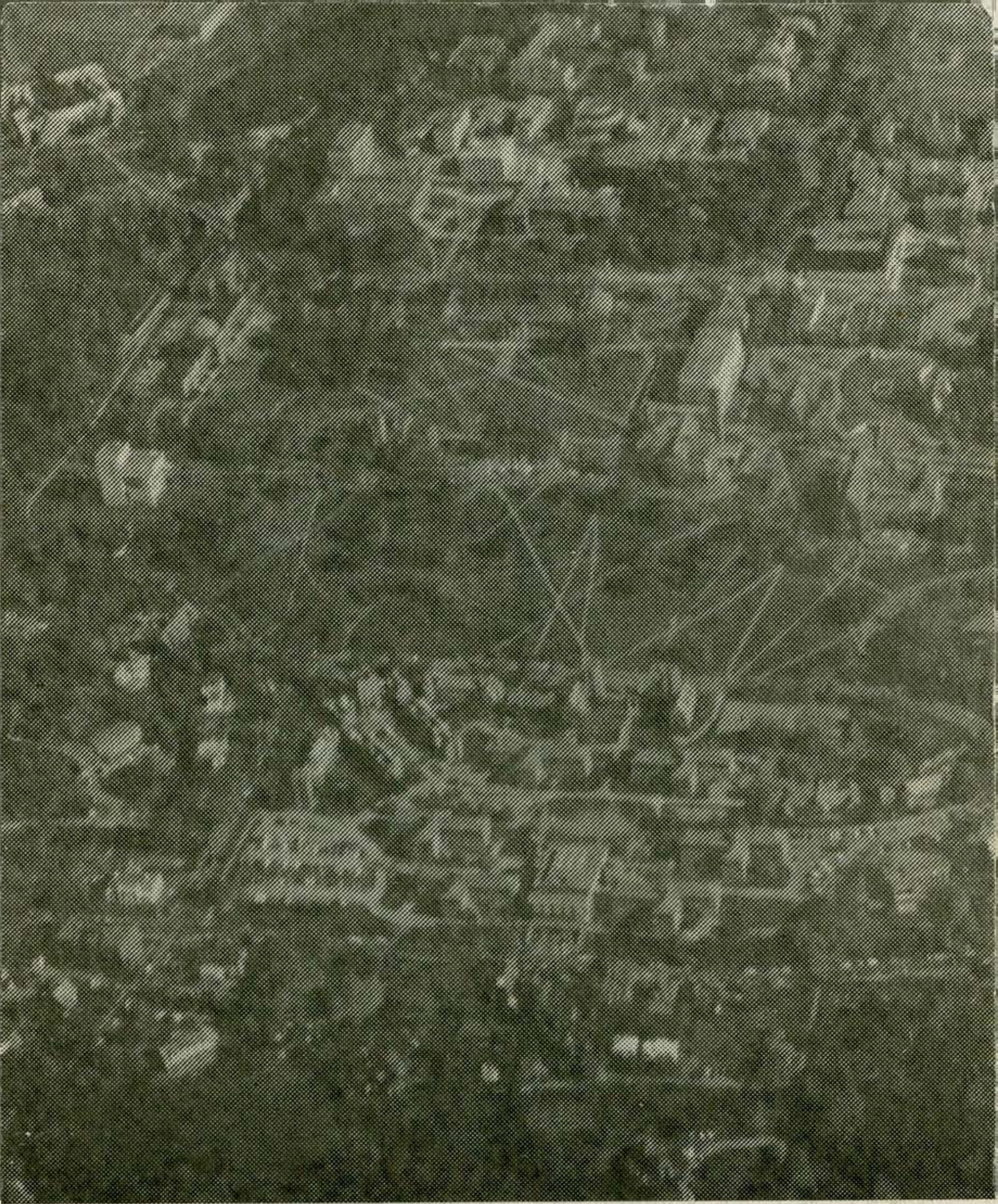


ABSTRACT RESOURCE-BOUND CLASSES

P. van Emde Boas

ABSTRACT RESOURCE-BOUND CLASSES

STELLINGEN

BEHORENDE BIJ HET PROEFSCHRIFT

ABSTRACT RESOURCE-BOUND CLASSES

VAN

PETER VAN EMDE BOAS

18 SEPTEMBER 1974

1

In iedere complexiteitsmaat kan iedere totale functie worden berekend met een programma met monotoon stijgende rekentijd.

Corollary 1.5.6, dit proefschrift.

2

De inbeddingsstelling, genoemd in 2.4.2, kan als volgt worden verscherpt: voor iedere complexiteitsmaat bestaat er een totale recursieve functie t met de eigenschap dat bij iedere recursieve partiele ordening $\leq$ op $\mathbb{N}$ een transformatie σ bestaat zodat aan de volgende voorwaarden is voldaan:

(i) $\sigma \in C_t$,

(ii) voor iedere $i \in \mathbb{N}$ geldt $\varphi_{\sigma(i)} \in F_t$,

(iii) voor iedere $i \in \mathbb{N}$ geldt $C_{\varphi_{\sigma(i)}} \subset C_t$,

(iv) voor iedere $i,j \in \mathbb{N}$ geldt $i \leq j \Leftrightarrow C_{\varphi_{\sigma(i)}} \subset C_{\varphi_{\sigma(j)}}$.

P. VAN EMDE BOAS, *Machine-independent complexity theory. Part 2, Resource-bound classes.* Math. Centre Tracts MCT 61 (in voorbereiding), Amsterdam, 1974.

3

De begrippen "opvraagbare rij" en "gelijkmatigheidsklasse" zijn, mede gezien voorbeeld 2.1.6, minder gelijkwaardig dan gesuggereerd wordt door de equivalentiestelling 2.1.5.

Hoofdstuk 2.1, dit proefschrift.

4

Een subbasis S voor een topologie heet *minimaal* indien iedere echte deelcollectie van S een echt zwakkere topologie genereert. Voor iedere metrizeerbare ruimte bestaat er een minimale subbasis voor de topologie.

P. VAN EMDE BOAS, *Minimally generated topologies.* Proc. conf. on Topology and appl. Herceg Novi 1968.

5

De door BALTHASAR ELIAS LUB aangegeven constructie van een aftelbare gegeneralizeerde rij die geen minimale subbasis voor de topologie toelaat, kan worden gebruikt om CW-complexen zonder minimale subbases te construeren.

B.E. LUB, *Sequences without minimal subbases.*
Rapport ZW 26/74, Mathematisch Centrum, Amsterdam.

6

Voor een eindige Abelse groep $G = \mathbb{Z}/m_1 \times \mathbb{Z}/m_2 \times \ldots \times \mathbb{Z}/m_k$, met $1 < m_1 | m_2 | \ldots | m_k$, definieren we de grootheden $\Lambda(G)$ en $\lambda(G)$ door: $\Lambda(G) = m_1 + m_2 + \ldots + m_k - k + 1$; $\lambda(G)$ = de maximale lengte van een rij elementen uit G met som nul die geen niet-triviale deelrij met som nul bevat. De gelijkheid $\Lambda(G) = \lambda(G)$ geldt o.m. in de volgende gevallen:

$k=3$, $m_1 = m_2 = 3$, $6 \times m_3$;
$k=3$, $m_i = 3 \times 2^{n_i}$;
$k=3$, $m_1 = 3$, $m_2 = 6 \times n \times d$, $m_3 = 6 \times n \times e$, waarbij n slechts factoren 2, 3,5, en 7 bevat en, hetzij d=1, hetzij d en e beiden een macht van eenzelfde priemgetal zijn.

P. VAN EMDE BOAS & D. KRUYSWIJK, *A combinatorial problem on finite Abelian groups III.*
Rapport ZW 08/69, Mathematisch Centrum, Amsterdam.

7

Bij de meeste behandelingen in de literatuur van de nergens differentieerbare functie van CELERIER: $f(X) = \sum_{n=0}^{\infty} a^{-n} \sin(a^n \pi X)$, wordt het geval dat a=2 niet bewezen. Ook in dit speciale geval is een elementair bewijs mogelijk.

P. VAN EMDE BOAS, *Nowhere differentiable contiuous functions, with an extended list of references.*
Rapport ZW 12/69, Mathematisch Centrum, Amsterdam.

8

In een relationele calculus voor de semantiek van programma-schema's laat de eigenschap dat p een ondeelbaar predicaat is zich karakterizeren door de axioma's $p \subset E$ en $p;U \cap U;p \subset p$.

W.P. DE ROEVER, *Operational, Mathematical and Axiomatized Semantics for Recursive Procedures and Data structures.*
Rapport ID 01/74, Mathematisch Centrum, Amsterdam.

9

De axiomatizering van de KLEENE standaard algebra's die is gegeven door J.H. CONWAY is onvolledig, tenzij bij impliciete conventie wordt aangenomen dat $\sum_{t}^{\{1\}} E_t = E_1$.

J.H. CONWAY, *Regular algebra and finite machines.*
Chapman & Hall, 1971.

10

De universele verzamelingenalgebra die is beschreven door A. MOSTOWSKI is ook in constructieve zin universeel: men kan, gegeven een recursieve partiele ordening, op effectieve wijze een inbedding van deze ordening in de MOSTOWSKI algebra construeren.

A. MOSTOWSKI, *Uber Gewisse Universelle Relationen.*
Ann. Soc. Polon. Math. 17 (1938) 117-118 ;
P. VAN EMDE BOAS, *Mostowski's universal set algebra.*
Rapport ZW 14/73, Mathematisch Centrum, Amsterdam.

11

De door J. VAN DE LUNE ingevoerde *Truncated-average limit* en de CESÀRO limiet zijn onafhankelijk.

P. VAN EMDE BOAS, *The truncated-average limit and the Cesàro limit are independent.*
Rapport ZW 21/74, Mathematisch Centrum, Amsterdam.

12

ACHMED probeert er achter te komen of een gerichte graaf zonder lussen op $n \geq 2$ punten, genummerd 1 t.e.m. n, die BALTHASAR in gedachte heeft, al dan niet een gerichte cykel bevat. Hiertoe mag ACHMED aan BALTHASAR vragen of er al dan niet een kant van i naar j loopt. Als ACHMED er niet uit komt alvorens alle $n(n-1)$ mogelijke kanten opgevraagd te hebben, wint BALTHASAR.
Indien BALTHASAR vals speelt, en zijn graaf opbouwt naar aanleiding van de vragen van ACHMED, met de bedoeling het hem moeilijk te maken, dan heeft BALTHASAR een gegarandeerde winst.

M.R. BEST, P. VAN EMDE BOAS & H.W. LENSTRA jr., *A sharpened version of the Aanderaa-Rosenberg conjecture.*
Rapport ZW 30/74, Mathematisch Centrum, Amsterdam.

13

Het assignment-axioma van HOARE is onhanteerbaar in situaties waarbij assignments aan herhaald geïndiceerde array elementen optreden zoals in a[a[1]] := a[a[2]].

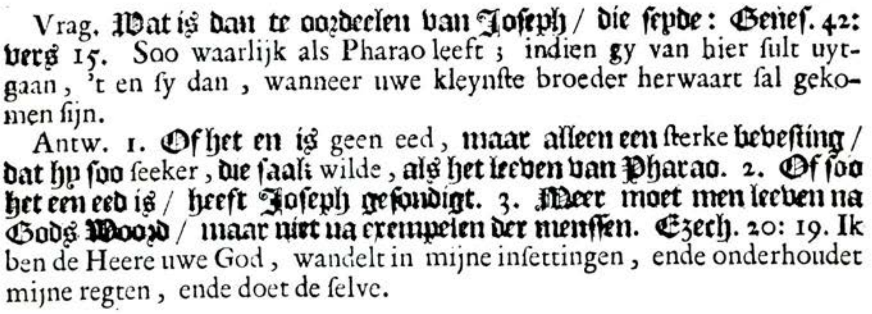
C.A.R. HOARE, *An axiomatic base for computer programming.*
Comm. Assoc. Comput. Mach., 12 (1969) 576-583.

14

Twee rechthoeken heten *onvergelijkbaar* indien het onmogelijk is door verschuiving en/of draaiing over 90° de ene rechthoek tot deelfiguur van de andere te maken. Er bestaan rechthoeken met gehele zijden die niet-triviale decomposities in onderling onvergelijkbare deelrechthoeken met gehele zijden toelaten. De kleinste oplossing bestaat uit zeven deelrechthoeken.

Elementary problems and solutions E 2422 [1973,691].
Amer. math. Monthly, 81 (1974) 664-666.

In de uitgave van de Heidelberger Katechismus, verzorgd door DAVID KNIBBE, treffen wij bij de katechizatie over de 102e vraag (37e Zondag) het volgende betoog aan.



Vrag. Wat is dan te oordeelen van Joſeph / die ſeyde: Geneſ. 42: vers 15. Soo waarlijk als Pharao leeft ; indien gy van hier ſult uyt-gaan , 't en ſy dan , wanneer uwe kleynſte broeder herwaart ſal gekomen ſijn.

Antw. 1. Of het en is geen eed , maar alleen een ſterke beveſting / dat hy ſoo ſeeker , die ſaak wilde , als het leeven van Pharao. 2. Of ſoo het een eed is / heeft Joſeph geſondigt. 3. Meer moet men leeven na Gods Woord / maar niet na exempelen der menſſen. Ezech. 20: 19. Ik ben de Heere uwe God , wandelt in mijne inſettingen , ende onderhoudet mijne regten , ende doet de ſelve.

Deze gedachtengang vertoont een niet te ontkennen overeenkomst met de welbekende redenering van de gebroken pot, die 1. niet was geleend, 2. ten tijde van het uitlenen reeds was beschadigd, en 3. in geheel goede toestand was teruggegeven. In beide gevallen is er sprake van een bewijs op grond van niet eenvoudige implicaties. Kennelijk is dit type argumenten niet alleen voor beoefenaren van de kunstmatige intelligentie problematisch.

DE LEERE DER GEREFORMEERDE KERK, Volgens de order van de HEYDELBERGSE KATECHISMUS Verklaard, bevestigt, en tot oeffening der Godsaligheyd toegepast. Vermeerderd, verbeeterd, en voor yder Sondag met een ontleedende TAFEL, EN Een kort ONDERWIJS, om een PREDICATIE met order te hooren en te herhaalen, verrijkt.
DOOR DAVID KNIBBE, Bedienaar van het Goddelijk Woord, tot LEYDEN.
DEN TWEEDEN DRUK. TOT LEYDEN, By JORDAAN LUGTMANS, Boekverkooper, 1696. Met Privilegie.

16

De onoverzienbaarheid van de hedendaagse wiskunde blijkt onder andere uit het ervaringsfeit dat de gemiddelde wiskundige niet in staat is van alle rubrieken die voorkomen in het AMS MOS 70 classificatiesysteem, ook maar bij benadering te weten wat het desbetreffende onderwerp inhoudt. Het is dan ook ondenkbaar dat één persoon, zonder hulp van collega's van diverse richtingen, een systematische catalogus voor een wiskundebibliotheek op verantwoorde wijze kan opbouwen of bijhouden. Daarnaast is het dringend gewenst dat de auteurs van geavanceerde boeken en rapporten zelf voor een indeling volgens dit systeem zorg dragen.

AMS (MOS) Subject classification scheme (1970).
Math. Reviews, 39 (1970) A1-A42.

17

Het is strijdig met de culturele functie van de Universiteit, dat de Dienst Bouw en Huisvesting van de Universiteit van Amsterdam zich bij herhaling schuldig maakt aan vormen van actieve of passieve verwaarlozing en/of verwoesting van de haar toevertrouwde historische monumenten in de stad Amsterdam, op een wijze die men eerder verwacht van de dienaren van POENE BEURSKRAKER of de HEILIGE BUREAUCRATIUS.

Folia Civitatis 18 Mei 1974 en 8 Juni 1974.

18

Het uit de kelder van het Stedelijk Museum afkomstige, niet convexe kunstwerk van C. KORTLANG, dat de Universiteit van Amsterdam heeft opgehangen in de voormalige consistoriekamer, die zij tijdelijk in gebruik heeft als receptieruimte bij oraties en promoties, is noch qua oppervlakte, noch qua uitwerking op diegenen die in deze ruimte vertoeven, in staat het hiaat op te vullen dat ontstaan is na de verwijdering van het voorheen aldaar aanwezige Meesterwerk van A. VAN PELT (1815-1895), voorstellende MAARTEN LUTHER voor de Rijksdag te Worms (1521). De voor het weghalen verantwoordelijke Lutherse gemeente, die handelde op grond van een gerechtvaardigde vrees voor het immer oproerige studentenvolkje, treft in dezen geen blaam.

19

De Weg tot de Wetenschap voert langs een militair complex.

ABSTRACT RESOURCE-BOUND CLASSES

ACADEMISCH PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN
DOCTOR IN DE WISKUNDE EN NATUURWETENSCHAPPEN
AAN DE UNIVERSITEIT VAN AMSTERDAM
OP GEZAG VAN DE RECTOR MAGNIFICUS
DR. A. DE FROE
HOOGLERAAR IN DE FACULTEIT DER GENEESKUNDE
IN HET OPENBAAR TE VERDEDIGEN
IN DE AULA DER UNIVERSITEIT
(TIJDELIJK IN DE LUTHERSE KERK, INGANG SINGEL 411, HOEK SPUI)
OP WOENSDAG 18 SEPTEMBER 1974 DES NAMIDDAGS TE 1.30 UUR

DOOR

PETER VAN EMDE BOAS

GEBOREN TE AMSTERDAM

1974
MATHEMATISCH CENTRUM, AMSTERDAM

PROMOTOR	: PROF.DR.IR. A. VAN WIJNGAARDEN
CO-PROMOTOR	: PROF.DR. P.C. BAAYEN
COREFERENT	: PROF.DR. J. HARTMANIS

CONTENTS

ABSTRACT

We present a survey on the theory of Resource-bound Classes in Abstract Complexity Theory. In particular the theory of Honesty Classes is developed in full analogy with the existing theory of Complexity Classes. Honesty Classes are defined by gathering (functions computable by) programs, whose run-times are bound almost everywhere by a function depending on the argument and the computed value. It is proved that several known theorems, like the Gap-, Operator Gap- and Union Theorem can be generalized to Honesty Classes, although the proofs need some non-trivial modifications. The Naming Theorem, however, is invalid for Honesty Classes.

The concept of an Acceptance Relation is developed, to explain these unexpected differences. In this framework a number of different types of Resource-bound Classes can be described. Moreover, it is argued that there exist two different ways in which a so called Abstract Resource-bound Class is restricted by its name, called a weak and a strong restriction. Since Complexity Classes are strong classes whereas Honesty Classes are weak, this explains the difference in behaviour of the two types of classes.

We introduce a new formalism to represent algorithms in recursion theory by mathematical expressions which do not differ much from the ones traditionally used in an informal way. In this way a number of ambiguities which are rooted in the lack of formalism are eliminated. In the appendix some of the more complicated algorithms used in Abstract Complexity Theory are represented in this formalism.

Keywords: Complexity Classes, Honesty Classes, Blum measure, Gap Theorem, Union Theorem, Naming Theorem, Honesty Procedures, Meyer-McCreight Algorithm, Acceptance Relation, measured set, Resource-bound Classes, Abstract Complexity Theory.

AMS MOS 70 classification: primary 68 A 20, 02 F 35, (68 M 15)
secondary 68 A 10, (68 K 99), 02 F 43.

Computing reviews classification: 5.25, 4.29, 5.29.

VOORWOORD VOOR DE LEEK.

Gegeven de grote waarde die de samenleving in het algemeen, en de subgroep van verwanten en vrienden van de promovendus in het bijzonder, toekent aan de promotie en de daarmee verband houdende gebeurtenissen, is het onvermijdelijk dat dit proefschrift ook onder ogen komt van niet wiskundig geschoolden. We moeten ernstig rekening houden met de mogelijkheid dat deze lezers reeds na het lezen van een drietal zinnen geheel overdonderd zijn. Tot overmaat van ramp is ook onder de bijgevoegde stellingen weinig leuks voor hen te vinden. Ik wil daarom, en tegelijk om eens en vooral af te rekenen met het fabeltje dat Wiskunde onbegrijpelijk is, in dit voorwoord een van de centrale onderdelen van dit proefschrift verwoorden in de volkstaal en het op deze wijze geheel en al verklaren en ook voor de leek begrijpelijk maken. Het betreft hier het stuk theorie, dat in paragraaf 3.4.4 op een meer traditionele wijze is behandeld.

Het is mij bovendien een groot genoegen in een jaar, waarin zovele gruwelverhalen de ronde doen over zekere prijsvormingsmechanismen in de wereldeconomie en over de daarvoor verantwoordelijk geachte duistere krachten, U op de hoogte te mogen brengen van een geheel nieuwe theorie over het economisch gebeuren. De feitelijke informatie, die in dit voorwoord verwerkt is, ontleen ik aan de verhandeling "Religious Principles and Oil Price Mechanisms; a Study on Socioeconomic Behaviour in Harad" van BALTHASAR E. LUB, werkzaam aan het Theologisch Seminarium van de Universiteit van Umbar. Voor de goede orde dien ik hierbij te melden dat dit geschrift het proefschrift is, waarop de auteur recentelijk summa cum laude de graad van doctor in de godgeleerdheid verwierf.

Voordat ik kan ingaan op de theorie van de jonge doctor, eerst enkele opmerkingen over Harad. Zoals U wellicht bekend zal zijn, telt dit land, volgens de laatste schattingen, een 20.000 zielen; hierbij dient te worden opgemerkt, dat er voorzover bekend, nog nooit een volkstelling is gehouden. Het land wordt sinds mensenheugenis geregeerd door een Iman. Aangezien er in het gehele land niets wil groeien, is de gehele bevolking werkzaam in de handel. Het zal U dan ook niet verbazen, dat tot ver in de twintigste eeuw Harad een straatarm land was; in deze toestand is pas wijziging gekomen na het aanboren van enkele oliebronnen.

De langdurige periode van armoede heeft het volk gelouterd, en men zal op deze wereld lang moeten zoeken om een volk te vinden, dat op vergelijkbare wijze de religieuze wetten naleeft. Dit houdt onder meer in, dat vrouwen en kinderen in Harad niets in te brengen hebben en in het vervolg van dit verhaal zullen we ons dan ook beperken tot de inwoners van het mannelijk geslacht, die de leeftijd van 20 jaren gepasseerd zijn en die we verder Haranen zullen noemen.

Een van de meest opvallende karakteristieken van Harad is de grote invloed van de godsdienstige wetten op het economisch handelen van de Haranen. Zo is het de Haraan verboden om tegelijkertijd als koper en als verkoper op te treden. Aangezien er in de practijk slechts één artikel is om in te handelen, te weten de reeds eerder genoemde olie, die iedere Haraan in onbeperkte hoeveelheden uit zijn achtertuintje haalt, kunnen we de Haranen gevoegelijk onderverdelen in kopers en verkopers. Andere bepalingen verbieden het de Haraan om vaker dan eens per jaar zijn bied- of laatprijs te herzien, of nog erger, uit de rangen van de verkopers toe te treden tot de groep der kopers (of omgekeerd). Slechts één keer per jaar, te weten op het feest van de Groote Verrekening, krijgt men de kans zijn prijs te herzien, terwijl de beslissing of men gedurende het komende jaar koper dan wel verkoper zal zijn, van geheel andere factoren afhangt, iets, waarop wij in het verdere betoog nader zullen ingaan. Overtreders van deze in de heilige boeken verankerde geboden, worden gestraft met verbanning naar een niet nader genoemde, doch vermoedelijk hoogst onaangename verblijfplaats; voorts worden hun bezittingen verdeeld onder het volk via de weg van plundering. Dit gebruik is overigens de oorzaak van het feit, dat de Internationale Concerns nooit enige invloed hebben weten te verwerven op de Haraanse economie.

De overgang van een Haraan van koper tot verkoper komt als volgt tot stand. Een koper die gewillig is de gedurende een jaar geldende olieprijs te betalen, zal al snel ontdekken, dat zijn beurs eerder is uitgeput dan de oliebron van zijn leveranciers. Na afloop van het jaar zal de man dan ook geheel berooid zijn en om hem niet geheel buiten spel te zetten, mag hij weer toetreden tot de rangen der verkopers.

Het is duidelijk, dat het niet erg aanlokkelijk is een jaar lang als koper te moeten optreden, gegeven het feit, dat men ŏf niet kan handelen, ŏf aan het eind van het jaar failliet is. De practijk leert dan ook, dat er geen Haranen te vinden zijn, die zich vrijwillig aanbieden om koper te worden. Om het economisch leven niet te verlammen, wordt daarom ieder jaar een aantal Haranen, die tijdens het voorafgaande jaar de ergernis van de Iman hebben opgewekt, tot koper gedegradeerd, met de verplichting koper te blijven, tot het faillissement er op volgt.

Aangezien alleen die kopers, die bereid zijn de vastgestelde prijs te betalen, failliet gaan, is het mogelijk door het hanteren van lage biedprijzen, zich een aantal jaren als koper te handhaven.

De Haraan beschouwt het als een grove belediging wanneer een ander hem rechtstreeks vraagt hoe hoog zijn bied- of laatprijs is; men mag slechts de tegenpartij een prijs noemen en vragen of hij voor deze prijs al dan niet bereid is te handelen. Aangezien reeds verscheidene toeristen hun onbekendheid met deze lichtgeraaktheid moesten bekopen met een messteek of een gebroken arm, moet ik de lezers - voorzover zij na het lezen van dit verhaal nog steeds van zins zijn om op korte termijn een bezoek aan dit mooie land te willen brengen - bezweren, toch vooral rekening te houden met dit gevoelige punt. De geleerden zijn het overigens niet eens over de oorsprong van dit gebruik. De vermaarde auteur ISMAEL B. MERODAK wijst op een passage in de heilige boeken, die enige overeenkomst vertoont met Genesis 18 Vs 17-33. Zijn opvattingen worden bestreden door de minder bekende socioloog HOSIA W. LABBERS jr., die zelf een theorie hanteert, uitgaande van een vervelingssyndroom. Ik wil echter geen verdere tijd verdoen aan deze voor ons niet ter zake doende kwestie en overgaan tot de bespreking van het ritueel waarmede de olieprijs jaarlijks wordt vastgesteld.

De Iman, als absoluut despoot, is de vaststeller van de jaarlijkse olieprijs. Hij is het immers die de handel in zijn land gaande moet houden. Aangezien de Haranen zelf vrij zijn in het kiezen van hun privé bied- en laatprijzen en het dus zeer wel denkbaar is dat de laagste biedprijs uitkomt beneden het niveau van de hoogste vraagprijs, is het de Iman niet mogelijk de verlangens van al zijn onderdanen tegelijkertijd te honoreren. De Iman is in de practijk reeds dan tevreden, als hij er in slaagt om één enkele transactie mogelijk te maken. Daarnaast is de Iman Zeer Rechtvaardig. Om te voorkomen dat slechts een deel van de Haranen bij de prijsvaststelling betrokken wordt, hanteert de Iman een hierarchie, waarin alle Haranen zijn opgenomen in volgorde van ancienniteit. Het is evenwel mogelijk om een Haraan zijn ancienniteit te ontnemen en in de practijk gebeurt dat zelfs regelmatig, n.l. iedere keer als de Haraan van status verandert. Het is duidelijk, dat iemand, die de toorn van de Iman opwekt, bij zijn degradatie tot koper tevens zijn ancienniteit kwijtraakt, maar ook op het faillissement staat het verlies van ancienniteit. In de loop der jaren zullen daarom die Haranen die niet van status veranderen, oprukken in de hierarchie. Laat ik tenslotte opmerken dat jongeren, die juist de leeftijd van twintig jaren gepasseerd zijn, onderaan in de hierarchie worden opgenomen, meestal als verkoper.

Ik wil bij de beschrijving van het feest der Groote Verrekening een aantal saillante,maar voor ons doel irrelevante onderdelen, onvermeld laten.

Na het uitspreken van een reeks gebeden door de Hogepriester en een rituele zuivering, volgt de inwijding der twintigjarigen die, zoals gezegd, onderaan in de hierarchie worden opgenomen. Hierna volgt de verbanning der onstandvastigen (maar dit onderdeel is de laatste jaren bij gebrek aan schuldigen niet meer opgevoerd).

Vervolgens houdt de Groot-discriminator een boetepreek, waarin hij allen, die de Iman gedurende het voorafgaande jaar onrecht hebben gedaan of anderszins hebben geërgerd, aanklaagt. De boosdoeners worden naar voren gesleurd, bespot en uitgejouwd; de kleren worden hun van het lijf getrokken en ten overstaan van de massa worden zij tot koper gedegradeerd. Het enige wat zij mogen behouden is hun beurs, want die zullen ze het jaar daarop hard nodig hebben.

Meestal wordt tegenwoordig de ceremonie op dit punt onderbroken voor het nuttigen van een bokaal wijn, maar dit is een verwatering van het oorspronkelijke ceremoneel, dat geen pauze kende.

Na de onderbreking stellen de Haranen zich op in de voorhof der tempel. De Iman betreedt het Heilige der Heiligen en zet zich op een daar speciaal voor dit doel ingerichte troon. Eenmaal gezeten spreekt hij een gebed uit waarin hij de Groote Rekenaar bidt om hem te inspireren tot het kiezen van een getal dat als beginprijs moet gaan dienen. In de practijk blijkt de keuze altijd het getal nul te zijn.

Vervolgens worden de Haranen een voor een tot de Iman toegelaten om te worden gehoord. Dit gebeurt in de volgorde waarin zij in de hierarchie zijn opgenomen, maar de Iman zal niet meer mensen bij zich roepen dan hij nodig heeft om de olieprijs vast te stellen.

Het doel van de Iman is een koper te vinden, die bereid is een prijs te betalen, waarvoor alle verkopers, die vóór hem bij de Iman geweest zijn, bereid zijn te handelen. Het is dan ook duidelijk dat de audientie snel is afgelopen in het geval dat de eerste Haraan een koper blijkt te zijn, want deze wordt uiteraard bereid geacht om te willen kopen voor de prijs van nul pegels (de pegel is de plaatselijke munteenheid, waarvan de waarde, tengevolge van het afwezig zijn van handelsverkeer met het buiten-land, moeilijk te bepalen is). We zullen dan ook maar aannemen, dat de eerste Haraan een verkoper is en derhalve zal de Iman met zijn wensen rekening moeten houden. De Iman gaat, gegeven de onmogelijkheid de verkoper rechtstreeks om zijn prijs te vragen, net zolang de prijs die hem voor ogen zweeft met een pegel verhogen, tot de verkoper ermee accoord gaat. Na aldus zijn zin te hebben gekregen mag de verkoper vertrekken.

Het is van belang te vermelden, dat de uiteindelijk vastgestelde prijs nooit lager zal zijn dan een tijdens het ritueel door de Iman uitgebracht tussenbod.

Zolang de voorafgaande audienties nog geen definitieve prijs hebben opgeleverd, wordt een volgende Haraan binnengeroepen. Als dit opnieuw een verkoper is wordt hij gelijk zijn voorganger-verkopers behandeld en tevreden gesteld. Blijkt de eerstvolgende Haraan echter een koper te zijn, dan wordt hem gevraagd of hij zich kan verenigen met de laatstgenoemde vraagprijs. Zo nee, dan kan hij het vertrek direct weer verlaten. Is de koper echter wel bereid om deze prijs te betalen, dan is hiermede de definitieve prijs vastgelegd. De laatste koper wordt uitgeroepen tot koper van het jaar en de prijs wordt onder paukengeroffel, bekkenslagen en het afsteken van vuurwerk verkondigd aan het volk. Deze prijs zal gedurende het gehele jaar gelden als de vaste olieprijs.

Er bestaan uiteraard diverse manieren om het ritueel te frustreren. Het is erg gemakkelijk voor een verkoper te doen alsof men voor geen enkele prijs bereid is te verkopen. De traditie leert, dat de Iman, door dit soort simulanten getergd, wel eens uit zijn rol wil vallen en onder bedreiging met een kromzwaard uiteindelijk toch nog een positieve reactie weet los te peuteren.

Ernstiger is de situatie die optreedt als de Iman aan het einde der hierarchie gekomen, nog steeds geen koper heeft gevonden. De Heilige Boeken schrijven voor, dat de Iman in dat geval zelf koper wordt, voor de laatst genoemde prijs; het is duidelijk dat in deze situatie de Iman moet aftreden en de geschiedenis van Harad kent vele burgertwisten en opvolgingstroebelen die op een dergelijk aftreden gevolgd zijn.

Het feest van de Groote Verrekening wordt afgesloten met een dankgebed en een orgie, maar de details daarvan wil ik U besparen. Veel interessanter is de evaluatie die B.E. LUB geeft van het hiervóór geschetste ritueel.

De zeergeleerde auteur maakt het aannemelijk, dat de Haranen er op uit zijn om enerzijds in vrede te leven met hun Iman, om op deze wijze verkoper te mogen blijven en aan de andere kant als verkoper graag zien gebeuren, dat hun vraagprijs gerespecteerd wordt. De kopers proberen op hun beurt het faillissement te ontlopen door een lage biedprijs te hanteren, maar meestal blijken enkelen onder hen zich te willen opofferen om maar van het koper zijn af te komen.

Vervolgens toont B.E. LUB aan, dat het gebruikte ritueel op optimale wijze aan de verlangens der Haranen tegemoet komt.

Het bewijs berust op een splitsing van de Haranen in drie groepen, afhankelijk van hun levenswandel (waarbij moet worden opgemerkt, dat B.E. LUB er bij de vorming van zijn model kennelijk van uitgaat, dat de Haranen onsterfelijk zijn). Allereerst zijn er die Haranen, die telkens opnieuw ruzie met de Iman krijgen en zich kort daarop failliet kopen. Deze Haranen slagen er kennelijk niet in hun doeleinden te realiseren.

In de tweede plaats zijn er de Haranen, die na enige lotswisselingen hun verdere levensdagen als verkoper slijten. Doordat zij nooit meer ruzie krijgen met de Iman, rukken zij zover op in de hierarchie, dat op den duur altijd met hun belangen rekening blijkt te worden gehouden.

De derde groep bestaat uit die Haranen, die uiteindelijk als koper door het leven moeten gaan en er niet meer in slagen, of niet meer de bereidheid opbrengen, nog een jaar de gevraagde olieprijs op te brengen. Jaarlijks brengen zij hun bezoeken aan de Iman, want zij zijn wegens hun standvastigheid opgerukt in de hierarchie, maar iedere keer is hun prijs te laag gekozen.

Het voorafgaande levert een verklaring voor de interne stabiliteit der Haraanse economie. Het zijn de standvastigen die, op hoge leeftijd gekomen, ver zijn opgerukt in de hierarchie en die nooit meer van rol zullen wisselen, die jaar na jaar de dienst uitmaken.

De oplettende lezer zal er na vergelijking met 3.4.4 en 3.4.1 misschien in slagen om de, in het betoog van B.E. LUB verpakte, MEYER-McCREIGHT algoritme te herkennen. Ik wil hierbij niet nalaten te vermelden, dat de gegeven beschrijving niet exact overeenstemt met de in het proefschrift gebodene, maar op bepaalde details uitgebreider is dan de in de literatuur bekende versies (vgl [HH 71] of [MMC 69]). Alhoewel enige barokke versieringen niet ontbreken, vormt zij toch de weerslag van het door mij gedurende de zomer van 1971 verkregen inzicht in deze als lastig bekend staande algoritme - een inzicht, dat de basis heeft gevormd voor het tot stand komen van dit proefschrift.

PREFACE

Abstract complexity theory is a subject in theoretical computer science, treating the mathematical properties of some universe of computability; more in particular it concentrates on the fact that the computations in such a universe use irreversibly some resource, in absence of which these computations are impossible.

The universe of computability is in general one of the formal systems from recursion theory, like for example the Turing-machine formalism, KLEENE's calculus of general recursive functions based on defining equations, or more abstractly the concept of an effective enumeration, as defined by ROGERS.

For the TURING formalism a measure for the resource used by the computation is given by the number of basic cycles executed during the course of the computation, or, alternatively, the number of tape squares "used". For the KLEENE formalism one could count the number of times a left-hand side is expanded by writing out a defining equation, but also the number z for which the KLEENE predicate T(i,x,z) holds can be interpreted to be the measure for the resource used by the i-th program at argument x.

Clearly for an abstractly defined effective enumeration the resource should be treated abstractly also. Such a treatment was given for the first time by M. BLUM, who in 1966 introduced the concept of a complexity measure. A complexity measure consists of an effective enumeration, equipped with a sequence of run-times, i.e. functions whose values represent the amount of resource used by the corresponding programs in the effective enumeration, provided that the programs terminate. This sequence must satisfy two conditions, called the BLUM axioms. In the first place the run-time has the same domain as the corresponding program; secondly the sequence of run-times is a so-called measured set; i.e. it should be decidable whether the run-time of program i at argument x equals y or not.

With this very general concept as a base, an impressive mathematical theory has been developed by M. BLUM, A. BORODIN, R.L. CONSTABLE, E.M. McCREIGHT, A. MEYER and several others whose contributions are discussed in this treatise.

From a mathematical point of view, abstract complexity theory provides us with a language which enables us to describe constructions in recursion theory in a fully machine-independent way. A disadvantage of the usual machine-independent treatment is the lack of formal rigidity used in the description and representation of the sometimes quite complicated algorithms and constructions used in the theory. The same criticism applies to the machine dependent theory as well.

Our personal interest in the subject mainly did originate from an endeavour to grasp the essential meaning of the MEYER-McCREIGHT algorithm, from its erroneously informal description in the survey paper by J. HARTMANIS and J.E. HOPCROFT [HH 71].

To substantiate our belief in the feasability of a more formal treatment, we present a formalism for the representation of algorithms in machine-independent recursion theory, using this formalism throughout this treatise. The formalism consists of a high level programming language, enriched by primitives for a complexity measure.

During the development of this formalism we discovered that a formalization of this type needs a further primitive, which enables one to generate, for a program represented by expressions in the programming language, an index in the effective enumeration under consideration.

Traditionally, the corresponding step in argumentations is made from outside the formal structure under discussion, by a reasoning like in "look at this function which I have described; clearly it is a computable and hence a recursive function (inessential use of CHURCH's thesis!) so there exists an index for it; get me such an index...".

Special care is taken to preserve those mathematical expressions which provide a single-line definition for a computable function. This is realized by extending the formalism with a mathematical representation style, giving these traditional expressions an unambiguous meaning. (As a matter of fact, nobody has ever worried about existing ambiguities in the usual language of abstract complexity theory.)

In abstract complexity theory, one of the main subjects is the behaviour of the different run-times of the distinct programs for a single function; in particular, much attention is paid to the so-called speed-up phenomenon. The contents of this treatise mostly belong to a second subject: the theory of resource-bound classes. These classes are defined by collecting all programs or functions whose run-times are bounded almost everywhere by some (recursive) function called the name of the corresponding resource-bound class.

To be more concrete, let $(\Phi_i)_i$ denote the sequence of run-times of the programs which are represented by the sequence $(\varphi_i)_i$. The complexity class F_t consists of all programs φ_i which for almost all arguments x in the domain of the name t satisfy the condition $\Phi_i(x) \le t(x)$. The class of functions computed by programs in F_t is denoted C_t. Analogously, for a two variable function R one defines the honesty class G_R by collecting all programs satisfying almost everywhere the condition $\Phi_i(x) \le R(x,\varphi_i(x))$ whenever the right-hand side is defined. The class of functions computed by programs in G_R is denoted H_R.

The main attention in complexity theory has been given to the complexity classes, whereas the honesty classes have been considered more or less to be equivalent to the measured sets. This conception originates from a well-known result by E.M. McCREIGHT which states that for total R the set H_R is recursively presented by a measured set, and that conversely the functions in a measured set all are R-honest for some total function R.

We have considered the honesty classes as an alternative type of resource-bound classes, investigating their properties as compared to the properties of the hierarchy of complexity classes. The result of this comparison can be found in the scheme at the end of this treatise.

During the process of generalizing the known results for complexity classes to honesty classes we discovered that, even in situations where the result on complexity classes remains valid for honesty classes, the classical proofs may break down and need repairing by non-trivial modifications. Moreover, a central result like the naming theorem of E.M. McCREIGHT becomes invalid for honesty classes. This suggests that there is more involved than a "slight modification of definitions".

Our analysis shows that what is involved is an essential difference in the appreciation of an infinite run-time; such a run-time is felt to be a violation in the case of the complexity classes, but it contributes no evidence against the honesty of the corresponding program. This analysis leads to the new abstract framework of an acceptance relation and a corresponding measured set of generalized run-times. Within this framework we can discuss both types of resource-bound classes and several other types at the same time.

Let $(\alpha_i)_i$ be some measured set. For a partial function t we denote by $\mathcal{F}_S(t)$ ($\mathcal{F}_W(t)$) the set of all indices i which for almost all x in the domain of t satisfy the condition $\alpha_i(x) \le t(x)$ ($\alpha_i(x) \le t(x)$ or $\alpha_i(x) = \infty$). $\mathcal{F}_S(t)$ ($\mathcal{F}_W(t)$) is called a strong (weak) abstract resource-bound class.

