue is ($i$, black) then terminate with output val if $\Phi_i(y) <$ val and
      proceed to the next item on the queue otherwise;
    if the queue is exhausted or val = 0 then terminate with output 0
```

```
end antisearchval.
process anticomp(y);
    compute testval by performing the call antisearchval(y); next set $t(y)$ = testval;
    if testval $\neq 0$ scan the complete priority queue for items (i, black) with $\Phi_i(y) <$
    testval, displacing these items. After the scan terminate.
end anticomp(y);
```

The critical sections in antisearchval and anticomp resemble those in searchval and comp in Section 3.

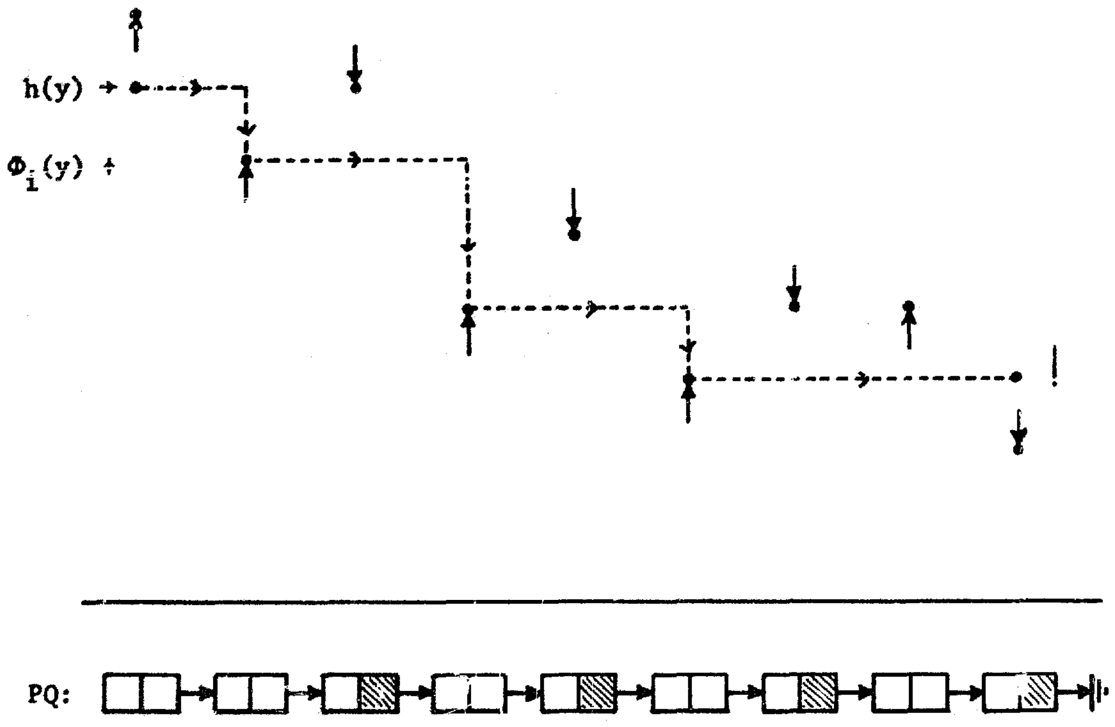

Fig. 4. The computation of antisearchval.

Again we consider the three possible behaviours of an item A with index i. The reasoning is the same as that for the classical MCM-algorithm with the order reversed at the right place. In Case 1 (A unstable) one has $\varphi_i \notin \mathscr{X}$ and $\varphi_i \notin A(t)$. In Case 2 (A white stable) one has $\varphi_i \in \mathscr{X}$ and due to the exhaustion of unstable higher priorities it is certain that antisearchval will proceed beyond item A almost everywhere, and $\Phi_i \gg t$ consequently; hence $\varphi_i \in A(t)$.

The interesting case is Case 3 (A black stable). As before $\varphi_i \in A(t)$. Moreover Φ_i is bounded from below almost everywhere by the pointwise minimum of h and a finite collection of run-times Φ_j of programs corresponding to white stable items. Assuming that h is an á priori upper bound for the class $\mathscr{X}$ it follows that Φ_i is bounded from below by the pointwise minimum of some larger but still finite class of run-times of programs contained in $\mathscr{X}$. As such $\varphi_i \in A(u)$ for each u such that $\mathscr{X} \subseteq A(u)$.

We therefore obtain:

Theorem 5.1. *Let h be an apupb for the class $\mathscr{X}$ which is discriminated by B. Then the anti-*MCM*-algorithm based on h and B computes a name t for the class $A(t)$ satisfying*

$$A(t) = \bigcap \{A(u) \mid \mathscr{X} \subset A(u)\}.$$

More specifically we have $\varphi_j \in A(t)$ provided Φ_j is bounded asymptotically from below by the pointwise minimum of the run-times of a finite collection of members of $\mathscr{X}$.

As corollaries one obtains results on intersections and increasing unions of anti-complexity classes. A more interesting corollary is Levin's greatest lower bound result which we discuss in the next section.

