stichting
mathematisch
centrum

MC

AFDELING ZUIVERE WISKUNDE
(DEPARTMENT OF PURE MATHEMATICS)

ZW 79/76 JULY

P. VAN EMDE BOAS

SOME APPLICATIONS OF THE MEYER-McCREIGHT ALGORITHM IN ABSTRACT COMPLEXITY THEORY

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

AMS(MOS) subject classification scheme (1970): 68A20, 02F20, 02F35

SOME APPLICATIONS OF THE MEYER-McCREIGHT ALGORITHM IN ABSTRACT COMPLEXITY THEORY

by

P. VAN EMDE BOAS

ABSTRACT

The Meyer-McCreight 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", obtaining various known and new results as corollaries of more general assertions.

KEY WORDS & PHRASES: *Naming theorem, Honesty theorem, Measured set, Meyer-McCreight algorithm, complexity classes, Hierarchies, Blum measure, complexity sequence.*

1. INTRODUCTION

The Meyer-McCreight 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 $\varphi_i(x) = y$ is not). 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 & HOPCROFT [5]. Applications of the Naming theorem for the construction of hierarchies over the ordinals are given by BASS & YOUNG [1].

Descriptions of the Meyer-McCreight algorithm have been published at many places; see e.g. MEYER & McCREIGHT [7], BASS & YOUNG [1], HARTMANIS & HOPCROFT [5], MEYER & MOLL [9], or the present author [3]. In the present paper we present a version using full parallellism 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 a test only finitely many times. This indicates that instead of renaming complexity classes the algorithm might 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 [7] gives the example of a union of two complexity classes which itself is not a complexity class.

It turns out that the Meyer-McCreight 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, 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 Meyer-McCreight 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 the name of the class as a lower bound of 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 complexisty sequence of a total recursive function has a greatest lowerbound [6].

Algorithms in this paper are described using an ALGOL 68 - like formalism, which should be self-explanatory. For more details on this formalism the reader is referred to [3].

2. NOTATIONS AND DEFINITIONS

We use a fixed recursive pairing function $\langle x,y \rangle$ with coordinate inverses π_1 and π_2. Let $((\varphi_i)_i)$, $(\Phi_i)_i)$ denote a complexity measure, cf BLUM [2]. This means that $(\varphi_i)_i$ is an *acceptable Gödelnumbering* and φ_i and Φ_i have equal domain for each i; finally 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 unless explicitly stated to be total. The *domain* of a function f is denoted $\mathcal{D}f$, and we write $f(x) < \infty$ ($f(x) = \infty$) for $x \in \mathcal{D}f$ ($x \notin \mathcal{D}f$). We say that a function f is computed by φ_i or equivalently i is an *index* for f provided $f = \varphi_i$ extensionally, i.e.,

$\forall x\,[f(x) = \varphi_i(x)]$. The inequality $f \leq g$ means that $\mathcal{D}g \subseteq \mathcal{D}f$ and $g(x) \geq f(x)$ for all $x \in \mathcal{D}g$.

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

A *transformation of programs* σ is a total recursive function operating on the indices of programs; mostly such transformations result from application of the s-m-n axiom or the Recursion theorem.

If R is a two-variable function we say that φ_i is R-*honest* provided $\overset{\infty}{\forall}x[\Phi_i(x) \leq R(x, \varphi_i(x))]$, the condition holding vacuously for $x \notin \mathcal{D}\varphi_i$. A sequence of functions $(\gamma_i)_i$ 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 $(\varphi_{\sigma(i)})_i$ 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 [7,9].

A Σ_2-*class of programs* is a set of programs $X = \{\varphi_i \mid i \in A\}$ where A is a Σ_2-class in the usual sense; consequently the set A can be presented as $\{i \mid \overset{\infty}{\forall}x[B(i,x)]\}$ where B is a total recursive predicate called the *discriminator* of X.

For partial 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*
$F_w(t)$	$= \{\varphi_i \mid \overset{\infty}{\forall}x[\Phi_i(x) \leq t(x) \text{ or } \varphi_i(x) = \infty]\}$	*the weak complexity class*
$A(t)$	$= \{\varphi_i \mid \overset{\infty}{\forall}x[x \in \mathcal{D}t \Rightarrow \Phi_i(x) \geq t(x)]\}$	*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. Complexity classes have been studied from the start of abstract complexity theory, cf [1,2,5,7,10]. Weak complexity classes were introduced by the present author for the analysis of the behaviour of honesty classes, cf [3] & [4]. The anti-complexity classes are introduced here for the first time in order to obtain as a corollary a recent result by L.A. LEVIN on the greatest lowerbound of complexity sequences [6].