It will be argued that the strong classes generalize the complexity classes, whereas the weak classes are a generalization of the honesty classes. For a number of important results in abstract complexity theory, such as for example the gap and operator-gap theorems, the union theorem and the naming theorem, (for their formulation, see the scheme, mentioned earlier) the known proofs for complexity classes yield proofs for the strong abstract resource-bound classes by a straightforward translation. For weak classes, however, the proofs of the operator-gap theorem and the union theorem, need an essential repairing, and the naming theorem becomes invalid.

The final sections of this treatise contain some results related to the MEYER-McCREIGHT algorithm which is involved in the proof of the naming theorem for strong classes. We present a further generalization of the union theorem. Moreover, by concentrating on the renaming function of the MEYER-McCREIGHT algorithm we are able to construct a closure operator which maps arbitrary Σ_2-sets of indices onto the smallest strong class containing the given set. This latter theory is partially generalized for weak classes, yielding some new results on the set theoretical closure properties of honesty classes.

The treatise is completed by providing programs for some of the more complicated algorithms, treated in the text. Several of these programs are represented using the non-deterministic feature of parallelism, to indicate the amount of freedom which the user has in choosing a sequential implementation.

TO THIS THESIS-EDITION

Since it was our intention to make this thesis self-contained, without, on the other hand, providing unnecessary details which may be found elsewhere, the chapters 1-2 up to 2.4 have been reduced in size, and many proofs have been omitted. These sections, including a number of minor new results, will be restored to full length in the edition of this treatise which will appear as a two-volume publication in the Mathematical Centre Tract series.

ACKNOWLEDGEMENTS

I am deeply indebted to the late J. de GROOT for guiding me into research in pure Mathematics, giving me complete freedom in selecting my subjects. Without this freedom the present thesis would not have been written.

My research in Abstract Complexity Theory originated from a visit to the Computer Science Department at Cornell University in April 1971. This visit was made possible by an introduction provided by R.D. ANDERSON, and a return ticket to the U.S.A. paid for by IBM Netherlands, at the occasion of a two-months assignment of my wife in Endicott, N.Y.

I am grateful to J. HARTMANIS for introducing me to the field of Abstract Complexity Theory, for not being scared away by my vague and erroneous objections during the first half year, and for his guidance as coreferee to this thesis during the subsequent period. Two of our meetings took place at the GMD in Bonn, where he invited me during his sabatical leave in 1971-1972, whereas another meeting occurred at Cornell in October 1973, made possible through the hospitality of the Computer Science Department.

My gratitude also concerns my thesis-supervisors A. van WIJNGAARDEN for his interest in the different languages involved in this thesis, and P.C. BAAYEN for his contributions to the internal consistency and the clarity of this treatise, and his guidance during my ten years at the Mathematical Centre.

I thank A. MEYER for his interest in the work and for his suggestions which have improved the results in section 3.4.2, and L.G.L.T. MEERTENS for valuable suggestions concerning the representation style for algorithms used in the thesis, and for his general interest in the constructivistic aspects of Abstract Complexity Theory.

People who influenced or contributed to my mathematical development in general are (in alphabetical order) T.J. DEKKER, A. HEYTING, F.E.J. KRUZEMAN ARETZ, H.W. LENSTRA jr., F. OORT, A.B. PAALMAN-de MIRANDA, W.P. de ROEVER, P.S. STOBBE, and A. TROELSTRA.

Finally I thank the people involved in the technical realization of this thesis. These people are T.M.V. JANSSEN (for a multitude of XEROX copies), J. HILLEBRAND (for editing the thesis), J.K. LENSTRA (for typographical advice), A. SCHUYT-FASEN (for typing the extensive manuscript), and D. ZWARST, J. SUIKER and J. SCHIPPER (for the printing).

COLOPHON

This thesis was typewritten on an IBM 89. For the mathematical text Prestige Elite 12" was used, whereas the citations and programs were typed in Light Italic 12". The mathematical symbols which were used are present on the typing units Symbol 12", APL-Blanco, Script, Symbol Special, and the Symbol Greek 10", for the IBM composer. The Orator 10" and the Courier 12" were used for the title pages which were designed by J.K. LENSTRA.

The front cover was designed by T. BAANDERS, using an aerial photograph of the campus of Cornell University, taken by the author from a plane flown by R.L. CONSTABLE. The lithograph for the cover was prepared by D. ZWARST.

T. BAANDERS also designed the invitation for the reception after the ceremony.

This thesis was printed at the Mathematical Centre on a Gestetner Offset duplicator.

PART 1

INTRODUCTION TO MACHINE-INDEPENDENT COMPLEXITY THEORY

{23 And Abraham drew near, and said, Wilt thou also destroy the righteous with the wicked?
24 Peradventure there be fifty righteous within the city: wilt thou also destroy and not spare the place for the fifty righteous that are therein?
25 That be far from thee to do after this manner, to slay the righteous with the wicked: and that the righteous should be as the wicked, that be far from thee: Shall not the Judge of all the earth do right?
26 And the Lord said, If I find in Sodom fifty righteous within the city, then I will spare all the place for their sakes.
27 And Abraham answered and said, Behold now, I have taken upon me to speak unto the Lord, which am but dust and ashes:
28 Peradventure there shall lack five of the fifty righteous: wilt thou destroy all the city for lack of five? And he said, If I find there forty and five, I will not destroy it.
29 And he spake unto him yet again, and said, Peradventure there shall be forty found there. And he said, I will not do it for forty's sake.
30 And he said unto him, Oh let not the Lord be angry, and I will speak: Peradventure there shall thirty be found there. And he said, I will not do it, if I find thirty there.
31 And he said, Behold now, I have taken upon me to speak unto the Lord: Peradventure there shall be twenty found there. And he said, I will not destroy it for twenty's sake.
32 And he said, Oh let not the Lord be angry, and I will speak yet but this once: Peradventure ten shall be found there. And he said, I will not destroy it for ten's sake.
Genesis, XVIII 23-32}

CHAPTER 1.1

ALGORITHMS AND THEIR REPRESENTATION IN MACHINE-INDEPENDENT RECURSION THEORY

1.1.1. THE PROBLEM OF PROGRAM REPRESENTATION IN RECURSION THEORY

Recursion theory is a branch of mathematics dealing with computable functions. Although by now most mathematicians are accustomed to at least one formal definition of "computable function", the word itself betrays an inherent ambiguity of this concept: mathematicians are inclined to consider functions as being static objects (a specific type of a relation) whereas the word "computation" suggests some dynamical behaviour.

Both aspects are present in a formal definition. In the KLEENE formalism the functions are represented by defining sets of equations; stepwise application of these definitions yields a computation for a function value. In the TURING formalism the computation is mathematically defined and the function is nothing but the input output relationship defined by a particular interpretation of what is going on in the Turing machine.

Although the gap between the two aspects has been closed by rigorous mathematical proofs, there is no traditional method to combine within a single mathematical framework functions defined by describing some computation method and functions defined by mathematical expressions. There are several reasons why such a framework is not yet developed.

In the first place, mathematical formulas have been used much longer than formally represented algorithms. The description of a function by an expression is in general shorter and more transparent then a description by an algorithm for the same function.

Finally, there exists no "programming language" yet which is generally accepted by mathematicians.

Yet there exist functions, defined in recursion theory, which are only definable by means of some algorithm for them, since a definition by means of mathematical equations becomes uncomprehensible. The recursive permutation constructed in the proof of the MYHILL isomorphism theorem (th. 1.4.8) (see also [Ro 67]) may be considered to be a typical example of such a function.

This leads to the deplorable situation in actual (machine-dependent or machine-independent) recursion theory that the larger part of the algorithms are described informally or even ambiguously. Excuse for this informality is sought by invoking ("inessentially") the thesis of CHURCH whenever needed.

To our opinion a great deal of formality and rigidity can be gained by application of a well designed programming language. This language should satisfy a number of more or less contradicting conditions which we formulate below.

(i) The language should be directed to the subject under discussion. In our situation, where we are dealing with Machine-Independent Complexity Theory, this means that the structure of a complexity measure should be easily accessible.

(ii) In order to be readable, the description of functions in our language should not be unnecessarily distinct from the usual mathematical denotation by expressions.

(iii) The language should be unambiguous. (This is clearly necessary but difficult to combine with (ii).)

(iv) The language should be easy to read for mathematicians without any programming experience (since many recursion theoreticians belong to this category). The meaning of a program should be clear even without knowing the language.

(v) It is not necessary that the language should be implemented.

In the sequel of this chapter a language designed to satisfy these conditions is proposed.

If we have been able to construct this language indeed in such a way that (iv) is satisfied this should make it possible for those readers which are not interested in the particularities of our language and the more complicated algorithms written in it, to skip the remainder of chapter 1.1 and to proceed to where the mathematics is resumed (cf. chapter 1.2). These readers should not be scared by the ALGOL-like formulas and instructions, keeping in mind a few usable translations like:

<u>*int*</u> *i = 6 ;*	"let i be six".
<u>*int*</u> *i ;*	"let i be an integer variable".
i := 6 ;	"i becomes six".

if p then S else T fi ;	"if p holds then do S; otherwise do T". (The same *if-then-else-fi* construction occurs also within expressions.)
for j to 100 do ...	"repeat for j = 0,1,2,...,100 ..."
*proc f = (int x) int: x*x+2*x+1* ;	"f is a function with an integral argument denoted x and integral value; $f(x) = x^2+2x+1$".

After having lured this way part of our readership into believing that the reading of our programs will be easy, we now turn to the technical problems related to (i), (ii) and (iii).

In order to have a good accessibility to the structure of a complexity measure we make the following proposals.

1) Our basic types are integers and booleans. By *integers* we mean the non-negative integers 0,1,2,3,... . *Boolean* values are *true* and *false*. Although usually the integers 0 and 1 are used to represent *true* and *false* we have preferred not to do so in our language. *Pairing and projection functions* will be implemented making it possible to interpret our integers as being pairs, and multiplets or finite sequences of integers.

2) By a *function* we will understand a partial recursive function defined on integers with integral values. Within our language we will at some places also allow functions whose values may be either integral or boolean, or even more general integral, boolean or one of the two *error conditions* *error* or *loop*.

3) The effective enumeration on which our complexity measure is based is explicitly available in our language; the value of the i-th program at argument x is simply denoted by $\varphi_i(x)$.

4) The run-times of the complexity measures are accessible by the decision procedure given by the second Blum axiom. The instruction to compute whether the i-th program terminates at argument x within y steps is simply denoted by $\Phi_i(x) \le y$.

5) The universal machine, and the s-n-m function corresponding to our effective enumeration are explicitly implemented in our language. Moreover, the language is introspective in the following sense: there exists an operator *index* which transforms a function-routine, i.e. a piece of pro-

gram text describing some function, into an index of this function in the effective enumeration.

6) Structuring of data types, expressions, clauses and procedures is defined as is the case in ALGOL 68; in fact the description of our language consists of the description of an extension of ALGOL 68, with a few modifications; this extension is then mapped onto our language by a syntactical transformation which makes it satisfy our requirement (ii) on readability.

In trying to satisfy the requirements (ii) and (iv) we should not forget to keep our language unambiguous. A typical example of possible ambiguities is the meaning of the inequality operator $\leq$. The algorithm used to decide $f(x) \leq g(x)$ is a quite different one, depending on whether f or g are ordinary functions or run-times of programs in the enumeration; in the second case the decision procedure given by the second Blum axiom is involved.

A more complete description of our language is given in the next two sections. First we describe an extension of ALGOL 68 with some particularities, redefined according to our needs, in §1.1.2. Next we replace in §1.1.3 a number of constructs by "mathematical representations" to make our programs readable. This replacement should be considered to be another extension.

In the sequel of chapter 1.1 the reader is supposed to know the concepts of an effective enumeration, complexity measure, transformation of programs and measured set. If these concepts are still unknown the reader should first read chapters 1.2 and 1.3.

1.1.2. DESCRIPTION OF A PROGRAMMING LANGUAGE FOR RECURSION THEORY - THE ALGOL 68 EXTENSION

Before introducing any new construct we indicate the following modifications to the definitions in the ALGOL 68 report:

(1) Representation alfabet

A certain number of Greek characters have become terminal production of 'letter token'. Occurrence of these tokens is reserved for denoting the programs and run-times of our Blum-measure (φ and Φ), transformations of programs (σ,τ,ρ,κ), measured sets (Υ,α), λ- and μ-expressions (λ,μ), operators (Γ), projection operators (π), the elaboration operator (Λ), the empty function (ε).

(2) Integers

Integral values are restricted to be non-negative. The set of non-negative integers is denoted by $\mathbb{N}$. The integer capacity is supposed to be infinite.

(3) Defaults for arrays and loops

All counting starts at zero instead of one. Hence lower bounds of arrays sliced from others are by default set to zero; *from* 0 can be omitted and zero indicates the first case in a case-clause.

(4) Conditions

There exists a plain type "condition" (indicated by *cond*) consisting of the two values *error* and *loop*; *error* represents the result of detection of some error condition; *loop* represents the result of willfully executing a statement like *l:goto l;* whereas a result *error* can be used by subsequent computations, *loop* cannot be input to any terminating computation. The conditions *error* and *loop* are used syntactical analogously to jumps.

(5) Boolean operators

The operators *and* and *or* are elaborated as should be the case when their second operand had been of the mode *proc bool* instead of *bool*. Consequently elaborating "*true or q*" or "*false and q*" the operand *q* is not elaborated.

This modification is motivated by the fact that we use many times the expression *p and q* in a situation where termination of *q* is guaranteed only if *p* holds. *iff* is an indication for equality between booleans.

(6) Indications

The symbols *eq*,*nq*,*lt*,*le*,*gt*,*ge* are not used as alternative representations for $=,\neq,<,\leq,>,\geq$ (they are reserved to denote inequalities involving functions from a measured set).

(7) Scope restrictions

The following example of an ALGOL 68 program is found to be illegal because of scope restrictions:

```
begin proc increasor = (ref int ii) proc void: ii +:= 1;
      int i := 0;
      proc void inc = increasor(i);
      inc; print(i) pr ? pr
end
```

Since it is clear that the above program is intended to print the number 1 (*inc* is supposed to possess the routine *i +:= 1* and not *ii +:= 1*), we will forget the involved scope restrictions. (This modification is motivated by requirement (v).).

Next we indicate how the features promised in the introduction are represented.

(8) United types

The following united modes are introduced.

```
mode result = union (int,bool),
     outcome = union (int,bool,cond)
```

(9) Operations on integral operands

Since there are no negative integers neither the monadic operator - nor the diadic operator - is implemented. There exists however a diadic operator ∸ with priority 6 and "declaration":

```
op ∸ = (int x,y) int: ¢ if x≥y then x−y else 0 fi ¢;
```

(note that the body of this declaration is outside our language).

The following pairing function and projection operators are available:

```
op π1 = (int x) int: ¢ first coordinate of x ¢;
op π2 = (int x) int: ¢ second coordinate of x ¢;
proc pair = (int x,y) int: ¢ integer representing the pair <x,y> ¢;
```

We request that pair(x,y) is a bijection from the set $\mathbb{N}^2$ onto $\mathbb{N}$. Moreover, pair is monotonically increasing in both arguments. Consequently pair(0,0) = 0. *π1* and *π2* are the coordinate mappings to pair. We have the following equalities:

```
pair(π1 x, π2 x) = x
π1 pair(x,~) = x
π2 pair(~,x) = x
```

Build from the above two-dimensional pairing and projection functions are sufficiently many higher dimensional ones. For example:

```
op π3,1 = (int x) int: π1 x;
op π3,2 = (int x) int: π1 π2 x;
op π3,3 = (int x) int: π2 π2 x;
proc triplet = (int x,y,z) int: pair(x,pair(y,z));
```

(10) Operations on boolean operands

Implication is implemented.

```
priority imp = 1;
      op imp = (bool p,q) bool: if p then q else true fi;
```

Elaboration of *p imp q* proceeds however like when *q* had been of the mode *proc bool*; *q* is not elaborated in *false imp q*.

The "assigning variants" of *and* and *or* are implemented:

```
priority ∧:= = 1, ∨:= = 1;

      op ∧:= = (ref bool pp, bool q) ref bool: p := p and q;
      op ∨:= = (ref bool pp, bool q) ref bool: p := p or q;
```

Again *q* is not elaborated if not needed, like in *bool p := true, q;*
p ∨:= q;   p := not p;   p ∧:= q;

(11) Least-number operator, bounded quantifiers, maxima and minima

The following procedures are given:

```
proc least number = (proc (int) bool p) int:

      (int z := 0; while not p(z) do z +:= 1 od; z);

proc bnd unv qua = (proc (int) bool p, int k) bool:

      (bool b := true; for i to k while b do b ∧:= p(i) od; b);

proc bnd ext qua = (proc (int) bool p, int k) bool:

      (bool b := false; for i to k while not b do b ∨:= p(i) od; b);

proc bnd least number = (proc (int) bool p, int k) int:

      (int z := 0; while not p(z) and z≤k do z +:= 1 od; z)
```

Note that (modulo side effects in p) one has

```
        bnd least number (p,k) = if bnd ext qua (p,k) then least number (p,k)
                                     else k+1 fi.
op max = ([ ] int r) int:
        (int z := 0; for i from ⌊r to ⌈r do
                     if z < r[i] then z := r[i] fi od;
                     z);
proc max = (proc (int) int f, int l,u) int:
        (if u<l then 0
        else int m := f(l);
             for j from l+1 to u do
                 if (int n = f(j)) > m then m := n fi od;
             m
        fi);
op min = ([  ] int r) outcome:
        int u = ⌈r, l = ⌊r;
        if u<l then error
        else (int z := r[l];
              for i from l+1 to u do
                  if z > r[i] then z := r[i] fi od;
              z)
        fi);
proc min = (proc (int) int f, int l,u) outcome:
        (if u<l then error
        else int m := f(l);
             for j from l+1 to u do
                 if int n = f(j); n<m then m := n fi od;
             m
        fi);
```

Maxima and minima can be computed both for linear arrays and for functions defined over finite segments. Note that the minimum over an empty domain leads to the condition *error*. Consequently whenever *min* or *min* are used the definedness of the result must be checked using a conformity case clause.

For calls of the above procedure alternative "mathematical representations" will be provided in the next section.

(12) Standard functions

The following special functions are used:

```
proc ε = (int x) int: (l:goto l);
op zero = (int x) int: 0;
proc zero = (int x) int: 0;
```

These procedures exist also for more arguments

```
proc ε2 = (int x,y) int: (l:goto l);
proc zero 2 = (int x,y) int: 0;
op even = (int x) bool:  (x∸x÷2*2) = 0;
proc even = (int x) bool: even x;
op odd = (int x) bool: not even x;
proc odd = (int x) bool: odd x;
```

(13) Effective enumeration and complexity measure

The basic elements of the BLUM measure under consideration (programs and run-times) are introduced by the definition of a couple of structured modes "computations" and "run-times" and a number of operators defined on values of these modes.

```
mode comp = struct(int ind,arg),
     rt = struct(int prog,arg);
```

comp(i,x) represents the computation of $\varphi_i(x)$ as an intentional object. Similarly, *rt(i,x)* represents the run-time of this computation as an intentional object. The value of a computation results from applying the elaboration operator Λ.

```
op Λ = (comp c) int: ¢ the value of φ_i(x) whenever defined
                       where i = ind of c and x = arg of c;
                       if φ_i(x) diverges so does the call Λc  ¢;
```

This operator Λ replaces the universal machine.

For a run-time it is essential that the question $\Phi_i(x) = y$ is decidable. This is introduced into our language by defining the following operator = :

```
op = = (rt r,  int y) bool: ¢ the value of Φ_i(x) = y, as computed using
                              the second Blum axiom where i = prog of r,
                              x = arg of r  ¢;
```

The numerical value of a run-time results from applying the elaboration operator Λ:

_op_ Λ = *(*_rt_ *r)* _int_*: least number ((*_int_ *y)* _bool_*: r=y);*

The first BLUM axiom is implemented by requesting that Λ *comp(i,x)* converges if and only if Λ _rt_*(i,x)* converges.

We still need the s-n-m function for our effective enumeration. Instead of providing this function by some fixed procedure declaration we provide a much stronger instrument: the operator _index_. The reason for this is the following. In general one uses the s-n-m axiom to provide a total function $\tau(k)$ which computes an index for the function S(-,k) which results from replacing in some (very complicated) recursive function S(-,-) the second argument by a fixed integer k. The s-n-m axiom provides this function τ, given some index for S. In general S itself is defined in our programming language itself, and since our effective enumeration contains all recursive functions an index for S exists. Hence in order to be able to apply the S-n-m function, we still need a way to go from a program for S to an index for S; this translation is performed by the operator _index_. Moreover, by introducing _index_ the s-n-m function becomes definable.

Now a huge problem arises since in order to define _index_ we must define which function is possessed by a function routine f which is given by a piece of program text during elaboration of our algorithm. Clearly this meaning depends on the actual values of all non-local identifiers in f which depend in their turn on the nest of active declarations. One might try to replace all non-local identifiers in f by denotations for their values (assuming sufficiently many denotations exist), but this replacement may be non-terminating because of the presence of recursive procedures which are referred to in f.

Still there is one type of non-local identifier which we want to be replaced by a denotation of its value. Since we are particularly interested to see how an index of the function S(-,k) depends on the numerical value of k we replace k by its value.

Therefore we present the following description for _index_:

_op_ _index_ = *(*_proc_ *(*_int_*)* _int_ *f)* _int_*:*

> _¢_ *an index for the function computed by the routine f' which results from the routine f by replacing all non-local integral identifiers occurring in f by denotations for the values possessed by these identifiers at the instance of the call* _index_ *f* _¢_

As an example we show how the s-n-m function is defined using *index*:

```
proc snm = (int i,j) int: index(int x) int: Λ comp(i,pair(j,x));
```

The operators Λ and *index* satisfy the following relation:

$f(x) = \Lambda\ \underline{comp}(\underline{index}\ f,x)$

(disregarding side-effects in f)

The operator *index* may be considered to be an axiomatization of the thesis of CHURCH: every function defined in our language is contained in the effective enumeration.

The operator Λ acts as an inverse to *index*:

```
op Λ = (int i) proc(int) int: (int x) int: Λ comp(i,x);
```

Note that not necessarily *index* Λ $i = i$. It is true, however, that f and Λ *index* f are extensionally equivalent, i.e. they compute the same function.

The following operators are introduced to have all possibilities of compairing run-times with integers and/or run-times:

```
op ≤ = (rt r, int y) bool: bnd ext qua ((int z) bool: r = z,y);
op < = (rt r, int y) bool: r≤y and not r=y;
op ≠ = (rt r, int y) bool: not (r=y);
op > = (rt r, int y) bool: not (r≤y);
op ≥ = (rt r, int y) bool: not (r<y);

op ≤ = (rt r,s) bool:
            (bool undecided := true, p; int z := 0;
             while undecided do
                   if r=z then p := true; undecided := false
                   elif s=z then p := false; undecided := false
                   else z +:= 1
                   fi od;
                   p
            );

op = = (rt r,s) bool: r≤s and s≤r;
op < = (rt r,s) bool: r≤s and not r=s;
```

In the next section we introduce the "mathematical representations" for the above constructs.

(14) Transformations of programs

The concept of a transformation of programs by definition is nothing but a total recursive function which is supposed to transform the index of a program into the index of another program. Intuitively one should think the transformation to be some construction which modifies a program text in a systematic way; this construction induces a total function mapping the index of the original program onto the index of its image. The transformation also modifies the computed function but since it is not certain that extensional equality between programs is preserved it is not possible to regard the transformation to be an operator.

Consider for example the transformation which maps a program φ_i onto a program computing $\underline{max}(\varphi_i(x),\Phi_i(x))$. Using the s-n-m axiom one proves mathematically the existence of a total function σ such that for each x $\varphi_{\sigma(i)}(x) = \underline{max}(\varphi_i(x),\Phi_i(x))$. Within our programming language, however, we want to be able to write a definition for σ. This has strongly motivated our definition of the operator *index*. In the above example we can write:

$\underline{proc}\ \sigma = (\underline{int}\ i)\ \underline{int}\colon \underline{index}(\underline{int}\ x)\ \underline{int}\colon \underline{max}(\Lambda\ \underline{comp}(i,x),\ \Lambda\ \underline{rt}(i,x));$

Remember that during a call of *index* *i* is replaced by a denotation of its current value.

The example shows that it is possible to define transformations of programs within our language without introducing any construct. Because of the specific importance of the mathematical concept we provide in the next section a mathematical representation for this type of a declaration of a transformation of programs.

A transformation-declaration may be formally defined as follows: Let E be a unitary integral clause and suppose that $i1,i2,\ldots,ik$ and x represent distinct integral mode identifiers whose scope contains E as a proper subrange. Then the following procedure declaration

$\underline{proc}\ \tau = (\underline{int}\ i1,i2,\ldots,ik)\ \underline{int}\colon \underline{index}(\underline{int}\ x)\ \underline{int}\colon E;$

is called a transformation declaration.

Note that (disregarding side effects) the following holds: Let $t1,\ldots,tk$ denote integral primaries, and let $E[t1/i1,t2/i2,\ldots,tk/ik,y/x]$ denote the result from substitution of $t1,\ldots,tk,y$ for $i1,\ldots,ik,x$ in E. Then one has:

$\Lambda\ \underline{comp}(\tau(t1,t2,\ldots,tk),y) = E[t1/i1,t2/i2,\ldots,tk/ik,y/x]$

(15) Measured sets

A measured set $(\gamma_i)_i$ is a sequence of functions for which a decision procedure for $\gamma_i(x) = y$ is given. Consequently if a measured set is dealt with in our language this decision procedure should be the data structure to start with. All other operations and data structures are derived from this structure.

```
mode ms = proc(int,int,int) bool;
```

A value γ of the mode ms is called improper if there exists integral i, x, y and z such that $y \neq z$ and both $\gamma(i,x,y)$ = *true* and $\gamma(i,x,z)$ = *true*.

```
mode mc = struct(ms ga, int ind, arg);
mode mf = struct(ms ga, int ind);
```

Whereas the *ms* value γ represents the decision procedure for $\gamma_i(x) = y$, the *mc* (measured computation) value (γ,i,x) represents the computation of $\gamma_i(x)$ as an intentional object and the *mf* (measured function) value represents the intentional function γ_i.

For dealing with measured sets in an unambiguous way the indications *eq*, *nq*, *lt*, *le*, *ge*, *gt* have been reserved. We give the necessary declarations below.

```
priority eq = 4, nq = 4, lt = 5, le = 5, ge = 5, gt = 5;

op eq = (mc p, int z) bool: (ga of p)(ind of p, arg of p, z)

op Λ = (mc p) int: least number ((int x) bool: p=x)
```

$\Lambda(\gamma,i,x)$ is the numerical value of $\gamma_i(x)$.

```
op Λ = (mf pp) proc(int) int: (int x) int: Λ mc(ga of pp, ind of pp,x);
```

$\Lambda(\gamma,i)$ is the function γ_i considered to be an ordinary integral function.

The operations *nq*, *lt*, *le*, *ge* and *gt* are derived from *eq*.

```
op le = (mc p, int y) bool: bnd ext qua ((int x) int: p eq x,y);

op lt = (mc p, int y) bool: p le y and not (p eq y);

op nq = (mc p, int y) bool: not (p eq y);

op gt = (mc p, int y) bool: not (p lt y);
op le = (mc p,q) bool:
        (bool undecided := true, b; int z := 0;
         while undecided do
              if p eq z then b := true; undecided := false
              elif q eq z then b := false; undecided := false
              else z +:= 1
              fi od;
              b
        );
op eq = (mc p,q) bool: p le q and q le p;
op lt = (mc p,q) bool: p le q and not (p eq q);
```

A transformation of programs τ is called measured if the sequence $(\varphi_{\tau(i)})_i$ is a measured set. Within our language this means that there exists a proper mc value γ such that

$$\gamma(i,x,z) = \underline{¢} \wedge \underline{comp}(\tau(i),x) = z\ \underline{¢}.$$

Now in general the right-hand side of this expression does not describe the algorithm which makes the relation $\varphi_{\tau(i)}(x) = z$ decidable. Consequently definition of the transformation τ alone is not sufficient to describe the corresponding ms value.

Therefore in defining a measured transformation one should have the decision procedure available.

```
mode mt = struct(ms ga, proc(int) int tau)
```

Given a measured set γ one may define a measured transformation as follows:

```
op tf = (ms γ) mt: (γ, (int i) int: index ∧ mf (γ, i))
```

1.1.3. MATHEMATICAL REPRESENTATIONS

The operators, structured values and other constructs defined in the preceding section are not used in this form in the sequel. In order to increase the readability of the represented algorithms we introduce by means of several extensions a number of mathematical representations. As a consequence many expressions will be looking like the ones mathematicians traditionally are using. There are a few exceptions. The numerical value of a run-time must be written explicitly using the operator Λ; a restriction leading to overredundant occurrences of this operator in situations where it is clear that the numerical value is needed (like in $R(x,\Lambda\Phi_i(x))$). A second unusual phenomenon is the occurrence of the indications *le*, *eq*, *nq*, *lt*, *ge* and *gt* in the context of a measured set or measured transformation.

Finally we describe a quite radical extension for transformation-declarations which hides completely the use of the operator *index*; only the occurrence of the symbol $\Leftarrow$ remembers the reader that in fact the transformation τ is the identifier declared in

$$\varphi_{\tau(i)}(x) \Leftarrow \underline{max}(\varphi_i(x),\Lambda\Phi_i(x)).$$

In the sequel of this section we let $E,T,T_1,\ldots,T_k,U$ denote integral unitary clauses. $I,J,X,Y,Z,I_1,\ldots,I_k$ denote integral identifiers. $P,Q,P_1,\ldots,P_k$ denote unitary boolean clauses. L denotes an integral unit list, $R(F)$ denotes a procedure-with-integral-parameter-boolean (integral) unit. Γ denotes a measured set identifier and Σ denotes a procedure identifier declared by a transformation declaration. Ξ denotes a measured transformation identifier.

(16) λ-notation

The procedure-with-integral-parameter-integral denotation

$$(\underline{int}\ X)\ \underline{int}\colon E$$

may be replaced by

$$\lambda X[E].$$

Similarly one may replace

$$(\underline{int}\ X)\ \underline{bool}\colon P \quad \text{by} \quad \lambda X[P].$$

This extension is not permitted in the declaration of recursive procedures: the declaration *proc* $p = \lambda X[E]$ is invalid if p occurs in E.

This extension generalizes for many-variable functions as well. For example

(int x,y) int: x+y	may be written	$\lambda x,y[x+y]$.

(17) Pairing and projection functions

We provide for the projection operators the following (non-linear) representations:

$\underline{\pi 1}$	π_1
$\underline{\pi 2}$	π_2
$\underline{\pi 3,1}$	π^3_1
$\underline{\pi 3,2}$	π^3_2
$\underline{\pi 3,3}$	π^3_3

Calls of the pairing functions pair, triplet etc. may be represented using < and >

$pair(T_1,T_2)$	may be written as	$<T_1,T_2>$
$triplet(T_1,T_2,T_3)$	may be written as	$<T_1,T_2,T_3>$

(18) Least-number operator, bounded quantifiers, maxima and minima

least number $(\lambda X[P])$ may be replaced by $\mu X[P]$.

It was noted by L.G.L.T. MEERTENS that a separate extension of the type

least number (R) is replaced by $\mu X[R(X)]$

is unnecessary and in fact ambiguous.

Using this latter extension both *least number* (R) and *least number* $(\lambda X[R(X)])$ are represented by $\mu X[R(X)]$.

The effect of the above extension can be collected without using it since clearly R and $\lambda X[R(X)]$ are equivalent routines.

The same observation holds for other extensions defined in section (18).

This extension introduces the μ-operator as usually written. For the bounded quantifiers the usual mathematical representations are also legal.

$bnd\ unv\ qua\ (\lambda X[P],T)$ becomes $\forall X \le T[P]$
$bnd\ ext\ qua\ (\lambda X[P],T)$ becomes $\exists X \le T[P]$

The analogous extension holds for the bounded least-number operation

$bnd\ least\ number\ (\lambda X[P],T)$ becomes $\mu X \le T[P]$

In applying these extensions the identifier X must be selected such that no clash of identifiers results.

For bounded quantification with the bound T not belonging to the domain over which is quantified we have the following extensions:

$\exists X \le T[X \ne T\ \underline{and}\ P]$ becomes $\exists X < T[P]$
$\forall X \le T[X = T\ \underline{or}\ P]$ becomes $\forall X < T[P]$

(Use of this extension is improper whenever evaluation of T has side-effects.)

For the calls of max and min we use the following mathematical representations:

$max(\lambda I[E],T,U)$ becomes $max\{E \mid T \le I \le U\}$
$min(\lambda I[E],T,U)$ becomes $min\{E \mid T \le I \le U\}$

Moreover, in the right-hand side representation

$0 \le I \le U$ may be replaced by $I \le U$.

In the mathematical text we allow, moreover, contraction of iterated maximalizations or minimalizations; for example

$max\{max\{E \mid I \le T\} \mid J \le T\}$ becomes $max\{E \mid J,I \le T\}$.

(These extensions are only a compromise to mathematicians.)

(19) Effective enumeration and complexity measure

In the preceding section we separated the intentional meaning of a computation and a run-time from their numerical values by introducing specific new modes for the intentional objects. Consequently in introducing the mathematical representations $\varphi_i(x)$ or $\Phi_i(x)$ we must choose whether these expressions denote the intentional or the extensional object. Our choice is motivated by the practical use. Nobody ever uses in some expression $\varphi_i(x)$ but for its numerical value, hence we let $\varphi_i(x)$ denote the integral unit Λ *comp(i,x)*. For a run-time it is however crucial that the decision procedure of the second Blum axiom may be invoked. Consequently we let $\Phi_i(x)$ denotedthe intentional object *rt(i,x)*. This means that whenever a run-time is used as argument for some function like in $R(x,\Phi_i(x))$ this representation becomes illegal; we must write $R(x,\Lambda\Phi_i(x))$.

A similar problem exists for the distinction between a program and the function computed by it. Mathematically a program is nothing but a function in the effective enumeration. Most mathematicians consider φ_i to be the computed function and if for some reason one wants to consider the program computing it one takes not the program itself but its index i. Since we are dealing in part 3 with situations where we have a three-level distinction between an index, a program encoded by it, and the function computed by this program where, moreover, two indices may encode the same program we must abstain from this convention.

Therefore we are obliged to use the following representations. If i is some index then φ_i denotes the program with index i and $\Lambda\varphi_i$ denotes the function computed by φ_i.

Note however, that the expression φ_i will not occur in any program without a parameter pack following it (there exists no mode "program"). Moreover, there is no ambiguity between $\varphi_i(x)$ and $\Lambda\varphi_i(x)$; the first expression is not a phrase in the second but the values of the two integral clauses are equal.

For the run-times Φ_i the situation leads to an inessential ambiguity. Since Φ_i is again an intentional representation of the step-counting function, which is not referred to in any program, there is no ambiguity in $\Phi_i(x)$. The numerical-value-of-the-running-time function is represented by $\Lambda\Phi_i$. Consequently the clause $\Lambda\Phi_i(x)$ is ambiguous since it may represent both of the following clauses.

$$\Lambda\ \underline{rt}(i,x) \quad \text{or} \quad ((\underline{int}\ x)\ \underline{int}\text{:}\ \Lambda\ \underline{rt}(i,x))(x).$$

Note however that the values of both clauses are equal.

We now give the formal definitions of our extensions:

The integral unit	$\Lambda\ \underline{comp}(E,T)$	is replaced by	$\varphi_E(T)$
The run-time unit	$\underline{rt}(E,T)$	is replaced by	$\Phi_E(T)$
The *proc(int) int* unit	ΛE	is replaced by	$\Lambda\varphi_E$
The *proc(int) int* unit	$(\underline{int}\ x)\ \underline{int}\text{:}\ \Lambda\Phi_E(x)$	is replaced by	$\Lambda\Phi_E$

Using the above extensions we are very often confronted in the mathematical text with an expression $\lambda x[S(x,\Lambda\Phi_i(x))]$ which sometimes is used as a subscript. For typographical reasons we use therefore in the text the following extension: Let S be a procedure with two integral parameters and integral (or boolean) value. Then $\lambda X[S(X,F(X))]$ may be denoted by $S \,\square\, F$. Moreover, $\lambda X[S(X,\Lambda\Phi_T(X))]$ may be denoted by $S \,\square\, \Phi_T$ (so the Λ may be omitted in this context).

(20) Transformations of programs

The transformation declaration

$$\underline{proc}\ \Sigma = (\underline{int}\ I_1,\ldots,I_k)\ \underline{int}\text{:}\ \underline{index}(\underline{int}\ X)\ \underline{int}\ E;$$

may be represented by

$$\varphi_{\Sigma(I_1,\ldots,I_k)}(X) \Leftarrow E;$$

This extension is motivated by the usual way to introduce transformations by expressions like:

"Let σ be a total function such that $\varphi_{\sigma(i)} = \underline{\max}(\varphi_i(x),\Phi_i(x))$".

The symbol $\Leftarrow$ may not be used in any other way; its use as operator indication is strictly forbidden.

It should be noted that the above extension is used mostly within the mathematical text.

(21) Measured sets

For measured sets we again have a distinction between the intentional object of a measured computation and its numerical value.

The *mc* unit	*mc*(Γ,E,T)	is represented by	$\Gamma_E(T)$
The *mf* unit	*mf*(Γ,E)	is represented by	Γ_E.

Consequently expressions like $\Gamma_E(T)$ *le* U are defined.

Again the combination $\Lambda\Gamma_E(T)$ is an ambiguous integral clause since it is not clear whether the function $\Lambda\Gamma_E$ is called or the measured computation $\Gamma_E(T)$ is enumerated. Both parsings, however, lead to the same value, disregarding side-effects. If side-effects are involved the extension is improper.

If Ξ is a measured transformation then *ga* *of* Ξ is a measured set and *tau* *of* Ξ is a transformation. Consequently *mc*(*ga* *of* Ξ,I,E) is an mc unit and Λ *comp*((*tau* *of* Ξ)(I),E) is an integral unit. Since it is clear by the operators in the context whether an integral clause or an mc clause is needed, we allow that both clauses are represented by $\varphi_{\Xi(T)}(E)$. Consequently both $\varphi_{\Xi(T)}(E)$ *le* U and $\varphi_{\Xi(T)}(E) \leq U$ make sense but the computations involved are distinct.

If the programmer creates an ambiguity by the introduction of new operators the above extension becomes improper.

(22) Convergence test

In the mathematical text the symbol ∞ is used to denote divergence like for example in $\varphi_i(x) = \infty$ ($\varphi_i(x)<\infty$) for "$\varphi_i(x)$ diverges" ("$\varphi_i(x)$ converges"). The use of these expressions is legalized by the following extensions:

($\varphi_E(T)$;*true*)	becomes	$\varphi_E(T) < \infty$
($\varphi_E(T)$;*false*)	becomes	$\varphi_E(T) = \infty$
($\Lambda\Phi_E(T)$;*true*)	becomes	$\Phi_E(T) < \infty$
($\Lambda\Phi_E(T)$;*false*)	becomes	$\Phi_E(T) = \infty$
($\Lambda\Gamma_E(T)$;*true*)	becomes	$\Gamma_E(T)$ *lt* ∞
($\Lambda\Gamma_E(T)$;*false*)	becomes	$\Gamma_E(T)$ *eq* ∞

Note that for a measured transformation Ξ both $\varphi_{\Xi(i)}(x)$ *lt* ∞ and $\varphi_{\Xi(i)}(x) < \infty$ make sense; the two expressions have the same value.

(23) Many-variable programs

By definition our enumeration consists of one-variable functions. The same holds for the functions in a measured set. Since by use of the pairing functions many-variable functions can be represented by one-variable functions, the restriction to single-variable functions is not enforced throughout this book. We assume to be given a sufficiently large collection of extensions like the two written below permitting the use of super indices to indicate the number of variables. Examples:

$$\varphi_E(<T,U>) \quad \text{is denoted} \quad \varphi_E^2(T,U)$$
$$\Phi_E(<T,U>) \quad \text{is denoted} \quad \Phi_E^2(T,U)$$

1.1.4. MANIPULATION OF UNBOUNDED ARRAYS, SUMMATION AND LINEAR LISTS

The linear array of increasing size is a type of data structure which is frequently used in algorithms within recursion theory. Instead of defining an extension to permit a declaration like $[0:\infty]$ *int* a; after elaboration of which each element of a is available, we use the construct of a flexible array, which exists in ALGOL 68. A disadvantage of this construct is that for each extension of the array the whole contents must be copied.

To facilitate the use of these flexible arrays we introduce a number of procedures. Whenever some flexible array of a certain type is declared, the corresponding manipulating routines are assumed to be delcared outside the particular program under consideration.

All arrays are supposed to have zero as lower bound. For each type we have three procedures available. The first procedure inserts a given value at a given place, creating if necessary the needed space; other newly created fields are initialized with a default value depending on the type; the value of the call is the length of the extended array.

A second procedure looks for a certain item from the array; if the wanted member is not yet present the array is extended and the default value is delivered.

The third procedure simply extends the array.