6. Greatest lower bounds of complexity sequences

Let f be a total function. The sequence of total recursive functions $(p_i)_{i=0}^{\infty}$ is called a *complexity sequence for f* provided the sequence $(p_i)_{i=0}^{\infty}$ is cofinal with the collection of run-times of f in the ordering $\backpropto$.

More explicitly:

$$\forall i\, \exists j\, [\varphi_j = f \text{ and } \Phi_j \propto p_i] \quad \text{and} \quad \forall i\, [\varphi_i = f \Rightarrow \exists j\, [p_j \propto \Phi_i]].$$

For more details and applications see [9 and 12].

Levin's result reads [7]:

Theorem 6.1. [Levin]: *If $(p_i)_{i=0}^{\infty}$ is a* r.e. *decreasing complexity sequence for f then there exists a recursive greatest lower bound t; i.e. if $\Phi_j \backpropto t$ then $\exists k\, [\Phi_j \backpropto p_k]$ and moreover $\forall k\, [p_k \backpropto t]$. Or equivalently $\forall i\, [\varphi_i = f \Rightarrow \Phi_i \backpropto t]$ and $\forall j\, [\Phi_j \backpropto t \Rightarrow \exists k\, [\varphi_k = f$ and $\Phi_j \backpropto \Phi_k]]$.*

Proof. Let $\mathscr{X}$ be the class of programs φ_i such that either $\Phi_i \backpropto p_j$ for some $j \in \mathbb{N}$, or equivalently $\Phi_i \backpropto \Phi_k$ for some index k such that $\varphi_k = f$. Since $(p_i)_{i=0}^{\infty}$ is a r.e. decreasing complexity sequence we obtain $\mathscr{X} = \bigcup_{i=0}^{\infty} A(p_i)$. Hence $\mathscr{X}$ is a Σ_2-class of programs. The function p_0 is an á priori upper bound for $\mathscr{X}$ since there exists an index j for f such that $\Phi_j \propto p_0$. Consider the name t computed by the anti-MCM algorithm based upon some discriminator for $\mathscr{X}$ and p_0. Then $\mathscr{X} \subseteq A(t)$ which indicates that t is a lower bound for the run-times of programs in $\mathscr{X}$; in particular $t \propto \Phi_j$ for each index j for f, so t is a lower bound. Conversely if $\varphi_j \in A(t)$ then Φ_j is bounded from below by the pointwise minimum of a finite collection of run-times of members of $\mathscr{X}$. Since $(p_i)_{i=0}^{\infty}$ is decreasing this shows that $\Phi_j \infty p_k$ for some index k. Hence $A(t) \subseteq \mathscr{X}$, so t is a greatest lower bound.

We conclude that $\mathscr{X} = A(t)$ (thus proving in fact the union theorem[1] for anti-complexity classes). This completes the proof. $\square$

The restriction that the complexity sequence $(p_i)_{i=0}^{\infty}$ is decreasing can be satisfied trivially for the Turing tape measure, or any other measure for which the parallel computation axiom holds [6]. Moreover it is known that functions with a sufficiently large speed-up have a decreasing complexity sequence, see e.g. [12].

Acknowledgements

I am indebted to A. Meyer and two anonymous referees for useful suggestions and textual improvements; and to P. Gács for providing me with the original proof of th. 6.1.

References

[1] L. Bass and P. Young, Ordinal hierarchies and naming complexity classes, *J. Assoc. Comput. Mach.* **19** (1972) 158–174.
[2] M. Blum, A machine—independent theory of the complexity of recursive functions, *J. Assoc. Comput. Mach.* **14** (1967) 322–336.
[3] P. van Emde Boas, Abstract resource-bound classes, Ph.D. Thesis, Math. Center Amsterdam (September 1974).
[4] P. van Emde Boas, The non-renameability of honesty classes, *Computing* **14** (1975) 183–193.
[5] J. Hartmanis and J. E. Hopcroft, An overview of the theory of computational complexity, *J. Assoc. Comput. Mach.* **18** (1971) 444–475.
[6] L. H. Landweber and E. L. Robertson, Recursive properties of abstract complexity classes, *J. Assoc. Comput. Mach.* **19** (1972) 296–308.
[7] L. A. Levin, On storage capacity for algorithms, *Soviet Math. Dokl.* **14** (1973) 1464–1466.
[8] E. M. McCreight and A. Meyer, Classes of computable functions defined by bounds on computation, *First ACM Symp. Theory of Computing* (1969) 79–88.
[9] A. R. Meyer and P. C. Fischer, Computational speed-up by effective operators, *J. Symbolic Logic* **37** (1972) 55–68.
[10] R. Moll and A. R. Meyer, Honest bounds for complexity classes of recursive functions, *J. Symbolic Logic* **39** (1974) 127–138.
[11] E. L. Robertson, Properties of complexity classes and sets in abstract complexity theory, Ph.D. Thesis University of Wisconsin (1970).
[12] C. P. Schnorr and G. Stumpf, A characterization of complexity sequences, *Z. Math. Logic Grundlagen Math.* **21** (1975) 47–56.

[1] Recently we obtained a copy of Levin's proof (in Russian); this proof actually resembles the proof of the Union Theorem in [8].