3. THE CLASSICAL MEYER-McCREIGHT ALGORITHM

The *Meyer-McCreight algorithm* (called the MMC algorithm hereafter) was introduced in order to prove the following theorem [7]:

THEOREM 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 executes during a dovetailed computation 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, this way enforcing the stable "good" programs to be respected in the limit.

The resulting names $(\varphi_{\sigma(i)})_i$ 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:

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

This shows that for each i the predicate B_i is total and consequently $\varphi_j \in F(\varphi_i)$ iff $\overset{\infty}{\forall}_x[B_i(j,x)]$.

Hence $F(\varphi_i)$ is nothing but the Σ_2-class of programs discriminated by B_i. It makes sense therefore to consider the behaviour of the MMC 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 parallellism (leaving the transformation to a dovetailed computation to the underlying operating system).

The *Meyer-McCreight algorithm based on* B consists of three modules 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* item 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 module is the so-called *governor*. Its task is to introduce all indices into the priority queue. The second module is called the *discriminator*. It is the unique module 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 module computation turns out to consist of an infinite sequence of processes running in parallel, called *comp(i)*. The process comp(i) tries to compute a value for the new name t at argument i, and after succeeding to provide such a value, it tests all black items on the priority queue against it, displacing those black indices which fail this test (which are punished this way). The processes comp(i) 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*. This later procedure reflects the structure of the complexity classes; MMC algorithms for weak complexity classes and anti-complexity classes are obtained by modifying this procedure searchval.

In diagram 1 we illustrate the modular decomposition explained above. All accesses to the priority queue occur within critical sections, i.e., if one module is performing a scan over the priority queue, or accessing a single item on it it does so undisturbed by other processes.

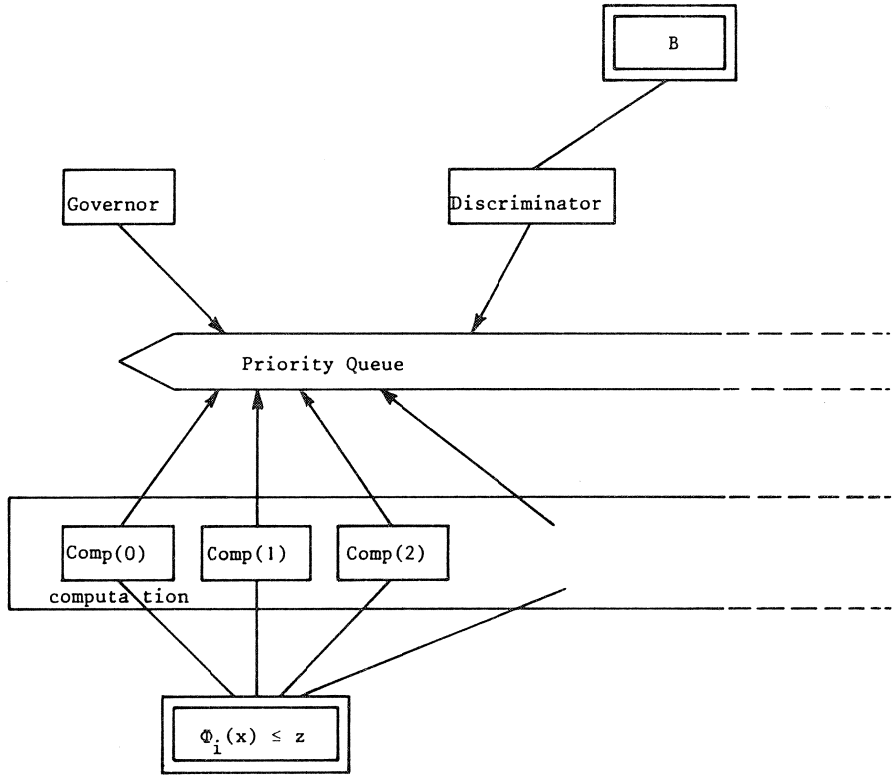

Diagram 1. Modular decomposition of the MMC-algorithm.