In the description below $\underline{M}$ denotes a mode indication and D denotes the default value corresponding to this mode. M is a sequence of symbols which looks sufficiently like $\underline{M}$ to see which mode is involved in the declaration (i.e. if *M* is *int* then *M* is *int* but if *M* is *struct(int a,b)*, *M* may become something like *str int a int b*).

```
proc insert M = (ref flex [ ] M ar, int ind, M val) int:
      (int ub = ⌈ar;
       if ind ≤ ub
          then ar[ind] := val; ub
          else [:ind] M temp;
                for j to ub do temp[j] := ar[j] od;
                for j from ub+1 to ind∸1 do temp[j] := D od;
                    temp[ind] := val;
                    ar := temp;
                    ind
       fi);
proc lookup M = (ref flex [ ] M ar, int ind) M:
      if ⌈ar ≥ ind
         then ar[ind]
         else insert M(ar,ind,D);D
      fi;
proc extend M = (ref flex [ ] M ar, int ind) int:
      (int ub = ⌈ar;
           if ub ≥ ind
              then ub
              else insert M(ar,ind,D)
           fi);
```

The default value for integers is *0* and for booleans the default value equals *false*.

The procedure below computes the sum of the values of an integral function over a finite interval:

```
proc sum = (proc(int) int f, int n) int;
            (int s := 0;
             for x to n do s +:= f(x);
             s);
```

To please mathematicians we allow for the call *sum(λX[E],T)* the non-linear representation $\Sigma_{X\le T}$E.

Linear list manipulation

For a table of function values or an increasing sequence the flexible array is a suitable data structure. For each item to be stored a fixed index in the array is available. In chapter 3.3 on the union theorem we will have growing collections of information where no preassigned mapping into the integers is defined. In this case the linear list is a more appropriate type of data structure.

Let M, *M* and *D* have the meaning as before.

```
mode llM = struct(M info, ref llM tail);

op head = (llM list) M: info of list;
op tail = (llM list) ref llM: tail of list;

proc clear M = (ref llM list) void: list := (D,nil);

proc attach M = (ref llM list, M item) ref llM:
     list := (item, heap llM := list);

proc lookup M = (llM list, M item) bool:
     if head list = item then true
     elif tail list :=: nil then false
     else lookup M (tail list, item) fi;

proc delete M = (ref llM list, M item) ref llM:
     if list :=: nil then nil
     elif head list ≠ item then list := (head list, delete M(tail list,item))
     elif tail list :=: nil then list := (D,nil); nil
     else ref llM hulp = delete M(tail list, item);
               if hulp :=: nil then list := (D,nil); nil
                                else list := hulp
               fi
     fi;
```

The default value of a linear list of items of the mode M equals *(D,nil)*.

The void clause

```
(M item; ref llM domain := list;
 while ref llM (domain) :≠: nil do
        item := head domain; domain := tail domain;
             (S) od);
```

may be represented by

```
          for item over list do S od.
```

The void clause

```
(M item; ref llM domain := list;
 while ref llM (domain) :≠: nil do
      item := head domain;
      domain := if item = stop then nil else tail domain fi;
               (S) od)
```

may be replaced by

```
          for item over list upto stop do S od.
```

1.1.5. MATHEMATICAL NOTATIONS

The larger part of this treatise consists of mathematical text and not of programs. The notations introduced in the preceding sections are used however also within the text; whenever there is printed an expression looking like a phrase in our programming language the corresponding mathematical object is meant.

There are however many concepts having no counterpart in the programming language, like sets, inequalities between functions etc. The present section describes their notations.

The reader should keep in mind the following reserved (but not exclusive) use of some characters:

f,g,h	denote	partial recursive functions in one variable
R,S	denote	partial recursive functions in many variables
i,j,k	denote	indices of programs
x,y,z	denote	arguments of functions or programs
σ,ρ,τ	denote	transformations of programs
A,B,E	denote	subsets of $\mathbb{N}$.

List of special symbols

symbol	{page of definition}	meaning
$\square$		end of proof
$\mathbb{N}$		set of non-negative integers

symbol	{page of definition}	meaning
$\mathbb{Z}$		set of positive and negative integers
$\mathcal{D}f, \mathcal{R}f$	30	domain (range) of the function f
$\mathcal{P}$	30	class of all partial recursive functions (programs)
$\mathcal{R}$	30	class of all total recursive functions (programs)
$\mathcal{P}^n, \mathcal{R}^n$		class of partial (total) recursive functions in n variables
$x \leq y$	30	inequality
$x < y$	30	strict inequality
$f \leq g$	30	inequality between functions
$f < g$	30	strict inequality between functions
$f \sqsubseteq g$	30	inclusion between functions
$f \underline{\propto} g$	30	"almost everywhere" inequality between functions
$f \underline{\propto} g(A)$	30	idem, relativized to a set $A \subseteq \mathbb{N}$
$f \mid A, \varphi_i \mid A$		restriction of a function (program) to $A \subseteq \mathbb{N}$
f *cheap* g	85	"f is cheaper than g"
f *cheapv* g	86	"f is cheaper than the values of g"
f *ncomp* g	69	"g cannot be computed within time f"
f *ncompv* g	69	"the values of g cannot be computed within time f"

These four relations again may be relativized to a subset $A \subset \mathbb{N}$.

$\forall, \exists$		unbounded quantifiers
$\overset{\infty}{\forall}_x$	30	for all x except finitely many
$\overset{\infty}{\exists}_x$	30	there exists infinitely many x
$\nexists_x$		there exists no x
$\leq_m, \leq_1$	43	many-one (one-one) reducibility
$\equiv$	43	recursive isomorphism
$\# A$	31	number of elements in set A
$[k,l], [k,l)$		segment $k \leq x \leq l$ $(k \leq x < l)$
$\in, \notin$		is element of (is no element of)

symbol	{page of definition}	meaning
$\emptyset$		empty set
$\Sigma_n, \Pi_n, \Delta_n$	45	degrees in the arithmetical hierarchy
C_t, F_t	64	complexity classes
C_t^W, F_t^W	65	weak complexity classes
H_R, G_R	65	honesty classes
C_t^E, F_t^E	68	complexity classes modulo sets of exceptional points
$H_r^{\wedge}, G_r^{\wedge}$	66	modified honesty classes
$SC_t, SC1_t, CT_1$	100	other types of resource bound classes
$(x_i)_i, (f_i)_i, (E_i)_i$	35	sequence of integers, functions, sets
$(\Upsilon_i)_i$	35	measured set
φ_i	34	i-th program in effective enumeration
Φ_i	39	run-time of i-th program
φ_i^n, Φ_i^n	35	idem, for n-variable programs
E	67	class of sets of exceptional points
A	95	general acceptance relation
Cpl, *Hon*, *Cplex*, *Honex*, $\underline{\Gamma}$, *Scpl*, *Scpl1*, *Tcpl*	97-100	specific acceptance relations
$F_S^A(t), F^A(t), F(t)$	95	strong abstract resource-bound class
$F_W^A(t), F_W(t)$	95	weak abstract resource-bound class
$G_W^A(t), G_S^A(t), G(t)$	96	abstract resource-bound class of programs
$H_W^A(t), H_S^A(t), H(t)$	96	abstract resource-bound class of functions
$(\alpha_i)_i$	104	measured set of run-times derived from acceptance relation
halts, *non-empty*, *finite*, *bound*, *cofinite*, *empty*	43-47	standard reference sets in the arithmetic hierarchy
$\Omega\, \underline{C}$	49	index set for a class of functions
$f^{(k)}(x)$		$f^{(0)}(x) = x; \quad f^{(k+1)}(x) = f(f^{(k)}(x))$.

Within descriptions of algorithms designated "informal" we use (among others) the following deviations of our own formalism:

(i) If S denotes some sequence of symbols which describes informally some computation or computable value then $\underline{¢}\ S\ \underline{¢}$ denotes a closed clause within our programming language whose elaboration yields the computation or value intended.

For example if A is a recursive set then $\#\{x \in A \mid x \leq n\}$ clearly is a computable integer; consequently $\underline{¢}\ \#\{x \in A \mid x \leq n\}\ \underline{¢}$ denotes a closed integral clause yielding the number of elements in A whose values are less than or equal to n.

(ii) If B(x) denotes some predicate then the expression

$$max\{\underline{if}\ B(X)\ \underline{then}\ F(X)\ \underline{else}\ 0\ \underline{fi} \mid A \leq X \leq B\}$$

can be represented also by

$$max\{F(X) \mid A \leq X \leq B\ \underline{and}\ B(X)\}$$

or even

$$\max_{A \leq X \leq B} \{F(X) \mid B(X)\}.$$

Finally if $B(X) = \lambda x[\underline{true}]$ we replace

$$\max_{A \leq X \leq B} \{F(X) \mid B(X)\}\ \ by\ \ \max_{A \leq X \leq B} \{F(X)\}.$$

Analogous representations are used for minima.

CHAPTER 1.2

RECURSIVE FUNCTIONS AND EFFECTIVE ENUMERATIONS

1.2.1. MATHEMATICAL CONVENTIONS *)

By a function we mean (unless stated otherwise) a *partial recursive function* from the set $\mathbb{N}$ of non-negative integers (including zero) into itself. Functions which are defined for all arguments are called *total*. $\mathcal{P}(\mathcal{R})$ denotes the collection of all (total) functions. The set of all arguments x for which f(x) is defined, the *domain* of f, is denoted $\mathcal{D}f$. We write $f(x) < \infty$ $(f(x)=\infty)$ for $x \in \mathcal{D}f$ $(x \notin \mathcal{D}f)$. Equality of functions $f = g$ denotes extensional equality ($\mathcal{D}f = \mathcal{D}g$ and $\forall x[x \in \mathcal{D}f \Rightarrow f(x) = g(x)]$).

The inequality $f \le g$ means $\mathcal{D}f \supseteq \mathcal{D}g$ and $g(x) \ge f(x)$ for all $x \in \mathcal{D}g$. Strict inequality $f < g$ means $\mathcal{D}f \supseteq \mathcal{D}g$ and $g(x) > f(x)$ for all $x \in \mathcal{D}g$. If $\mathcal{D}f \supseteq \mathcal{D}g$ and $g(x) = f(x)$ for $x \in \mathcal{D}g$ then we write $g \sqsubseteq f$.

The range of a function f is denoted $\mathcal{R}f$. For finite k the inequality $k \le \infty$ is taken true whereas $\infty \le k$ is taken false. The inequality $\infty \le \infty$ is also considered true.

If P is some predicate then we write $\overset{\infty}{\forall}x[P(x)]$ for "P(x) holds for all except finitely many x" and $\overset{\infty}{\exists}x[P(x)]$ for "there exist infinitely many x such that P(x)". Using these notations we define the following "almost everywhere" inequality between functions:

$$f \underline{\propto} g \quad \underline{\text{iff}} \quad \overset{\infty}{\forall}x[f(x) \le g(x)].$$

This inequality may be relativized to a subset $A \subseteq \mathbb{N}$; $f \underline{\propto} g$ (A) means $\overset{\infty}{\forall}x[x \in A \Rightarrow f(x) \le g(x)]$. Note that $f \underline{\propto} g$ whenever $\mathcal{D}g$ is finite. $\mu z[P(z)]$ denotes "the least z such that P(z)".

We use a fixed recursive pairing function $\langle x,y\rangle$ with coordinate projections π_1 and π_2; $\pi_1\langle x,y\rangle = x$; $\pi_2\langle x,y\rangle = y$; $\langle \pi_1 x, \pi_2 x\rangle = x$. Moreover, $\langle x,y\rangle$ is increasing in both arguments and consequently $\langle 0,0\rangle = 0$. Using

*) The conventions introduced in this section cover a few notations not introduced in chapter 1.1, although not all notations are new. Some notations, introduced already in chapter 1.1, are repeated for the sake of the readers who have skipped chapter 1.1.

this pairing function many-variable functions are introduced in a formalism consisting of one-variable functions.

We let ε (zero) denote the function which is everywhere undefined (zero).

If R is a two-variable function and t a single variable function then the function $\lambda x[R(x,t(x))]$ is denoted $R\Box t$.

It is a practice in recursion theory to identify (recursive) sets, predicates and total 0-1 valued functions using the interpretation of characteristic functions and the interpretation 0 = *true*, 1 = *false* or vice versa. We abstain from these identifications. If a Boolean function is intended the truth values *true* and *false* are used explicitly, and for recursive sets the notations from set theory are used.

If A is some set then #A denotes the number of elements in A. #A = ∞ if A is not finite.

In the sequel "increasing" will always mean "strictly monotonically increasing" and "non-decreasing" will stand for "monotonically non-decreasing".

1.2.2. THE ORIGIN OF MACHINE-INDEPENDENT RECURSION THEORY

The concept of a recursive function is one of the important results of 20-th century mathematics. During the years 1930-5 a number of distinct formalized definitions of "computability" were given. Afterwards these different formalisms were proved to be equivalent; i.e. each formalism yields the same class of computable functions.

The best known formalisms are those given by KLEENE and TURING.

KLEENE considers functions determined by defining systems of equations. In his formalism the recursive functions form the smallest class of functions containing a few base functions (zero, successor and projections) which is closed under the schemes of substitution, primitive recursion and minimalization. If the scheme of minimalization is excluded one defines this way the class of primitive recursive functions. Reading the defining equations from left to right yields a method to evaluate a given function at a given argument; consequently all functions in the KLEENE formalism are indeed computable.

The formalism introduced by TURING is based on the description of an abstract machine (Turing machine) which performs a computation. Although the computation is described in terms of physical entities (a machine operating on a two-way, potentially infinite tape, consisting of squares upon which symbols are printed and erased) the input-output relationship determined by this machine can be interpreted to be a function from $\mathbb{N}$ into $\mathbb{N}$. Again the origin of the function indicates that it is in fact computable.

For neither of the two formalisms it is clear that each function which is felt to be computable is computable in the sense of the formalism. An impression of the power of the two formalisms results from trying to prove their equivalence. Such proofs can be found in KLEENE, *Introduction to Meta-mathematics* [K152] or DAVIS, *Computability and Unsolvability* [Da58].

The proof of the equivalence of the formalism yields some very important corollaries. In both formalisms there exists a "canonical" method to enumerate the programs c.q. systems of defining equations, this way enumerating the collection of computable functions. This makes it possible to introduce the universal function u which is defined by

u(i,x) = "the value of the i-th program in the list at argument x".

This universal function u itself can be shown to be recursive. Consequently one may replace the collection of all Turing machines by a single one (*the universal machine*) which operates on encodings of all other machines.

A second important corollary is the definition of the so-called *Kleene predicate* T. This is a primitive recursive (and hence total) boolean function which satisfies

T(i,x,z) = "z is an encoding of the complete computation of the i-th machine at argument x".

By use of this predicate it is possible to describe formally some constructions which interfere with computable functions at the level of the computations determining the function values.

A third result which is easily derived from the technical tools developed for the preceding results is the so-called *s-n-m-theorem*. If R is a recursive function in two variables then for each fixed k, R(k,-) is a single-variable function which is computable. Now both R and R(k,-) have an index in the canonical enumeration. The s-n-m-theorem expresses that an index for R(k,-) can be computed recursively from an index for R and k.

After having worked through the proofs of the preceding results the reader may be convinced that each function which he feels to be computable, is formally computable within the formalisms. Still this is a mathematically unprovable assertion which is known under the name of "Thesis of CHURCH": *Each computable function is recursive.*

The two formalisms described above are not usable to represent complicated algorithms. This has led to the use of informal descriptions of algorithms which are "clearly" recursive. This practice is justified by invoking the thesis of CHURCH. However, using this thesis this way is called "inessential use" since one believes that a formalized description could be written down, given the time and the paper.

Each of the formalisms for the recursive functions mentioned above leads to a so-called *machine-dependent theory*. It is hardly possible to separate between the mathematical content of an algorithm and its "physical" implementation.

The base for a machine-independent approach was given by ROGERS [Ro58] who introduced the concept of an *effective enumeration*. In this formalism the basic structure is given by the universal machine and the s-n-m-function. There exists however no equivalent to the KLEENE predicate. Still it is possible to derive the so-called *recursion theorem* within this framework.

The concept of an effective enumeration was extended by BLUM [B167] to the concept of a *complexity measure*. In this formalism an analogue for the KLEENE predicate is given. In the resulting mathematical system it is fairly good possible to discuss computable functions at the level of their computations; for example the construction of a "dovetailed" computation can be formalized. Still it is felt by the author that in defining the so-called *transformations of programs* the formal discussion is interrupted by an argumentation like: "look at this function; clearly it is computable and consequently there exists an index for it in the enumeration; let i be such an index ...". In order to formalize this argumentation the operator *index* was introduced in the preceding chapter.

In the sequel of this treatise we present a survey of *machine-independent recursion theory*.

1.2.3. ABSTRACT EFFECTIVE ENUMERATIONS AND THE RECURSION THEOREM

{... *and, lo, a great multitude, which no man could number, of all nations, and kindreds, and people, and tongues, stood before the throne, ...*

Revelation VII,9}.

DEFINITION 1.2.1. An *effective enumeration* $(\varphi_i)_i$ is a sequence of partial recursive functions (in one variable) called *programs*, which satisfies the following properties:

(0) Each partial recursive function $f\colon \mathbb{N} \to \mathbb{N}$ occurs somewhere in the sequence $(\varphi_i)_i$.

(1) [Universal machine]. There exists an index u such that $\forall x,y[\varphi_u(<x,y>) = \varphi_x(y)]$.

(2) [s-n-m axiom]. There exists a total recursive function snm such that

$$\forall i,x,y[\varphi_{snm(i,x)}(y) = \varphi_i(<x,y>)].$$

Although our programs φ_i are single-variable functions by definition we introduce many-variable functions in our enumeration by using the pairing function. An occasional super-index indicates use of this interpretation: for example $\varphi_i^2(x,y)$ equals $\varphi_i(<x,y>)$.

By definition the programs φ_i are nothing but functions. We want to be able to separate the intentional object of an abstract computing process from its extensional meaning which is the computed function. Therefore we denote the function computed by the program φ_i by $\Lambda\varphi_i$. This symbol Λ may be omitted in many circumstances where it is clear that the numerical value is intended. In particular $\varphi_i(x)$ denotes always the numerical result. Also in equalities between programs and/or functions the Λ is omitted. Note that equality between programs means extensional equality; there is no concept like "program equivalence" in our theory.

The above interpretation of the symbol φ_i occurs in papers by J. HARTMANIS and A. BORODIN. An alternative interpretation which is used by M. BLUM, A. MEYER, P. YOUNG and others where φ_i denotes the function computed by program i and where no distinction is made between index and program, is less suitable for this treatise. In part 3 we have situations where an index encodes more information than just a program, and in this situation a three-leveled approach (index, program and function) is needed.

The fact that φ_i computes f is also expressed by saying "i is an index for f".

DEFINITION 1.2.2. A *transformation of programs* is a total recursive function (working on indices of programs).

In chapter 1.1 the background of this concept was fully explained. Transformations of programs are defined by implicit use of the s-n-m axiom. For example the function $\varphi_i^2(y,x)$ being recursive in i, x and y there exists by the s-n-m axiom a total function σ for which

$$\varphi^2_{\sigma(i)}(x,y) = \varphi_{\sigma(i)}(<x,y>) = \varphi_i(<y,x>) = \varphi_i^2(y,x).$$

Using the formalism of section 1.1.2 a formal definition of σ becomes

proc σ = *(int i) int: index(int x) int:* Λ *comp(i,pair(π2x,π1x))*

or using the mathematical representation of 1.1.3

$$\varphi_{\sigma(i)}(x) \Leftarrow \varphi_i(<\pi_2 x, \pi_1 x>).$$

DEFINITION 1.2.3. By a *sequence of programs (functions)* we understand a sequence of programs (functions) which is enumerated by a transformation of programs. Hence if σ is a transformation of programs then $(\varphi_{\sigma(i)})_i$ is a sequence of programs and $(\Lambda\varphi_{\sigma(i)})_i$ is a sequence of functions.

For the concept of a measured set we have two equivalent definitions:

DEFINITION 1.2.4. A *measured set* γ is a recursive ternary predicate such that for each i and x there exists at most one value y for which $\gamma(i,x,y)$ = *true*.

DEFINITION 1.2.5. A *measured set* γ is a sequence of functions $(\gamma_i)_i$ with the property that the ternary predicate $\gamma_i(x) = y$ [*)] is recursive in i, x and y.

*) In order to be consistent with chapter 1.1 we should write $\gamma_i(x)$ *eq* y instead of $\gamma_i(x) = y$.

Clearly the two definitions describe the same concept. From the ternary predicate γ one derives the sequence (γ_i) by taking $\gamma_i = \varphi_{\sigma(i)}$, the transformation σ being defined by

$$\varphi_{\sigma(i)}(x) \Leftarrow \mu z[\gamma(i,x,z)].$$

Conversely the ternary predicate γ is implicitly present in the second definition, and it is clear that $\gamma_i(x) = y$ for at most one value y.

DEFINITION 1.2.6. The transformation of programs σ is called a *measured transformation* if the sequence $(\varphi_{\sigma(i)})_i$ is a measured set.

There exist sequences of functions which are not a measured set. For example the sequence $(\varphi_i)_i$ is not measured since the predicate $\varphi_i(x) = y$ is not recursive (cf. chapter 1.4).

The recursion theorem of KLEENE states that every transformation of programs has a fixed point, i.e. a program extensionally equivalent with its image. Moreover, the index of the fixed point depends uniformly on further parameters.

THEOREM 1.2.7. [Recursion theorem]. Let σ be a transformation of programs. Then there exists an index j such that $\varphi_j = \varphi_{\sigma(j)}$.

PROOF. Let ρ be defined by:

$$\varphi_{\rho(j)}(x) \Leftarrow \varphi_{\varphi_j(j)}(x)$$

and let k be an index for the function $\lambda i[\sigma(\rho(i))]$. Then $j = \rho(k)$ is the requested fixed point since for every x

$$\varphi_j(x) = \varphi_{\rho(k)}(x) = \varphi_{\varphi_k(k)}(x) = \varphi_{\sigma(\rho(k))}(x) = \varphi_{\sigma(j)}(x). \quad \Box$$

Note that the above proof seems to have no intuitive meaning at all. (The reader may convince himself of this fact by closing the book and trying to repeat the argumentation).

THEOREM 1.2.8. [Uniform recursion theorem]. Let $\sigma\colon \mathbb{N}^2 \to \mathbb{N}$ be a transformation of programs. Then there exists a transformation τ satisfying for each j $\varphi_{\sigma(\tau(j),j)} = \varphi_{\tau(j)}$.

PROOF. Let $\rho\colon \mathbb{N}^2 \to \mathbb{N}$ be defined by

$$\varphi_{\rho(i,j)}(x) \Leftarrow \varphi_{\varphi_i^2(i,j)}(x).$$

Take for k an index such that

$$\varphi_k^2(i,j) = \sigma(\rho(i,j),j).$$

Then the transformation $\tau = \lambda j[\rho(k,j)]$ satisfies the fixed point conditions. For each j and x we have

$$\varphi_{\tau(j)}(x) = \varphi_{\rho(k,j)}(x) = \varphi_{\varphi_k^2(k,j)}(x) = \varphi_{\sigma(\rho(k,j),j)}(x) =$$

$$= \varphi_{\sigma(\tau(j),j)}(x). \quad \square$$

1.2.4. THE CONCEPT OF AN OPERATOR

DEFINITION 1.2.9. A *total effective operator* Γ is a transformation of programs σ with the following properties (writing $\Gamma(\varphi_i)$ for $\varphi_{\sigma(i)}$):

(1) [operator] Γ preserves functional equality: $\varphi_i = \varphi_j \Rightarrow \Gamma(\varphi_i) = \Gamma(\varphi_j)$,

(2) [totality] φ_i total $\Rightarrow \Gamma(\varphi_i)$ total,

(3) [continuity] if $\Gamma(\varphi_i)(x) = y$ then there exists a finite set $F \subset \mathcal{D}\varphi_i$ so that $\Gamma(\varphi_i|F)(x) = y$ and, moreover, for any index j such that $\varphi_j|F = \varphi_i|F$ one has $\Gamma(\varphi_j|F)(x) = y$ also.

If F is a finite set such that $\varphi_i|F$ completely determines the value of $\Gamma(\varphi_i)(x)$ then we say that the support of (the computation of) $\Gamma(\varphi_i)$ on x is contained within F. If a total effective operator Γ and a total function φ_i are given one can effectively compute a finite set F containing the support of $\Gamma(\varphi_i)$ on x.

One should visualize a total effective operator as a "procedure" which computes $\Gamma(t)$ using a program for t as a subroutine; in computing $\Gamma(t)(x)$ (which computation always terminates if t is total) the values of t at a finite set of arguments of t (the support of $\Gamma(t)$ on x) are used; the result only depends on these values and not on the way these values are computed. (In fact this way the support of $\Gamma(\varphi_i)$ on x is determined.)

The procedure may be based on paralellism; if the procedure is purely sequential one gets a smaller subclass of the total effective operators (the so-called *subroutine operators* [Sy 71]).

By dovetailing a procedure using paralellism can be translated into a sequential procedure; however, the computations of t(x) should be dovetailed also. Another problem is that in general one cannot compute a minimal support of a computation of such an operator. To illustrate this consider the example below:

Example 1.2.10. Let σ_k be defined

```
$\varphi_{\sigma_k(i)}(x) \Leftarrow$ (par begin if $\varphi_i(0) < \infty$ and $\varphi_k(k) < \infty$ then goto found fi,
                          if $\varphi_i(0) < \infty$ and $\varphi_i(1) < \infty$ then goto found fi
                 end;
          found: 0);
```

Clearly σ_k satisfies for each k the conditions (1), (2) and (3) thus defining a total effective operator Γ_k. However, to decide whether the minimal support of $\Gamma_k(t)$ on x is {0} or {0,1} one must first solve the halting problem which is unsolvable (cf. chapter 1.4).

The more general concepts of an operator, such as recursive operator, partial recursive operator, and general recursive operator [Ro 67] are not used in this treatise.

CHAPTER 1.3

COMPLEXITY MEASURES

As indicated in the preceding chapter the concept of an effective enumeration is not powerful enough to treat computable functions at the level of their computations. There is no analogue for the KLEENE predicate.

To create a more powerful formalism M. BLUM has introduced for each program φ_i a *step-counting function* (or *run-time*) Φ_i. One should think of $\Phi_i(x)$ to be the "amount of resource used by the computation of $\varphi_i(x)$". The behaviour of the functions $(\Phi_i)_i$ is regulated by the two so-called *Blum axioms*. The pair $\Phi = ((\varphi_i)_i,(\Phi_i)_i)$ is called a *complexity measure* or *Blum measure*.

A complexity measure may be considered to be an abstract universe of computation. The working of the machinery is unknown; moreover, programs cannot be freely combined although alternative programs for the intended combinations exist in the enumeration. However the formalism allows the interruption of computations taking to much resource.

<u>DEFINITION 1.3.1</u>. A *complexity measure* Φ is a pair $((\varphi_i)_i,(\Phi_i)_i)$ consisting of two sequences of (partial recursive) functions satisfying the following axioms:

(0) The sequence $(\varphi_i)_i$ is an effective enumeration of partial recursive functions.

(1) For each i $\mathcal{D}\varphi_i = \mathcal{D}\Phi_i$.

(2) The sequence $(\Phi_i)_i$ is a measured set (i.e. $\Phi_i(x) = y$ is decidable).

As before with the programs we should separate the intentional run-times Φ_i from the numerical functions which they are by definition. Again the later objects are denoted $\Lambda\Phi_i$. Contrasting to the preceding section it is not permitted to suppress occurrence of the symbol Λ freely. For example in determining the validity of $\Phi_i(x) = y$ one wants to use the decision procedure given by the second Blum axiom, which terminates, whereas computation of $\Phi_i(x)$ may diverge. For this reason we reserved in chapter 1.1 the notation $\Phi_i(x)$ for the intentional object; the numerical value is denoted $\Lambda\Phi_i(x)$. This convention leads to many occurrences of the symbol Λ in circumstances where the numerical value of the run-time is used as argument for some computation. The denotation $R\square\Phi_i$ from $\lambda x[R(x,\Lambda\Phi_i(x))]$ is an exception to this rule, which is motivated by the frequent occurrences of such functions as index in some other expression. Within our text we omit in fact many occurrences of Λ in circumstances where the intentionality of the run-time is irrelevant.

Note that the decision procedure for $\Phi_i(x) = y$ is an analogue for the Kleene predicate $T(i,x,y)$ if we consider "the encoding of the computation of the i-th program of argument x" to be the run-time of the i-th program at x. For the effective enumeration of all Turing machines there exist two well-known examples of complexity measures. In the so-called *Turing time-measure*, $\Phi_i(x)$ equals "the number of steps" taken by machine i at argument x, a "step" denoting the reading and/or printing of a single symbol followed by an internal transition and a possible replacement of the reading-head over one tape square. In the *Turing space measure* $\Phi_i(x)$ equals the number of tape squares used during the computation. In the latter case this number may be finite also for a divergent computation and in order to satisfy axiom (1) the function $\Phi_i(x)$ must be set in diverge if this (decidable) phenomenon occurs.

Consider some more or less concrete instruction code for some computer model, which is strong enough to allow computation of all recursive functions. One should like to have in these circumstances a natural complexity measure, with φ_i being the i-th program written in the instruction code (using some encoding of programs by integers) and with $\Phi_i(x)$ being the number of elementary instructions executed during computation of the i-th program at argument x. In practice there are almost always hidden snags in this definition. Many instruction codes permit situations leading to divergent computations. As an example one might consider the divergent ALGOL 60 program

```
begin switch S := S[1]; goto S[1] end
```

in which occurs a non-terminating but (at the level of the ALGOL 60 semantics) indecomposable statement. At a more machine-oriented level instructions using indirect addressing (à la KNUTH [Kn 68 - 2.2.2, Ex.3]) or indirectly executed instructions (like the DO-instruction in the EL X8 machine code [EL 66]) are examples of instructions which may fail to terminate. This situation becomes worse if by the use of microprogramming the power of the instruction code is increased. Consider a computer equipped with an instruction which computes a zero with integral coordinates for a many-variable polynomial with integral coefficients. As is indicated by the unsolvability of HILBERT's tenth problem [Da 73] this instruction has an unsolvable halting problem.

Although the reader will look in vain for a formal proof in this treatise the following assertion is crucial at this place.

ASSERTION 1.3.2. There exists a complexity measure.

In fact the Turing machine measures mentioned above are well-defined Moreover, computer-model inspired measures without non-terminating instructions in their instruction codes exist also; for an example of a formally defined type of "register machines" we refer to [SS 63] or [Hm 71]. In a recent textbook E. ENGELER [En 73] uses such a model of computation as a foundation for recursion theory.

We conclude this section with some remarks one the strength of the two Blum axioms.

In the first place the two Blum axioms are independent. Taking $\Phi_i = \varphi_i$ the first axiom is satisfied but the second is not since $(\varphi_i)_i$ is not a measured set (cf. chap.1.4). If we take Φ_i = zero for each i then $(\Phi_i)_i$ is measured but the first axiom fails.

The existence of a single complexity measure Φ shows that it is possible to extend each effective enumeration $(\varphi_i)_i$ to a complexity measure by "borrowing" the run-times from the measure Φ. Let u be the universal program for $(\psi_i)_i$ i.e. $u(i,x) = \psi_i(x)$. Since u is recursive there exists an index j for u in the enumeration $(\varphi_i)_i$. Hence $\psi_i(x) = \varphi_j^2(i,x)$. Now define $\Psi_i(x) = \Phi_j^2(i,x)$. It is not difficult to prove that $((\psi_i)_i,(\Psi_i)_i)$ is indeed a complexity measure.

From the fact that the sequence $(\Phi_i)_i$ is a measured set one finds a transformation ρ such that $\Phi_i = \varphi_{\rho(i)}$. One defines ρ by

$$\varphi_{\rho(i)}(x) \Leftarrow \mu z[\Phi_i(x) = z].$$

Moreover, the following predicates are seen to be recursive: $\Phi_i(x) \neq z$, $\Phi_i(x) < z$, $\Phi_i(x) \leq z$, $\Phi_i(x) > z$ and $\Phi_i(x) \geq z$. For the predicates $\Phi_i(x) \leq \Phi_j(y)$, $\Phi_i(x) < \Phi_j(y)$ and $\Phi_i(x) = \Phi_j(y)$ the second Blum axiom suggests a computation method consisting of "running the two machines in parallel to find out which one terminates first"; see chapter 1.1 for a formal definition.

If $\Phi = ((\varphi_i)_i, (\Phi_i)_i)$ and $\Phi' = ((\varphi_i)_i, (\Phi'_i)_i)$ are two measures on the same effective enumeration then the following combinations yield complexity measures also:

(i) $\lambda x[\Lambda\Phi_i(x) + \Lambda\Phi'_i(x)]$

(ii) $\lambda x[g(\Lambda\Phi_i(x))]$ provided g is a total function such that there exists a total h satisfying $g(x) = y \Rightarrow x \le h(y)$.

(iii) $\lambda x[\underline{if}\ \langle i,x\rangle \in A\ \underline{then}\ \Lambda\Phi_i(x)\ \underline{else}\ \Lambda\Phi'_i(x)\ \underline{fi}]$ supposed $A \subset \mathbb{N}$ is recursive.

(iv) $\lambda x[\underline{if}\ \langle i,x\rangle \in B\ \underline{then}\ 0\ \underline{else}\ \Lambda\Phi_i(x)\ \underline{fi}]$ supposed $B \subset \mathbb{N}$ is recursive and $\varphi_i(x) < \infty$ for $\langle i,x\rangle \in B$.

These constructions show how measures may be combined, compressed, expanded, and conditionally selected, or even set to zero at a recursive set of converging computations. Application of these and similar constructions may lead to "pathological" examples for complexity measures. For example it is not difficult to design a measure where zero and $\lambda x[x+1]$ both are computed free of charge, whereas their composition $\lambda x[1]$ cannot be computed within polynomially bounded time.

One of the important unsolved problems in abstract complexity theory consists of the characterization of "naturalness" of measures. Several proposed extensions of the Blum axioms have been found to fail in this respect. The reader is referred to [Hm 73].

CHAPTER 1.4

SOME CONCEPTS FROM RECURSION THEORY

In this chapter we discuss the concepts of unsolvability, and many-one (one-one) reducibility. Furthermore we mention the arithmetical hierarchy, indicating some standard reference sets in this hierarchy. Finally the recursive enumerable sets are defined.

As was suggested several times before, not all sets and functions are recursive. The best known example of an unsolvable problem is the so-called halting problem. Let the set *halts* be defined by

$$halts = \{i \mid \varphi_i(i) < \infty\}.$$

Then *halts* is not recursive:

PROPOSITION 1.4.1. For no index n one has

$$\varphi_n = \lambda x[\underline{if}\ \varphi_x(x) < \infty\ \underline{then}\ 0\ \underline{else}\ 1\ \underline{fi}].$$

PROOF. Suppose n exists, then the following function is total recursive:

$$f = \lambda x[\underline{if}\ \varphi_n(x) = 0\ \underline{then}\ \varphi_x(x)+1\ \underline{else}\ 0\ \underline{fi}]$$

However, if j is an index for f then f(j) = f(j) + 1. Contradiction. □

The fact that not all problems are recursive has lead to a number of reducibility concepts. One says that problem 1 is reduced to problem 2 if a solution of problem 2 yields a solution to problem 1. We describe two reducibility concepts.

DEFINITION 1.4.2. Let A and B be two (not necessarily recursive) subsets of $\mathbb{N}$. We say that the total recursive function s *m-reduces A to B* if for all x $s(x) \in B$ *iff* $x \in A$; notation $A \le_m B$ (by s). If s is, moreover, a 1-1 function we say that s *1-reduces A to B*; notation $A \le_1 B$. If s is a bijection *(recursive permutation)* then A and B are called recursively isomorphic; notation $A \equiv B$.

It is not difficult to show that $\leq_m$ and $\leq_1$ are pre-orderings on the power-set $P(\mathbb{N})$. Defining $A \equiv_1 B$ ($A \equiv_m B$) by $A \leq_1 B$ _and_ $B \leq_1 A$ ($A \leq_m B$ _and_ $B \leq_m A$) $\equiv$, $\equiv_1$ and $\equiv_m$ become equivalence relations. The equivalence classes modulo $\equiv_1$ ($\equiv_m$) are called 1-degrees (m-degrees).

The relation between $\equiv$ and $\equiv_1$ follows from the following theorem:

THEOREM 1.4.3. [MYHILL isomorphism principle]. $A \equiv B$ _iff_ $A \equiv_1 B$.

For a proof the reader may consult ROGERS [Ro 67], or the appendix. The proof involves a method to construct from two injective functions f and g such that $A \leq_1 B$ (by f) and $B \leq_1 A$ (by g) a recursive permutation s such that $A \equiv B$ (by s). The algorithm for s is a nice example of an essentially non-terminating algorithm, which can hardly be represented adequately by the usual mathematical expressions. (The reader should compare the representation in the appendix with the one given by ROGERS.)

EXAMPLE 1.4.4. Let *total* be defined by

$$total = \{i \mid \mathcal{D}\varphi_i = \mathbb{N}\}.$$

then *halts* $\leq_m$ *total* by σ when σ is defined by

$$\varphi_{\sigma(i)}(x) \Leftarrow \varphi_i(i).$$

Note that $A \leq_m B$ and B recursive implies that A is recursive.

PROPOSITION 1.4.5. The sequence $(\varphi_i)_i$ is not a measured set.

PROOF. Let $C = \{<i,x,y> \mid \varphi_i(x) = y\}$. Then *halts* $\leq_m C$ by s if s is defined from σ as follows:

$$\varphi_{\sigma(i)}(x) \Leftarrow \underline{if}\ \varphi_i(i) < \infty\ \underline{then}\ 0\ \underline{else}\ \infty\ \underline{fi}$$

(note that the else part is never executed) and

$$s = \lambda i[<\sigma(i),0,0>].$$

Consequently if C is recursive then so is *halts*. Contradiction. □

The *arithmetical hierarchy* classifies sets by the structure of their descriptions by defining formulas. By a defining formula we understand an expression of the form

$$F = Q_1y_1,Q_2y_2,\dots,Q_ky_k[E]$$

where E is some total recursive predicate with free variables $x,y_1,y_2,\dots$ $\dots,y_k$ and $Q_1,\dots,Q_k$ are (unbounded) existential or universal quantifiers. Consequently all free variables in E except x are bounded in F. The set defined by F is the set $\{x \mid F\} = \{x \mid Q_1y_1,Q_2y_2,\dots,Q_ky_k[E]\}$.

EXAMPLES: $\mathit{halts} = \{x \mid \exists y[\Phi_x(x) = y]\}$,

$$\mathit{total} = \{x \mid \forall y\exists z[\Phi_x(y) = z]\}.$$

Expressions like F are called expressions in *prenex normal form*. The quantifier block $Q_1y_1,Q_2y_2,\dots,Q_ky_k$ is called the *prefix* of F. It is not difficult to prove that without loss of generality we may assume that the quantifiers are alternating: if $Q_i = \exists$ then $Q_{i+1} = \forall$ and vice versa. Otherwise two equal quantifiers may be contracted into a single one: e.g.
$\dots,\exists y_1,\exists y_2,\dots[P(\dots,y_1,y_2,\dots)]$ is replaced by
$\dots,\exists y,\dots[P(\dots,\pi_1y,\pi_2y,\dots)]$ etc.

We say that F is of type Π_k if the prefix of x consists of k alternating quantifier blocks starting with $\forall$; similarly F is of type Σ_k if the prefix of k consists of k alternating quantifier blocks starting with $\exists$.

A set defined by a Σ_k (Π_k) expression is called a Σ_k (Π_k)-set. Sets which are both Σ_k and Π_k are called Δ_k-sets.

It is clear that a Σ_k-set or a Π_k-set is a Δ_n-set for $n > k$; this is proved by the use of "dummy" variables.

If Ξ denotes one of the types Π_k, Σ_k or Δ_k then $\underline{\Xi}$ denotes the class of all Ξ-sets. We have the following inclusions.

$$\begin{tikzcd}
 & \underline{\Pi}_1 \arrow[dr, dash, "\subseteq"] & & \underline{\Pi}_2 \arrow[dr, dash, "\subseteq"] & & \underline{\Pi}_3 \arrow[dr, dash, "\subseteq"] & & {} \\
\underline{\Delta}_1 \arrow[ur, dash, "\subseteq"] \arrow[dr, dash, "\subseteq"'] & & \underline{\Delta}_2 \arrow[ur, dash, "\subseteq"] \arrow[dr, dash, "\subseteq"'] & & \underline{\Delta}_3 \arrow[ur, dash, "\subseteq"] \arrow[dr, dash, "\subseteq"'] & & \underline{\Delta}_4 \arrow[ur, dash, "\subseteq"] \arrow[dr, dash, "\subseteq"'] & \\
 & \underline{\Sigma}_1 \arrow[ur, dash, "\subseteq"'] & & \underline{\Sigma}_2 \arrow[ur, dash, "\subseteq"'] & & \underline{\Sigma}_3 \arrow[ur, dash, "\subseteq"'] & & {}
\end{tikzcd}$$

FACT 1.4.6. [Hierarchy theorem]. All inclusions in the above diagram are proper inclusions.

For a formal treatment of the arithmetical hierarchy and a proof of the hierarchy theorem the reader is referred to ROGERS [Ro 67].