The crucial procedure within the MMC 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, white) which have higher priority one has $\Phi_i(y) \leq z$; moreover the black item used this way should have the highest possible priority. Below we present a program for searchval; its behaviour is illustrated in diagram 2.

```
proc searchval = (int y) int:

   (int val := 0; item cand := head priorityqueue;
   do    if cand = nil then val +:= 1
       elif colour of cand = white

         then while Φ index of cand(y) > val do val +:= 1 od;
              cand := succ cand

       elif Φ index of cand(y) ≤ val
         then cand := succ cand
```

```
        else goto ready
        fi
od ; ready : val
) # searchval #
```

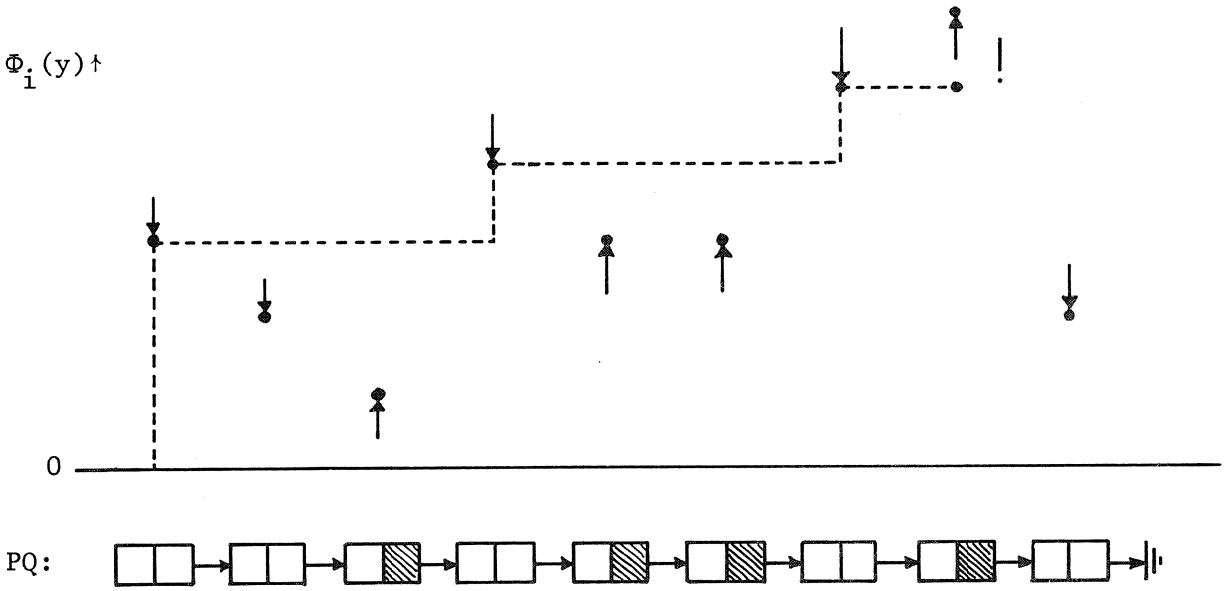

Diagram 2. The computation of searchval.

From diagram 2 and the program for 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 searchtime 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) with fail to terminite. In fact we have closed the priority queue by a symbolic white item (nil) with infinite run-time, this way preventing exhaustion of the priority queue.

The three modules and the MMC algorithm are suggested by the programs below:

```
proc governor = void:
   (int i:= -1;
      do append ((i+:= 1, white), priorityqueue);
            wastetime
      od);
proc discriminator = void:
      (int x:= 0;
       do for item over priorityqueue
          do if colour of item = white and
                B(index of item, x) = false
          then displace (item)
          fi
          od;
          x+:=1
       od);

proc comp = (int arg) void:
     (int testval := searchval (arg); t[arg] := testval;

       for item over priorityqueue
       do if colour of item = black and
            Φ index of item(arg) > testval
          then displace (item)
          fi
       od);

proc computation = (int arg) void:

        par begin comp (arg), computation (arg+1) end;

#MMC-algorithm#

par begin governor, discriminator, computation(0) end
```

This completes our description of the MMC-algorithm. Its behaviour is analysed by considering the three possible behaviours of an item during the infinite computation. An item is called *instable* 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 X and the new name computed by the MMC algorithm is denoted by t.

Case 1: Item A is instable.

Let i denote the index of A. Since A is displaced infinitely often by the discriminator there are infinitely many values of x such that B(i,x) = false. Consequently $\varphi_i \notin X$. On the other hand A is displaced infinitely often by some comp(y) process. Since each process 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 discriminator only finitely many times. Therefore $\varphi_i \in X$. After having reached its stable position on the priority queue 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 upto 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 reached its stable black position it will occur only finitely many times withing a process comp(y) that the procedure searchval stops at black item preceding A. Consequently for almost all y item A will be considered in searchval and be rejected as having a run-time $\Phi_i(y)$ not exceeding the current value of val, which equals the maximum of the run-times of white items, which (for almost all y) have already become stable. 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 X.

Gathering all evidence collected thus far we obtain:

THEOREM 2. *Let* B *be a discriminator for the* Σ_2*-class* X. *Then the* MMC *algorithm based on* B *computes a name* t *such that the class* F(t) *satisfies* $F(t) = \cap\{F(u) \mid X \subseteq F(u)\}$.

Denoting the class F(t) by $\bar{X}$ we have

ASSERTION 3. $\varphi_j \in \bar{X}$ provided there exists a finite collection

$$\{\varphi_{i_1}, \ldots \varphi_{i_k}\} \subseteq X \text{ such that } \overset{\infty}{\forall} x[\Phi_j(x) \leq \max_{\ell \leq k} \{\Phi_{i_\ell}(x)\}]$$

The "measured set" property results from:

ASSERTION 4. Let $(B_i)_i$ be a uniform recursive sequence of discriminators and let the MMC algorithm based on B_i compute the name t_i. Then the sequence $(t_i)_i$ is a measured set.

PROOFS. First consider the MMC-algorithm based on B. If $\varphi_i \in X$ then the item with index i cannot become instable and therefore $\varphi_i \in F(t)$. This proves $X \subseteq F(t)$. Next let $\varphi_i \in F(t)$. Then the item with index i cannot become instable and therefore this item becomes white stable or black stable. In the first case Assertion 3 holds trivially since $\varphi_i \in X$, whereas for black stable items Assertion 3 is derived from the analysis of Case 3 above. Let $\{\varphi_{i_1}, \ldots, \varphi_{i_k}\} \subseteq X$ be the set of programs corresponding to the white stable items preceding the item with index i. If $X \subseteq F(u)$ then $\Phi_{i_\ell} \propto u$ for $\ell = 1, \ldots, k$ and therefore $\lambda x[\max_{\ell \leq 1} \{\Phi_{i_\ell}(x)\}] \propto u$ also. Since by Assertion 3 $\Phi_i \propto \lambda x[\max_{\ell \leq k} \{\Phi_{i_\ell}(x)\}]$ we derive $\Phi_i \propto u$ and therefore $\varphi_i \in F(u)$. This shows $F(t) \subseteq \cap\{F(u) \mid X \subseteq F(u)\}$. The inclusion $\cap\{F(u) \mid X \subseteq F(u)\} \subseteq F(t)$ is trivial since $X \subseteq F(t)$. There remains to prove Assertion 4. If $(B_i)_i$ is uniform sequence of discriminators we can design an MMC algorithm based on the sequence $(B_i)_i$ using the value of i as an additional input parameter. This MMC algorithm can be transformed into a decision procedure for $t_i(y) = z$ as follows:

To decide $t_i(y) = z$ the MMC 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 + 1. In the first case output whether the result of searchval(y) equals z, and output false otherwise. This completes the proof. □

Clearly if the collection X is already a complexity class we have $\bar{X} = X$; in particular $\bar{\bar{X}} = \bar{X}$. As a consequence Theorem 1 now becomes a corollary of Theorem 2 and Assertion 4. It is also clear from Theorem 2 that $X \subseteq Y$ implies $\bar{X} \subseteq \bar{Y}$; hence the operator $X \to \bar{X}$ acts as a closure operator. Theorem 2 also gives $\overline{X \cap Y} = \bar{X} \cap \bar{Y}$. From this one obtains:

COROLLARY 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. Our next corollary reads:

THEOREM 6. *The union of an increasing sequence of complexity classes* $\bigcup_{i=0}^{\infty} F(t_i)$ *is again a complexity class.*

For total names $(t_i)_i$ satisfying $t_i \leq t_{i+1}$ this result was proved already by McCREIGHT [7].

PROOF. We consider an MMC algorithm manipulating items whose indices form a pair consisting of a program-index i and an index j of a function in the sequence $(t_i)_i$. The discriminator $B(<i,j>,x)$ investigates whether $\Phi_i \propto t_j$. The run-time of an item with index $<i,j>$ equals the run-time Φ_i.