It is not difficult to show that the complement of a Σ_k-set is a Π_k-set and vice versa. Moreover if B is a Ξ-set and $A \leq_m B$ then A is Ξ also.

DEFINITION 1.4.7. A Ξ-set is called *Ξ-complete* if every Ξ-set B can be m-reduced to A.

Clearly the complement of a complete set is again complete.

PROPOSITION 1.4.8. A Δ_1-set is recursive.

PROOF. Let $A = \{x \mid \forall y[P(x,y)]\} = \{x \mid \exists z[Q(x,z)]\}$, P and Q total recursive. Now $\mathbb{N} \setminus A = \{x \mid \exists y[\underline{not}\ P(x,y)]\}$. Since $\mathbb{N} = A \cup \mathbb{N} \setminus A$ we have

$$\forall x \exists y[\underline{not}\ P(x,y)\ \underline{or}\ Q(x,y)].$$

Hence the following function g is total

$$g = \lambda x[\mu z[\underline{not}\ P(x,z)\ \underline{or}\ Q(x,z)]].$$

Now we can define A by

$$A = \{x \mid Q(x,g(x))\}.$$

which shows A to be recursive. □

The sets in Σ_1 are called *recursively enumerable sets*. This name is explained by the following proposition:

PROPOSITION 1.4.9. The following assertions are equivalent:

(i) A is recursively enumerable
(ii) $A = \mathcal{D}f$ for some $f \in \mathcal{P}$
(iii) $A = \mathcal{R}f$ for some $f \in \mathcal{R}$ or $A = \emptyset$
(iv) $A = \mathcal{R}f$ for some $f \in \mathcal{P}$.

PROOF: (i) $\Rightarrow$ (ii). Let $A = \{x \mid \exists y[E(x,y)]\}$, E total recursive, then for $f = \lambda x[\mu y[E(x,y)]]$, $A = \mathcal{D}f$.

(ii) $\Rightarrow$ (iii). Trivial for $A = \emptyset$. Otherwise let $A = \mathcal{D}f$ then $\mathcal{D}f$ is not empty. Let i be an index for f; the following expression yields a "first" element of $\mathcal{D}f$:

$$x_0 = \pi_1 \mu z[\Phi_i(\pi_1 z) = \pi_2 z].$$

Now define g by

$$g = \lambda x[\underline{if}\ \Phi_i(\pi_1 x) = \pi_2 x\ \underline{then}\ \pi_1 x\ \underline{else}\ x_0\ \underline{fi}].$$

Then g is total and $\mathcal{R}g = \mathcal{D}f$.

(iii) $\Rightarrow$ (iv). Trivial since $\emptyset = \mathcal{R}\varepsilon$.

(iv) $\Rightarrow$ (i). Let $A = \mathcal{R}f$ and let j be an index for f. Then $A = \{x \mid \exists y[\Phi_j(\pi_1 y) = \pi_2 y\ \underline{and}\ \varphi_j(\pi_1 y) = x]\}$. □

We should emphasize that totality of the recursive predicate "$\Phi_j(\pi_1 y) = \pi_2 y\ \underline{and}\ \varphi_j(\pi_1 y) = x$" is based on our specific interpretation of the operator $\underline{and}$; in $\underline{false}\ \underline{and}\ q$ the value of q is not elaborated.

We introduce some further standard reference sets in the arithmetical hierarchy:

$$\begin{aligned}
\textit{non-empty} &= \{i \mid \mathcal{D}\varphi_i \neq \emptyset\}\\
\textit{empty} &= \{i \mid \mathcal{D}\varphi_i = \emptyset\}\\
\textit{finite} &= \{i \mid \#\mathcal{D}\varphi_i < \infty\}\\
\textit{bound} &= \{i \mid \mathcal{D}\varphi_i = \mathbb{N}\ \underline{and}\ \#\mathcal{R}\varphi_i < \infty\}\\
\textit{cofinite} &= \{i \mid \#(\mathbb{N} \setminus \mathcal{D}\varphi_i) < \infty\}.
\end{aligned}$$

We have *non-empty* $\equiv_m$ *halts*, both sets being Σ_1-complete. Consequently *empty* is Π_1-complete. *finite* is a Σ_2-set, since $x \in$ *finite* $\underline{iff}$ $\exists y \forall z[\pi_1 z \leq y\ \underline{or}\ \Phi_x(\pi_1 z) \neq \pi_2 z]$. This latter condition may also be written like

$$\overset{\infty}{\forall} z[\Phi_x(\pi_1 z) \neq \pi_2 z].$$

This suggests a relation between Σ_2-sets and sets defined by $\overset{\infty}{\forall}$ expressions. In fact the two classes of sets are equal as follows from the following lemma:

LEMMA 1.4.10. Let A be a Σ_2-set. Then there exists a total recursive predicate B such that

$$i \in A \text{ \textit{iff} } \overset{\infty}{\forall} x[B(i,x)].$$

PROOF. Let $A = \{i \mid \exists y \forall x[P(i,x,y)]\}$. We define B by:

$$B(i,x) = \exists y{\le}x[\forall z{\le}x[P(i,z,y)] \text{ \underline{and} }$$
$$\text{\underline{not} } \exists w{<}y[\forall z{<}x[P(i,z,w)] \text{ \underline{and} \underline{not} } P(i,x,w)]].$$

To understand this horrible definition the reader should consider the diagram below:

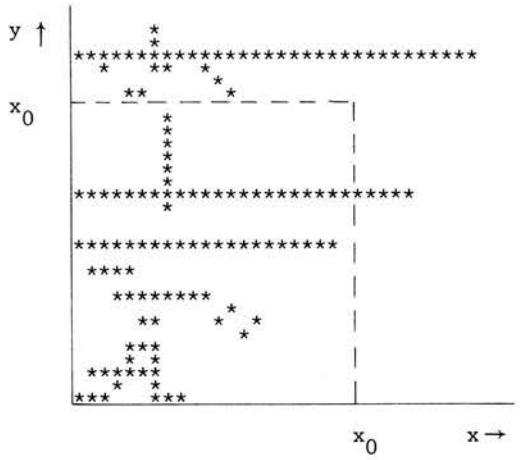


Diagram 1.4.11

The asterisks denote the pairs $\langle x,y\rangle$ for which $P(i,x,y)$ = *true* (i is fixed for this diagram).

The value of $B(i,x_0)$ depends on the configuration of asterisks within the square $0 \le x,y \le x_0$.

The clause "$\forall z{\le}x_0[P(i,z,y)]$" means that the y-th row in the diagram is filled with asterisks upto x_0.

The clause "$\underline{not}$ $\exists w<y[\forall z<x_0[P(i,z,w)]$ $\underline{and}$ $\underline{not}$ $P(i,x_0,w)]$" means that there is no row in the diagram below the y-th row which is filled with asterisks upto x_0-1.

Now $i \in A$ whenever there exists a y such that the y-th row consists of asterisks completely. A row containing asterisks upto x_0 may be a candidate for such a row. The first clause detects the presence of such a candidate. However, in order to be sure that our candidate is "good" upto infinity we should not freely replace it by another if we have detected a mistake by our candidate. The second clause ensures that $B(i,x)$ is false for $x = x_0$ if some former candidate is found to perform its first mistake at x_0.

Therefore if a good row exists after finitely many rejections of wrong candidates the lowest good row becomes the current candidate and remains so forever. Consequently $i \in A$ implies $\overset{\infty}{\forall}x[B(i,x)]$. Conversely if no good row exists each possible candidate will be rejected or there will be at x no candidate at all. Hence $i \notin A$ implies $\overset{\infty}{\exists}x[B(i,x) = \underline{\textit{false}}]$. This completes the proof.

It is not difficult to prove using this lemma that *finite* is Σ_2-complete.

total can be proved to be Π_2 and even Π_2-complete by reducing $\mathbb{N} \setminus$ *finite* to it. *bound* is a Δ_3-set which is not complete; this is not very amazing since there exist no Δ_3-complete sets (cf. [Ro 67]).

Finally the set *cofinite* is Σ_3. One can prove it to be Σ_3-complete using the lemma below:

<u>LEMMA 1.4.12</u>. Let A be a Σ_3-set. Then there exists a total predicate B such that

$$i \in A \ \underline{\textit{iff}}\ \overset{\infty}{\forall}x\exists z[B(i,x,z)].$$

The proof of this lemma, which is more or less analogous to the proof of the preceding lemma, will appear in [EB 74].

All the standard reference sets are so-called *index sets*. If F is some collection of functions then $\Omega F = \{i \mid \lambda\varphi_i \in F\}$ is called the index set corresponding to F. Except for the trivial cases $F = \emptyset$ or $P \in F$ the index sets ΩF are never recursive. This fact is known as RICE's theorem:

<u>THEOREM 1.4.13</u>. [RICE]. If $F \neq \emptyset, P$ then ΩF is not recursive.

<u>PROOF</u>. Replacing if necessary ΩF by $\mathbb{N} \setminus \Omega F = \Omega \bar{F}$ where $\bar{F}$ is the complement of F in P we may assume that $\varepsilon \notin F$. Let f be some member of F. Then $\mathcal{D}f \neq \emptyset$. Define the transformation σ by

$$\varphi_{\sigma(i)}(x) \Leftarrow \underline{if}\ \varphi_i(i) < \infty\ \underline{then}\ f(x)\ \underline{else}\ \infty\ \underline{fi}.$$

Then $\underline{halts} \leq_m \Omega F$ by σ. $\square$

<u>DEFINITION 1.4.14</u>. If A is a Ξ-set and if σ is a transformation of programs then the class of functions $\{\Lambda\varphi_{\sigma(i)} \mid i \in A\}$ is called *Ξ-presentable* (by σ). In particular if $A = \mathbb{N}$ the class of functions $\{\Lambda\varphi_{\sigma(i)} \mid i \in \mathbb{N}\}$ is a *recursively presentable class*. Note that a Σ_1-presentable class is also recursively presentable.

CHAPTER 1.5

GENERAL PROPERTIES OF COMPLEXITY MEASURES

{parturiunt montes, nascetur ridiculus mus. Horatius}

In order that a complexity measure is a reasonable concept it should be useful to have some facts from every day experience which could be formulated and formally proved within the language of this concept. This chapter is dedicated to this kind of translations of intuitive ideas like inefficient computations and algorithms constructed by combination of other algorithms.

Daily life experience learns that it is difficult to design an efficient algorithm but that it is very easy to spoil a good program by including inessential, time wasting instructions. This inefficiency is useful insofar that it proves the existence of infinitely many programs for each recursive function, a fact known as the ROGERS' padding lemma. The proof of this lemma presented below was first given in McCREIGHT's unpublished thesis [MC 69].

LEMMA 1.5.1. [Inefficiency lemma - BLUM]. There exists a transformation of programs σ satisfying

(i) $\mathcal{D}\varphi_{\sigma(i,j)} = \mathcal{D}\varphi_i \cap \mathcal{D}\varphi_j$

(ii) $\forall x[x \in \mathcal{D}\varphi_{\sigma(i,j)}$ $\underline{imp}$ $\varphi_{\sigma(i,j)}(x) = \varphi_i(x)$ $\underline{and}$ $\Phi_{\sigma(i,j)}(x) > \varphi_j(x)]$.

In particular for total φ_j the transformation $\varphi_i \longrightarrow \varphi_{\sigma(i,j)}$ replaces the program φ_i by a program for the same function having a run-time which exceeds $\varphi_j(x)$ for all arguments x.

PROOF. Let τ be defined by

$$\varphi_{\tau(i,j,k)}(x) \Leftarrow \underline{if}\ \Phi_k(x) \le \varphi_j(x)\ \underline{then}\ \varphi_k(x)+1\ \underline{else}\ \varphi_i(x)\ \underline{fi}.$$

Let τ be the fixed-point transformation such that $\varphi_{\sigma(i,j)} = \varphi_{\tau(i,j,\sigma(i,j))}$. Existence of σ is proved by the recursion theorem. Then $\varphi_{\sigma(i,j)}$ satisfies the equation

$$\varphi_{\sigma(i,j)}(x) = \underline{if}\ \Phi_{\sigma(i,j)}(x) \le \varphi_j(x)\ \underline{then}\ \varphi_{\sigma(i,j)}(x)+1\ \underline{else}\ \varphi_i(x)\ \underline{fi}.$$

It is clear from this equation that $\varphi_{\sigma(i,j)}(x)$ is not computed by selecting the *then* part at any argument x; hence $\varphi_{\sigma(i,j)}(x) = \varphi_i(x)$ whenever defined. Moreover $\varphi_{\sigma(i,j)}(x)$ diverges if either the condition does not converge (i.e. $x \notin \mathcal{D}\varphi_j$) or if the subsequent computation of $\varphi_i(x)$ fails to terminate ($x \notin \mathcal{D}\varphi_i$). Therefore $\mathcal{D}\varphi_{\sigma(i,j)} = \mathcal{D}\varphi_i \cap \mathcal{D}\varphi_j$. □

By replacing φ_j by Φ_i (or better $\varphi_{\rho(i)}$ where ρ is the transformation satisfying $\Phi_i = \varphi_{\rho(i)}$) we conclude that each program φ_i can be replaced by a more expensive one, except when φ_i happens to be a program for the empty function. Iterating this construction one finds a sequence of distinct indices for the function $\Lambda\varphi_i$.

More formally let $P(j,0) = j$ and $P(j,k+1) = \sigma(P(j,k),\rho(P(j,k)))$. Then for each k $\varphi_j = \varphi_{P(j,k)}$ and $\Lambda\Phi_{P(j,k+1)} > \Lambda\Phi_{P(j,k)}$.

If $\varphi_j \neq \varepsilon$ it is clear that $P(j,k) \neq P(j,n)$ if $k \neq n$. For indices j for the empty functions either the sequence $(P(j,k))_k$ is again infinite or it becomes periodic ($P(j,k) = P(j,k+m)$ for k sufficiently large). Now the second development cannot occur for all indices j for ε since otherwise the test "does $(P(j,k))_k$ becomes periodic?" yields a method to enumerate the Π_1-complete set *empty*.

Moreover it is not difficult to construct an index j for ε such that $(P(j,k))_k$ becomes not periodic:

Define the transformation ψ by:

$$\varphi_{\psi(j)}(x) \Leftarrow \underline{if}\ \pi_1 x \neq \pi_2 x\ \underline{and}\ P(j,\pi_1 x) = P(j,\pi_2 x)\ \underline{then}\ 1\ \underline{else}\ \underline{loop}\ \underline{fi}.$$

If j_0 is the fixed point under ψ: $\varphi_{j_0} = \varphi_{\psi(j_0)}$ then the then-part becomes contradictory; consequently $\varphi_{j_0} = \varepsilon$ and moreover $P(j_0,k) \neq P(j_0,n)$ whenever $k \neq n$.

A slight modification of the function P consisting of the replacement of values $P(j,k)$ which have become periodic by values from the sequence $(P(j_0,k))_k$ which still are "unused", yields a real "padding function" as claimed by the ROGERS' *Padding lemma*.

LEMMA 1.5.2. [ROGERS' Padding lemma]. There exists a transformation of programs π such that for each index j the sequence $(\pi(j,k))_k$ is a sequence of distinct indices for $\Lambda\varphi_j$.

By further modifications the transformation π can be made increasing in both arguments and moreover a 1-1 function. Use of the padding lemma leads to the following corollaries.

COROLLARY 1.5.3. Let σ be a transformation of programs. Then there exists an increasing transformation τ functionally equivalent to σ; formally $\varphi_{\sigma(i)} = \varphi_{\tau(i)}$ and $\tau(i+1) > \tau(i)$.

COROLLARY 1.5.4. [ROGERS - Isomorphism between effective enumerations]. Let $(\varphi_i)_i$ and $(\psi_i)_i$ be effective enumerations then there exists a recursive permutation κ such that $\Lambda\varphi_i = \Lambda\psi_{\kappa(i)}$.

In the proof of the second corollary we use the extension of $(\psi_i)_i$ to a complexity measure from chapter 1.3 and the MYHILL Isomorphism principle.

Finally the standard reference sets of the preceding chapters, which by definitions are index sets all can be shown to be complete in their corresponding 1-degree; if $A \leq_m B$ by f and B is an index set then there exists a 1-1 function f' such that $A \leq_1 B$ by f'.

A problem frequently considered in abstract complexity theory is the existence of sufficiently many increasing run-times (cf. M. BLUM [Bl 67] and N. LYNCH [Ly 72]). A complete solution is given by the following

LEMMA 1.5.5. [Monotonicity lemma]. There exists a transformation τ such that:

(i) $\mathcal{D}\varphi_{\tau(i)}$ is the largest segment $[0,x)$ contained within $\mathcal{D}\varphi_i$ (hence $\mathcal{D}\varphi_i = \mathcal{D}\varphi_{\tau(i)} = \mathbb{N}$ for total φ_i).

(ii) $\Lambda\Phi_{\tau(i)}$ is increasing on $\mathcal{D}\varphi_{\tau(i)}$.

(iii) $\varphi_{\tau(i)} \sqsubseteq \varphi_i$.

PROOF. Let σ be the transformation described in the inefficiency lemma. Define the transformation ρ by

$$\varphi_{\rho(i,j)}(x) \Leftarrow \underline{if}\ x = 0\ \underline{then}\ \Lambda\Phi_i(0)$$
$$\underline{else}\ \underline{max}(\ \Phi_i(x), \Lambda\Phi_{\sigma(i,j)}(x-1))\ \underline{fi}.$$

Consequently $0 \in \mathcal{D}\varphi_{\rho(i,j)}$ *iff* $0 \in \mathcal{D}\varphi_i$ and for $x > 0$
$x \in \mathcal{D}\varphi_{\rho(i,j)}$ *iff* $x-1 \in \mathcal{D}\varphi_i \cap \mathcal{D}\varphi_j$ *and* $x \in \mathcal{D}\varphi_i$.

Let κ be the fixed-point transformation such that $\varphi_{\kappa(i)} = \varphi_{\rho(i,\kappa(i))}$.
Hence:

$$\varphi_{\kappa(i)}(x) = \underline{if}\ x = 0\ \underline{then}\ \Lambda\Phi_i(0)\ \underline{else}\ \underline{max}(\Lambda\Phi_i(x),\Lambda\Phi_{\sigma(i,\kappa(i))}(x-1))\ \underline{fi}.$$

Consequently $0 \in \mathcal{D}\varphi_{\kappa(i)}$ *iff* $0 \in \mathcal{D}\varphi_i$ and for $x > 0$
$x \in \mathcal{D}\varphi_{\kappa(i)}$ *iff* $x-1 \in \mathcal{D}\varphi_{\kappa(i)}$ *and* $x-1, x \in \mathcal{D}\varphi_i$.
This shows that $\mathcal{D}\varphi_{\kappa(i)}$ is the largest segment $[0,x)$ contained in $\mathcal{D}\varphi_i$.

Let $\tau = \lambda i[\sigma(i,\kappa(i))] = \sigma \,\Box\, \kappa$. Since $\mathcal{D}\varphi_{\tau(i)} = \mathcal{D}\varphi_i \cap \mathcal{D}\varphi_{\kappa(i)} = \mathcal{D}\varphi_{\kappa(i)}$, (i) is satisfied. (iii) follows from the definition of σ whereas (ii) is derived from:

$$\Lambda\Phi_{\tau(i)}(x) \geq \Lambda\Phi_{\sigma(i,\kappa(i))}(x) > \varphi_{\kappa(i)}(x) =$$

$$= \underline{if}\ x=0\ \underline{then}\ \Lambda\Phi_i(0)\ \underline{else}\ \underline{max}(\Lambda\Phi_i(x),\Lambda\Phi_{\tau(i)}(x-1))\ \underline{fi}.$$

Consequently $\Lambda\Phi_{\tau(i)}(x) > \Lambda\Phi_{\tau(i)}(x-1)$ for $x > 0$. □

COROLLARY 1.5.6. Each total function is computed by a program with increasing run-time.

Intuitively functions which are expensive in one measure should be expensive for other measures also. This is not completely true since measures can be constructed were a decidable set of total functions can be computed "for nothing" as was seen in chapter 1.3. However, "at large" the run-times of a function in different measures are related.

PROPOSITION 1.5.7. [Recursive relatedness between measures]. Let $((\varphi_i)_i,(\Phi_i)_i)$ and $((\varphi_i)_i,(\Phi_i^*)_i)$ be two complexity measures on a single enumeration. Then there exists a total recursive function R satisfying:

$$\forall i \overset{\infty}{\forall} x[\Lambda\Phi_i(x) \leq R\Box\Phi_i^*(x)\ \underline{and}\ \Lambda\Phi_i^*(x) \leq R\Box\Phi_i(x)].$$

PROOF. Let P be defined by

$$P = \lambda i,x,z[\underline{max}(\underline{if}\ \Phi_i(x) = z\ \underline{then}\ \Lambda\Phi_i^*(x)\ \underline{else}\ 0\ \underline{fi},$$

$$\underline{if}\ \Phi_i^*(x) = z\ \underline{then}\ \Lambda\Phi_i(x)\ \underline{else}\ 0\ \underline{fi})]$$

and define R by

$$R = \lambda x,z[max\{P(i,x,z) \mid i \leq x\}].$$

Then for $x > i$ R satisfies

$$R\Box\Phi_i(x) \geq \Lambda\Phi_i^\star(x) \text{ and } R\Box\Phi_i^\star(x) \geq \Lambda\Phi_i(x). \quad \Box$$

A similar result holds for complexity measures on different enumerations. This shows that (abstractly) compilers introduce a recursively bounded "overhead".

A similar recursive "overhead-function" is involved in the combining of several algorithms to a single one. This fact is known as the "combining lemma".

<u>LEMMA 1.5.8</u>. [Combining lemma]. Let σ be a transformation of programs satisfying $\mathcal{D}\varphi_{\sigma(i,j)} \subseteq \mathcal{D}\varphi_i \cap \mathcal{D}\varphi_j$. Then there exists a total function R satisfying:

$$\forall i,j \overset{\infty}{\forall} x[\Lambda\Phi_{\sigma(i,j)}(x) \leq R(x,\Lambda\Phi_i(x),\Lambda\Phi_j(x))].$$

<u>PROOF</u>. The domain condition $\mathcal{D}\varphi_{\sigma(i,j)} \subseteq \mathcal{D}\varphi_i \cap \mathcal{D}\varphi_j$ shows that the function P defined below is total:

$$P = \lambda i,j,x,y,z[\underline{if}\ \Phi_i(x) = y\ \underline{and}\ \Phi_j(x) = z\ \underline{then}\ \Lambda\Phi_{\sigma(i,j)}(x)\ \underline{else}\ 0\ \underline{fi}].$$

The function R is defined by

$$R = \lambda x,y,z[max\{P(i,j,x,y,z) \mid i,j \leq x\}].$$

Hence for $x \geq i,j$ one has $R(x,\Lambda\Phi_i(x),\Lambda\Phi_j(x)) \geq \Lambda\Phi_{\sigma(i,j)}(x)$. $\Box$

The combining lemma as formulated above was given by HARTMANIS & HOPCROFT [HH 71]. For a "metamathematical" generalization of this lemma the reader is referred to G. AUSIELLO [Au 70].

In any measure the run-time of a program recursively bounds the size of this program, whereas no bounding the other way around exists.

<u>PROPOSITION 1.5.9</u>. There exists a total function R such that for each i $\varphi_i \underline{\propto} R\Box\Phi_i$ but for no total function R one has $\Lambda\Phi_i \underline{\propto} R\Box\varphi_i$.

<u>PROOF</u>. If $P = \lambda i,x,z[\underline{if}\ \Phi_i(x) = z\ \underline{then}\ \varphi_i(x)\ \underline{else}\ 0\ \underline{fi}]$ and $R = \lambda x,z[max\{P(i,x,z) \mid i \leq x\}]$, then clearly $\varphi_i \underline{\propto} R\Box\Phi_i$.

To prove the second assertion we use a diagonalization argument. Let R be total and define f by

$$f = \lambda x[\underline{if}\ \Phi_{\pi_1 x}(x) \le R(x,0)\ \underline{then}\ 1 \dot{-} \varphi_{\pi_1 x}(x)\ \underline{else}\ 0\ \underline{fi}].$$

Note that the then-part converges whenever it is selected; hence f is a total function. If k is an index and $\pi_1 y = k$ the definition of f leads to:

$$f(y) = \underline{if}\ \Phi_k(y) \le R(y,0)\ \underline{then}\ 1 \dot{-} f(y)\ \underline{else}\ 0\ \underline{fi}.$$

So the then-part becomes contradictory. Consequently $f(y) = 0$ and $\Phi_k(y) > R(y,f(y))$. This shows that

$$\overset{\infty}{\exists} x[\Lambda\Phi_k(x) > R\square\varphi_k(x)].$$

Hence not only for a given index i the assertion $\Lambda\Phi_i \underline{\propto} R\square\varphi_i$ is false but there exist functions f such that this assertion fails for each index k for f. □

For specific programs it may be possible that $\Lambda\Phi_i \underline{\propto} R\square\varphi_i$; such programs are called R-honest.

<u>DEFINITION 1.5.10</u>. Let R be a (total) function. A program φ_i is called *R-honest* whenever $\Lambda\Phi_i \underline{\propto} R\square\varphi_i$. A function f is R-honest whenever it is computed by some R-honest program.

Our last proposition shows that for a given total R not all recursive functions are R-honest.

A specific method to combine programs is to let two programs run in parallel, terminating the computation, the moment the fastest computation terminates. Intuitively such a computation should have a run-time which equals the minimum of the run-times of the two simulated computations.

The *parallel-computation axiom*, which is one of the axioms which have been proposed to separate between natural and pathological measures, formalizes this intuition. This axiom reads:

<u>AXIOM 1.5.11</u>. [Parallel computation axiom - [LR 72]]. There exists a transformation κ satisfying:

(i) $\varphi_{\kappa(i,j)}(x) = \underline{\textit{if}}\ \Phi_i(x) \le \Phi_j(x)\ \underline{\textit{then}}\ \varphi_i(x)\ \underline{\textit{else}}\ \varphi_j(x)\ \underline{\textit{fi}}$

(ii) $\Lambda\Phi_{\kappa(i,j)}(x) = \underline{\textit{min}}(\Lambda\Phi_i(x), \Lambda\Phi_j(x))$.

This axiom is satisfied for the model of many-tape, many-heads Turing machines, with a read-only head on their input tape. Clearly the axiom is not satisfied in all measures. However each complexity measure can be extended to a measure for which the axiom holds by introducing sufficiently many new programs. Cf. [EB 74].

CHAPTER 1.6

THE SPEED-UP PHENOMENON

The speed-up phenomenon is one of the central results in abstract complexity theory. Although the remaining parts of this treatise - which deal with resource-bound classes and their generalizations - have almost no relations to the speed-up phenomenon a short explanation is felt to be appropriate at this time.

Consider the Turing machine model of computation. By combining two squares into one and by processing this way two symbols in a single step such a machine can be replaced by a new one operating about twice as fast. This shows that all functions may be speeded-up linearly. It might be that this is the best one can hope for; up to a linear factor each function has an optimal program computing this function. In fact the above assertion is false. Regardless the amount of relativity in the definition of "optimality", there exist always functions having no "optimal" program at all.

This fact is formalized by the so-called speed-up theorem, first given by M. BLUM [Bl 67].

<u>THEOREM 1.6.1</u>. [Speed-up theorem]. Let R be a total function. Then there exists a (0-1 valued) total function f with the following property:

For every index i for f there exists an index j for f such that $R\square\Phi_j \overset{\infty}{\leq} \Lambda\Phi_i$.

It has been shown by MEYER and FISHER [MF 72] that the theorem remains valid if the function R is replaced by a total effective operator Γ; in this case still functions f exist such that for each index i for f an index j for f exists such that $\Gamma\Lambda\Phi_j \overset{\infty}{\leq} \Phi_i$.

This theorem eliminates at once the hope of uniformly optimizing all programs at once. Thinking R to be a function like 2^{x+y}, the theorem shows that each program for f may be replaced by another which is exponentially faster (which again on its turn is surpassed by a still more efficient one, etc.).

The theory has learned moreover that although the faster programs are proved to exist it is not in general possible to find them. In fact it can be shown that for non-trivial speed-ups the faster programs have indices which do not depend recursively on the indices for the slower ones (cf. [Bl 71], [HY 71] or [MF 72]).

A central concept in the theory of the speed-up phenomenon is the concept of a complexity sequence. This is a sequence of functions cofinal in the $\underline{\propto}$-order with the collection of run-times for a given function f.

DEFINITION 1.6.2. A sequence of functions $(p_i)_i$ is called a *complexity sequence* for the function f, provided that

(i) $\forall i[\mathcal{D}\varphi_i = \mathcal{D}f]$
(ii) $\forall j[f = \varphi_j \Rightarrow \exists i[p_i \underline{\propto} \Lambda\varphi_j]]$
(iii) $\forall i \exists j[f = \varphi_j$ *and* $\Lambda\Phi_j \underline{\propto} p_i]$.

From this definition one sees that a function f with a complexity sequence $(p_i)_i$ satisfying: $\forall i \exists j[R \Box p_j \underline{\propto} p_i]$ is an R-speed-up able function. In fact the speed-up theorem is proved by constructing a function with such a complexity sequence.

A more complete survey on the speed-up theory will be included in [EB 74].

PART 2

RESOURCE-BOUND CLASSES

{*27 So the servants of the householder came and said unto him, Sir, didst not thou sow good seed in thy field? from whence then hath it tares?*
28 He said unto them, An enemy hath done this. The servants said unto him, Wilt thou then that we go and gather them up?
29 But he said, Nay; lest while ye gather up the tares, ye root up also the wheat with them.
30 Let both grow together until the harvest: and in the time of harvest I will say to the reapers, Gather ye together first the tares, and bind them in bundles to burn them: but gather the wheat into my barn.
St. Matthew, XIII 27-30}

CHAPTER 2.1

DEFINITIONS

2.1.1. INTRODUCTION

Resource-bound classes are a particular type of subrecursive classes of functions. The arithmetical hierarchy discussed in part 1, classifies non-recursive functions and sets by the complexity of their definitions, and the class of recursive functions is a non-structured class at the base of this hierarchy. Subrecursive classes are subsets of the class of recursive functions itself. Mostly they are defined in terms of some hierarchy. Examples of such hierarchies are among others:

the hierarchy defined by R. PETER [Pe 50] based on multiple recursion,

the *GRZEGORCZYCK hierarchy* defined in terms of bounded primitive recursive functions, which is the base class of the PETER hierarchy [Gz 53],

the hierarchy of *predicatably computable functions* defined by R.W. RITCHIE [Rr 63].

The first two hierarchies are defined in terms of program structure whereas the last hierarchy is defined in terms of the Turing tape measure.

For an extensive survey on the classical examples of subrecursive hierarchies the reader is referred to the first part of the thesis of R. MOLL [Mo 73].

Given the concept of a complexity measure one can define several types of resource-bound classes in terms of this measure. The most investigated ones are the *complexity classes*, which consist of all functions which may be computed by some program whose run-time is bounded almost everywhere by some (partial) recursive function. If the maximal run-time may depend on both the argument and the computed value one gets the so-called *honesty classes* (cf. chapter 1.5).

The approach to subrecursive classes by the way of resource-bound classes has several advantages over the classical approach.

Parts of the theory (like for example diagonalization techniques) are measure independent; using resource-bound classes they can be treated this way.

The classical examples can be defined in terms of resource-bound classes for some suitable complexity measure; we have therefore a good generalization.

The hierarchy of resource-bound classes which is indexed by the system of partial recursive functions is much richer than hierarchies indexed by natural numbers or ordinal notations. Moreover hierarchies indexed by ordinal notations can be defined, and the properties of these hierarchies (like degeneration and non-uniqueness) can be analyzed [Ba 70, BY 73].

During the sequel we take for $\Phi = ((\varphi_i)_i, (\Phi_i)_i)$ a fixed complexity measure.

2.1.2. TYPES OF RESOURCE-BOUND CLASSES

DEFINITION 2.1. Let t be a partial (recursive) function. The *complexity class of programs* F_t is the set of programs φ_i satisfying:

$$\overset{\infty}{\forall} x[x \in \mathcal{D}t \Rightarrow \Phi_i(x) \leq t(x)]$$

The *complexity class of functions* C_t is the set of all functions computed by some program in F_t. The function t is called a *name* for C_t respectively F_t.

It should be mentioned that this definition differs from definitions given by other authors. In the first place F_t is a class of programs and not of indices, a distinction motivated by the theory developed in Part 3. Furthermore there are no domain conditions enforced like $\mathcal{D}\varphi_i = \mathcal{D}t$ (cf. [LR 72]).

By the "almost everywhere" condition in the definition, programs are included in F_t even if they behave extravagantly on the first $10^{10^{10}}$ arguments only becoming nice at still larger arguments. The "almost everywhere" condition is exploited heavily in the proofs. Moreover, for "natural" measures "almost everywhere" should become "everywhere" by modifying the program in such a way that an initial segment of the function is computed by table look-up, thus eliminating the bad behaviour at small arguments.

Clearly we have $t \underline{\propto} u \Rightarrow C_t \subseteq C_u$ and $F_t \subseteq F_u$. Moreover, functions t and u satisfying $\overset{\infty}{\forall} x[t(x) = u(x)]$ clearly are names for the same classes. As we shall see the converse of this assertion is false. The same class may be named by highly different functions.

The class named by a function with finite domain contains all recursive functions or programs. The smallest class consists of all functions which can be computed without charge; zero (or any other function which is almost everywhere equal to zero) is a name for this class.

Note that a class containing a function with finite domain must have a name with finite domain as well, consequently a class, which contains a single function with finite domain contains every other function also.

From the definition of F_t it is easy to see that the set $\{i \mid \varphi_i \in F_t\}$ is a Σ_2-set. In fact it is a Σ_2-complete set provided t is sufficiently large (see chapter 2.3). Consequently C_t is Σ_2-presentable. In chapter 2.3 it will be shown that C_t is in fact recursively presentable, provided t is large enough.

<u>DEFINITION 2.1.2</u>. Let R be a function with two arguments. The *honesty class of programs* G_R consists of all programs φ_i satisfying:

$$\overset{\infty}{\forall} x[\langle x,\varphi_i(x)\rangle \in \mathcal{D}R \Rightarrow \Phi_i(x) \leq R(x,\varphi_i(x))]$$

(where by convention $\langle x,\varphi_i(x)\rangle \notin \mathcal{D}R$ for all $x \notin \mathcal{D}\varphi_i$).

The *honesty class of functions* H_R consists of all functions computed by programs in G_R.

The "honesty" of a function suggests that its run-time is bounded recursively by the size of the function. For larger values a larger run-time is permitted and for divergent computations there is no restriction on the run-time at all. Consequently each honesty class contains all functions c.q. programs with finite domain.

Several other types of resource-bound classes are defined by considering honesty classes with a special type of names.

<u>DEFINITION 2.1.3</u>. Let t be a recursive function. The *weak complexity class of programs* F_t^W consists of all programs φ_i satisfying:

$$\overset{\infty}{\forall} x[\varphi_i(x) = \infty \ \underline{or}\ (x \in \mathcal{D}t \Rightarrow \Phi_i(x) \leq t(x))].$$

The *weak complexity class of functions* C_t^W consists of all functions computed by programs in F_t^W. If we write $T = \lambda x,y[t(x)]$ then clearly $C_t^W = H_T$ and $F_t^W = G_T$.

The weak complexity classes behave like honesty classes. Note however that C_t and C_t^W contain the same total functions. Consequently the weak complexity classes are a generalization of the ordinary complexity classes consisting of total functions, which are frequently considered in the literature.

DEFINITION 2.1.4. Let r be a recursive function and let $R = \lambda x,y[r(\underline{max}(x,y))]$. The *modified honesty classes* $H_r^\wedge$ and $G_r^\wedge$ are defined by

$$H_r^\wedge = H_R \quad \text{and} \quad G_r^\wedge = G_R.$$

These modified honesty classes are introduced to solve a (for general honesty classes still unsolved) problem in chapter 3.4.

As before we have $H_T \subseteq H_U$ for $T \underline{\propto} U$. $H_{\text{zero }2}$ resp. $H_{\varepsilon 2}$ is the smallest resp. largest honesty class. Again $\{i \mid \varphi_i \in H_R\}$ is a Σ_2-class of indices.

An important result concerning honesty classes is the so-called "equivalence" between honest set and measured sets.

THEOREM 2.1.5. Let R be a total function. Then there exists a measured set $(\gamma_i)_i$ such that $\{\gamma_i \mid i \in \mathbb{N}\} = H_R$. Conversity, if $(\gamma_i)_i$ is a measured set, then there exists a total function R such that $\{\gamma_i \mid i \in \mathbb{N}\} \subseteq H_R$.

This theorem, due to E.M. McCREIGHT, is very frequently mentioned in the literature, but mostly the proof is omitted. Moreover, from the formulation in [MCM 69] the present author was lured into believing that the set G_R itself is a measured set. In general this is not true. The example below (which is derived from the "counterexample" in [EB 71]) shows how the honest set is "scrambled" non-recursively; given an index of an honest program it is impossible to generate recursively an index for the corresponding program in the measured set.

A proof of the theorem is found in the unpublished thesis of E.M. McCREIGHT [MC 69] and also in the thesis of R. MOLL [Mo 73]. We give the proof in chapter 2.3 (th. 2.3.8).

EXAMPLE 2.1.6. Let σ be a transformation increasing in both arguments satisfying $\varphi_{\sigma(i,j)}(x) = \underline{\mathit{if}}\ x > j\ \underline{\mathit{then}}\ 0\ \underline{\mathit{else}}\ \varphi_i(x)\ \underline{\mathit{fi}}$. Since σ is non-decreasing it is decidable whether $k \in R\sigma$ and, if so, the indices i and j such that $k = \sigma(i,j)$ are computable. Hence we define a new measure Φ^* by:

$$\Phi_k^*(x) = \begin{cases} \Phi_k(x) + x & \text{for } k \notin R\sigma \\ \Phi_i(x) & \text{for } k = \sigma(i,j) \text{ \underline{and} } x \le j \\ 0 & \text{for } k = \sigma(i,j) \text{ \underline{and} } x > j. \end{cases}$$

Clearly $\Phi^* = \{(\varphi_i)_i, (\Phi_i)_i\}$ is a complexity measure.
Now $G_{\text{zero } 2}$ consists of all programs with finite domain and all programs $\varphi_{\sigma(i,j)}$. Let $(\gamma_i)_i$ be a measured set enumerating $H_{\text{zero } 2}$. Suppose that τ is a recursive function such that $\Lambda\gamma_{\tau(k)} = \Lambda\varphi_k$ for those k such that $\varphi_k \in G_{\text{zero } 2}$. Then a contradiction arises as follows: We have

$$\varphi_i(x) = y \ \underline{\mathit{iff}}\ \varphi_{\sigma(i,x)}(x) = y.$$

The latter relation by assumption is equivalent to

$$\gamma_{\tau(\sigma(i,x))}(x) = y$$

which relation is decidable by the definition of a measured set. Hence $\varphi_i(x) = y$ is decidable, quod non. □

REMARK 2.1.7: In the theorem it is essential that the function R is total. For example let R be the partial function $\lambda x,y[\underline{if}\ y=0\ \underline{then}\ \underline{loop}\ \underline{else}\ 0\ \underline{fi}]$. Now H_R contains among others the function

$$k = \lambda x[\underline{if}\ \varphi_x(x) < \infty\ \underline{then}\ 0\ \underline{else}\ \infty\ \underline{fi}]$$

which is a member of no measured set ($k(x) = 0$ being equivalent to the halting problem).

The "almost everywhere" condition in the definition of a complexity class has been replaced by an even more general condition by L.J. BASS [Ba 70]. His conclusion was that the complexity classes "modulo sets of exceptional points" defined this way behave not much better than ordinary complexity classes.

DEFINITION 2.1.8. A class E consisting of subsets of $\mathbb{N}$ is called a *class of sets of exceptional points* provided

(i) each member of E is recursive

(ii) $\mathcal{E}$ contains all finite sets

(iii) $\mathcal{E}$ is closed under finite union: $E, F \in \mathcal{E} \Rightarrow E \cup F \in \mathcal{E}$

(iv) $\mathbb{N} \notin \mathcal{E}$.

The class $\mathcal{E}$ is called recursively presentable if there exists a transformation η satisfying

(i) $\forall i[\eta(i) \in \underline{\textit{total}}\ \underline{\textit{and}}\ R\varphi_{\eta(i)} \subseteq \{0,1\}]$

(ii) $E \in \mathcal{E}\ \underline{\textit{iff}}\ \exists i[E = \{x \mid \varphi_{\eta(i)}(x) = 0\}]$.

Note that by the padding lemma we may assume that η is an increasing function.