Let t_{inf} be the name computed by this MMC-algorithm. If $\varphi_i \in F(t_j)$ then the item with index $<i,j>$ must stabilize and consequently $\Phi_i \propto t_{inf}$. On the other hand if the item with index $<i,k>$ stabilizes it either becomes white stable (and then $\varphi_i \in F_{t_k}$) or it becomes black stable. Let $\{<i_1,k_1>,\ldots,<i_s,k_s>\}$ be the finite collection of indices of white stable indices preceding $<i,k>$, 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 Φ_i 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 $\varphi_i \in F(t_{k_0})$. This shows that $F(t_{inf}) = \bigcup_{i=0}^{\infty} F(t_i)$. □

As an application where non-recursive names occur I mention:

COROLLARY 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.*

This result was obtained already by E.L. ROBERTSON [10]. It is proved by using a (non-recursive) discriminator for the infinite intersection $X = \bigcap_{i=0}^{\infty} F(t_i)$. Clearly $X = \bar{X}$ since $\bigcap_{i=0}^{\infty} F(t_i) \supseteq \cap\{F(u) \mid \bigcap_{i=0}^{\infty} F(t_i) \subseteq F(u)\}$.

4. A MEYER-McCREIGHT 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 8. *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 Meyer-McCreight algorithm for weak classes can be designed. In order to obtain such an algorithm we reconsider the computations of the procedure searchval. The two modifications one has to perform are:

1) If searchval is to use a black item it has to make sure that its run-time, is finite. This is an easy modification; instead of using a black item, when its run-time is found to exceed the current value of val, this black item is marked as a candidate and the computation proceeds up to the time where one of the candidate - black items turns out to have a run-time equal to val + 1.

2) If searchval is attempting to respect a white item by increasing val to its run-time, this run-time better should be finite.

It is the second modification which causes the troubles reflected in Theorem 8. Once a white item with an infinite run-time at all arguments gets itself installed in the front of the priority queue about 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 9. A *wizard* is a predicate b such that for a given value x the predicate $\lambda i[b(i,x)]$ either is total or otherwise undefined for all i (depending on x). The set $\{x \mid \lambda i[b(i,x)] \text{ is total}\}$ is called the *reach* of b and is denoted $\mathcal{D}'b$. If X is a $\dot{\Sigma}_2$-class of programs and $Z \subseteq \mathbb{N}$ then we say that the wizard b is *justified for* X *on* Z provided

1) $\mathcal{D}'b = Z$ and
2) $\forall i \overset{\infty}{\forall} x[\varphi_i \in X \text{ and } x \in Z \Rightarrow (\Phi_i(x) < \infty \text{ iff } b(i,x))]$.

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 to diverge, increasing the value of val until one of the candidate black items is found to have a finite run-time. Its computations are illustrated in diagram 3.

```
proc weaksearchval = (int y) int:
    (bool shalliproceed = b(0,y); # if this succeeds then y ∈ Db'#
     int val := 0; item cand := head priorityqueue;
                   itemlist good := nil;
     proc increaseval = void:

        (for blackit over good do
             if Φ index of blackit(y) = val + 1
             then goto ready fi od;
             val +:= 1) # increaseval #

     do if cand = nil then increaseval
       elif colour of cand = white
            then if b(index of cand,y)
            then while Φ index of cand(y) > val do increaseval od fi;
               cand := succ cand

       elif Φ index of cand(y) ≤ val then cand := succ cand
       else apend (cand, good); cand := succ cand
       fi
    od; ready: val
    )# weaksearchval #.
```

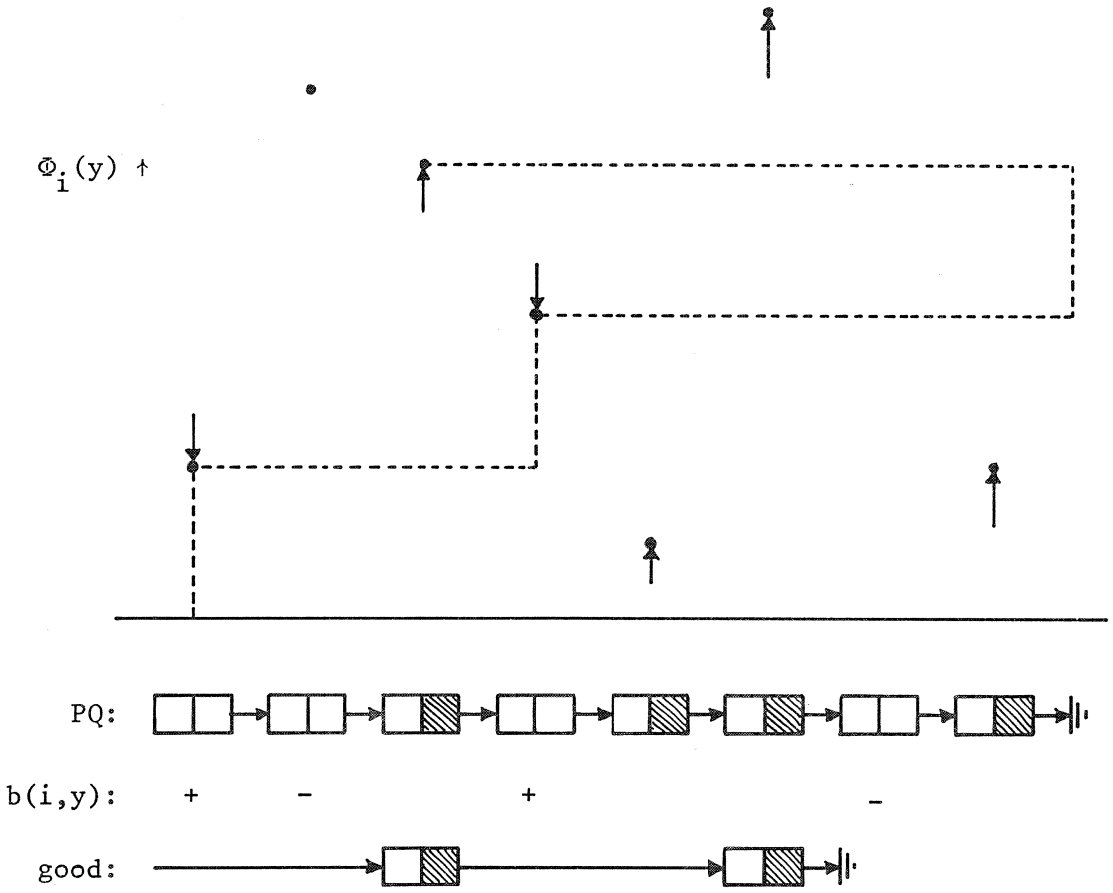

Diagram 3. The computation of weak search val

The *weak MMC algorithm* is obtained by replacing the procedure comp by

```
proc weakcomp = (int arg) void :
        (int testval := weaksearchval (arg); t[arg] := testval;
         for val from testval + 1 do
            for item over priorityqueue
              do if colour of item = black and
                    Φ index of item(arg) = val
                 then displace (item)
```

```
            fi
        od                  od
    )# weakcomp #.
```

A black item is displaced when its finite run-time is found to exceed the value of the new name t. Note that an item is displaced at most a single time by a single call of weakcomp.

Leaving the rest of the algorithm the way it was we investigate its behaviour. Assume that the Σ_2-class of programs X is discriminated by B and let the wizard b be justified for X on $Z \subseteq \mathbb{N}$. The test bool shall i proceed = b(0,y) at the beginning of weaksearchval makes sure that $y \in \mathcal{D}'b$.

Therefore we have $\mathcal{D}t \subseteq \mathcal{D}'b = Z$. On the other hand knowing that $y \in \mathcal{D}'b$ we conclude that weaksearchval terminates provided there exists a black item whose run-time at y is finite and exceeds the run-times of the white items whose indices i satisfy b(i,y) = true. Actually weaksearchval 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 instable.

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

Case 2: Item A is white stable.

If i denotes the index of A then as before $\varphi_i \in X$. 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 blocked by item A for at most finitely many arguments.

If weak search val terminates before accessing 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 $\mathcal{D}t \subseteq Z$ 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 stabilized there are at most finitely many arguments z where weaksearch-val (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. We conclude that the program φ_i is contained within $F_w(u)$ for each u such that $\mathcal{D}u \subseteq Z$ and $X \subseteq F_w(u)$.

Hence we have shown:

THEOREM 10. *If* b *is justified for* X *on* A *and if* X *is discriminated by* B *then the weak* MMC *algorithm based on* B *and* b, *computes a name* t *satisfying:*
1) $X \subseteq F_w(u)$ *and*
2) *if* $X \subseteq F_w(u)$ *and* $\mathcal{D}u \subset A$ *then* $F_w(t) \subseteq F_w(u)$

As applications we mention:

COROLLARY 11. *The intersection of weak complexity classes is again a weak complexity class.*

COROLLARY 12. *The increasing union of weak complexity classes with total names is again a weak complexity class.*

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

For Corollary 11 let $\varphi_i = u$ and $\varphi_j = v$ be the (partial) names for two weak classes. Take $Z = \mathcal{D}u \cup \mathcal{D}v$ 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) = $= (\Phi_k(x) \leq w(x))$. Then clearly $Z = \mathcal{D}b' = \mathcal{D}w = \mathcal{D}u \cup \mathcal{D}v$ and b is justified for $F_w(t) \cap F_w(u)$ on Z. The weak Meyer-McCreight 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 $F_w(t) = F_w(u) \cap F_w(v)$.