DEFINITION 2.1.9. Let $\mathcal{E}$ be a class of sets of exceptional points. Then the complexity classes (mod $\mathcal{E}$) $F_t^{\mathcal{E}}$ and $C_t^{\mathcal{E}}$ are defined by:

$$\varphi_i \in F_t^{\mathcal{E}}\ \underline{\textit{iff}}\ \exists_{E \in \mathcal{E}} \forall x[x \in E\ \underline{\textit{or}}\ (x \in \mathcal{D}t \Rightarrow \Phi_i(x) \leq t(x)]$$

$$f \in C_t^{\mathcal{E}}\ \underline{\textit{iff}}\ \exists_i[f = \Lambda\varphi_i\ \underline{\textit{and}}\ \varphi_i \in F_t^{\mathcal{E}}].$$

The behaviour of the classes $C_t^{\mathcal{E}}$ and $F_t^{\mathcal{E}}$ depends essentially on whether $\mathcal{E}$ is recursively presentable or not. In the first case the classes are in fact a special type of complexity classes with partial names, as we shall prove in part 3. Moreover one may use in this case the following lemma:

LEMMA 2.1.10. Let $\mathcal{E}$ be a recursively presentable class of sets of exceptional points. Then there exists an infinite recursive set A such that $\# A \cap E$ is finite for each $E \in \mathcal{E}$.

PROOF. Let A be the range of a recursive function f such that $f(x) = 0$ and $f(x+1)$ is the least number greater than $f(x)$ which is not contained in $\bigcup_{i \leq x} E_i$. By the definition of a class of sets of exceptional points f is total. $\square$

If, however, $\mathcal{E}$ is not assumed to be recursively presentable several important theorems on complexity classes become invalid. Examples are mentioned in the chapters 2-4.

The other types of resource-bound classes like honestly classes and weak complexity classes may be relativized to a class $\mathcal{E}$ analogously.

CHAPTER 2.2

DIAGONALIZATION TECHNIQUES AND COMPRESSION THEOREMS

The inefficiency lemma of chapter 1.5 shows that a given function f is computed by arbitrarily expensive programs. This construction does not yield however an "expensive" function, as is given by the diagonalization construction in the proof that the run-time of a function cannot be bounded recursively by its size.

There are several ways in which a function can be expensive:

Let f be a (recursive) function. The simplest way for F to be expensive is that $f \notin C_t$; i.e. one has:

$$\Lambda \varphi_j = f \Rightarrow \overset{\infty}{\exists} x[\Phi_j(x) > t(x)].$$

A more essential way of being expensive is that each program for f has almost everywhere a run-time larger than t:

$$\Lambda \varphi_j = f \Rightarrow \overset{\infty}{\forall} x[\Phi_j(x) > t(x)].$$

This relation is denoted by t *ncomp* f (f cannot be computed within time t).

Finally it is possible that the individual values of f are expensive:

$$\forall j \overset{\infty}{\forall} x[\varphi_j(x) = f(x) \Rightarrow \Phi_j(x) > t(x)].$$

i.e. if φ_j computes the same value at x as f does its run-time $\Phi_j(x)$ is large (except at finitely many arguments)

This last relation is denoted by t *ncompv* f (the values of f cannot be computed within time t).

Expensive functions are constructed by diagonalization procedures. Assume for the moment that t is a total function.

Define the functions f and g by:

$$f = \lambda x[\underline{if}\ \Phi_{\pi_1 x}(x) \le t(x)\ \underline{then}\ 1 \dot{-} \varphi_{\pi_1 x}(x)\ \underline{else}\ 0\ \underline{fi}],$$

$$g = \lambda x[\mu z[\forall i \le x[\Phi_i(x) > t(x)\ \underline{or}\ \varphi_i(x) \ne z]]].$$

Then $f \notin C_t$ and t *ncompv* g (hence also t *ncomp* g). A more general assertion is proved below.

Note that f is a zero-one-valued function whereas $g(x) \leq x+1$. It is clear that for sufficiently large t no zero-one-valued function h satisfies t *ncompv* h since the values 0 and 1 are "cheaply" computed by programs for the constant functions $\lambda x[0]$ and $\lambda x[1]$.

M. BLUM describes in his proof of the so-called compression theorem [Bl 67] a diagonalization procedure which yields for total t a 0-1 valued total function h satisfying t *ncomp* h.

The diagonalization constructions involved in the definition of the above f and g are more or less uniform in a program for t. For partial t however the above definitions may run astray and one must use some more sophisticated techniques. One uses the fact that the domain of a function is recursively enumerable, and the characterization of the recursively enumerable sets in chapter 1-4.

Uniforming the proofs given at that place one concludes the existence of a pair of transformations α and β such that for i such that $\mathcal{D}\varphi_i$ is infinite

(i) $\varphi_{\alpha(i)}$ is a 1-1 function such that $\mathcal{R}\varphi_{\alpha(i)} = \mathcal{D}\varphi_i$.
(ii) $\varphi_{\beta(i)}$ is a monotonically increasing function such that $\mathcal{R}\varphi_{\beta(i)} \subseteq \mathcal{D}\varphi_i$.

Since $\mathcal{R}\varphi_{\beta(i)}$ is recursive we can define a total result valued function $\varphi_{\gamma(i)}$ such that $\varphi_{\gamma(i)}(x) = $ *false* if $x \notin \mathcal{R}\varphi_{\beta(i)}$ and $\varphi_{\gamma(i)}(x) = k$ whenever $\varphi_{\beta(i)}(k) = x$. For $\varphi_{\alpha(i)}$ we can find a partial inverse $\varphi_{\eta(i)}$ such that $\mathcal{D}\varphi_{\eta(i)} = \mathcal{R}\varphi_{\alpha(i)}$ and $\varphi_{\eta(i)}(x) = y$ *iff* $\varphi_{\alpha(i)}(y) = x$.

Next consider the following transformations:

$$\varphi_{\delta(i)}(x) \Leftarrow \underline{case}\ \varphi_{\gamma(i)}(x)\ \underline{in}\ (\underline{int}\ n)\colon\ \underline{if}\ \Phi_{\pi_1 n}(x) \leq \varphi_i(x)\ \underline{then}\ 1 \dot{-} \varphi_{\pi_1 n}(x)\ \underline{else}\ 0\ \underline{fi}\quad \underline{out}\quad 0\quad \underline{esac};$$ †)

$$\varphi_{\delta'(i)}(x) \Leftarrow (\underline{int}\ n = \varphi_{\eta(i)}(x);\ \mu z[\forall_{k \leq n}[\Phi_k(x) > \varphi_i(x)\ \underline{or}\ \varphi_k(x) \neq z]]).$$

†) This case conformity clause should be read like: "If $\varphi_{\gamma(i)}(x)$ takes the integral values n then .... otherwise 0."

PROPOSITION 2.2.1.

(i) $\varphi_{\delta(i)}$ is total and for i such that $\mathcal{D}\varphi_i$ is infinite $\varphi_{\delta(i)} \notin C_{\varphi_i}$.
(ii) $\mathcal{D}\varphi_{\delta'(i)} = \mathcal{D}\varphi_i$ and for i with $\mathcal{D}\varphi_i$ is infinite φ_i *ncompv* $\varphi_{\delta'(i)}$. Finally there exists a total function R such that $\Lambda\Phi_{\delta'(i)} \overset{\propto}{-} R\Box\Phi_i$.

PROOF.

(i) Let $\varphi_{\delta(i)} = \varphi_k$, and let φ_i be infinite. Assume $\pi_1 n = k$. Then for $x = \varphi_{\beta(i)}(n)$ one has $\varphi_k(x) = \varphi_{\delta(i)}(x) =$ $=$ *if* $\Phi_k(x) \le \varphi_i(x)$ *then* $1 \dot{-} \varphi_k(x)$ *else* 0 *fi* and consequently $\Phi_k(x) > \varphi_i(x)$. Since $\varphi_{\beta(i)}$ is total this argument shows that $\varphi_{\delta(i)} \notin C_{\varphi_i}$.
(ii) From the definition of δ' it is clear that $\mathcal{D}\varphi_{\delta'(i)} = \mathcal{D}\varphi_{\eta(i)} = \mathcal{R}\varphi_{\alpha(i)} =$ $= \mathcal{D}\varphi_i$. If $\mathcal{D}\varphi_i$ is infinite then $\varphi_{\eta(i)}$ is total. For $x \in \mathcal{D}\varphi_{\eta(i)}$ with $n = \varphi_{\eta(i)}(x) \ge k$ one has

$$\varphi_{\delta'(i)}(x) = \mu z[\forall j \le n[\Phi_j(x) > \varphi_i(x) \ \underline{or}\ \varphi_j(x) \ne z]]$$

consequently

$$\Phi_k(x) > \varphi_i(x) \ \underline{or}\ \varphi_k(x) \ne \varphi_{\delta'(i)}(x).$$

This proves φ_i *ncompv* $\varphi_{\delta'(i)}$. Existence of R follows from $\mathcal{D}\varphi_{\delta'(i)} = \mathcal{D}\varphi_i$ by using the combining lemma (1.5.8). □

We say that a function f is *compressed between* two functions t and u provided t *ncomp* f and $f \in C_u$. The *compression theorem* [Bl 67] states that there exists a total function R such that for all indices i with $\mathcal{D}\varphi_i$ infinite, a function exists which is compresses inbetween φ_i and $R\Box\Phi_i$. Without further restrictions on the large behaviour of f this is a corollary to the above proposition. In fact one can construct a zero-one valued function satisfying the conditions. The proof uses a diagonalization method given by BLUM, which we mentioned before.

One might ask whether the upper bound $R\Box\Phi_i$ may be replaced by an upper bound of the form $R\Box\varphi_i$. The answer is negative as shown by the *Gap theorem*.

THEOREM 2.2.2. [Gap theorem] [Bo 72]. For every total R such that $R(x,y) \ge y$ there exist (arbitrarily large) total functions t such that $C_t = C_{R\Box t}$.

The situation is even worse, as indicated by the much stronger result:

THEOREM 2.2.3: [Operator-gap theorem] [Co 72]. For every total effective operator Γ satisfying $\Gamma(t) \geq t$ there exist total functions t such that $C_t = C_{\Gamma(t)}$.

These theorems are proved in chapter 3.2.

The gap-theorems show that uniform extension of all complexity classes is not possible. However:

THEOREM 2.2.4. Let $(\gamma_i)_i$ be a measured set. Then there exists a total function R such that for each i such that $\mathcal{D}\gamma_i$ is infinite, there exists a function f which is compressed inbetween γ_i and $R\square\gamma_i$.

This result follows from the fact that measured sets are honest; consequently there exists a transformation δ and a total function S such that $\Lambda\varphi_{\delta(i)} = \gamma_i$ and $\Lambda\Phi_{\delta(i)} \underline{\propto} S\square\gamma_i$. The compression theorem now yields the result.

The gap-phenomenon may be escaped by restricting oneself to names selected from a measured set. The *naming theorem* shows that measured sets exist, which contain names for all complexity classes.

THEOREM 2.2.5. [Naming theorem] [MC 69]. There exists a measured transformation of programs ν such that for each i $C\varphi_i = C\varphi_{\nu(i)}$.

The naming theorem is proved in part 3.

Combining the two results we see that a uniform procedure to extend complexity classes exists. The larger class however depends on a given index for a name of the class and not on the name itself.

We have considered up to now only the problem of diagonalizing over complexity classes. For the other types of resource-bound classes analogous diagonalization constructions can be defined; moreover the constructions for ordinary complexity classes are good for some other types too.

For complexity classes modulo sets of exceptional points diagonalization is possible unless the domain of the name of the class is included in some exceptional set. This follows as a corollary to the compression theorem.

For honesty classes the situation is more interesting. We can show that the class H_R can be extended provided $\mathcal{D}R$ contains the graph of a recursive function with infinite domain. This later condition is fulfilled whenever $\pi_1 \mathcal{D}R$ is infinite.

LEMMA 2.2.6. There exist transformations ρ, τ and θ, satisfying the following condition: If $\pi_1 \mathcal{D}\varphi_i^2$ is infinite then

(i) $\varphi_{\rho(i)}$ is total and 1-1, $\mathcal{R}\varphi_{\rho(i)} \subseteq \mathcal{D}\varphi_i^2$.
(ii) $\lambda x[\pi_1\varphi_{\rho(i)}(x)]$ is increasing
(iii) $\pi_2\varphi_{\rho(i)}(x) = \varphi_{\tau(i)}(\pi_1\varphi_{\rho(i)}(x))$
(iv) $\varphi_{\tau(i)}$ is total, $\varphi_{\theta(i)} \sqsubseteq \varphi_{\tau(i)}$ and $\mathcal{D}\varphi_{\theta(i)} = \pi_1\mathcal{R}\varphi_{\rho(i)}$.

PROOF. There exists a transformation κ such that $\varphi_{\kappa(i)}$ is a 1-1 enumeration of $\mathcal{D}\varphi_i^2$. ρ, τ and θ are constructed from κ by:

$$\varphi_{\rho(i)}(x) = \underline{\textit{if}}\ x=0\ \underline{\textit{then}}\ \varphi_{\kappa(i)}(0)\ \underline{\textit{else}}$$
$$\varphi_{\kappa(i)}(\mu z[\pi_1\varphi_{\kappa(i)}(z) > \varphi_{\rho(i)}(x-1)])\ \underline{\textit{fi}}.$$

(In this "definition" we have implicitly used the recursion theorem, hence $\varphi_{\rho(i)}(x) \Leftarrow \ldots$ would have been illegal.)

$$\varphi_{\theta(i)}(x) \Leftarrow \pi_2\varphi_{\rho(i)}(\mu z[\pi_1\varphi_{\rho(i)}(z) = x])$$

$$\varphi_{\tau(i)}(x) \Leftarrow \pi_2\varphi_{\rho(i)}(\mu z[\pi_1\varphi_{\rho(i)}(z) \geq x]). \quad \square$$

An example of a diagonalizing procedure is the transformation δ below:

```
$\varphi_{\delta(i)}(x) \Leftarrow$ ($\underline{int}$ $k = \mu z[\pi_1\varphi_{\rho(i)}(z) \geq x]$;
          $\underline{int}$ $xx = \pi_1\varphi_{\rho(i)}(k)$;
          $\underline{if}$ $xx > x$ $\underline{then}$ $\varphi_{\tau(i)}(x)$
             $\underline{else}$ $\underline{int}$ $y = \pi_2\varphi_{\rho(i)}(k)$;
                   ¢ hence $\langle x,y\rangle \in \mathcal{D}\varphi_i^2$ ¢
                   $\underline{int}$ $z = \varphi_i^2(x,y)$;
                   $\underline{if}$ $\Phi_{\pi_1 k}(x) \leq z$ $\underline{and}$ $\varphi_{\pi_1 k}(x) = y$ $\underline{then}$ $y+1$ $\underline{else}$ $y$ $\underline{fi}$
             $\underline{fi}$
          );
```

Informally: to evaluate $\varphi_{\delta(i)}$ for x see whether some pair $\langle x,y\rangle$ is enumerated by $\varphi_{\rho(i)}$. If $\pi_1\varphi_{\rho(i)}$ becomes too large take $\varphi_{\tau(i)}(x)$; otherwise let k be the rank of <x,y> in the enumeration by $\varphi_{\rho(i)}$ and let z be the value of $\varphi_i^2(x,y)$ (which by now is known to be finite). If the k-th program at x terminates within z steps and computes the value y then $\varphi_{\delta(i)}$ is set to compute a different function by setting $\varphi_{\delta(i)}(x) = y+1$, otherwise the value y is safe.

The above diagonalization procedure yields for $\pi_1\mathcal{D}\varphi_i^2$ infinite a total function $\Lambda\varphi_{\delta(i)}(x)$ which is not included in $H_{\varphi_i^2}$.

An interesting variant of δ is the following transformation:

```
$\varphi_{\delta'(i)}(x) \Leftarrow$ ($\underline{int}$ $k = \mu z[\pi_1\varphi_{\rho(i)}(z) = x]$;
            $\underline{int}$ $y = \pi_2\varphi_{\rho(i)}(k)$;
            $\underline{int}$ $z = \varphi_i^2(x,y)$;
            $\underline{if}$ $\Phi_{\pi_1 k}(x) \le z$ $\underline{then}$ $\underline{loop}$ $\underline{else}$ $y$ $\underline{fi}$);
```

For indices i such that $\pi_1\mathcal{D}\varphi_i^2$ is infinite we have $\varphi_{\delta'(i)} \sqsubseteq \varphi_{\theta(i)}$ and $\varphi_{\delta'(i)} \notin H_{\varphi_i^2}$.

It should be noted that for honesty classes we have only looked at the simplest "expensiveness" relation "f $\notin H_R$". One might consider also the relation "for each program φ_j for f the relation $\Phi_j(x) > R(x,f(x))$ holds almost everywhere". We shall not consider this topic any further.

Because of the technicalities involved in lemma 2.2.6 there is no nice analogue of the compression theorem for arbitrary honesty classes. For honesty classes with total names such an analogue is a straightforward corollary to the ordinary compression theorem.

Finally it should be noted that the two gap-theorems can be generalized for honesty classes. This subject is treated in chapter 3.2. We will also show that the assertion of the naming theorem becomes invalid for honesty classes: there exists no uniform method to rename honesty classes by a measured set of names (cf. chapter 3.4).

CHAPTER 2.3.

ARITHMETICAL COMPLEXITY OF RESOURCE-BOUND CLASSES

2.3.1. CLASSES OF PROGRAMS

By their definitions the classes of programs are Σ_2-sets. This can be seen as follows:

$$\varphi_i \in F_{\varphi_j} \ \underline{\mathit{iff}}\ \exists y \forall x_1 \forall x_2 \forall x_3 [\Phi_j(x_1) \neq x_2 \ \underline{\mathit{or}}\ \varphi_j(x_1) \neq x_3 \ \underline{\mathit{or}}\ x_1 \leq y \ \underline{\mathit{or}}\ \Phi_i(x_1) < x_3],$$

$$\varphi_i \in G_{\varphi_j^2} \ \underline{\mathit{iff}}\ \exists y \forall x_1 \forall x_2 \forall x_3 \forall x_4 \forall x_5$$
$$[\Phi_j^2(x_1,x_2) \neq x_3 \ \underline{\mathit{or}}\ \varphi_j^2(x_1,x_2) \neq x_4 \ \underline{\mathit{or}}\ x_1 \leq y \ \underline{\mathit{or}}$$
$$\Phi_j(x_1) \neq x_5 \ \underline{\mathit{or}}\ x_5 \leq x_4 \ \underline{\mathit{or}}\ \varphi_j(x_1) \neq x_2].$$

For the class $F^{E}_{\varphi_i}$ where $E = (E_i)_i$ is recursively presentable one must replace "$x_1 \leq y$" by "$x_1 \in E_y$" in the above expression to provide a Σ_2-definition for $F^{E}_{\varphi_i}$.

We first consider the classes F_t. In [Bl 66] M. BLUM shows that for the Turing time measure the class F_t with $t = \lambda x[x+1]$ is not recursively enumerable. F.D. LEWIS shows in [Ee 70] that for sufficiently large total t the set of indices $\{i \mid \varphi_i \in F_t\} \cap \underline{\mathit{total}}$ is Σ_2-complete; for small t this set may be of any recursive enumerable degree. His proof is based upon diagonalization.

Below we prove the same result using the recursion theorem.

<u>LEMMA 2.3.1</u>. For sufficiently large total t there exists a transformation τ satisfying:

(i) $\varphi_{\tau(i)}$ is total.
(ii) $\varphi_{\tau(i)} \in F_t$ $\underline{\mathit{iff}}$ $i \in \mathit{finite}$.

<u>COROLLARY 2.3.2</u>. For sufficiently large total t, F_t is Σ_2-complete.

<u>PROOF</u>. Define the transformation σ by:

```
$\varphi_{\sigma(i,j,k)} \Leftarrow$ if $\Phi_j(\pi_1 x) \neq \pi_2 x$ then $0$
                  elif $\Phi_i(x) \leq \varphi_k(x)$ then $\varphi_i(x)+1$
                  else $1$
                  fi
```

By the recursion theorem there exists a transformation ρ such that $\varphi_{\sigma(\rho(j,k),j,k)} = \varphi_{\rho(j,k)}$. Repeating the argumentation from the inefficiency lemma one verifies that

$$\varphi_{\rho(j,k)}(\langle x,y\rangle) = \underline{if}\ \Phi_j(x) \neq y\ \underline{then}\ 0\ \underline{elif}\ \varphi_k(\langle x,y\rangle) < \infty\ \underline{then}\ 1\ \underline{else}\ \infty\ \underline{fi}$$

and

$$\Phi_{\rho(j,k)}(\langle x,y\rangle) > \varphi_k(\langle x,y\rangle) \quad \text{whenever} \quad \varphi_{\rho(j,k)}(\langle x,y\rangle) = 1.$$

Since $\varphi_{\rho(j,k)}(x) = 0$ *iff* $\Phi_j(\pi_1 x) = \pi_2 x$ we can define a total function φ_n by:

$$\varphi_n(x) = max\{\underline{if}\ \Phi_j(\pi_1 x) = \pi_2 x\ \underline{then}\ \Phi_{\rho(j,k)}(x)\ \underline{else}\ 0\ \underline{fi} \mid j,k \leq x\}.$$

Now we have

$$\forall j \forall k \overset{\infty}{\forall} x[\varphi_{\rho(j,k)} = 0 \Rightarrow \Phi_{\rho(j,k)}(x) < \varphi_n(x)].$$

Let $t = \varphi_k$ be a total function such that $\varphi_n \underline{\propto} t$, and define τ by $\tau(j) = \rho(j,k)$. Then τ is the transformation requested by the lemma. $\varphi_{\tau(j)}$ is total (since φ_k is total), and for almost all x we have $\Phi_{\tau(j)}(x) <$ $< \varphi_n(x) \leq t(x)$ if $\Phi_j(\pi_1 x) \neq \pi_2 x$ whereas $\Phi_{\tau(j)}(x) > \varphi_k(x) = t(x)$ if $\Phi_j(\pi_1 x) = \pi_2 x$.
Clearly j $\in$ *finite* *iff* $\varphi_{\tau(j)} \in F_t$. □

If $\mathcal{E}$ is a recursively presentable class of sets of exceptional points the following modification shows that $C_t^{\mathcal{E}}$ is Σ_2-complete for sufficiently large t. Construct an infinite recursive set A with # A ∩ E finite for each E ∈ $\mathcal{E}$ (see lemma 2.1.10). Now $\varphi_{\rho(j,k)}(x)$ is going to be a program computing an "expensive" value 1 only if x is the k-th element of A (in some enumeration of A) and if $\Phi_j(\pi_1 k) = \pi_2 k$. The remainder of the construction is unchanged.

For honesty classes we can do without the "sufficiently large" assumption on the name.

PROPOSITION 2.3.3. Let R be a total function. Then G_R is Σ_2-complete.

PROOF. By the inefficiency lemma we can define a transformation τ such that $\varphi_{\tau(i)} = \varphi_i$ *and* $\Phi_{\tau(i)} > R\square\varphi_i$. Now $\varphi_{\tau(i)} \in G_R$ *iff* $i \in$ *finite*. □

One should beware not to confuse our classes of programs with the index sets for the corresponding classes of functions (cf. chapter 1.4). For example we consider the classes ΩC_t. Let σ be defined by:

$$\varphi_{\sigma(i)}(x) \Leftarrow \underline{if}\ \varphi_i(x) < \infty\ \underline{then}\ 0\ \underline{else}\ \infty\ \underline{fi}.$$

It is not difficult to construct a measured set of 0-∞ valued functions containing all 0-∞ functions with cofinite domain. From this one concludes the existence of some function t_0 such that C_{t_0} contains all those functions. Now for total $t \geq t_0$, σ reduces the Σ_3-complete set *cofinite* to ΩC_t. This shows that ΩC_t is Σ_3-complete for sufficiently large total t.

By the absense of domain conditions this result differs from corresponding results by E.L. ROBERTSON [Rb 71].

2.3.2. CLASSES OF FUNCTIONS

It is a well-known result that a complexity class C_t is recursively presentable provided this class contains all finite modifications of one of its members. See [Le 70], [LR 72], [Bo 72], [HH 71]. This can be derived from a general enumerability criterium which we describe below.

The non-enumerability results on resource-bound classes are more interesting. F.D. LEWIS and E.L. ROBERTSON constructed independently a non-enumerable complexity class for some specific measure. (It has been proposed as one of the "naturalness"-criteria that such classes do not exist.)

The other result states the non-enumerability of the class $C_t^E \cap R$, where E is a recursively presentable class of sets of exceptional points containing an infinite set. This result holds only because of domain conditions, for by our general enumerability principle we can prove that the classes C_t^E are recursively presentable for sufficiently large t.

DEFINITION 2.3.4. Let X be a class of functions. A *way-out strategy* for X is a recursive transformation which maps (indices of) finite functions onto programs for members of X extending these finite functions.

The index of a finite function is understood to be the index of a sequence whose elements are the points in the graph of this function.

Note that if σ is a way-out strategy for X and if $X \in Y$ then σ is also a way-out strategy for Y.

LEMMA 2.3.5. Let X be a Σ_2-presentable class of functions and let σ be a way-out strategy for X, then X is recursively presentable.

PROOF. Our proof is a straightforward translation of the earlier proof of enumerability of C_t (cf, [Bo 72]).

Let A be a Σ_2-set such that $X = \{\Lambda\varphi_i \mid i \in A\}$. There exists a total recursive Boolean function B such that:

$$i \in A \ \underline{\textit{iff}} \ \overset{\infty}{\forall} x[B(i,x)].$$

We define a transformation $\tau(i,j)$ such that $\varphi_{\tau(i,j)}$ simulates a dovetailed computation of φ_i. Simultaneously the values of $B(i,y)$ for $y \geq j$ are computed. Upon finding an argument $y \geq j$ with $B(i,y) = \underline{\textit{false}}$ the way-out strategy is invoked and the program φ_i is replaced by an extension in X of the finite segment of φ_i which was enumerated already. The function computed by $\varphi_{\tau(i,j)}$ is the function whose graph is enumerated in this way. (A formalized definition of τ is given in the appendix.)

To see that $(\varphi_{\tau(i,j)})_{i,j}$ is indeed a recursive presentation of X we consider two cases.

(i) $i \notin A$. In this case the way-out strategy is invoked regardless the value of j. Consequently $\varphi_{\tau(i,j)}$ is a member of X.

(ii) $i \in A$. Now there exists a value j such that for all $y \geq j$ $B(i,y)$ holds. Consequently the way-out strategy is never invoked and therefore $\varphi_{\tau(i,j)} = \varphi_i$ which function happens to be a member of X. Moreover each member of X is included in this way in the sequence $\varphi_{\tau(i,j)}$. □

The next lemma provides us with classes for which a way-out strategy exists.

LEMMA 2.3.6. Let $(\varphi_{j_k})_k$ be a sequence of total functions, then there exists a total function t such that for each k, $\varphi_{j_k} \in F_t$.

PROOF. Take $t = \max\{\Lambda\Phi_{j_k}(x) \mid k \leq x\}$. $\square$

It is not hard to see that for the class of functions which are total and almost everywhere zero, a way-out strategy can be given. Moreover, this class is entirely included in C_{t_0} for some total t_0. From this we conclude:

COROLLARY 2.3.7. For sufficiently large t the classes C_t^E are recursively presentable. The same holds for the classes C_t^E provided E is recursively presentable.

(The second part follows since $C_t \subseteq C_t^E$; hence a way-out strategy for C_t is one for C_t^E also.)

For honesty classes the enumerability construction yields in fact the so-called "equivalence between honest and measured sets" (cf. chapter 2.1).

THEOREM 2.3.8. (=2.1.5). Let R be total. Then H_R is presented by a measured set of programs. Conversely each measured set is subset of an honesty class with total name.

PROOF. Let $(\gamma_i)_i$ be a measured set, and let σ be a transformation such that $\gamma_i = \varphi_{\sigma(i)}$. If we define the total function R by
$R = \lambda x,y[max\{\underline{if}\ \gamma_i(x) = y\ \underline{then}\ \Phi_{\sigma(i)}(x)\ \underline{else}\ 0\ \underline{fi} \mid i \leq x\}]$ then clearly $(\gamma_i)_i \subseteq H_R$. This proves the second assertion.

To prove the first assumption we use a way-out strategy into the class of functions with finite domain which are known to be R-honest regardless the size of R. This way H_R is recursively presented. The problem is to produce a measured transformation which presents H_R.

Define K by:

```
K = λi,j,k,x,y[if x ≤ j then(if $\Phi_i(x) \leq k$ and $\varphi_i(x) = y$ then 0 else 1 fi)
               elif $\exists z \leq j[\Phi_i(z) > k$ and $\Phi_i(z) \leq x]$ or
                    $\exists z \leq x[j < z$ and $\Phi_i(z) \leq x$ and $\Phi_i(z) > R(z,\varphi_i(z))]$ then 1
               elif $\Phi_i(x) \leq R(x,y)$ and $\varphi_i(x) = y$ then 0
               else 1 fi].
```

K is a total 0-1-valued function. Since $K(i,j,k,x,y) = 0$ implies $\varphi_i(x) = y$

for given i,j,k,x, there exists at most a single y such that $K(i,j,k,x,y)=0$. Consequently we may interpretate K to be the decision procedure for some measured transformation τ defined by

$$\varphi_{\tau(i,j,k)}(x) \Leftarrow \mu z[K(i,j,k,x,z)].$$

Clearly $\varphi_{\tau(i,j,k)} \sqsubseteq \varphi_i$. Now the definition of $\varphi_{\tau(i,j,k)}(x)$ depends on whether $x \le j$ or not.

If $x \le j$ we have $\varphi_{\tau(i,j,k)}(x) = \varphi_i(x)$ provided $\Phi_i(x) \le k$; otherwise $\varphi_{\tau(i,j,k)}(x) = \infty$. For $x \le j$ we have $\varphi_{\tau(i,j,k)}(x) = \varphi_i(x)$ provided the following three conditions are satisfied.

(i) $\Phi_i(x) \le R(x,\varphi_i(x))$ i.e. φ_i is R-honest at x.
(ii) It is impossible to detect a violation $k < \Phi_i(y) \le x$ for $0 \le y \le j$.
(iii) It is impossible to detect a violation $R(y,\varphi_i(y)) < \Phi_i(y) \le x$ for $j < y < x$.

If one of the conditions is violated, $\varphi_{\tau(i,j,k)}(x) = \infty$. Clearly the conditions (ii) and (iii) once violated for x_0 remain violated for all larger x. Consequently $\varphi_{\tau(i,j,k)}$ is a finite function unless

(*) for $x \le j$, $\Phi_i(x) = \infty$ or $\Phi_i(x) \le k$ and
(**) for $j < x$, $\Phi_i(x) < R(x,\varphi_i(x))$.

Validity of (*) and (**) implies that φ_i is R-honest. Conversely for R-honest φ_i, parameters j and k can be selected such that (*) and (**) are satisfied. In this case $\varphi_{\tau(i,j,k)} = \varphi_i$. This shows that indeed H_R is recursively presented by τ. □

Next we consider non-enumerable classes.

DEFINITION 2.3.9. Let σ be a transformation. If the sequence $(\psi_i)_i$ defined by $\psi_i = \varphi_{\sigma(i)}$ is an effective enumeration then the pair of sequences $((\psi_i)_i,(\Psi_i)_i) = \Psi$ with $\Psi_i = \Phi_{\sigma(i)}$ is called a *sub-measure* of the measure $((\varphi_i)_i,(\Phi_i)_i) = \Phi$. Notation $\Psi \subseteq \Phi$ (by σ).

THEOREM 2.3.10. Let Φ be some measure. Then for sufficiently large t, a sub-measure $\Psi \subseteq \Phi$ can be defined such that C_t^{Ψ} is not recursively presentable.

(The super-index Ψ in C_t^{Ψ} refers to the measure relative to which the class is defined.)

PROOF. The set $\mathbb{N} \setminus \mathit{halts}$ is a standard example of a Π_1-complete set. We define a transformation ρ by

$$\varphi_{\rho(i)}(x) \Leftarrow \underline{if}\ \Phi_i(i) \le x\ \underline{then}\ \underline{loop}\ \underline{else}\ i\ \underline{fi}.$$

Consequently $\varphi_{\rho(i)}$ is total if $\varphi_i(i)$ diverges; otherwise $\mathcal{D}\varphi_{\rho(i)}$ is finite. Since $x \in \mathcal{D}\varphi_{\rho(i)}$ $\underline{iff}$ $\Phi_i(i) > x$, there exists a total function t_0 satisfying:

$$\forall i \overset{\infty}{\forall} x[x \in \mathcal{D}\varphi_{\rho(i)} \Rightarrow \Phi_{\rho(i)}(x) \le t_0(x)].$$

Assume that t is a total function, $t \ge t_0$.

By the inefficiency lemma one constructs a transformation τ such that $\varphi_{\tau(i)} = \varphi_i$ and $\Lambda\Phi_{\tau(i)} > t$. We now define our sub-measure Ψ by defining σ.

Let $\sigma = \lambda i[\underline{if}\ \underline{even}\ i\ \underline{then}\ \tau(i\div 2)\ \underline{else}\ \rho(i\div 2)\ \underline{fi}]$; then $(\varphi_{\tau(i)})_i$ consists of all programs $\varphi_{\rho(i)}$ and $\varphi_{\tau(i)}$.

It is not difficult to prove that the sequence $(\varphi_{\sigma(i)})_i$ is indeed an effective enumeration. However, by the choice of τ the class F_t^{Ψ} only contains programs $\varphi_{\rho(i)}$; in particular one has:

$$F_t^{\Psi} = \{\varphi_{\rho(i)} \mid i \in \mathbb{N} \setminus \mathit{halts}\}.$$

Consequently $C_t^{\Psi} = \{\lambda x[k] \mid k \in \mathbb{N} \setminus \mathit{halts}\}$.
Assume that C_t^{Ψ} is recursively presented by η:

$$\{\Lambda\varphi_{\eta(i)} \mid i \in \mathbb{N}\} = C_t^{\Psi}.$$

Then also

$$\{\varphi_{\eta(i)}(0) \mid i \in \mathbb{N}\} = \mathbb{N} \setminus \mathit{halts}.$$

This is a contradiction since $\mathbb{N} \setminus \mathit{halts}$ is known not to be recursively enumerable. □

In his thesis [Ba 70] L. BASS proves that for sufficiently large total t the classes C_t^{E} are not recursively presentable provided E contains an infinite set. Seemingly this contradicts our earlier result for recursively presentable E. However there is no contradiction since the classes considered by L. BASS consist of total functions only.

PROPOSITION 2.3.11. If the recursively presentable class of sets of exceptional points $\mathcal{E}$ contains an infinite set E then there exists for sufficiently large t no recursive presentation for $C_t \cap \mathcal{R}$.

PROOF. Writing "$x \in E$" for "$g(x) = 0$" where g is the characteristic function for E, we define the transformation τ by:

$$\varphi_{\tau(i)}(x) \Leftarrow \underline{if}\ x \in E\ \underline{then}\ \varphi_i(x)\ \underline{else}\ 0\ \underline{fi}.$$

Let t_0 be defined by:

$$t_0 = \lambda x[\underline{if}\ x \in E\ \underline{then}\ 0\ \underline{else}\ max\{\Phi_{\tau(i)}(x) \mid i \leq x\}\ \underline{fi}].$$

Now for $t \geq t_0$ one has $\varphi_{\tau(i)} \in F_t^E$. Let η be a recursive presentation of $C_t^E \cap \mathcal{R}$. Define a function n so that

$$\begin{aligned} n(x) = {} & \underline{if}\ x \notin E\ \underline{then}\ 0 \\ & \underline{elif}\ x = 0\ \underline{then}\ 1 \\ & \underline{else}\ \mu z \leq x[\forall y{<}x[n(y) < z]] \\ & \underline{fi}. \end{aligned}$$

So for $x \in E$, $n(x) = \#\ [0,x] \cap E$.

Define $f = \varphi_k$ by $f = \lambda x[\underline{if}\ x \in E\ \underline{then}\ \varphi_{\eta(n(x))}(x){+}1\ \underline{else}\ 0\ \underline{fi}]$. Since $\varphi_k = \varphi_{\tau(k)}$, $f \in C_t^E$. Moreover, f is total. However by considering the m-th element of E one finds that $f \neq \varphi_{\eta(m)}$. Consequently $f \notin \{\varphi_{\eta(i)} \mid i \in \mathbb{N}\}$. This completes the proof. □

Clearly this proof collapses if we try to diagonalize over a presentation for the complete set $C_t^{\mathcal{R}}$.

An analogous contradiction for ordinary classes does not arise. The classes $C_t \cap \mathcal{R}$ are recursively presentable for sufficiently large total t. The proof is based on our enumeration technique, using a way-out strategy into the set of total functions which are almost everywhere zero, using moreover an extra test against divergence on an initial segment.

CHAPTER 2.4.

SET THEORETICAL PROPERTIES OF RESOURCE-BOUND CLASSES

2.4.1. CLOSURE PROPERTIES OF RESOURCE-BOUND CLASSES

Resource-bound classes are sets of programs or functions, and consequently their behaviour under set theoretical operations and relations has received the attention of several investigators.

To conclude our survey on the known theory of resource-bound classes we mention a number of results relating these set theoretical properties of resource-bound classes. Proofs in this chapter are omitted. They can be found in the literature referenced to, and also in the more extended version of this chapter in [EB 74]. We restrict outself to complexity and honesty classes.

FINITE INTERSECTION

It is clear from the definition that resource-bound classes of programs are closed under finite intersection, provided the names are total. For example the set $F_t \cap F_u = F_v$ where $v = \lambda x[\underline{min}(t(x),u(x))]$. If t and u are partial functions v still is a "name" for $F_t \cap F_u$. However v need not be recursive.

This problem is solved by replacing t and u by names from a fixed measured set (this is possible by the naming theorem). The minimum of two measured functions is again measured and hence a recursive function.

For honesty classes we have the same problem. However, we no longer can apply the naming theorem since this theorem is invalid for honesty classes (cf. 3.4.2). Still the intersection of two honesty classes having partial names can be shown to be an honesty class. We will give the proof in 3.4.5.

The theory on classes of functions is uninteresting. By specific examples one shows that for specific measures the intersection of certain specific complexity or honesty classes is not again such a class.

There are no known results indicating whether this is a pathological behaviour at the bottom of the hierarchy or whether this behaviour occurs "generically".

FINITE UNION

Neither the classes of programs nor the classes of functions are closed under finite union. A theorem by E.M. McCREIGHT [MC 69] shows that for sufficiently large total t a total function t' can be constructed such that $C_t \cup C_{t'}$ is not a complexity class. The same result then follows for the classes F_t.

By a non-trivial modification an analogous result can be proved for honesty classes. The difficulty is to find total R and R' such that $H_R \cup H_{R'}$ is not an honesty class. If R and R' are partial an easy example is given by R = λx,y[*if* x=0 *then* *loop* *else* 0 *fi*] and R' = λx,y[*if* x=1 *then* *loop* *else* 0 *fi*]. Under this assumption H_R ($H_{R'}$) contains arbitrary expensive 0-∞ (1-∞) valued functions, but no non-trivial 0-1-∞ valued function is included in their union. Proofs are given in [EB 74].

INTERSECTION OF A DECREASING CHAIN

For both classes of functions and programs the results are negative. For complexity classes of programs an example of a decreasing chain of classes such that the intersection is not a class has been given by E.L. ROBERTSON [Rb 71]. His example can be translated to give an example for honesty classes as well.

The result for classes of functions which is based upon the speed-up theorem is due to L.J. BASS [Ba 70]. His proof can be generalized for honesty classes, but one uses essentially the presence of partial functions in a honesty class. (See [EB 74]).

UNION OF AN INCREASING CHAIN

This is the only set theoretical closure property for which the results are positive.

By the *union theorem* of E.M. McCREIGHT [MC 69], the union of a sequence C_{t_i} (F_{t_i}) with total names t_i such that $t_i \leq t_{i+1}$ is again a complexity class.

In part 3 this theorem is proved together with a number of generalizations. First we show that the names $(t_i)_i$ may be partial as well, supposed the relation $t_i \leq t_{i+1}$ includes the domain condition $\mathcal{D}t_i \supseteq \mathcal{D}t_{i+1}$.

Moreover the condition on the names may be replaced by the condition "$F_{t_i} \subseteq F_{t_{i+1}}$" on the classes F_{t_i} themselves. (This latter generalization does not hold for the classes of functions.)

For honesty classes we have some weaker results. The union theorem holds, provided $(R_i)_i$ is an increasing sequence. The theorem holds also if $(R_i)_i$ is a total sequence such that $G_{R_i} \subseteq G_{R_{i+1}}$. It is unknown whether this last assertion holds also for partial $(R_i)_i$. An implicit connection between the union theorem and the naming theorem is responsible for these weaker results.

Investigation whether the union theorem holds also for the classes C_t^{E} shows that this depends on whether the class E is recursively presentable or not. If E is recursively presentable the theorem holds, as will be derived from our abstract approach in Part 2. For non-recursively presentable E there exists a counterexample. Cf. [Ba 70].

2.4.2. EMBEDDING THEOREMS

A *partial order* R on $\mathbb{N}$ is an antisymmetric and transitive relation on $\mathbb{N}$. It is called *recursive* provided the relation xRy is recursive.

Let S be a partial order on some set X and let f be a (partial) function from $\mathbb{N}$ into X. We say that a partial order R on $\mathbb{N}$ is *embedded in* (X,S,f) *by* g provided $\mathcal{R}g \subseteq \mathcal{D}f$ <u>and</u> $f(g(x))\, S\, f(g(y))$ <u>*iff*</u> xRy. The embedding is called *effective* if g is a recursive function.

The triple (X,S,f) is called a *universal partial order* provided any recursive partial order R on $\mathbb{N}$ can be embedded effectively into it.

Abstract complexity theory has yielded several universal partial orders.

The simplest example is described as follows:
Let X be the system of all complexity classes, ordered by inclusion. f maps the integer j onto the class with name φ_j. Then $(X,\subseteq,f)$ is universal.

This result is essentially due to E.M. McCREIGHT [MC 69]. He describes the construction in a context where he proves the existence of uncountably many different complexity classes (with non-recursive names).

A more interesting universal partial order is the following. For X we take the set $\mathcal{R}$ of total recursive functions, and for f we take the map Λ. A partial order on $\mathcal{R}$ is defined as follows. We say that f is cheaper than g provided some program for f is faster almost everywhere than each program for g. Notation f <u>*cheap*</u> g.

More formally:

$$f \underline{\mathit{cheap}}\ g\ \underline{\mathit{iff}}\ \exists i[f = \Lambda\varphi_i\ \underline{\mathit{and}}\ \Lambda\Phi_i\ \underline{\mathit{ncomp}}\ g].$$

An analogous relation *cheapv* is defined by

$$f\ \underline{\mathit{cheapv}}\ g\ \underline{\mathit{iff}}\ \exists i[f = \Lambda\varphi_i\ \underline{\mathit{and}}\ \Lambda\Phi_i\ \underline{\mathit{ncompv}}\ g].$$

The *embedding theorem* by E.M. McCREIGHT [MC 69] states that $(R,\mathit{cheapv},\Lambda)$ is universal (from this the universality of $(X,\subseteq,f)$ becomes a corollary.)

This result has been strengthened by D. ALTON [Al 73] and R. MOLL [Mo 73] who have shown that the "representing functions" may be separated by large "gaps". For a total effective operator Γ one defines a relation R_Γ by:

$$f\ R_\Gamma\ g\ \underline{\mathit{iff}}\ \exists i[f = \Lambda\varphi_i\ \underline{\mathit{and}}\ \Gamma(\Lambda\Phi_i)\ \underline{\mathit{ncomp}}\ g].$$

The generalized embedding theorem states that the triple (R,R_Γ,Λ) still is universal in the sense that each partial order can be embedded into it.

For honesty classes only the order by inclusion makes sense. It is almost trivial that this yields a universal partial order.

At this place we should mention an interesting degeneracy of the complexity classes modulo sets of exceptional points. The definition of a class of sets of exceptional points is more or less dual to the definition of a free filter in set theory. Following this idea we may consider a free ultrafilter F on $\mathbb{N}$. For E we take the class of all recursive complements of members of F. It is not hard to prove that E is a class of sets of exceptional points, which has the property that for each covering of $\mathbb{N}$ by two disjoint recursive sets A and B, either $A \in E$ or $B \in E$.

This has the strange consequence that each pair of classes C_t^E and C_u^E is inclusion-comparable, provided their names are total. Consequently the complexity classes mod E with total names are linearly ordered by inclusion!

PART 3

ABSTRACT RESOURCE-BOUND CLASSES

{*If We opened for the unbelievers a gate in heaven and they ascended through it higher and higher, still they would say: "Our eyes were dazzled: truly, we must have been bewitched".*
Koran 15, ed. N.J. Dawood}

CHAPTER 3.1.

ACCEPTANCE RELATIONS

3.1.1. INTRODUCTION

The concept of a complexity class of recursive functions has proved to be a useful tool in the theory of recursive functions. It has been used to define new hierarchies of classes of recursive functions and it has given a new understanding of other hierarchies which were defined previously by other means.

Furthermore the system of complexity classes has interesting properties on its own. The most important of these were formulated in Part 2 of this treatise. In particular, the compression theorems, gap theorems as well as the union and naming theorem should be mentioned.

The above theorems can be divided into two classes. In the first place we have those theorems which use in their proof a diagonalization construction. The compression theorems are examples of this type of result. Also the speed-up theorems (which are not theorems on complexity classes) can be considered to be of this type. We always use the finiteness of a run-time $\Phi_i(x)$ in the construction of some function f to make f and φ_i different by defining $f(x) = \varphi_i(x) + 1$. This means that we use the first Blum axiom.

In the second place we have theorems which only use the system of run-times $(\Phi_i)_i$ as well as the fact that these run-times form a measured set. The gap theorems and the union and naming theorems are theorems of this second type. The proofs do not use the first Blum axiom, and consequently no program φ_i is ever computed. What is used is the fact that the relation $\Phi_i(x) \leq z$ is decidable.

In Part 2 of this treatise we discussed the honesty classes as an alternative way to define subrecursive hierarchies. We stated that the gap theorems and the union theorem were true for honesty classes, but the proofs were not given.

Our motivation is that these theorems should be proved in an abstract formulation from which one derives them for complexity classes, honesty classes and lots of other types of similar classes at the same time.

The present part of this treatise discusses the above mentioned abstraction. The concept of an *acceptance relation* which is formally introduced in 3.1.3 is an abstraction from the fact that for the run-times $(\Phi_i)_i$ the relation $\Phi_i(x) \leq y$ is decidable (i.e. the property of a measured set).

As we shall see in 3.1.4 this acceptance relation remains equivalent to some measured set of generalized "run-times". However the two-valued logic of the relation $\Phi_i(x) \leq y$ is replaced by a "three-valued logic" of the acceptance relation as result of the introduction of the "truth-value" *void* which represents the answer "does not apply".

Although for both complexity classes and honesty classes an underlying acceptance relation can be defined, there remains a difference in the way the acceptance relation "bounds" the classes. In 3.1.2 we analyze the concept of an honesty class and we arive at the conclusion that honesty classes need some "alternative way of bounding" which we shall call *weak restriction* constrasting the *strong restriction* used in the definition of a complexity class.

This notion of weak restriction forms the justification for introducing the acceptance relation. In the case of strong restriction, *false* and *void* are identified, but for weak restriction their difference is crucial.

In general the proof of a theorem on strongly restricted classes is a straightforward translation of the proof given for the corresponding theorem on complexity classes in the literature. For the weakly restricted classes we use some tricks, varying from a "one-stage look-ahead" in the proof of the operator-gap theorem to a "two-phased test on violations" in the proof of the union theorem.

A surprising difference between strong and weak classes is the nonexistence of a naming theorem for weak classes. This negative result is the main theorem of this treatise.

The remaining sections of chapter 3.1 contain the definitions of abstract resource-bound classes, and a number of examples. Furthermore chapter 3.2 gives the proofs of the gap theorems. In chapter 3.3 we present two independent generalizations of the union theorem. Chapter 3.4 contains the discussion of the naming theorem, the negative result announced above and some related topics.

Throughout this part of the present treatise programs and algorithms are defined informally. In the appendix the reader will find a number of formalized descriptions, using the programming language introduced in Part 1, of some of the more complicated algorithms discussed in the text. This way the author hopes to separate the essential features of the designs of these algorithms from the inessential particularities of some implementation.

3.1.2. THE HONESTY CONDITION AS A THREE-VALUED PREDICATE

The definition of a complexity class of programs F_t can be given as

$$\varphi_i \in F_t \ \underline{\mathit{iff}} \ \overset{\infty}{\forall} x[\Phi_i(x) \leq t(x)]$$

where we use the following interpretation in case one or two sides of the inequality diverge.

The value t(x) should be considered to be a test-value. If the computation of t(x) diverges we have no testvalue and hence no test. This means that in testing the program φ_i no test is performed for the argument x. The formula given above should therefore read:

$$\varphi_i \in F_t \ \underline{\mathit{iff}} \ \overset{\infty}{\forall} x[x \in \mathcal{D}t \ \underline{\mathit{imp}} \ \Phi_i(x) \leq t(x)].$$

If t(x) converges and yields a value z we recall that $\Phi_i(x) \leq z$ stands for the finite disjunction:

$$\Phi_i(x) = 0 \ \underline{\mathit{or}} \ \Phi_i(x) = 1 \ \underline{\mathit{or}} \ \dots \ \underline{\mathit{or}} \ \Phi_i(x) = z$$

which can be tested componentwise. In this interpretation $\Phi_i(x) \leq z$ is *false* whenever $\Phi_i(x)$ diverges.

There is a difference with the case where $\Phi_i(x)$ is finite but greater than z. If $z < \Phi_i(x) < \infty$ then $\Phi_i(x) \leq z'$ will become *true* for a sufficiently large z'. If, however, $\Phi_i(x)$ diverges then $\Phi_i(x) \leq z'$ will never be *true* no matter how large z' is chosen.

The reader should keep this artifical distinction in mind while reading the sequel of this section.

In Part 2 we used a definition of an honesty class which can be formulated to read:

φ_i is R-honest if it satisfies the R-honesty condition $\Phi_i(x) \leq R(x,\varphi_i(x))$ for almost all x and

the honesty class H_R is the set of all recursive functions computed by an R-honest program.

The honesty condition $\Phi_i(x) \leq R(x,\varphi_i(x))$ is an *implicit condition*. Since $\varphi_i(x)$ occurs at the right-hand side, one must first compute $\varphi_i(x)$ to see whether φ_i is honest at x or not.

In trying to "localise" the honesty condition we can consider the following equivalent interpretations:

(i) [Global]. Enumerate the graph of φ_i and for each pair $\langle x,\varphi_i(x)\rangle$ enumerated, compute $R(x,\varphi_i(x))$. If this converges (R is not assumed to be total) test whether $\Phi_i(x) \leq R(x,\varphi_i(x))$. If the answer is no we have found a violation. The number of violations detected has to be finite.

(ii) [Argumentwise]. Compute $\varphi_i(x)$ and if this converges compute $R(x,\varphi_i(x))$. If this also converges, test whether $\Phi_i(x) \leq R(x,\varphi_i(x))$. If the answer is no then φ_i is not R-honest at x. The number of arguments x at which φ_i is not R-honest should be finite.

Both interpretations have the disadvantage that they invite us to execute infinite computations, which do not contribute any negative evidence. Furthermore we cannot isolate the bound function R from the honesty condition. Therefore, we still localize further; this obviates the second disadvantage mentioned above, but does not solve the first disadvantage.

(iii) [Local]. We say that φ_i *satisfies the honesty condition with value z at the argument-pair* $\langle x,y\rangle$ if $\Phi_i(x) \leq z$ <u>and</u> $\varphi_i(x) = y$.
We say that φ_i *violates the honesty condition with value z at the argument-pair* $\langle x,y\rangle$ if $\Phi_i(x) > z$ <u>and</u> $\varphi_i(x) = y$.
In all other cases ($\varphi_i(x) = \infty$ <u>or</u> $\varphi_i(x) \neq y$) we say that *the honesty condition with value z at the argument pair* $\langle x,y\rangle$ *does not apply to* φ_i.
φ_i should violate the honesty condition with value $R(x,y)$ at the argument-pair $\langle x,y\rangle$ for at most finitely many pairs $\langle x,y\rangle$.

In this local interpretation we have isolated the function R from the honesty condition. The price we have to pay is the introduction of a third answer "the honesty condition does not apply". This, however, is not unreasonable.

The happenstance that $R(x,y) = 0$ has no influence on R-honesty of φ_i at x if $\varphi_i(x)$ happens to be distinct from y. Also, if $\varphi_i(x)$ diverges then φ_i is R-honest at x, regardless the values of $R(x,y)$.

The computational situation is more complicated. We can test recursively whether φ_i satisfies an honesty condition with value z at $<x,y>$. To do so, we first test whether $\Phi_i(x) \le z$; if this is not the case then φ_i violates the honesty condition at $<x,y>$, or the honesty condition does not apply. If $\Phi_i(x) \le z$ we compute $\varphi_i(x)$ and if the answer is y then φ_i satisfies the honesty condition; otherwise the honesty condition does not apply.

After having decided that φ_i does not satisfy the honesty condition with value z at $<x,y>$, the situation is more complicated. Now we must compute $\varphi_i(x)$ to decide whether the condition was violated, or whether the condition did not apply. Again the danger of an infinite computation arises.

To give an overview of the formal properties of the three-valued predicate which is suggested above, we write $\underline{\text{Hon}}(i,x,y,z)$ for "the honesty condition with value z applied to φ_i at the argument-pair $<x,y>$". The possible outcomes are: "is satisfied", "is violated" and "does not apply".

We have the following properties:

(a) If $z' > z$ and $\underline{\text{Hon}}(i,x,y,z)$ is satisfied then $\underline{\text{Hon}}(i,x,y,z')$ is also satisfied.

(b) If $\underline{\text{Hon}}(i,x,y,z)$ does not apply then $\underline{\text{Hon}}(i,x,y,z')$ does not apply for all z'.

(c) If $\underline{\text{Hon}}(i,x,y,z)$ is violated then there exists a $z' > z$ so that $\underline{\text{Hon}}(i,x,y,z')$ is satisfied.

(d) The quadruples $<i,x,y,z>$ for which $\underline{\text{Hon}}(i,x,y,z)$ is satisfied form a recursive set.

(e) The quadruples $<i,x,y,z>$ for which $\underline{\text{Hon}}(i,x,y,z)$ is violated form a rerecursively enumerable set.

The properties (a) to (e) represent in fact everything we use about the honesty relation in the proofs of the union and gap theorems. They play a role similar to that of the fact that the run-times of programs form a measured set in the theory of complexity classes. Furthermore (e) can be derived from (a) to (d), and (b) and (c) can be formulated in a weaker way. In such a weaker formulation the properties described above will be used as axioms for the concept of an acceptance relation in 3.1.3.

Below, we give an interpretation which shows that a similar three-valued predicate can be defined to represent a complexity condition.

Let Cpl be defined by:

$$\underline{\text{Cpl}}(i,x,z) = \begin{cases} \text{is satisfied if } \Phi_i(x) \le z \\ \text{is violated if } z < \Phi_i(x) < \infty \\ \text{does not apply if } \Phi_i(x) \text{ diverges.} \end{cases}$$

Now Cpl has the same formal properties (a) ... (e) as Hon had before. However, in our interpretation of complexity classes at the beginning of this section, we have treated the two cases "is violated" and "does not apply" as being the same.

This identification is in fact the crucial difference between the definition of a complexity class and the definition of an honesty class. The difference is not the three-valuedness of the underlying relation but the way the classes are defined in terms of this three-valued predicate.

In order to be a member of the complexity class of programs F_t the program φ_i should satisfy for almost all $x \in \mathcal{D}t$ the condition $\underline{\text{Cpl}}(i,x,t(x))$. This type of restriction could be called *strong restriction*.

In order to be an R-honest program, the program φ_i should violate the condition $\underline{\text{Hon}}(i,x,y,R(x,y))$ for at most finitely many pairs $\langle x,y\rangle$ in $\mathcal{D}R$. This type of restriction could be called *weak restriction*.

The notions suggested above will be defined formally in the next section.

Although we give lots of other examples, the complexity classes and the honesty classes can be considered to be the "categorical" examples. The honesty classes however have the disadvantage of having two-dimensional names. We have available however another example of weakly restricted classes; the *weak complexity classes*. These classes were defined by taking the relation Cpl defined above and applying weak restriction.

The strongly restricted classes behave in many aspects similarly to the complexity classes. The whole theory would become uninteresting if this were the same for weakly restricted classes too. There are however a number of differences, the most remarkable being the absence of a naming theorem for weakly restricted classes.

3.1.3. FORMAL DEFINITIONS AND EXAMPLES OF ABSTRACT RESOURCE-BOUND CLASSES

DEFINITION 3.1.1. An *acceptance relation* A is a set-theoretical total function with three number arguments (say i, x, and z) and values in the three-element set $\{\underline{true}, \underline{false}, \underline{void}\}$ which satisfies the following axioms:

A1. Monotonicity: If $z < z'$ then

(A1a) $A(i,x,z) = \underline{true}\ \underline{imp}\ A(i,x,z') = \underline{true}$

(A1b) $A(i,x,z) = \underline{void}\ \underline{imp}\ A(i,x,z') = \underline{void}$

furthermore

(A1c) $A(i,x,z) = \underline{false}\ \underline{imp}\ \exists z'[A(i,x,z') = \underline{true}]$

A2. Computability

(A2) The predicate $A(i,x,z) = \underline{true}$ is recursive in i, x, and z.

REMARK 3.1.2. One should visualize the arguments i, x and z as playing the role of "index", "argument" and "testvalue". We shall have examples where i or x encode more information than does the index of a program or some argument of a computation.

REMARK 3.1.3. In the next section we shall prove that the concept of an acceptance relation is recursively equivalent to the concept of a measured set, which is in fact the content of the second Blum axiom. For future use one could think of a third axiom, expressing the fact that $A(i,x,z)$ is *true* forces a computation to terminate. Up to now we have no reasonable candidate for this axiom, and since we don't need it we omit it.

DEFINITION 3.1.4. Let A be an acceptance relation, and let t be a partial recursive function. The *set of indices strongly* A-*restricted by* t, denoted $F^A_S(t)$ is defined by

$$F^A_S(t) = \{i \mid \overset{\infty}{\forall} x[t(x) < \infty\ \underline{imp}\ A(i,x,t(x)) = \underline{true}]\}.$$

The *set of indices weakly* A-*restricted by* t, denoted $F^A_W(t)$ is defined by

$$F^A_W(t) = \{i \mid \overset{\infty}{\forall} x[t(x) < \infty\ \underline{imp}\ A(i,x,t(x)) \neq \underline{false}]\}.$$

CONVENTION 3.1.5. The notations introduced above are subjected to the following rules of simplification:

If the acceptance relation intended is clear from the context or if this relation is otherwise irrelevant we write $F_S(t)$ ($F_W(t)$) instead of $F_S^A(t)$ ($F_W^A(t)$).

If it is clear from the context that we have a strongly restricted class we write $F(t)$ ($F^A(t)$) instead of $F_S(t)$ ($F_S^A(t)$).

The index as such in general has no "physical meaning". However, in most of the examples i encodes some program and this program again computes some function. This means that the sets of indices defined above represent sets of programs and functions as well.

This encoding is formalized in our next convention.

CONVENTION 3.1.6. If the index i of an acceptance relation encodes a program then this program is denoted by *prog* i. If the index represents a function this function is denoted by *fun* i. If *prog* is defined we have always *fun* i = Λ(*prog* i).

In our examples we either have *prog* i = φ_i or *prog* i = $\varphi_{\pi_1 i}$. In the first case we usually use the word program instead of index.

The sets of programs and functions corresponding to the sets of indices defined above are the following:

DEFINITION 3.1.7. Let A be an acceptance relation for which *prog* and/or *fun* are defined. Then we have the following sets of programs (functions)

$G_S^A(t) = \{\underline{prog}\ i \mid i \in F_S^A(t)\}$ set of programs strongly A-restricted by t

$H_S^A(t) = \{\underline{fun}\ i \mid i \in F_S^A(t)\}$ set of functions strongly A-restricted by t

$G_W^A(t) = \{\underline{prog}\ i \mid i \in F_W^A(t)\}$ set of programs weakly A-restricted by t

$H_W^A(t) = \{\underline{fun}\ i \mid i \in F_W^A(t)\}$ set of functions weakly A-restricted by t.

Convention 3.1.5 extends to these classes as well:

CONVENTION 3.1.8. If the acceptance relation A is clear from the context we suppress the occurrence of the symbol A in the above notations. Furthermore, the symbol S can be deleted if it is clear from the context that the classes are strongly restricted.

All the classes defined in 3.1.4 and 3.1.7 together are called *Abstract Resource-Bound Classes*. We use the abbreviation ARBC for abstract resource-bound class. We also speak of strong classes (weak classes) instead of strongly restricted classes (weakly restricted classes).

The following definition introduces a technical term which is used frequently in our discussions.

<u>DEFINITION 3.1.9.</u> Let A be an acceptance relation and let t be a function. Let i be an index and let x be an argument. A *strong violation by i at x against t* occurs if $A(i,x,t(x))$ is defined and $A(i,x,t(x)) \neq \underline{\textit{true}}$. A *weak violation by i at x against t* occurs if $A(i,x,t(x))$ is defined and $A(i,x,t(x)) = \underline{\textit{false}}$.

The violations introduced above can be grouped together to form the violations by a given index, c.q. the violations at a certain argument etc.. The words "weak" and "strong" are used in an unnatural way. A weak violation is also a strong violation but conversely a strong violation is not necessarily also a weak violation. This peculiar use of the words "strong" and "weak" is motivated by the fact that an index i is contained in a strong (weak) class provided that there exist only finitely many strong (weak) violations by i against t.

We now give a number of examples of old and new classes which are abstract resource-bound classes.

<u>EXAMPLE 3.1.10.</u> Let *Cpl* be defined by

$$Cpl(i,x,z) = \begin{cases} \underline{\textit{true}}\ \underline{\textit{iff}}\ \Phi_i(x) \leq z \\ \underline{\textit{false}}\ \underline{\textit{iff}}\ z < \Phi_i(x) < \infty \\ \underline{\textit{void}}\ \underline{\textit{iff}}\ \Phi_i(x) = \infty. \end{cases}$$

This acceptance relation can be called the *complexity condition*. We have $\underline{\textit{prog}}\ i = \varphi_i$. We recognize the following complexity classes

$$C_t = H_S^{Cpl}(t) \quad \text{and} \quad F_t = G_S^{Cpl}(t).$$

Furthermore, there are the weakly *Cpl*-restricted classes, which were defined in part 2 as a particular type of honesty classes;

$$C_t^W = H_W^{Cpl}(t) \quad \text{and} \quad F_t^W = G_W^{Cpl}(t)$$

These weak complexity classes form an unnatural alternative for the usual complexity classes. To be in a weak complexity class a program must either be cheap to compute or otherwise the program must diverge. This could be the interpretation of "bounding" from someone who only has to pay for terminating computations. If the program loops forever (and is terminated abnormally by the operator) the computing centre pays. It is clear that this is not a realistic interpretation.

EXAMPLE 3.1.11. Let *Hon* be defined by

$$Hon(i,x,z) = \begin{cases} \underline{\mathit{true}}\ \underline{\mathit{iff}}\ \Phi_i(\pi_1 x) \le z\ \underline{\mathit{and}}\ \varphi_i(\pi_1 x) = \pi_2 x \\ \underline{\mathit{false}}\ \underline{\mathit{iff}}\ \Phi_i(\pi_1 x) > z\ \underline{\mathit{and}}\ \Phi_i(\pi_1 x) = \pi_2 x \\ \underline{\mathit{void}}\ \underline{\mathit{iff}}\ \varphi_i(\pi_1 x) \ne \pi_2 x\ \underline{\mathit{or}}\ \varphi_i(\pi_1 x) = \infty. \end{cases}$$

This acceptance relation is the honesty condition which was discussed in section 3.1.2. The honesty classes are now given by $H_R = H_{\mathcal{W}}^{Hon}(r)$ and $G_R = G_{\mathcal{W}}^{Hon}(r)$ where $r(<x,y>) = R(x,y)$. We have again $\underline{prog}\ i = \varphi_i$.

EXAMPLE 3.1.12. Let $(\gamma_i)_i$ be a measured set of functions. We define a corresponding acceptance relation Γ by:

$$\Gamma(i,x,z) = \begin{cases} \underline{\mathit{true}}\ \underline{\mathit{iff}}\ \gamma_i(x) \le z \\ \underline{\mathit{false}}\ \underline{\mathit{iff}}\ z < \gamma_i(x) < \infty \\ \underline{\mathit{void}}\ \underline{\mathit{iff}}\ \gamma_i(x) = \infty \end{cases}$$

In this example $\underline{prog}$ is not defined. One can define $\underline{fun}$ by putting $\underline{fun}\ i = \gamma_i$, but this does not correspond to the special case that $(\gamma_i)_i$ is the measured set of the run-times of the programs $(\Phi_i)_i$, in which case Γ and *Cpl* become the same acceptance relation.

The example of a measured set can be used in several situations to construct examples without having to extend the measured set to a complete system of run-times (by making all other programs much more expensive).

EXAMPLE 3.1.13. Let $E = (E_i)_i$ be a recursive presentable class of sets of exceptional points (cf. [Ba 70],[BY 71] and also Part 2). The acceptance relation *Cplex* is defined by:

$$Cplex(i,x,z) = \begin{cases} \underline{true}\ \underline{iff}\ x \in E_{\pi_2 i}\ \underline{or}\ \Phi_{\pi_1 i}(x) \le z \\ \underline{false}\ \underline{iff}\ x \notin E_{\pi_2 i}\ \underline{and}\ z < \Phi_{\pi_1 i}(x) < \infty \\ \underline{void}\ \underline{iff}\ x \notin E_{\pi_2 i}\ \underline{and}\ \Phi_{\pi_1 i}(x) = \infty. \end{cases}$$

In this example $\underline{prog}$ is defined by $\underline{prog}\ i = \varphi_{\pi_1 i}$.
The definition of the complexity classes modulo the class of sets of exceptional points $\mathcal{E}$ can be given as

$$C_t^{\mathcal{E}} = H_S^{Cplex}(t) \quad \text{and} \quad F_t^{\mathcal{E}} = G_S^{Cplex}(t).$$

In this example we have encoded the sets of exceptional points in the index of the program. In his thesis [Ba 70] BASS uses a similar trick in the proofs of the union and naming theorem for the classes $C_t^{\mathcal{E}}$ with recursively presentable $\mathcal{E}$. He encodes the sets of exceptional points in the priority numbers featuring in the algorithms used in these proofs.

The honesty condition may be relativized modulo sets of exceptional points in the same way.

<u>EXAMPLE 3.1.14</u>. Let $\mathcal{E}$ be as above. We define *Honex* by:

$$Honex(i,x,z) = \begin{cases} \underline{true}\ \underline{iff}\ \pi_1 x \in E_{\pi_2 i}\ \underline{or}\ (\Phi_{\pi_1 i}(\pi_1 x) \le z\ \underline{and}\ \varphi_{\pi_1 i}(\pi_1 x) = \pi_2 x) \\ \underline{false}\ \underline{iff}\ \pi_1 x \notin E_{\pi_2 i}\ \underline{and}\ \Phi_{\pi_1 i}(\pi_1 x) > z\ \underline{and}\ \varphi_{\pi_1 i}(\pi_1 x) = \pi_2 x \\ \underline{void}\ \text{otherwise.} \end{cases}$$

Again define $\underline{prog}$ by $\underline{prog}\ i = \varphi_{\pi_1 i}$ and let $r(\langle x,y\rangle) = R(x,y)$. Now we can introduce the honesty classes modulo sets of exceptional points

$$H_R^{\mathcal{E}} = H_W^{Honex}(r) \quad \text{and} \quad G_R^{\mathcal{E}} = G_W^{Honex}(r).$$

<u>EXAMPLE 3.1.15</u>. Let *Scpl* be the acceptance relation defined by:

$$Scpl(i,x,z) = \begin{cases} \underline{true}\ \underline{iff}\ \sum_{y \le x} \Phi_i(y) \le z \\ \underline{false}\ \underline{iff}\ z < \sum_{y \le x} \Phi_i(x) < \infty \\ \underline{void}\ \text{otherwise.} \end{cases}$$

Define $\underline{prog}$ i = φ_i. The strong classes corresponding to this acceptance relation may be called the *summed complexity classes*.

$$SC_t = H_S^{Scpl}(t).$$

Note that the members of SC_t are total functions, whenever $\mathcal{D}t$ is infinite. In this way, the condition of totality which is enforced by many authors, holds automatically.

There is an element of arbitrariness in this definition. If a program has long run-times in its initial segment it can be thrown out of the summed complexity class, although its asymptotic behaviour is excellent. This imperfection is eliminated in our next alternative definition of a summed complexity class.

EXAMPLE 3.1.16. Let *Scpl1* be defined by:

$$Scpl1(i,x,z) = \begin{cases} \underline{true}\ \underline{iff}\ \sum_{y\leq x} \Phi_{\pi_1 i}(y) \leq z + \pi_2 i \\ \underline{false}\ \underline{iff}\ z + \pi_2 i < \sum_{y\leq x} \Phi_{\pi_1 i}(y) < \infty \\ \underline{void}\ \text{otherwise.} \end{cases}$$

We define $\underline{prog}$ by $\underline{prog}$ i = $\varphi_{\pi_1 i}$. Now the summed complexity class $SC1_t = H_S^{Scpl1}(t)$ no longer carries the disadvantage mentioned above.

EXAMPLE 3.1.17. A type of classes which goes still further in the direction of the total complexity introduced by J.A. FELDMAN and P.C. SHIELDS [FS 72] can be defined if we replace the initial segments of $\mathbb{N}$ by the system of finite sets.
Let $(D_x)_x$ be a fixed enumeration of all finite subsets of $\mathbb{N}$. Then define the acceptance relation *Tcpl* by:

$$Tcpl(i,x,z) = \begin{cases} \underline{true}\ \underline{iff}\ \sum_{y\in D_x} \Phi_i(y) \leq z \\ \underline{false}\ \underline{iff}\ z < \sum_{y\in D_x} \Phi_i(y) < \infty \\ \underline{void}\ \text{otherwise.} \end{cases}$$

Again $\underline{prog}$ i = φ_i. For each function $\iota: P_f(\mathbb{N}) \to \mathbb{N}$ we define $CT_\iota = H_S^{Tcpl}(m)$ where $m(x) = \iota(D_x)$. CT_ι can be called a total complexity class. Again, a modification like example 3.1.16 is possible.

After this list of examples (which could easily be extended) we consider the totality of the functions and programs in our classes. In almost all examples the classes defined above contain partial functions or programs, even if the names are total. Many authors have restricted themselves in the study of complexity classes to total functions and programs, and have enforced different types of domain conditions in the definitions of classes with partial names.

We have not considered such extra domain conditions since it is clear that the classes defined using these conditions become more complex in the sense of the arithmetical hierarchy. Furthermore, for a number of theorems the restriction to total programs and functions does not influence the results. If a theorem states two classes to be equal, then these classes will also contain the same total functions.

If we restrict ourselves to total functions and programs the difference between strong and weak complexity classes disappears: For each partial function t we have:

$$C_t \cap R = C_t^W \cap R$$

and a similar equality holds for the classes of programs.

It is useful to consider at several occasions abstract resource-bound classes consisting of functions having more than one argument. The number of arguments (if greater than one) appears as an extra index in the notations. For example:

C_R^2 is the complexity class of all two-variable functions computed almost everywhere (i.e. for all pairs <x,y> with finitely many exceptions) within R(x,y) steps.

Using these extended notations we can formulate a relation between honest sets and weak complexity classes of semicharacteristic functions.

If f is a (partial) function then the *semicharacteristic function of the graph* of f, denoted $\underline{scg}$ f, is the function defined by:

$$\underline{scg}\ f = \lambda x,y[\underline{if}\ f(x) = y\ \underline{then}\ 0\ \underline{else}\ \infty\ \underline{fi}].$$

Let τ be a transformation of programs satisfying $\varphi^2_{\tau(i)} = \underline{scg}\ \varphi_i$. We may assume (after modification of the measure) that we also have

$$\Phi^2_{\tau(i)}(x,y) = \underline{if}\ \varphi_i(x) = y\ \underline{then}\ \Phi_i(x)\ \underline{else}\ \infty\ \underline{fi}.$$

ASSERTION 3.1.18. Under the above assumptions $G_R \leq_m F_R^{2W}$ by τ.

PROOF. We have

$$\begin{aligned}\varphi_i \in G_R\ &\underline{iff}\ \overset{\infty}{\forall}\langle x,y\rangle[\varphi_i(x) \neq y\ \underline{or}\ \Phi_i(x) \leq R(x,y)]\\ &\underline{iff}\ \overset{\infty}{\forall}\langle x,y\rangle[\varphi^2_{\tau(i)}(x,y) = \infty\ \underline{or}\ \Phi^2_{\tau(i)}(x,y) \leq R(x,y)]\\ &\underline{iff}\ \varphi^2_{\tau(i)} \in F_R^{2W}.\ \square\end{aligned}$$

In general there only exists a total function K so that

$$\forall i \overset{\infty}{\forall}\langle x,y\rangle[\varphi_i(x) = y\ \underline{imp}\ \Phi^2_{\tau(i)}(x,y) \leq K(x,y,\Phi_i(x))].$$

This only yields the implication

$$\varphi_i \in G_R\ \underline{imp}\ \varphi^2_{\tau(i)} \in F^{2W}_{\lambda x,y[K(x,y,R(x,y))]}\ ,$$

the converse implication not being generally true.

For the classes of programs we derive from Ass. 3.1.18:

$$f \in H_R\ \underline{imp}\ \underline{scg}\ f \in C_R^{2W}.$$

The converse implication may be spoiled by cheap programs for *scg* f which are not contained in the list of special programs $(\varphi^2_{\tau(i)})_i$.

3.1.4. BASIC PROPERTIES OF ACCEPTANCE RELATIONS.

The following lemma lists the essential properties of an acceptance relation.

LEMMA 3.1.19. Let A be an acceptance relation. Then we have:

(a) $A(i,x,z) = \underline{false}\ \underline{and}\ z' < z\ \underline{imp}\ A(i,x,z') = \underline{false}$

(b) $A(i,x,z) = \underline{void}\ \underline{and}\ z' < z\ \underline{imp}\ A(i,x,z') = \underline{void}$

(c) $A(i,x,z) = \underline{void}\ \underline{imp}\ \forall w[A(i,x,w) = \underline{void}]$

(d) $A(i,x,z) = \underline{false}\ \underline{and}\ A(i,x,z') = \underline{true}\ \underline{imp}\ z < z'$

(e) $\underline{not}(A(i,x,z) = \underline{\mathit{false}}\ \underline{\mathit{and}}\ A(i,x,z') = \underline{\mathit{void}})$
$\underline{not}(A(i,x,z) = \underline{\mathit{true}}\ \underline{\mathit{and}}\ A(i,x,z') = \underline{\mathit{void}})$

(f) $A(i,x,0) = \underline{\mathit{void}}\ \underline{\mathit{or}}\ \mu z[A(i,x,z) = \underline{\mathit{true}}] < \infty$

(g) the triplets $<i,x,z>$ with $A(i,x,z) = \underline{\mathit{false}}$ form a recursively enumerable set.

PROOF. (a) to (e) are evident from the definitions.

(f) If $A(i,x,0) = \underline{\mathit{true}}$ then the μ-operator yields 0. If $A(i,x,0) = \underline{\mathit{false}}$ then there exists a w for which $A(i,x,w)$ becomes $\underline{\mathit{true}}$ and consequently the μ-operator yields a finite value.

(g) By (A1c) and (c) we have

$$A(i,x,z) = \underline{\mathit{false}}\ \underline{\mathit{iff}}\ A(i,x,z) \neq \underline{\mathit{true}}\ \underline{\mathit{and}}\ \exists w[A(i,x,w) = \underline{\mathit{true}}].$$

The right-hand side clearly is a recursively enumerable predicate. □

PROPOSITION 3.1.20. Let A be an acceptance relation. Define the functions $(\alpha_i)_i$ by:

$$\alpha_i(x) = \mu z[A(i,x,z) = \underline{\mathit{true}}].$$

Then $(\alpha_i)_i$ is a measured set.

PROOF. The following equivalence is true:

$$\alpha_i(x) = y\ \underline{\mathit{iff}}\ A(i,x,y) = \underline{\mathit{true}}\ \underline{\mathit{and}}\ (y = 0\ \underline{\mathit{or}}\ A(i,x,y \dot{-} 1) \neq \underline{\mathit{true}}).$$

The right-hand side clearly is recursive in i, x and y. □

In example 3.1.12 we have constructed an acceptance relation from a measured set. Now we have given a converse construction. It is intuitively clear that the two constructions are inverses of each other. We shall prove in fact that this correspondence is a recursive equivalence between the concepts of an acceptance relation and a measured set.

In the case of the acceptance relation *Cpl* the measured set defined above is again the set of the run-times of the programs in the underlying complexity measure. This fact inspires the following definition:

DEFINITION 3.1.21. Let A be an acceptance relation. Let the sequence $(\alpha_i^A)_i$ be defined by $\alpha_i^A(x) = \mu z[A(i,x,z) = \underline{true}]$. Then $\alpha_i^A(x)$ is called the *A-run-time of index i at argument x*. The function α_i^A is called the *A-run-time of index i* and the measured set $(\alpha_i^A)_i$ is called the *set of A-run-times*.

By convention the symbol A is not written if the acceptance relation intended is clear from the context or otherwise irrelevant. If $\underline{prog}\ i = \varphi_i$ the run-time of index i is also called the run-time of the program φ_i. This definition allows us to discuss abstract resource-bound classes within the language of complexity classes. One should however be careful:

The *Hon*-run-time of φ_i at $<x,y>$ equals $\Phi_i(x)$ iff $\varphi_i(x) = y$, otherwise the *Hon*-run-time of φ_i at $<x,y>$ is infinite. This example shows that the *Hon*-run-time of φ_i is not the same as the "physical" run-time of φ_i.

To describe the recursive equivalence between measured sets and acceptance relations we first must define which are the "indices" of an acceptance relation or a measured set.

A measured set $(\gamma_i)_i$ is given by the recursive predicate $P(i,x,y)$ which is $\underline{true}$ if $\gamma_i(x)$ equals y and $\underline{false}$ otherwise. The index for a program computing P can be considered to be an index for the measured set.*)

The acceptance relation $A(i,x,z)$ as a three-valued function is in general not recursive. However, the relation is determined completely by the set of triples $<i,x,z>$ for which $A(i,x,z) = \underline{true}$. An index for an acceptance relation can be considered to be the index of a program computing the characteristic function of this set. If $(\alpha_i)_i$ is the measured set of the A-run-times, this is precisely an index for the recursive predicate $\alpha_i(x) \le z$.

The recursive isomorphism between measured sets and acceptance relations is in fact nothing but the "equivalence" between the predicates $\alpha_i(x) \le z$ and $\alpha_i(x) = z$.

In the sequel, let N be the set of indices of acceptance relations and let M be the set of indices of measured sets, both as defined below. We prove that $N \equiv M$ using the well-known MYHILL isomorphism principle (cf. 1.4.3). By this principle it is sufficient to show $N \le_1 M$ and $N \le_1 M$. In

*) This choice is consistent with our representation of measured sets in our algorithmic language in chapter 1-1.

fact it is even sufficient to prove that $N \leq_m M$ and $N \leq_m M$, since both N and M are index sets, defined in terms of the functions computed by these programs, and therefore we may apply the padding lemma to transform many-one reductions into one-one reductions (cf. chapter 1.5).

DEFINITION 3.1.22. An *index of an acceptance relation* j is an index of a program φ^3_j so that:

(a) $\varphi^3_j(i,x,z) = 0$ *or* $\varphi^3_j(i,x,z) = 1$.
(i.e. $\Lambda(\varphi^3_j)$ is a total characteristic function).

(b) if $z < z'$ then $\varphi^3_j(i,x,z) = 0$ *imp* $\varphi^3_j(i,x,z') = 0$.

By (b) we have also for $z < z'$ $\varphi^3_j(i,x,z') = 1$ *imp* $\varphi^3_j(i,x,z) = 1$.
The set of indices of acceptance relations is denoted by N.

DEFINITION 3.1.23. An *index of a measured set* j is an index of a program φ^3_j so that:

(a) $\varphi^3_j(i,x,z) = 0$ *or* $\varphi^3_j(i,x,z) = 1$.
(i.e. $\Lambda(\varphi^3_j)$ is total characteristic function)

(b) $\varphi^3_j(i,x,z) = 0$ *and* $\varphi^3_j(i,x,z') = 0$ *imp* $z = z'$.

The set of indices of measured sets is denoted by M.

PROPOSITION 3.1.24. $N \equiv M$.

PROOF. As explained above, it is sufficient to prove $N \leq_m M$ and $M \leq_m N$.
$M \leq_m N$: Define the transformation σ by:

```
$\varphi^3_{\sigma(j)}(i,x,z) \Leftarrow$ if $\exists y \leq z[\varphi^3_j(i,x,y) > 1]$ then $2$
                    else int $y = \mu v \leq z[\varphi^3_j(i,x,v) = 0]$;
                         if $y > z$ then $1$
                         else int $w = \mu v \leq z[v > y$ and $\varphi^3_j(i,x,v) = 0]$;
                              if $w > z$ then $0$ else $1$ fi
                         fi
                    fi
```

The first clause tests both whether $\varphi^3_j(i,x,y)$ is defined for $0 \leq y \leq z$ and whether no forbidden values occur in this interval; consequently $\varphi^3_{\sigma(j)}$ is a total characteristic function iff φ^3_j is one.

In the second clause it is computed whether $\varphi^3_j(i,x,y) = 0$ holds for none, one or more than one value $z \leq y$. Only if there exists precisely a single value $z \leq y$ such that $\varphi^3_j(i,x,y) = 0$ one has $\varphi^3_{\sigma(j)}(i,x,y) = 0$. From this, one concludes that $\sigma(j)$ is the index of an acceptance relation iff j is the index of a measured set. Hence $M \leq_m N$ by σ.