For Corollary 12 let $(t_i)_i$ be the sequence of total names and let $X = \bigcup_{i=0}^{\infty} F_w(t_i)$. As before in the proof of Theorem 6 we consider items whose indices are pairs consisting of a program index and an index of a name in the sequence. The run-time of an index is the run-time of its program. Let

$u(x) = \max_{i \le x} \{t_i(x)\}$. As wizard for X we take the predicate $b(i,x) = (\Phi_{\pi_1 i}(x) \le u(x))$. Clearly this wizard is justified for X on $\mathbb{N}$. 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 MEYER-McCREIGHT ALGORITHM FOR ANTI-COMPLEXITY CLASSES

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

Clearly each run-time of a total program in X is an apupb for X.

Below we present a description of the procedure *antisearchval* and its calling routine *anticomp*. Replacing comp by anticomp one obtains the *anti-Meyer-McCrieght algorithm based upon the discriminator* B *and the á priori upperbound* h. The computation of antisearchval is illustrated in diagram 4.

```
proc antisearchval = (int y) int:

(int val := h(y); item cand := head priority queue;
 do if cand = nil then if val = 0 then goto ready else val -:= 1 fi
   elif colour of cand = white
       then while Φ index of cand(y) < val do val -:= 1 od;
            cand := succ cand

   elif Φ index of cand(y) ≥ val
       then cand := succ cand
```

```
   else goto ready
   fi
 od; ready : val
 )# antisearchval #;

proc anticomp (int arg) void:
    (int testval = antisearchval (arg); t[arg] := testval;

     for item over priorityqueue
     do if colour of item = black and
           Φ_index of item(arg) < testval
        then displace (item)
        fi
     od)# anticomp #;
```

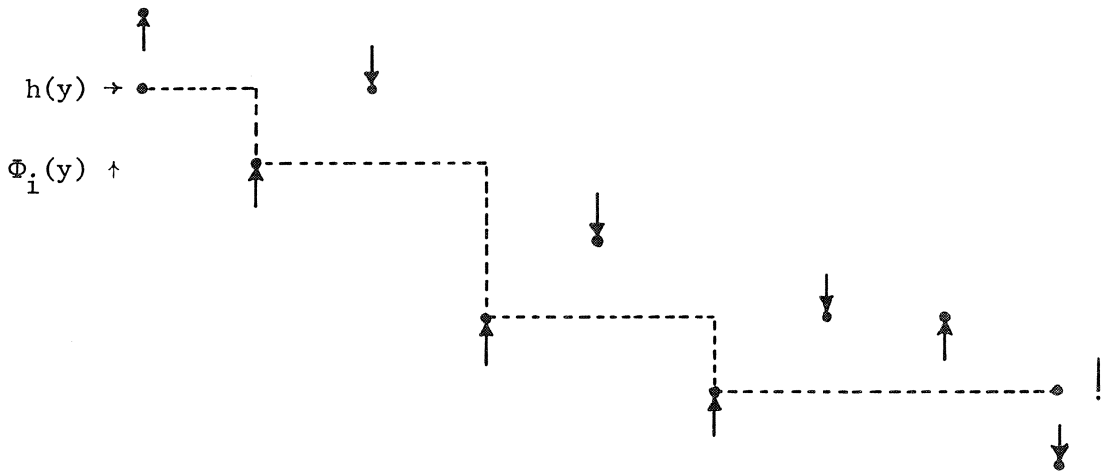

Diagram 4. The computation of antisearchval.

Again we consider the three possible behaviours of an item A with index i. The reasoning is exactly as for the classical MMC-algorithm with the order reversed at the right place. In Case 1 (A stable) one has $\varphi_i \notin X$ and

$\varphi_i \notin A(t)$. In Case 2 (A white stable) one has $\varphi_i \in X$ and due to the exhaustion of instable higher priorities it is certain that antisearchval will proceed beyond item A almost everywhere, and $\Phi_i \infty t$ 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 upperbound for the class 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 X. As such $\varphi_i \in A(u)$ for each u such that $X \subseteq A(u)$.