$N \leq_m M$: Define the transformation τ by:

```
$\varphi^3_{\tau(j)}(i,x,z) \Leftarrow$ if $\exists y \leq z[\varphi^3_j(i,x,y) > 1]$ then $2$
                     elif $\varphi^3_j(i,x,z) = 0$ and $(z = 0$ or $\varphi^3_j(i,x,z \dot{-} 1) = 1)$ then $0$
                     elif $\varphi^3_j(i,x,z) = 1$ and $z > 0$ and $\varphi^2_j(i,x,z \dot{-} 1) = 0$ then $0$
                     else $1$ fi
```

Again it is clear that $\varphi^3_{\tau(j)}$ is a total characteristic function iff φ^3_j is one. Moreover, a necessary and sufficient condition for $\varphi^3_{\tau(j)}(i,x,z)$ to be zero is $(z = 0$ *and* $\varphi^3_j(i,x,z) = 0)$ *or* $(z > 0$ *and* $\varphi^3_j(i,x,z) \neq \varphi^3_j(i,x,z \dot{-} 1))$.

If $\varphi^3_{\tau(j)}(i,x,z)$ has to be zero for at most one value of z (i and x being fixed), there should be at most a single change of value for $\varphi^3_j(i,x,z)$; moreover this should be a change from 1 to 0 since otherwise $\varphi_{\tau(j)}(i,x,0) = 0$.

From this one derives that $\tau(j)$ is the index of a measured set iff j is the index of an acceptance relation, proving $N \leq_m M$ by τ.

This completes the proof of 3.1.24. □

The conclusion of 3.1.24 could be that the concept of an acceptance relation can be eliminated from the theory. If we were to consider only strong classes, this is indeed the case; for strong classes *false* and *void* are identified, and the measured set of the A-run-times is sufficient to formulate everything we need. For weak classes, however, the three-valuedness of the acceptance relation becomes crucial, as this three-valuedness visualizes the difference between the finite-but-to-large run-time and the infinite run-time. This provides a motivation to preserve the concept of an acceptance relation.

Another motivation may arise by giving a third axiom as suggested in 3.1.3.

CHAPTER 3.2.

GAP AND OPERATOR GAP

3.2.1. INTRODUCTION

In this chapter we prove the generalizations of the gap theorem of A. BORODIN [Bo 72] and the operator-gap theorem of R.L. CONSTABLE [Co 72] for abstract resource-bound classes. These theorems can be formulated in the following way:

THEOREM 3.2.1. [Gap and operator gap]. Let A be an acceptance relation and let Γ be a total effective operator, satisfying $\Gamma(t) \geq t$. Let R be a total function in two variables satisfying $R(x,y) \geq y$. Then there exist arbitrarily large total functions t_1 and t_2 so that the following equalities hold:

$$F_S^A(t_1) = F_S^A(R\Box t_1); \quad F_W^A(t_1) = F_W^A(R\Box t_1)$$
$$F_S^A(t_2) = F_S^A(\Gamma(t_2)); \quad F_W^A(t_2) = F_W^A(\Gamma(t_2)).$$

The main subject in this chapter is the operator-gap theorem for weak classes. This theorem is proved by construction of a function t which satisfies the conditions formulated in the lemma below:

LEMMA 3.2.2. Let A be an acceptance relation, let Γ be a total effective operator with $\Gamma(t) \geq t$ and let f be a fixed total recursive function. Then there exists a total recursive function $t \geq f$ so that for each index j the following implication holds:

$$\overset{\infty}{\exists} x[A(j,x,t(x)) \neq \underline{\mathit{true}} \ \underline{\mathit{and}} \ A(j,x,\Gamma(t)(x)) = \underline{\mathit{true}}]$$

implies

$$\overset{\infty}{\exists} x[A(j,x,\Gamma(t)(x)) = \underline{\mathit{false}}].$$

This is a stronger condition than the condition enforced in the proof of the operator-gap theorem for complexity classes, where we have:

$$\overset{\infty}{\exists}x[A(j,x,t(x)) \neq \underline{true}]$$

implies

$$\overset{\infty}{\exists}x[A(j,x,\Gamma(t)(x)) \neq \underline{true}].$$

The function t which is constructed in the proof of lemma 3.3.2 yields an operator-gap for both strong and weak classes. By considering the composition-gap theorem to be a special case of the operator-gap theorem, we can derive the four possible gap theorems from the most difficult one, the proof of which is given in this chapter. However, the old proofs of the composition-gap theorem remain valid for weak classes.

First we prove that the gap theorems can be derived from lemma 3.2.2.

<u>PROOF</u> of 3.2.1 (a). [Operator-gap]. Let A be an acceptance relation, and let Γ be a total effective operator and let f be a total function. If $\Gamma(t) \geq t$ for each total recursive t then there exists a function $t \geq f$ so that both:

$$F_W^A(t) = F_W^A(\Gamma(t)) \quad \underline{and} \quad F_S^A(t) = F_S^A(\Gamma(t)).$$

By lemma 3.2.2 there exists a function $t \geq f$ so that for each index i the following implication holds:

$$\overset{\infty}{\exists}x[A(i,x,t(x)) \neq \underline{true} \;\; \underline{and} \;\; A(i,x,\Gamma(t)(x)) = \underline{true}]$$

implies

$$\overset{\infty}{\exists}x[A(i,x,\Gamma(t)(x)) = \underline{false}].$$

Now suppose that $i \in F_W^A(\Gamma(t))$ *and* $i \notin F_W^A(t)$, then we have by definition:

$$\overset{\infty}{\forall}x[A(i,x,\Gamma(t)(x)) \neq \underline{false}] \;\; \underline{and} \;\; \overset{\infty}{\exists}x[A(i,x,t(x)) = \underline{false}]$$

and consequently, since $A(i,x,t(x)) = \underline{false}$ $\underline{and}$ $A(i,x,\Gamma(t)(x)) = \underline{void}$ is impossible

$$\overset{\infty}{\exists}x[A(i,x,t(x)) = \underline{false} \;\; \underline{and} \;\; A(i,x,\Gamma(t)(x)) = \underline{true}].$$

By the choice of t we conclude

$$\overset{\infty}{\exists}x[A(i,x,\Gamma(t)(x)) = \underline{\mathit{false}}]$$

which is a contradiction. Therefore $F_W^A(\Gamma(t)) \subset F_W^A(t)$. The converse inclusion is trivial since $\Gamma(t) \geq t$.

For strong classes $F_S^A(t) \subset F_S^A(\Gamma(t))$ again is trivial. If $i \in F_S^A(\Gamma(t))$ and $i \notin F_S^A(t)$ then we conclude

$$\overset{\infty}{\exists}x[A(i,x,t(x)) \neq \underline{\mathit{true}}] \ \ \underline{\mathit{and}} \ \ \overset{\infty}{\forall}x[A(i,x,\Gamma(t)(x)) = \underline{\mathit{true}}]$$

hence we have again

$$\overset{\infty}{\exists}x[A(i,x,t(x)) \neq \underline{\mathit{true}} \ \ \underline{\mathit{and}} \ \ A(i,x,\Gamma(t)(x)) = \underline{\mathit{true}}]$$

which implies by the choice of t

$$\overset{\infty}{\exists}x[A(i,x,\Gamma(t)(x)) = \underline{\mathit{false}}].$$

This is a contradiction, therefore $F_S^A(\Gamma(t)) \subset F_S^A(t)$. This completes the proof. □

PROOF of 3.2.1 (b). [Composition-gap theorem]. Let A be an acceptance relation and let R be a total function satisfying $R(x,y) \geq y$. Then there exist arbitrarily large total functions t so that both $F_S^A(t) = F_S^A(R\square t)$ and $F_W^A(t) = F_W^A(R\square t)$.

As indicated below, the old proof for the composition-gap theorem yields a function t which satisfies the condition of lemma 3.2.2. We need no modifications since the composition-gap algorithm is based on extension by "local gap-sections of length one" which are automatically closed.

LEMMA 3.2.3. Let A be an acceptance relation and let R be a total recursive function with $R(x,y) \geq y$. Let f be a total function. Then there exists a total function $t \geq f$ so that for each index i the following assertion holds:

$$\forall x{>}i[A(i,x,R(x,t(x))) = \underline{\mathit{true}} \ \ \underline{\mathit{imp}} \ \ A(i,x,t(x)) = \underline{\mathit{true}}].$$

PROOF. First we remark that this assertion implies the condition in lemma 3.2.2. Suppose that

$$\overset{\infty}{\exists} x[A(i,x,R(x,t(x))) = \underline{true}\ \ \underline{and}\ \ A(i,x,t(x)) \neq \underline{true}]$$

then also

$$\overset{\infty}{\exists} x>i[A(i,x,R(x,t(x))) = \underline{true}\ \ \underline{and}\ \ A(i,x,t(x)) \neq \underline{true}]$$

which contradicts the above assertion. Hence in order to prove 3.2.1(b) it is sufficient to prove 3.2.3.

The function t is constructed the following way:

Define T by the recursive definition:

$$T(x,k) = \underline{if}\ k=0\ \underline{then}\ f(x)\ \underline{else}\ R(x,T(x,k-1))+1\ \underline{fi}.$$

Now T is a total function and for each k and x one has $T(x,k) < T(x,k+1)$. Next we define the function n by:

$$n = \lambda x[\mu k \leq x[\forall i<x[A(i,x,T(x,k)) = \underline{true}\ \ \underline{or}\ \ A(i,x,T(x,k+1)) \neq \underline{true}]]].$$

Thus n(x) is the lowest value of k so that none of the x run-times $\alpha_i(x)$ with $i < x$ is contained within the interval $(T(x,k),T(x,k+1)]$. Since these intervals are disjoint for different k there exists a number $n(x) \leq x$ with this property.

Now $t = T\square n$ is a total function with the property asked for by the lemma. □ □

Although our gap theorems are formulated in terms of classes of indices it is clear that they hold also for classes of programs or functions whenever *prog* and *fun* are defined for a given acceptance relation. Most generalizations thus generated are well-known; only the operator-gap theorem for honesty classes needs the full strength of the operator-gap theorem for weak classes.

COROLLARY 3.2.4. Let Γ be a total effective operator from R^2 to R^2 satisfying $\Gamma(T) \geq T$. Then there exist arbitrarily large total functions T so that $H_T = H_{\Gamma(T)}$.

{*Iam aliquantum spatii ex eo loco ubi pugnatum est aufugerat, cum respiciens videt magnis intervallis sequentes.*
Livius, ab Urbe Condita I.25.8}

3.2.2. THE OPERATOR-GAP ALGORITHM FOR STRONG CLASSES, AND ITS MODIFICATIONS

The proof of lemma 3.2.2 is based on an algorithm which is defined in §3.2.3 in an informal way. A program for this algorithm is given in the appendix. The algorithm is a modification of the algorithm constructed for the proof of the operator-gap theorem by P. YOUNG [Yo 73].

In this section a description is given of the algorithm originally given by P. YOUNG (formulated within the language of acceptance relations). Next it is explained why this algorithm itself does not yield the result for weak classes.

In the following Γ is a total effective operator which satisfies $\Gamma(t) \geq t$; f is a fixed total recursive function and A is a fixed acceptance relation.

The algorithm of P. YOUNG is a stagewise algorithm which computes at each *stage* x the values of t for all arguments z within a segment $[y_0+1,y_1]$. At the beginning of *stage* x the values of t are known on the initial segment $[0,y_0]$. The value of y_1 is also computed during *stage* x.

The computation during *stage* x can be described as follows:

(1) generate x+1 programs for functions t_j extending $t \mid [0,y_0]$ so that $t_{j+1} > \Gamma(t_j)$ on $[y_0+1,\infty)$.
(2) generate x+1 integers $z_j > y_0$ so that for each $v \in [y_0,z_{j+1}]$ the value of $\Gamma(t_j)(v)$ can be computed using only values of t_j on arguments in the segment $[0,z_j]$. (The z_j are computed downward, for $j = x,x-1,\ldots,0$.) Make sure that $z_j > z_{j+1}$. t_j is replaced by its restriction $t_j \mid [0,z_j]$.
(3) For each $i < x$, $0 \leq j \leq x$ we test whether there exists an argument $z \in [y_0+1,z_j]$ where $A(i,z,t_j(z)) \neq$ true. If such an argument z exists we say that i *violates the extension* t_j. No $i < x$ violates the non-existent extension t_{x+1}.

(4) If i violates the extension t_j and does not violate the extension t_{j+1} then we declare the gap-section $<t_j,t_{j+1}>$ *unsafe for i.*

Note that each index i has at most one gap-section which is declared unsafe for it. Since we have x indices and x+1 gap-sections we safely may execute (5).

(5) Select a gap-section $<t_j,t_{j+1}>$ which is not declared unsafe for any index. Extend t by t_j over the segment $[y_0+1,z_j]$; put $y_0 = z_j$, and proceed to the next stage.

It is not difficult to verify that the function t constructed in this way satisfies the condition:

$$\overset{\infty}{\exists} x[A(i,x,t(x)) \neq \underline{true}]$$

implies

$$\overset{\infty}{\exists} x[A(i,x,\Gamma(t)(x)) \neq \underline{true}].$$

Now the operator gap theorem for strong classes is a straightforward corollary.

To illustrate the above algorithm we represent in diagram 3.2.5 the behaviour of a single index i with respect to the local gap-sections $<t_i,t_{i+1}>$ created in (2). The vertical lines represent A-run-times of index i. If $\alpha_i(x)$ is finite this is indicated by a bounded vertical line terminating at but not including the point $<x,\alpha_i(x)>$. An unbounded line corresponds to an infinite value for $\alpha_i(x)$. Consequently at intersections of the graph of t_j with a bounded line one has $A(i,x,t_j(x)) = \underline{\mathit{false}}$, where for unbounded lines one has $A(i,x,t_j(x)) = \underline{\mathit{void}}$.

The local gap-section $<t_j,t_{j+1}>$ may be visualized as the area in between the curves t_j and t_{j+1}. Since $z_{j+1} < z$, it is an open local gap-section.

In the situation described by the diagram the gap-section $<t_{j+2},t_{j+3}>$ is unsafe for i since there is an A-run-time of i which violates t_{j+2} whereas i respects t_{j+3} on the interval $[y_0,z_{j+3}]$.

The concepts of a gap-section $<t_j,t_{j+1}>$ which is unsafe for an index i, upon which one algorithm is based, is defined in terms of strong violations; the gap-section $<t_j,t_{j+1}>$ is unsafe for i, provided i strongly violates t_j over the interval $[y_0+1,z_j]$ without strongly violating t_{j+1} over the interval $[y_0+1,z_{j+1}]$.

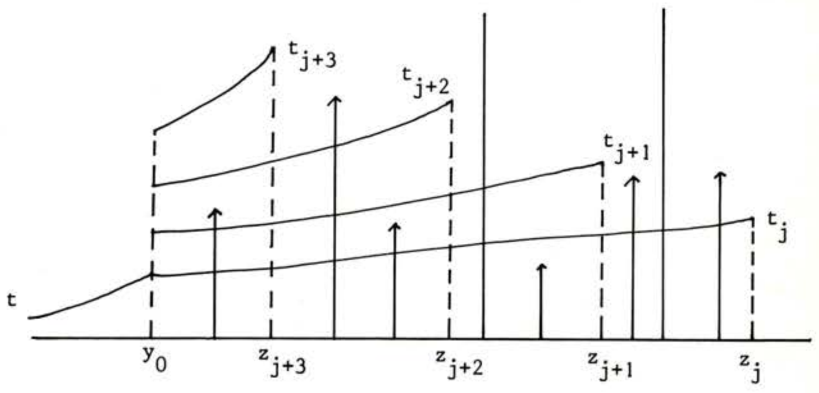


Diagram 3.2.5

To get an algorithm which works correctly for weak classes we must consider weak violations instead of strong ones. Consequently we will need a new unsafety concept. This new unsafety concept is defined as follows:

DEFINITION 3.2.6. The gap-section $<t_j,t_{j+1}>$ is called *weakly unsafe for i* provided i weakly violates t_j over the interval $[y_0+1,z_j]$ without weakly violating t_{j+1} over the interval $[y_0+1,z_{j+1}]$.

The reader may convince himself that, given this concept of "weakly unsafe", the earlier argumentations remain valid. In particular one can prove that each index has at most one gap-section which is weakly unsafe for it. Hence using the pigeon-hole principle as before one proves the existence of an extension t_j such that the gap-section $<t_j,t_{j+1}>$ is not weakly unsafe for any index $i < x$.

There is however a hidden snag in this argumentation. In the algorithm as given by P. YOUNG it is not possible to determine whether a gap-section is weakly unsafe for an index i or not. To understand this we should remember that the tests are executed along the graphs of the functions t_j. The tests moreover only yield answers of the type $A(i,x,t_j(x))$ = true or $A(i,x,t_j(x)) \neq$ true.

We consider the situation in more detail in diagram 3.2.7 below. The diagram represents all possible behaviours at an argument x of the run-time α_i with respect to the open local gap-section $<t_j,t_{j+1}>$. Situations $(x_1),\dots,(x_7)$ are discussed separately.

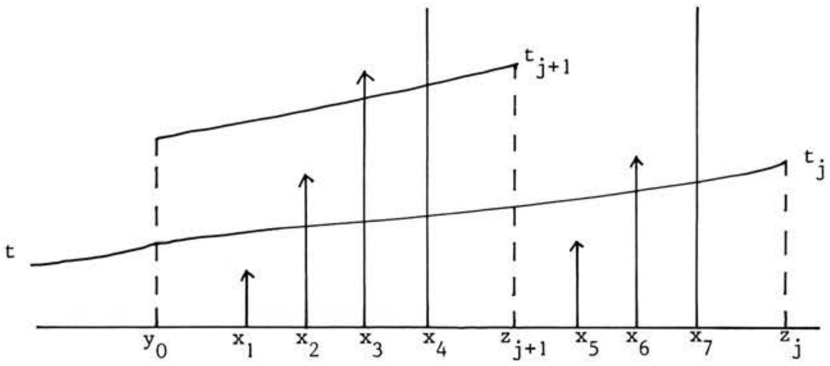


Diagram 3.2.7

($x=x_1$) $A(i,x,t_j(x)) = A(i,x,t_{j+1}(x) =$ *true*; no problem.

($x=x_2$) $A(i,x,t_j(x)) \neq$ *true*; $A(i,x,t_{j+1}(x)) =$ *true*; in this situation we positively know that i weakly violates t_j at x_2.

($x=x_3$) or ($x=x_4$) $A(i,x,t_j(x)) \neq$ *true*; $A(i,x,t_{j+1}(x)) \neq$ *true*. In this situation we only know that if i weakly violates t_j at x it also weakly violates t_{j+1} at x, but it is not possible, without execution of further tests to determine whether there are weak violations or not.

($x=x_5$) $A(i,x,t_j(x)) =$ *true*. Although the value of $t_{j+1}(x)$ is undefined there is no problem since whenever t_j is selected to extend t, we still have $\Gamma(t) \geq t$ and therefore $A(i,x,\Gamma(t)(x)) =$ *true* will be valid also.

($x=x_6$) or ($x=x_7$). $A(i,x,t_j(x)) \neq$ *true* and $t_{j+1}(x)$ is undefined. Again it is impossible to determine whether there is a weak violation or not.

Our conclusion is that the information which we have gathered is essentially incomplete. There may exist run-times $\alpha_i(x)$ which are larger than all values $t_j(x)$ for which $x \in \mathcal{D}t_j$. Let us call such a run-time an *unchecked large run-time*. The other run-times $\alpha_i(x)$ are called *checked run-times*.

The problem is that we cannot say whether or not an unchecked large run-time $\alpha_i(x)$ at x weakly violates the extensions t_j for which $x \in \mathcal{D}t_j$.

Now we return to the definition of "weakly unsafe". The gap-section $<t_j,t_{j+1}>$ is weakly unsafe for i provided there exists a weak violation by i against t_j and there exists no weak violation by i against t_{j+1}. Since it is not possible to determine whether or not weak violations are perpetrated by unchecked large run-times, we look for a more general concept of "unsafety", such that all weakly unsafe gap-sections are also unsafe in this more general sense.

DEFINITION 3.2.8. We say that the gap-section $<t_j, t_{j+1}>$ is *potentially weakly unsafe* for i if the following situation occurs;

(α) t_j is weakly violated by a checked run-time, or there exists an unchecked large run-time $\alpha_i(x)$ with $x \in \mathcal{D}t_j \setminus \mathcal{D}t_{j+1}$,

(β) t_{j+1} is not weakly violated by a checked run-time.

Note that it is decidable whether $<t_j, t_{j+1}>$ is potentially weakly unsafe for i.

LEMMA 3.2.9. If the gap-section $<t_j, t_{j+1}>$ is weakly unsafe for i then it is also potentially weakly unsafe.

PROOF. Clearly condition (β) holds since otherwise i weakly violates t_{j+1}. Let $\alpha_i(x)$ be a run-time weakly violating t_j. If $\alpha_i(x)$ is a checked run-time we must have a situation like at $x = x_2$ in diagram 3.2.9. If $\alpha_i(x)$ is unchecked then the situation is like at $x = x_3$ or at $x = x_6$. In the first case however, t_{j+1} is weakly violated and consequently $<t_j, t_{j+1}>$ is weakly safe for i, contradicting our assumption. Hence we have a situation like at $x = x_6$. Now the situations $x = x_2$ and $x = x_6$ both make for (α) to be satisfied. □

The solution therefore should be to replace the concept "weakly unsafe" by "potentially weakly unsafe". But this leads to new complications since it is possible that more than one gap-section is declared potentially weakly unsafe by a single index i. This situation is illustrated in diagram 3.2.10.

Diagram 3.2.10 shows that many gap-sections are spoiled by unchecked large run-times of α_i.

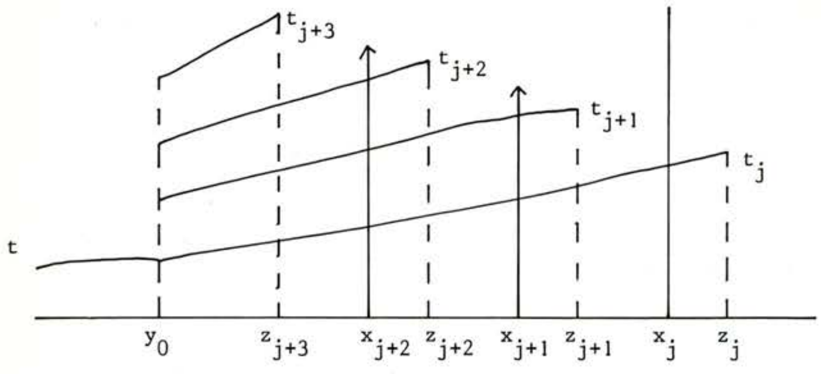


Diagram 3.2.10

From the definition of "potentially weakly unsafe" one reads that only the unchecked large run-times $\alpha_i(x)$ with $x \in \mathcal{D}t_j \setminus \mathcal{D}t_{j+1}$ are dangerous. These run-times could be eliminated if, by some trick, we would be able to extend t_{j+1} over $\mathcal{D}t_j$ in such a way that the relation "$t_{j+1} \geq \Gamma(t)$ if t is an extension of t_j" remains valid.

Suppose that such a trick is found. Then we can reformulate the definition of potentially weakly unsafe, which now becomes:

DEFINITION 3.2.11. The gap section $<t_j,t_{j+1}>$ is *potentially weakly unsafe'* for i if the following situation occurs:

(α') t_j is weakly violated by a checked run-time,
(β) t_{j+1} is not weakly violated by a checked run-time.

Note that in this formulation the unchecked large run-times no longer figure. Moreover, if $\mathcal{D}t_j = \mathcal{D}t_{j+1}$ there is no longer a difference between "weakly unsafe" and "potentially weakly unsafe". To see this assume that $\mathcal{D}t_j = \mathcal{D}t_{j+1}$. We know already that "weakly unsafe" implies "potentially weakly unsafe"; the latter is equivalent with "potentially weakly unsafe'" since $\mathcal{D}t_j = \mathcal{D}t_{j+1}$. Conversely from ($\alpha$') and ($\beta$) it is clear that $<t_j,t_{j+1}>$ is weakly unsafe for i.

This situation is illustrated in diagram 3.2.12.

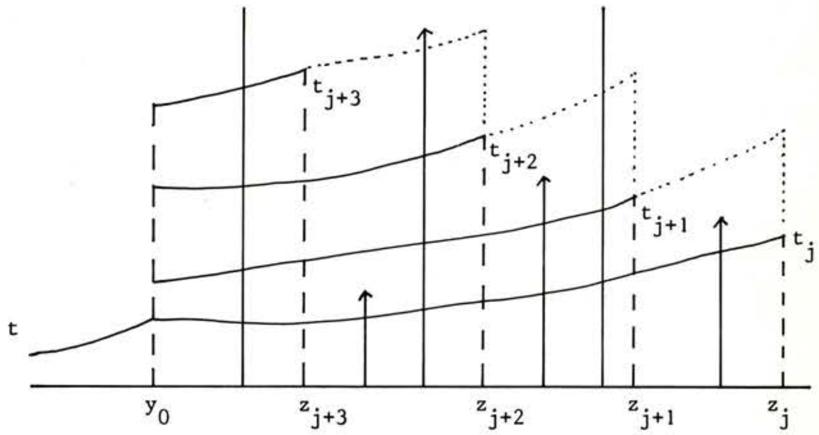


Diagram 3.2.12

In the situation represented in diagram 3.2.12 the gap-section $<t_{j+1},t_{j+2}>$ is weakly unsafe for i and the other two gap-sections are safe for i. Without use of the extension of the t_j all gap-sections are potentially weakly unsafe for i.

The trick used to extend the t_{j+1} over $\mathcal{D}t_j$ is a one-stage look-ahead in the algorithm. The reason that t_{j+1} is not defined over $\mathcal{D}t_j$ is that t_{j+1} must be an upperbound for $\Gamma(t)$ assuming that t is an extension of t_j. To compute $\Gamma(t)(x)$ for $x \in \mathcal{D}t_j$ we may need values $t(x)$ for $x \notin \mathcal{D}t_j$, which are not yet fixed.

However, by selecting t to be an extension of t_j we restrict the possible values of t on a larger domain, because t is going to be an extension of one of the x+2 extensions $u_{i,j}$ generated during *stage* x+1. Without loss of generality we may enforce that all extensions $u_{i,j}$ are defined over a domain sufficiently large such that $\Gamma(u_{i,j})(z)$ is defined for all $z \in \mathcal{D}t_j$.

Therefore, in order to generate an upperbound for $\Gamma(t)$ on $\mathcal{D}t_j$ based on the assumption that t is an extension of t_j, the x+2 next stage-extensions $u_{i,j}$ and the values of $\Gamma(u_{i,j})(z)$ for $z \in \mathcal{D}t_j$ only need to be computed one stage in advance. This way we compute closed local gap-sections.

This one-stage look-ahead is illustrated in diagram 3.2.13.

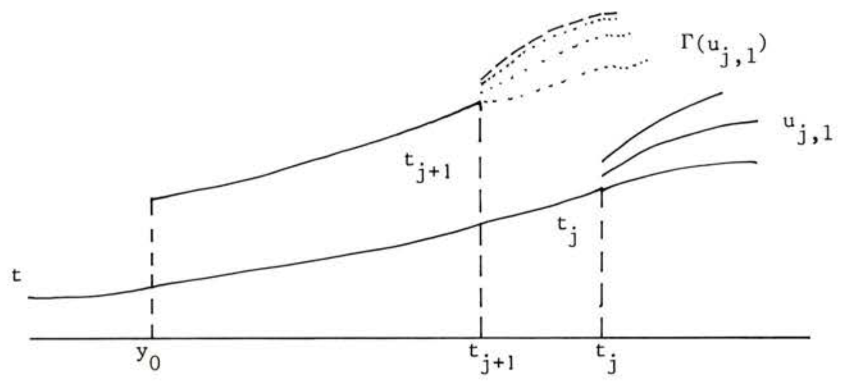


Diagram 3.2.13

The informal discussion above contains all essential ideas behind the operator gap algorithm for weak classes. The next section contains an informal description of a program executing the one-stage look-ahead. Next it will be that the function computed by this algorithm satisfies the conditions of lemma 3.2.2.

3.2.3. THE OPERATOR-GAP ALGORITHM FOR WEAK CLASSES

The algorithm described in this section computes the function t which is claimed by lemma 3.2.2. This lemma is repeated below:

LEMMA 3.2.2. Let A be an acceptance relation, let Γ be a total effective operator with $\Gamma(t) \geq t$ and let f be a total function. Then there exists a function $t \geq f$ such that for every index j the following holds:

$$\overset{\infty}{\exists} x[A(j,x,t(x)) \neq \underline{\mathit{true}} \ \ \underline{\mathit{and}} \ \ A(j,x,\Gamma(t)(x)) = \underline{\mathit{true}}]$$

implies

$$\overset{\infty}{\exists} x[A(j,x,\Gamma(t)(x)) = \underline{\mathit{false}}].$$

The algorithm is executed stage-wise. We describe *stage* k. Assume that at the beginning of *stage* k $t(x)$ is defined over the interval $[0,y_0]$;

furthermore there are k+1 programs of functions t_i, $i = 0,\dots,k$ which satisfy:

(1) $t_i(x) = t(x)$ for $x \le y_0$, $0 \le i \le k$.
(2) $t_{i+1}(x) \ge \Gamma(t_i)(x)$ for $x > y_0$, $0 \le i < k$.
(3) $t_i(x) \ge f(x)$ for all x, $0 \le i \le k$.
(4) $t_i(x)$ is non-decreasing in x (by $\Gamma(t) \ge t$ and by (2) $t_i(x)$ is non-decreasing in i as well).

Finally there exist k+1 pointers z_i $(0 \le i \le k)$ which satisfy:

(5) $z_k \ge y_0+1$; $z_i \ge z_{i+1}+1$ for $0 \le i < k$.
(6) The support of $\Gamma(t_k)$ on $[0,y_0+1]$ is contained in $[0,z_k]$.
(7) The support of $\Gamma(t_i)$ on $[0,z_{i+1}]$ is contained in $[0,z_i]$ for $0 \le i < k$.

Remark that the above conditions are satisfied after execution of (1) and (2) in the algorithm of P. YOUNG which is described in the preceding section.

The computations of *stage* k can be described as follows:

(1) For $0 \le j \le k$ we construct k+2 programs for the next stage-extensions $u_{j,l}$ which are defined by:

$$u_{j,0} = \lambda x[\underline{if}\ x \le z_j\ \underline{then}\ t_j(x)\ \underline{else}\ \underline{max}(f(x),u_{j,0}(x-1))\ \underline{fi}],$$

$$u_{j,l} = \lambda x[\underline{if}\ x \le z_j\ \underline{then}\ t_j(x)\ \underline{else}\ \underline{max}(f(x),u_{j,l}(x-1),u_{j,l-1}(x),\Gamma(u_{j,l-1})(x))\ \underline{fi}]$$

for $0 < l < k+1$.

(2) For $0 \le j \le k$, $k+1 \ge l \ge 0$ we construct a pointer $v_{j,l}$ which satisfies:

the support of $\Gamma(u_{j,k+1})$ on $[0,z_j+1]$ is contained within $[0,v_{j,k+1}]$ for $0 \le j \le k$,

the support of $\Gamma(u_{j,l})$ on $[0,v_{j,l+1}]$ is contained within $[0,v_{j,l}]$ for $0 \le j,l \le k$, moreover $v_{j,l} > v_{j,l+1}$.

¢ the pointers $v_{j,l}$ are computed downwards for $l = k+1,k,\dots,0$ ¢
¢ these computations correspond to (1) and (2) in the algorithm of P. YOUNG ¢

(3) For $0 \le j \le k$ we compute a function segment g_j on the interval $[y_0+1,z_j]$ which is defined by:

```
$g_j = \lambda z[\underline{if}\ z \notin [y_0+1, z_j]\ \underline{then}\ \underline{loop}$
        $\underline{elif}\ z \le z_{j+1}\ \underline{then}\ t_{j+1}(z)$
        $\underline{else}\ max\{\Gamma(u_{j,l})(z) \mid l \le k+1\}\ \underline{fi}]$
```

¢ By (2) the support of $\Gamma(u_{j,1})$ on $[y_0+1,z_j]$ is contained within $[0,v_{j,1}]$; furthermore by assumption $\Gamma(t_j)(z) = \Gamma(u_{j,1})(z) \le t_{j+1}(z)$ for $z \le z_{j+1}$. ¢

(4) For $0 \le j \le k$ and $0 \le i \le k-1$ we check whether there exists a $z \in [y_0+1,z_j]$ with $A(i,z,t_j(z)) \ne \underline{true}\ \underline{and}\ A(i,z,g_j(z)) = \underline{true}$. If such a z exists we say that *i enters the j-th local gap-section.*

(5) For $i = 0,\dots,k-1$ select the largest j such that the index i enters the j-th local gap-section. If such a j exists (say j_i) then we declare the j_i-th local gap-section *weakly unsafe for i.*

(6) Select a (the lowest) j such that the j-th local gap-section is not declared unsafe for any index i, with $0 \le i \le k-1$. Denote this j by j_0.
¢ Such an index j exists by the pigeon-hole principle. ¢

(7) Extend t by t_{j_0} over $[y_0+1,z_{j_0}]$; put $y_0 := z_{j_0}$, for $0 \le j \le k+1$, put $t_j := u_{j_0,j}$, $z_j := v_{j_0,j}$.

(8) Proceed to *stage* k+1.
¢ Remark that by execution of (7) the assumptions (1),...,(7) become correct before entering the next stage. ¢

To initialize the program one executes onece (1) and (2) using k=0 and a function t which is defined over the empty segment [0,-1]. This yields two programs t_0 and $t_1 \ge f$ and two pointers z_0 and z_1. We put $y_0 = -1$. Now the assumptions (1),...,(7) as assumed to hold before entering *stage* 1 are correct.

Next we start *stage* 1.

We should emphasize that the algorithm manipulates on programs for the functions t_i and that during elaboration of the algorithm these programs are executed at several arguments. The programming environment in which the above algorithm can be formally described must include therefore facilities which permit to create procedures at run-time.

To prove lemma 3.2.2 we present an informal and intuitive correctness proof for the above algorithm.

The reader should convince himself that each stage in the algorithm terminates. One uses the fact that for each total effective operator Γ and each total function t and each argument x, an integer y can be found effectively so that the support of $\Gamma(t)(x)$ is contained within the interval $[0,y]$ (although the y found this way is in general not minimal, cf. §1.2.4).

It is clear from the description that after a finite number of stages t is defined over a finite segment; moreover $t(x)$ is defined before or at *stage* x+1 since each stage properly extends the defined part of t.

Now let i be an index with the property:

$$\overset{\infty}{\exists}x[A(i,x,t(x)) \neq \underline{\mathit{true}} \ \ \underline{\mathit{and}} \ \ A(i,x,\Gamma(t)(x)) = \underline{\mathit{true}}],$$

we prove that this implies:

$$\overset{\infty}{\exists}x[A(i,x,\Gamma(t)(x)) = \underline{\mathit{false}}].$$

First we discard those arguments and values of t which are defined before or at *stage* i.

Let x be an argument so that $t(x)$ is defined at *stage* k with $k > i$ and so that

$$A(i,x,t(x)) \neq \underline{\mathit{true}} \ \ \underline{\mathit{and}} \ \ A(i,x,\Gamma(t)(x)) = \underline{\mathit{true}}.$$

By abuse of notation we give y_0, z_j, t_j, $u_{j,l}$, $v_{j,l}$, g_j and j_0 the meaning they have during execution of *stage* k (6).
Hence $t \mid [0,z_{j_0}] = t_{j_0} \mid [0,z_{j_0}]$ for a j_0 so that $x \in [y_0+1,z_{j_0}]$ and so that the j_0-th gap-section is not declared weakly unsafe for any index i' with $0 \leq i' \leq k-1$; in particular the j_0-th gap-section is not declared weakly unsafe for i.

Now by assumption i enters the j_0-th gap-section, for let t be extended during *stage* k+1 by the next stage-extension u_{j_0,l_0}. Then the support of $\Gamma(t)(x) = \Gamma(u_{j_0,l_0})(x)$ is contained within $[0,v_{j_0,l_0}]$ and by definition of g_{j_0} we have

$$g_{j_0}(x) \geq \Gamma(u_{j_0,l_0})(x) = \Gamma(t)(x).$$

Therefore $A(i,x,t_{j_0}(x)) \neq \underline{\mathit{true}}\ \underline{\mathit{and}}\ A(i,x,g_{j_0}(x)) = \underline{\mathit{true}}$. Since the j_0-th gap-section is not declared weakly unsafe for i we conclude that i enters also a j-th gap-section with $j > j_0$. This implies that for some

$z \in [y_0+1, z_j]$ we have

$$A(i,z,t_j(z)) \neq \underline{\mathit{true}}\ \underline{\mathit{and}}\ A(i,z,g_j(z)) = \underline{\mathit{true}}$$

hence $A(i,z,t_j(z)) = \underline{\mathit{false}}$.

Since $j > j_0$ one has $t_j(z) \geq t_{j_0+1}(z) \geq \Gamma(t_{j_0})(z) = \Gamma(t)(z)$ and therefor $A(i,z,t_j(z)) = A(i,z,\Gamma(t)(z)) = \underline{\mathit{false}}$.

Our conclusion is that for each *stage* k where values t(x) are defined so that $A(i,x,t(x)) = \underline{\mathit{false}}$ and $A(i,x,\Gamma(t)(x)) = \underline{\mathit{true}}$ at least one new value t(z) is defined with $A(i,z,\Gamma(t)(z)) = \underline{\mathit{false}}$.

Using the fact that only finitely many values of t are defined during a single stage we conclude that

$$\overset{\infty}{\exists} x[A(i,x,\Gamma(t)(x)) = \underline{\mathit{false}}]. \quad \square$$

CHAPTER 3.3

THE UNION THEOREM

3.3.0. INTRODUCTION

This chapter contains a discussion on the union theorem of E.M. McCREIGHT and A. MEYER [MCM 69]. This theorem states that the union of a sequence of complexity classes named by a non-decreasing sequence of total names is again a complexity class. This complexity class is named by a total function, which is computable from programs for the names in the sequence. The theorem is a typical example of a theorem which can be translated straightforward to yield a union theorem for strong abstract resource-bound classes.

There are however further generalizations, which state that the theorem holds also for weak classes and partial names. The condition of monotonicity of the names includes in this case also the condition that the sequence of domains of the names is a non-increasing sequence of sets.

In this chapter we concentrate on the most complicated case (weak classes with partial names). The other cases are left to the reader for verification.

In 3.4.3 and 3.4.5 we will meet another type of generalized union theorems where the condition of monotonicity of names is replaced by the condition that the classes of indices themselves form an increasing sequence.

An application of the union theorem is the translation of complexity classes modulo sets of exceptional points into complexity classes with partial names.

In section 1 we give the proof of the classical union theorem; next in section 2 the modifications needed for the generalization are explained. In section 3 we present the design of our union algorithm, followed by a correctness proof in section 4. (A formal program can be found in the appendix.) Section 5 contains applications of the union theorem.

REMARK. The terminology "j respects t" which is used frequently in this chapter is equivalent to $j \in F^A_S(t)$ $(j \in F^A_W(t))$ depending on the type of restriction considered.

3.3.1. THE CLASSICAL UNION THEOREM

In the context of strong abstract resource-bound classes the formulation of the union theorem reads:

THEOREM 3.3.1. [Union]. Let $(t_i)_i$ be a sequence of total functions so that $\forall i \forall x[t_i(x) \le t_{i+1}(x)]$. Then there exists a total function t_{inf} so that:

$$\bigcup_i F_S^A(t_i) = F_S^A(t_{inf}).$$

PROOF. The function t_{inf} is computed by a stagewise algorithm. We present the description of *stage* x below.

```
stage x: (1) guess[x] := x;
         (2) for k ≤ x do compute (t_k(x)) od;
         (3) val := t_x(x);
         (4) for k ≤ x do
                 if not (A(k,x,t_guess[k](x)) = true) then
                    val := min(val,t_guess[k](x));
                    guess[k] := x+1
                 fi od;
         (5) t_inf(x) := val;
         (6) goto stage x+1
```

The algorithm is started at *stage* 0.

This algorithm is called the *standard algorithm* during the next sections of this chapter.

It is clear from the totality of the t_i that $t_{inf}(x)$ is computed at the end of *stage* x. Hence t_{inf} is total.

To prove the equality $\bigcup_i F_S^A(t_i) = F_S^A(t_{inf})$ we look for a fixed j at the two possible behaviours of the value of guess[j].

Case 1) guess[j] is unstable (i.e. guess[j] grows unboundedly):

This means that the run-time $\alpha_j(x)$ is larger than $t_i(x)$ for arbitrarily large x and i; hence $j \notin \bigcup_i F(t_i)$. Furthermore for each *stage* x where guess[j] is redefined a violation by j at x against t_{inf} is created. Hence $j \notin F(t_{inf})$.

Case 2) guess[j] stabilizes at i.

This implies that j does not violate t_i at arguments x with x larger than the stagenumber of the stage during which guess[j] := i is executed. Hence $j \in F(t_i) \subset \bigcup_i F(t_i)$.