We therefore obtain:

THEOREM 13. *Let* h *be an* apupb *for the class* X *which is discriminated by* B. *Then the anti-*MMC *algorithm based on* h *and* B *computes a name* t *for the class* A(t) *satisfying*

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

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* X.

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

6. GREATEST LOWERBOUNDS OF COMPLEXITY SEQUENCES

Let f be a total function. The sequence $(p_i)_i$ is called a *complexity sequence for* f provided the sequence $(p_i)_i$ is co-final with the collection of run-times of f in the ordening ∞. More explicitly:

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

For more details and applications see [8] & [11].

LEVIN's result reads [6]:

THEOREM 14. [LEVIN]: *If* $(p_i)_i$ *is a decreasing complexity sequence for* f *then there exists a greatest lower bound* t; i.e. *if* $\Phi_j \gg t$ *then* $\exists k[\Phi_j \gg p_k]$ *and moreover* $\forall k[p_k \gg t]$. *Or equivalently* $\forall i[\varphi_i = f \Rightarrow \Phi_i \gg t]$ *and* $\forall j[\Phi_j \gg t \Rightarrow \exists k[\varphi_k = f$ *and* $\Phi_j \gg \Phi_k]]$.

PROOF. Let X be the class of programs φ_i such that either $\Phi_i \gg p_j$ for some $j \in \mathbb{N}$, or equivalently $\Phi_i \gg \Phi_k$ for some index k such that $\varphi_k = f$. Since $(p_i)_i$ is a decreasing complexity sequence we obtain $X = \bigcup_{i=0}^{\infty} A(p_i)$. Hence X is a Σ_2-class of programs. The function p_0 is an á priori upperbound for X since there exists an index j for f such that $\Phi_j \propto p_0$. Consider the name t computed by the anti-MMC algorithm based upon some discriminator for X and p_0. Then $X \subseteq A(t)$ which indicates that t is a lower bound for the run-times of programs in X ; in particular $t \propto \Phi_j$ for each index j for f. 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 X. Since $(p_i)_i$ is decreasing this shows that $\Phi_j \gg p_k$ for some index k. Hence $A(t) \subseteq X$.

We conclude that $X = A(t)$ (thus proving in fact the union theorem for anti-complexity classes). At the same time the above relation $\Phi_j \gg p_k$ implies the "hard" part of Levin's theorem. This completes the proof. □

The restriction that the complexity sequence $(p_i)_i$ is decreasing trivially can be satisfied for the Turing tape measure, or any other measure for which the parallel computation axiom holds. Moreover it is known that functions with a sufficiently large speed-up have a decreasing complexity sequence, see e.g. [11].

REFERENCES

[1] BASS, L. & P. YOUNG, *Ordinal hierarchies and naming complexity classes*, J. Assoc. Comput. Mach. 19 (1972) 158-174.

[2] BLUM, M., *A machine - independent theory of the complexity of recursive functions*, J. Assoc. Comput. Mach. 14 (1967) 322-336.

[3] EMDE BOAS, P. VAN, *Abstract resource-bound classes*, Ph.d. Thesis, Sept. 74 ed. Math. Center Amsterdam.

[4] EMDE BOAS, P. VAN, *The non-renameability of honesty classes*, Computing 14 (1975) 183-193.

[5] HARTMANIS, J. & J.E. HOPCROFT, *An overview of the theory of computational complexity*, J. Assoc. Comput. Mach. 18 (1971) 444-475.

[6] LEVIN, L.A., *On storage copacity for algorithms*, Soviet Math. Dokl. 14 (1973) 1464-1466.

[7] McCREIGHT, E.M. & A. MEYER, *Classes of computable functions defined by bounds on computation*, First ACM Symp. Th. of Computing (1969) 79-88.

[8] MEYER, A.R. & P.C. FISCHER, *Computational speed-up by effective operators*, J. Symbolic Logic 37 (1972) 55-68.

[9] MOLL, R. & A.R. MEYER, *Honest bounds for complexity classes of recursive functions*, J. Symb. Logic 39 127-138 (1974).

[10] ROBERTSON, E.L., *Properties of complexity classes and sets in abstract complexity theory*, Ph.D. thesis Univ. of Wisconsin (1970).

[11] SCHNORR, C.P. & G. Stumpf, *A characterization of complexity sequences*, Zeitschr. f. Math. Logik u. Grundlagen d. Math. 21 (1975) 47-56.