For each x the value of $t_{inf}(x)$ is defined during *stage* x; moreover, this value euqals $t_k(x)$ for some $k \leq x$. However, if $k \leq x$ then a value guess[j], which equalled k at the beginning of *stage* x, equals x+1 at the end of *stage* x. Hence a "low" value of $t_{inf}(x)$ implies the disappearance of a low guessvalue. At each moment during the computation only a finite number of guess-values less then l are alive in the algorithm; moreover after termination of *stage* l this number only decreases.

One concludes from this that $t_{inf}(x) \geq t_i(x)$ for almost all x. Since $\alpha_j(x) \leq t_i(x)$ almost everywhere we derive the relation

$$\overset{\infty}{\forall} x[\alpha_j(x) \leq t_{inf}(x)]$$

which implies $j \in F(t_{inf})$.

In both case we have found

$$j \in \bigcup_i F^A_S(t_i) \ \underline{\textit{iff}}\ j \in F^A_S(t_{inf}).$$

This equivalence proves the theorem. □

Readers familiar with the proof of the naming theorem should recognize the similarity between this proof and the above argument. In fact, the above reasoning is a simplification of the original proof of the union theorem, which was more related to the naming theorem. We return to this subject in 3.4.3.

The arguments in the foregoing proof represent the essential ideas of the generalization. The value guess[j] specifies simultaneously the bound index which index j is supposed to respect, and a priority number. The value of $t_{inf}(x)$ could be defined as the value of $t_k(x)$ where k is "the lowest bound-index violated against at x". In future this k shall be called the *bound-index used at x*.

The correctness of the program (or some generalization of it) can be expressed by means of the following four correctness claims:

Claim 1: If guess[j] is unstable, then j violates t_{inf} infinitely many times.

Claim 2: If j violates t_k infinitely often, then guess[j] does not stabilize at a value $\leq k$.

Claim 3: If j respects a bound t_k then guess[j] stabilizes.

Claim 4: If guess[j] stabilizes then j respects t_{inf}.

We formulate two more assertions, which seem rather complicated for the present case, but which are designed for the generalizations:

Assertion 1. Let $t_{\text{inf}}(x)$ be defined (at *stage* z) using bound-index k. Let j be an index and let i be the value of guess[j] at the moment $t_{\text{inf}}(x)$ is defined. Then we have:

$$i > k \ \underline{\textit{or}}\ \ A(j,x,t_i(x)) = A(j,x,t_{\text{inf}}(x)).$$

For the standard algorithm Assertion 1 is valid. We know that $z = x$. Assume that $i \leq k$. Clearly it is impossible that guess[j] has been redefined during *stage* x, for in this case $i > x \geq k$. Consequently we must have $A(j,x,t_i(x)) = \underline{\textit{true}}$, but now $i \leq k$ implies $A(j,x,t_k(x)) = A(j,x,t_{\text{inf}}(x)) = \underline{\textit{true}}$, which proves the assertion.

Assertion 1 becomes interesting when we treat the case of weak restriction.

Assertion 2. For each n the bound-index used at x, denoted by k(x), satisfies

$$\overset{\infty}{\forall} x[k(x) > n].$$

By this assertion (which was used explicitly in the above proof) $t_{\text{inf}}(x) \geq t_n(x)$ almost everywhere. The correctness of the assertion is based on an exhaustion of low guess-values, which are mortal when they are used as bound-index at x.

{*Dieses war der erste Streich*
Doch der Zweite folgt sogleich.
Max und Moritz Wilhelm Busch}

3.3.2. MODIFICATIONS TO THE UNION ALGORITHM

The modifications needed to construct a union algorithm which applies to weak classes with partial names, are of different types. To tackle the partiality of the names, we replace the computation of the bounds t_i by an enumeration of their graphs. To deal with the weak restriction we replace the one-phased test on strong violations by a two-phased test on weak violations. Finally there are modifications to the synchronization of the algorithm.

These types are discussed separately; application of a well-chosen selection yields the union algorithms for the two generalizations which are not treated explicitly in this treatise.

MODIFICATIONS TO DEAL WITH PARTIAL BOUNDS

The idea is simple. Instead of computing the values of $t_i(x)$ which are needed at a certain moment, a dovetailed computation is used to enumerate all values of $t_i(x)$ $(i,x \in \mathbb{N})$ simultaneously.

In the standard algorithm the tests on violations at x are executed when the values of $t_i(x)$ are known for $i = 0,1,\ldots,x$. The example below shows however that these tests should not be delayed upto this moment in our new algorithm.

<u>EXAMPLE 3.3.2.</u> [Do not wait until $t_0(x),\ldots,t_x(x)$ are known]. Let

$$t_i = \lambda x[\underline{if}\ x \le i\ \underline{then}\ \underline{loop}\ \underline{else}\ 0\ \underline{fi}]$$

For each i we have $F(t_i) = F(\text{zero})$. However for each x $t_x(x)$ is undefined; if we have to wait until all values $t_i(x)$ with $i \le x$ are computed, we have to wait forever. Hence an algorithm, based on this principle computes the empty function ε. However

$$\bigcup_i F(t_i) = F(\text{zero}) \ne F(\varepsilon).$$

Our policy is therefore to use each value $t_i(x)$ which is enumerated at once for the testing of the indices j with guess[j] = i. It is however possible that other indices j' with guess[j'] = i' < i are not yet tested at x (as $t_{i'}(x)$ was not yet enumerated).

In general these indices j' might force the value of $t_{inf}(x)$ to become smaller. In this respect the value of guess[j] acts as a priority number. The indices with lower guess-value should be tested first.

At this point the condition of non-increasing domains can be applied. If $t_k(x)$ converges, then $t_i(x)$ converges also for $i < k$. Therefore we can compute all $t_i(x)$ for $i < k$ if we know that $t_k(x)$ has been enumerated. This allows us to "fill-in" the "holes" in the table of known values of $t_i(x)$.

Alternative solutions to this problem are:

(i) Replace the programs computing $t_i(x)$ by a single program $T(i,x)$ which has a run-time monotonically increasing in i. (This is possible by a variant of the monotonicity lemma 1.5.5.)

(ii) Use the table of known values only as far as it is complete.

Each of these modifications allows us to generate the values of $t_i(x)$ in an usable order.

In the standard algorithm the value of $t_x(x)$ is used also as an escape definition if the situation arises that none of the indices introduced so far violates at x. This escape definition is used to enforce totality of t_{inf}. If partial bounds are used, these escape definitions are not needed. It is however possible to apply the definition $t_{inf}(x) = t_x(x)$ whenever $t_x(x)$ is enumerated and execution of all tests induced by this appearance does not yield a value for $t_{inf}(x)$. The proof, however, becomes more complex.

A complication which is unconceivable in the standard algorithm is the fact that tests at x are performed at widely different stages in the computation. This opens the possibility that a single index j is tested several times at the same argument x. This situation may arise in the following way:

Index j has guess[j] = i at a certain stage.
$t_i(x)$ is enumerated.
A test shows that j violates t_i at x.
guess[j] is redefined (guess[j] := i' where i' is chosen large enough that
    no value of $t_{i'}(x)$ has yet been enumerated).
At the time $t_{i'}(x)$ is enumerated we are invited to perform another test on
    j at x.

This multiple testing at a single argument could easely lead to infinite testing at a single argument. Then guess[j] could become unstable because of a violation of j at a single argument x against all bounds t_i, and this way Claim 3 collapses. It is therefore clear that infinite testing must be prevented. Two solutions which can be given are:

(1) Each test on j at x is registered to prevent multiple testing.

(2) No values $t_i(x)$ with $i > x$ are enumerated. This does not necessarily prevent multiple testing (depending on the way guess[j] is redefined) but prevents infinite testing.

In our program in 3.3.3 we use the second technique as this is easier for implementation. We need no extra data structure to store all tests which are executed. As we will see our method of redefining guess[j] prevents in fact multiple testing as well.

MODIFICATIONS NEEDED TO DEAL WITH WEAK RESTRICTION

From the definition of weak restriction one can derive that weak violations by an index j against a bound t_i cannot be found by a single test; they have to be enumerated.

Given a value $t_i(x)$ we can test whether $A(j,x,t_i(x)) =$ *true* or not. If this is not the case we have to search for a value $z > t_i(x)$ with $A(j,x,z) =$ *true* to discriminate between $A(j,x,t_i(x)) =$ *void* and $A(j,x,t_i(x)) =$ *false*. In our program such an index j is called *suspect at x* and a so-called *suspect report* (SR) is placed on the *suspect list* (SL). The suspect report contains the following information:

(1) the suspect index	j
(2) the argument	x
(3) the bound-index	i
(4) the value	$t_i(x)$

It is called a *suspect report on j at x against i with value $t_i(x)$.*

In our algorithm i always equals the value of guess[j] at the time the SR is created.

We will consider at returning occasions in the course of the computation all items on the suspect list, to see whether some of them represent weak violations. To do so we use an ever increasing test value z to test $A(j,x,z) =$ *true*, whenever an SR on j at x is present. If for a certain SR this test yields a positive answer, a weak violation is detected. The suspect report $<j,x,i,t_i(x)>$ is transformed into a *violation report* (VR) having the same format and values. Violations reports are placed in a *violation queue* (VQ) to be used in the computation of $t_{inf}(x)$.

In order for Claim 2 to be valid, it is necessary that guess[j] is redefined if a violation by j against some bound index $i \geq$ guess[j] is discovered (only a finite number of violations may be overlooked). At the same time it is not permitted to redefine guess[j] before it is certain that a corresponding violation of j against t_{inf} is (or has been) created, since otherwise Claim 1 becomes invalid.

This problem is solved by giving a correct method to select the bound-index used at x. In the standard algorithm this bound index k equals "the lowest bound-index violated against at x". If we should be able to use a similar definition of k as function of x in our modified algorithm, i.e. something like "the lowest bound-index weakly violated against at x", our problem would have been solved. However, the above description represents a non-computable function.

In the situation where $t_{i'}$ has been violated against at x there may exist suspect reports at x on the suspect list against bounds t_i with $i < i'$, which might be (or might not be) transformed into violation reports at much later stages, and this question is undecidable.

The correctness assertion Assertion 1 formulated in the preceding section now becomes relevant. Let j be an index having an SR $<j,x,i,t_i(x)>$ on the suspect list. If, at the time where it is discovered that j violates t_i at x, the value of $t_{inf}(x)$ is not yet defined then we still have the freedom to select the bound-index k used at x in such a way that $k \leq i$. It is not a good policy to postpone the definition of the value $t_{inf}(x)$, until all SR's at x against bound indices $i' \leq i$ have become VR's, for it is improbable that all of them will do so.

Consequently it is thinkable that the violation by j at x against t_i is discovered at a time where $t_{inf}(x)$ already is defined, using a bound-index $k > i$.

Now assume that the bound-index k used at x was selected in such a way that Assertion 1 holds. If guess j was redefined during the period starting at the time where the SR $<j,x,i,t_i(x)>$ was created and the current time, we may assume that at the occasion of this redefinition a violation by j against t_{inf} was created (or was recognized to have been created at a still earlier time). Consequently we are free to consider, in this case the new detected violation as having been punished "in advance". (In our final algorithm, a "clean-up" of the suspect list at the occasion of the redefinition of guess[j], makes this reasoning unnecessary.)

However, if guess j was not redefined we must redefine it at the current stage in the algorithm. At the same time we no longer have the freedom to create a violation by j against t_{inf} at x since the bound-index k used at x has already been selected, and this may have resulted into the problem that k > i. But in this situation we may apply Assertion 1, which yields the relation

$$A(j,x,t_i(x)) = A(j,x,t_k(x)).$$

This equality shows that is is irrelevant, for the case that j actually violates t_i at x, whether this violation is "punished" by setting $t_{inf}(x) = t_i(x)$ or $t_{inf}(x) = t_k(x)$.

It is therefore sufficient to select the bound-index used at x in such a way that Assertion 1 holds. This selection proceeds as follows:

The first time an SR at x is transformed into a VR (by finding an SR $<j,x,i,t_i(x)>$ and a testvalue z with $A(j,x,z) =$ *true*) this testvalue z is used to test all SRs at x on the suspect list. These tests lead to a number of violation reports. The bound-index used at x is the lowest bound-index violated against at this first occasion.

The above method is consistent with Assertion 1. Let $<j',x,k,t_k(x)>$ be the VR used to define $t_{inf}(x)$. If an index j has guess[j] $\geq$ k then there is no problem. If guess[j] = i < k and if there is an SR $<j,x,i,t_i(x)>$ on the suspect list, then we know that for the test value z $A(j,x,z) \neq$ *true*. However, we know also that for the index j' $A(j',x,t_k(x)) \neq$ *true* whereas $A(j',x,z) =$ *true*. Hence we have the inequalities: $z \geq t_k(x) \geq t_i(x)$ which implies

$$A(j,x,t_i(x)) = A(j,x,t_k(x)) = A(j,x,z) \neq \textit{true}.$$

If i < k and if there is no SR on j at x on the suspect list then we use the fact that j was tested at x against t_i and was accepted. (This fact does not yet result from the design explained so far; we will have to enforce it.) In this case we have therefore $A(j,x,t_i(x)) =$ *true* $= A(j,x,t_k(x))$.

MODIFICATIONS TO THE SYNCHRONIZATION

In the preceding argumentation some other correctness conditions are hinted at, which still have to be enforced. These conditions are related to the global synchronization of the algorithm. In the standard algorithm this synchronization is performed by the stagenumber x. This stagenumber represents concurrently:

(a) the maximal argument x for which the bounds $t_i(x)$ are computed
(b) the maximal bound-index i for which the $t_i(x)$ are computed
(c) the largest index j introduced in the computation
(d) the largest guess-value created by introduction at or redefinition before <u>stage</u> x; (at redefinition during <u>stage</u> x a guess-value is set equal to x+1 in order to make it larger than all guess-values existing before).

This accumulation of functions seems irrelevant. For reasons of modularity we introduce four different indicators for these maximal values named *maxarg*, *maxbnd*, *maxind* and *maxguess*.

In our modified union algorithm two more functions of the stagenumber will be visible; these are represented by the variables *maxcomp* and *maxtest*. The variable *maxcomp* represents the number of steps used in the enumeration of the $t_i(x)$ and *maxtest* is the ever increasing testvalue used to detect weak violations.

To enforce that no index j misses a test against a guessed bound t_i when $t_i(x)$ is already known before guess[j] gets the value i, it is sufficient to keep *maxguess* $\geq$ *maxbnd*.

The indicators *maxarg*, *maxbnd* and *maxind* control the values $t_i(x)$ which are enumerated and which tests are executed.

It is not possible to enforce in this way that all SRs on j presents on the suspect list are SRs against the bound t_i where i equals the current value of guess[j]. To enforce this relation one should remove from the suspect list all SRs on j whenever one of these SRs is transformed in a VR. This modification smoothes the argumentation, but is is not necessary in order to prove the correctness of the algorithm.

Using the strategy that no value of $t_i(x)$ is enumerated for i > x the phenomenon of multiple testing of an index j at a single argument x is prevented by keeping *maxguess* > *maxarg*. If guess[j] is then redefined, its value is larger than all arguments x for which values of $t_i(x)$ were enumerated, and therefore also larger than all arguments at which j was tested before. These arguments x are never used again to test j since the corresponding values of $t_{\text{guess}[j]}(x)$ are never enumerated.

To be able to use the value $t_x(x)$ as an escape definition for $t_{\inf}(x)$ one should keep *maxtest* ≥ the largest value of $t_i(x)$ enumerated so far; this is needed to keep <u>Assertion 1</u> correct.

3.3.3. A STAGEWISE UNION ALGORITHM FOR WEAK CLASSES WITH PARTIAL NAMES

In this section a stagewise algorithm is presented which computes $t_{\inf}$ in this case of weak classes with partial names. The description is an informal one; it is the skeleton of the program which can be found in the appendix. For example all declarations are deleted.

It is assumed that the following options are active:

(a) no escape definitions are used.
(b) no $t_i(x)$ with $i > x$ is enumerated.
(c) if a violation by j is detected all suspect reports on j are deleted.
(d) multiple testing is prevented by having *maxguess* = *maxarg*.

These options are not essential, but they simplify the argumentation in the next section.

The sequence of names t_i is computed by the single program $\varphi^2_t(i,x)$ with run-time $\Phi^2_t(i,x)$; $\varphi^2_t(i,x) = t_i(x)$.

The guess-values are stored in the infinite array *) guess. We need also an infinite array to keep track on the values $t_i(x)$ which already have been enumerated. This array is called *tcomp*. The element tcomp[x] is the first bound-index i such that $t_i(x)$ is still unknown.

The values of $t_{\inf}$ are stored in the infinite array tinf. The operator <u>*undefined*</u> tests whether a certain variable has been given a value by the program or not.

The program uses the ALGOL 68 semantics of a loop: to execute a loop the values of the bounds on the controlled variable are elaborated and copied and afterwards the loop is executed, using these static values.

*) In the formal representations these infinite arrays are represented by flexible arrays. Insertion and inspection of values is performed by the procedures described in section 1.1.5.

Therefore a loop terminates if the bounds determine a finite domain, and if for each value in this domain the embraced clause terminates.

Suspect reports and violation reports are structured values of the mode *struct(int ind,arg,bnd,val)*.

Each stage consists of 6 sections, called the blocks of the algorithm. *stage* x is described below.

```
¢ 0 ¢ synchronization: maxcomp := maxtest := maxguess := maxarg :=
                           maxind := maxbnd := x;

¢ 1 ¢ introduction: guess[maxind] := maxguess; tcomp[maxarg]:= 0;

¢ 2 ¢ enumeration and test on suspectness:

    for z to maxarg do
    begin newcomp := tcomp[z]-1 ¢ largest known i for which $t_i(x)$ is known*) ¢;
       for i from tcomp[z] to min(z,maxbnd) do ¢ min(z,maxbnd) = z ¢
          if $\Phi_t^2(i,z) \le$ maxcomp then newcomp := i fi od;
       ¢ which new values come out at this stage? ¢

       for i from tcomp[z] to newcomp do
       begin val := $\varphi_t^2(i,z)$ ¢ compute value ¢;
          for j to maxind do
             if guess[j] = i and not (A(j,z,val) = true)
             then attach (suspect list, (j,z,i,val)) fi od
          ¢ and use this value to test all indices introduced so far ¢
       end od;

       tcomp[z] := newcomp+1
       ¢ these values are enumerated and used for tests ¢
    end ¢ 2 ¢ od;

¢ 3 ¢ working through the suspect list:

    for item over suspect list do
       if A(ind of item, arg of item, maxtest) = true
       then attach (violation queue, item) fi ¢ violation detected ¢ od;
```

*) Since the dyadic operator " - " is not implemented this trick to detect whether new values are enumerated is illegal in our formalized algorithm.

```
      for item over violation queue do
         begin
            for item1 over suspect list do
               if ind of item = ind of item1
               then delete (item1, suspect list) fi
            ¢ delete all suspect reports on violator ¢ od;

            guess[ind of item] := maxguess + 1
         end ¢ 3 ¢ od;
¢ 4 ¢ definition of tinf:
      for z to maxarg do
         if undefined tinf[z] then
            begin violbnd := maxbnd + 1 ¢ lowest bound violated against at z ¢;
                  val := 0 ¢ eventual value of tinf[z] ¢;

                  for item over violation queue do
                     if arg of item = z and bnd of item < violbnd
                     then begin violbnd := bnd of item;
                                val := val of item
                          end
                     fi od;

                  if violbnd ≤ maxbnd ¢ serious violation at z detected ¢
                  then tinf[z] := val fi
            end
      fi ¢ 4 ¢ od;
¢ 5 ¢ cleanup: clear (violation queue);

      goto stage x+1
```

The program is initiated by starting *stage* 0 with all values undefined.

Note that the following non-interference relations hold:

(1),...,(5) leave the variables *maxguess*, *maxarg*, *maxbnd*, *maxind*, and *maxcomp* invariant.
(0),(3),(4) and (5) leave *guess* invariant.
(0),(1),(4) and (5) leave the suspect list invariant; (2) only extends this list and (3) only deletes from this list.
(0),(1),(2) and (4) leave the violation queue invariant; (3) loads this queue and (5) clears this queue completely.
(0),(1),(2),(3) and (5) leave *tinf* invariant.
tcomp is only used in (2).

3.3.4. CORRECTNESS OF THE UNION ALGORITHM

By lack of appropriate tools, it is not possible to present a rigid correctness proof of the union algorithm. The assertions formulated in this section mostly consist of a close inspection of the algorithm during its execution. In a few cases an assertion is shown to be invariant under the execution of the algorithm, by showing it to be preserved under the individual parts of some modular decomposition; this is in essence an inductive assertion method.

Readers who are already convinced of the correctness of the union theorem by the explanation in section 3.3.2 and who are willing to believe that the algorithm is designed according to our objectives, do better to skip this section.

In this section the computation of the program in the preceding section is analyzed. A number of assertions are now formulated, yielding Assertion 1 and Assertion 2 and the four correctness claims. These four claims together prove that the function tinf computed by the program indeed satisfies:

$$\bigcup_i F^A_{\omega}(t_i) = F^A_{\omega}(\text{tinf}).$$

By assumption the functions $t_i(x)$ satisfy:

$$\forall i,i',x[t_{i'}(x) \le \infty \ \underline{and}\ i < i' \ \underline{imp}\ t_i(x) \le t_{i'}(x)].$$

In the discussion the folloqing abbreviations are used:

(1) The status of the computation is mostly considered when its execution is in between two blocks of a stage. This moment is denoted as:
$\nprec$n,(1)$\nsucc$ for *stage* n, beginning of block (1) and
$\nprec$n,[1]$\nsucc$ for *stage* n, end of block (1).

(2) A suspect or violation report is denoted as a quadruple $<j,x,i,t_i(x)>$; we write $<j,x,i,t_i(x)> \in$ SL or $<j,x,i,t_i(x)> \in$ VQ to denote that this suspect (violation)-report is present on the suspect list (violation queue).

(3) Any identifier which is used in the program denotes in the discussion its current value (at the moment considered).

FACT 3.3.3. [Termination]. Each stage terminates.

PROOF. It is sufficient to show that each block terminates. For the blocks (0), (1) and (5) this is trivial. Block (2) consists of a for-loop embracing two disjoint for-loops, which contain only elementary actions and elementary tests, with exception of the assignment val := $\varphi_t^2(i,z)$. This latter assignment is executed only if newcomp ≥ i > tcomp[z], indicating that for a certain i' ≥ i we have $\Phi_t^2(i',z) \leq$ maxcomp. Now $t_{i'}(z) < \infty$ and therefore $t_i(z) < \infty$ also. Hence the computation of $\varphi_t^2(i,z)$ terminates.

Block (3) consists of a for-loop over the SL followed by a nested for-loop over the VQ and the SL; these loops contain only elementary actions and tests. Since the number of items on the SL and the VQ is bounded by maxind × maxarg × maxbnd, both the SL and the VQ are finite.

Block (4) consists of a for-loop embracing a foor-loop over the VQ which contains only elementary actions.

This completes the proof. □

COROLLARY 3.3.4. Each stage is executed. □

FACT 3.3.5. Let $\Phi_t^2(i,x) = y < \infty$, $i \leq x$. Then $\varphi_t^2(i,x) = t_i(x)$ is enumerated during precisely one *stage* n with $x \leq n \leq$ *max*(x,y). Moreover the predicate "$t_i(x)$ is enumerated before or at *stage* z" is equivalent to the predicate P defined by

$$P(i,x,z) = z \geq x \ \underline{and}\ \exists j[i \leq j \leq x \ \underline{and}\ \Phi_t^2(j,x) \leq z].$$

Hence n = $\mu z[P(i,x,z)]$.

PROOF. In order to have $t_i(x)$ enumerated at *stage* z it is necessary that during execution of the second loop in block (2) of *stage* z the relation tcomp[x] ≤ i ≤ newcomp holds. Afterwards but before ≮x,[2]≯ the assignment tcomp[x] := newcomp + 1 is executed. Consequently if $t_i(x)$ is enumerated during *stage* z one has tcomp[x] ≤ i at ≮z,(2)≯ and tcomp[x] > i at ≮z,[2]≯. Since the value of tcomp[x] does not decrease during the elaboration of the algorithm this shows that $t_i(x)$ is enumerated at most once.

The converse implication is easily seen to be true also: if tcomp[x] ≤ i at ≮z,(2)≯ and tcomp[x] > i at ≮z,[2]≯ then $t_i(x)$ must be enumerated at *stage* z.

Next assume that $t_i(x)$ is enumerated at *stage* z, i.e. tcomp[x] $\leq$ i at ⪡z,(2)⪢. In order that newcomp is raised above i during the execution of the first loop of block (2) of *stage* z one needs an index j with $i \leq j \leq x$ and $\Phi_j(x) \leq z$. Since moreover, $z \geq x$ we conclude that $P(i,x,z)$ holds. It is clear that $P(i,x,z)$ implies that $P(i,x,z')$ holds also for $z' > z$. This shows that $P(i,x,z)$ holds whenever $t_i(x)$ is enumerated before or at *stage* z.

Conversely, assuming that tcomp[x] $\leq$ i at ⪡z,(2)⪢ (so $t_i(x)$ is not enumerated before *stage* z) and that $P(i,x,z)$ holds, we read from the program that newcomp is increased and becomes indeed larger than i, and that t_i indeed is enumerated. Hence $P(i,x,z)$ implies that $t_i(x)$ is enumerated before or at *stage* z.

The proof is completed by observing that $P(i,x,z)$ is invalid for $z < x$ whereas $P(i,x,\mathit{max}(x,y))$ holds true. □

From the fact that the predicate P satisfies the relation

$$P(i,x,z) \ \underline{and}\ j < i \ \underline{imp}\ P(j,x,z)$$

we derive:

COROLLARY 3.3.6. [Correctness of enumeration]. If $i \leq x$ and if $t_i(x) < \infty$ then $t_i(x)$ is enumerated at precisely one *stage* z. If moreover $j < i$ then $t_j(x)$ is enumerated before or at *stage* z.

FACT 3.3.7. If guess[j] is (re-)defined and gets value i then no value of $t_i(x)$ is yet enumerated.

PROOF. guess[j] is defined for the first time at ⪡j,(1)⪢. In this situation guess[j] := maxguess $\geq$ maxbnd = j. At this stage no values of $t_i(x)$ are yet enumerated hence we should look at values enumerated before *stage* j. At that time however maxbnd < j, so no values of $t_i(x)$ are yet enumerated.

If guess[j] is redefined during (3) of *stage* n the new value becomes maxguess + 1 > maxbnd. Therefore no values of $t_{\text{maxguess}+1}(x)$ are yet known. □

COROLLARY 3.3.8. [Correctness of testing]. If at ⪡n,(3)⪢ guess[j] = i and if there exists a value $t_{i'}(x)$ with $i \leq i' \leq x$ which is enumerated before ⪡n,(3)⪢ then j is tested at x against i during execution of (2) of a stage before ⪡n,(3)⪢.

PROOF. Using the predicate P from 3.3.5 we conclude that P(i',x,n) holds. Now $i \leq i'$ implies that P(i,x,n) holds also, hence $t_i(x)$ is enumerated during execution of *stage* m (2) with $m \leq n$. From 3.3.7 it follows that $\langle m,(2)\rangle$ does not preceed the moment that guess[j] = i becomes valid. Now we can conclude that the test $A(j,x,t_i(x))$ = *true* ? is executed during execution of *stage* m (2).

FACT 3.3.9. [Correctness of manipulation with reports]. Let A1(j) be the assertion:
There exists a number (possibly zero) of SRs on j on the SL, all against the same bound t_i where the bound-index i equals the current value of guess[j], and the VQ contains no VR on j.

Let A2(j) be the assertion:
There exists a non-zero number of VRs on j in the VQ, all against the same bound t_i where the boundindex i is less then the current value of guess[j] and the SL contains no SRs on j.

Then for each j and at each moment in between two blocks during the computation the disjunction A1(j) *or* A2(j) holds.

Remark that A1(j) and A2(j) are never simultaneously true.

PROOF. A1(j) is trivial before $\langle j,(2)\rangle$. By studying the effect of execution of the blocks (0),...,(5) one proves that each of these blocks leaves the correctness of A1(j) *or* A2(j) invariant.

(0) and (1) both leave the SL and the VQ unchanged. Furthermore with exception of (1) of *stage* j, the value of guess[j] is not changed either. Hence after $\langle j,(2)\rangle$ the blocks (0) and (1) leave both A1(j) and A2(j) unaltered.

At $\langle n,(2)\rangle$ the VQ is still empty and therefore correctness of A1(j) *or* A2(j) implies that A1(j) holds. During execution of (2) the SL is left unaltered or extended by some further SRs; those of these SRs which are SRs on j are SRs against $t_{\text{guess}[j]}$. For this reason A1(j) holds also after completion of (2).

The effect of execution of block (3) depends on whether a violation by j is detected at the current stage or not. At $\langle n,(3)\rangle$ the VQ is empty and consequently A2(j) is not true. We may assume therefore that A1(j) holds at $\langle n,(3)\rangle$.

If no violation by j is detected then guess[j] remains unchanged; no

VR on j is placed in the VQ and all SRs on j remain undisturbed on the SL. Therefore A1(j) is still valid at $\langle n,[3]\rangle$.

If a violation by j is detected, then at least one SR on j is transformed into a VR on j which is placed in the VQ; the value of guess[j] is redefined and therefore increased, and all remaining SRs on j are deleted from the SL. Now A1(j) is no longer true but A2(j) is correct at $\langle n,[3]\rangle$.

Execution of (4) leaves the SL, VQ, and the value of guess[j] unchanged. If at $\langle n,(5)\rangle$ the assertion A1(j) holds the VQ contains no VRs on j either at $\langle n,(5)\rangle$ or at $\langle n,[5]\rangle$, whereas the SL and guess[j] are invariant under (5). If at $\langle n,(5)\rangle$ the assertion A2(j) holds then by the clearing of the VQ A2(j) becomes false but A1(j) becomes trivially true. Hence (5) leaves A1(j) *or* A2(j) invariant in this case too. This completes the proof. □

FACT 3.3.10. If guess[j] = i at a certain stage and if $A(j,x,t_i(x))$ = *false* for an $x \geq i$ then guess[j] is redefined at a future stage (and consequently guess[j] > i will become true).

PROOF. Assume by hypothesis to be shown contradictory that guess[j] stabilizes at i. Since $t_i(x) < \infty$ and $i \leq x$ the value $t_i(x)$ will be enumerated at a moment when guess[j] = i is already valid. Now the test $A(j,x,t_i(x))$ = = *true* ? is executed, and by hypothesis the answer is negative. This leads to the creation of an SR on j; $\langle j,x,i,t_i(x)\rangle \in$ SL becomes true.

Since SRs are only deleted from the SL during execution of (3) at which occasion the guess-values of the corresponding indices are redefined, one derives from the fact that guess[j] has stabilized at i that $\langle j,x,i,t_i(x)\rangle \in$ SL forever.

$A(j,x,t_i(x))$ = *false* implies that there exists a z so that for all w > z one has $A(j,x,w)$ = *true*. Consequently execution of *stage* w (3) where maxtest = w > z would result in the transfer of the SR $\langle j,x,i,t_i(x)\rangle$ to the VQ and to the redefinition of guess[j]. This is a contradiction.

This shows that guess[j] is indeed redefined. □

COROLLARY 3.3.11. [Claim 2]. If j violates t_i infinitely often then guess[j] does not stabilize at a value $k \leq i$.

PROOF. First consider the case k = i. As follows from 3.3.10, the assumption guess[j] = i at a certain moment implies guess[j] > i in some future. If k < i then the fact that j violates t_i infinitely often implies that j violates t_k infinitely often also. □

ASSERTION 3.3.12. [Assertion 1]. If tinf[x] is defined during *stage* n (4) and if guess[j] = i at $\nless n,(4)\ngtr$ and if k is the bound-index used at x then either $i \geq k$ or $A(j,x,t_i(x)) = A(j,x,t_k(x))$.

PROOF. If guess[j] is redefined during execution of *stage* n (3) then $k \leq \text{maxbnd} < \text{maxguess} + 1 = \text{guess}[j]$ and the assertion is proved. The case $i \geq k$ is also trivial, therefore we restrict ourselves to the case that $i < k$ and (consequently) guess[j] not redefined during *stage* n (3).

When $t_k(x)$ is enumerated, $k \leq x$ and consequently $i \leq x$ also. $t_i(x)$ is enumerated after guess[j] = i has become valid and $A(j,x,t_i(x))$ = *true* ? has been tested at this occasion (3.4.8). Depending on the outcome of this test there are two possibilities:

(i) $A(j,x,t_i(x))$ = *true*. Now $t_i(x) \leq t_k(x)$ implies $A(j,x,t_k(x))$ = *true* as well and we are ready.

(ii) $A(j,x,t_i(x)) \neq$ *true*. This results in the creation of the SR $\langle j,x,i,t_i(x)\rangle$ and since guess[j] = i is still valid at $\nless n,(4)\ngtr$ we still have $\langle j,x,i,t_i(x)\rangle \in$ SL at $\nless n,(4)\ngtr$.
The absense of the VR $\langle j,x,i,t_i(x)\rangle$ in the VQ at $\nless n,(4)\ngtr$, implies that $A(j,x,\text{maxtest}) \neq$ *true*; for the index j' which violates against t_k at x, and which is used to define tinf[x] one has $A(j',x,\text{maxtest})$ = *true*.
Therefore: $t_i(x) \leq t_k(x) < \text{maxtest}$ which implies:
$A(j,x,t_i(x)) = A(j,x,t_k(x)) = A(j,x,\text{maxtest}) \neq$ *true*.

This completes the proof. □

COROLLARY 3.3.13. If during *stage* n (3) a VR on j at x is created then j violates tinf at x.

PROOF. Let $\langle j,x,i,t_i(x)\rangle$ be the VR on j at x and let this VR be produced at *stage* n (3). Let tinf[x] be defined at *stage* m (4). Let i' be the value of guess[j] at $\nless m,(4)\ngtr$.

Since *stage* m is the first stage during which VRs at x are produced (this follows directly from the program text of block (4)) one concludes both that m exists and that $m \leq n$.

We consider several possibilities:

(i) $n = m$.
By our choice of the bound-index used at x we conclude that $i \geq k$. If $A(j,x,t_i(x))$ = *false* then $A(j,x,\text{tinf}[x]) = A(j,x,t_k(x))$ = *false* also.

(ii) $n > m$ and $i' = i$.

From <u>Assertion 1</u> we now conclude that either $i' \geq k$ in which case $A(j,x,t_{i'}(x))$ = *false* which implies that $A(j,x,tinf[x]) =$ $= A(j,x,t_k(x))$ = *false* also or otherwise $A(j,x,t_{i'}(x)) = A(j,x,t_k(x))$ which also implies $A(j,x,tinf[x])$ = *false*.

(iii) $n > m$ and $i' \neq i$.

In this case $i > i'$ since i is a more recent value of guess[j] then i', When the redefinition guess[j] = i was executed $t_k(x)$ was already enumerated, hence $i > k$. Combining this with the fact that $A(j,x,t_i(x))$ = *false* one derives that $A(j,x,tinf[x]) = A(j,x,t_k(x)) =$ = *false*. □

Note that case (iii) is actually made impossible by the fact that maxguess = maxarg. The assumed redefinition of guess[j] implies $i > x$ and consequently $t_i(x)$ is never enumerated, and the assumed VR $<j,x,i,t_i(x)>$ is never created.

Case (iii) is included in the proof to show that the correctness does not depend on the option maxguess ≥ maxarg.

The next assertion is again based on the choice maxguess ≥ maxarg. This assertion is inessential for the remainder of the proof.

<u>FACT 3.3.14</u>. [Prevention of multiple testing]. If $<j,x,i,t_i(x)> \in$ VQ at ≮n,(4)≯ and if $<j,x',i',t_i(x')> \in$ VQ at ≮n',(4)≯ then $x \neq x'$ or the two VRs are equal and $n = n'$.

<u>PROOF</u>. If $n = n'$ then the two VRs are derived from two SRs which were simultaneously on the SL at ≮n,[2]≯. Then $i = i'$ and consequently $x \neq x'$ since otherwise the two SRs were equal.

If n and n' are distinct we may assume that $n < n'$. At ≮n,[5]≯ there is no SR on j present on the SL and the VQ is empty. Moreover guess[j] = = maxguess + 1 > maxarg ≥ x. Because i' is at least this value of guess j at ≮n,[5]≯ and $i' \leq x'$ we conclude that $x' > x$.

This completes the proof. □

<u>COROLLARY 3.3.15</u>. [<u>Claim 1</u>]. If guess[j] is unstable then j violates tinf infinitely often.

<u>PROOF</u>. If guess[j] is redefined at *stage* n_a (3) then at ≮n_a,(4)≯ a VR

$<j,x_a,i_a,t_{i_a}(x_a)> \in VQ$ where i_a equals the value of guess[j] at $\langle\!\langle n_a,(3)\rangle\!\rangle$.

Consequently as follows from 3.3.13 j violates tinf at x_a. Moreover, $n_a < n_b$ implies $i_a < i_b$ and by our enumeration technique we have $x_a \geq i_a$.

If there exists an infinite sequence of stage numbers n_a with this property then the sequence of corresponding arguments x_a contains infinitely many elements as well, and consequently j violates tinf infinitely often. □

Note that from 3.3.14 we derive straightforward $n_a \neq n_b$ *imp* $x_a \neq x_b$.

FACT 3.4.16. [Assertion 2]. For each k the number of arguments where the bound-index used at x (denoted by k(x)) satisfies $k(x) \leq k$ is finite.

PROOF. After *stage* k+1 we have maxguess > k. From that moment on each value guess[j] which is redefined will be made larger than k. The set G_k defined by $G_k = \{j \mid \text{guess}[j] \leq k\}$ is a finite set which decreases after $\langle\!\langle k+1,[0]\rangle\!\rangle$.

We prove that for each argument x where tinf[x] is defined and where $k(x) \leq k$ an index j is deleted from G_k. Assertion 2 is now a consequence of the exhaustion of the finite set G_k.

Let tinf[x] be defined at *stage* n, (4) with n > k. Then the following implications hold (x and n being constant):

$k(x) \leq k$

imp

there exists a VR $<j,x,l,t_i(x)> \in VQ$ at $\langle\!\langle n,(4)\rangle\!\rangle$ with $l \leq k$

imp

there is an SR $<j,x,l,t_l(x)> \in SL$ at $\langle\!\langle n,(3)\rangle\!\rangle$ with guess[j] = $l \leq k$ at $\langle\!\langle n,(3)\rangle\!\rangle$ and during execution of *stage* n (3) this SR becomes a VR and guess[j] is redefined to be equal maxguess + 1 > k

imp

there exists a j with $j \in G_k$ at $\langle\!\langle n,(3)\rangle\!\rangle$ and $j \notin G_k$ at $\langle\!\langle n,(4)\rangle\!\rangle$.

If $k(x) \leq k$ for infinitely many x, then for infinitely many of these arguments tinf[x] is defined after $\langle\!\langle k+1,[0]\rangle\!\rangle$. At each stage tinf[x] is defined for at most a finite number of arguments x. Consequently infinitely many elements are deleted from the finite set formed by the members of G_k at $\langle\!\langle k+1,[0]\rangle\!\rangle$. This is a contradiction. □

COROLLARY 3.3.17. [Claim 4]. If guess[j] stabilizes then j respects tinf.

PROOF. As follows from Claim 2 guess[j] does not stabilize unless j respects some t_k. However the fact that $k(x) \geq k$ almost everywhere implies that tinf[x] $\geq t_k(x)$ for almost all x. Therefore j respects tinf as well. □

ASSERTION 3.3.18. [Claim 3]. If j respects some bound t_k then guess[j] stabilizes.

PROOF. Suppose that guess[j] does not stabilize. Then guess[j] is redefined during *stage* n_a for infinitely many n_a. Then redefinitions correspond to an infinite sequence of VRs $<j,x_a,i_a,t_{i_a}(x_a)>$ where i_a strictly increases and where $x_a \geq i_a$. Therefore there exist arbitrary large x and i with $A(j,x,t_i(x))$ = *false*.

If j respects t_k then there exists a z so that for all $x \geq z$ where $t_k(x)$ is defined $A(j,x,t_k(x)) \neq$ *false*. The monotonicity of the sequence $(t_i)_i$ implies:

$$\exists z \forall x \geq z \forall i \geq k [t_i(x) < \infty \ \underline{imp}\ A(j,x,t_i(x)) \neq \textit{false}].$$

This contradicts the existence of arbitrarily large x and i with $A(j,x,t_i(x))$ = *false*. □

THEOREM 3.3.19. [Union theorem]. The function tinf computed by the program in 3.4.3 satisfies:

$$\bigcup_i F_{\omega}^{A}(t_i) = F_{\omega}^{A}(\text{tinf}).$$

PROOF. Let $j \in \bigcup_i F_{\omega}^{A}(t_i)$. Then there exists a k so that $j \in F_{\omega}^{A}(t_k)$; now by Claim 3 guess[j] stabilizes and by Claim 4 $j \in F_{\omega}^{A}(\text{tinf})$.

If $j \notin \bigcup_i F_{\omega}^{A}(t_i)$ then there exists no k so that $j \in F_{\omega}^{A}(t_k)$. Therefore guess j does not stabilize at a value $\leq k$ for each k by Claim 1. This shows that guess[j] is instable, and consequently, by Claim 2, $j \notin F_{\omega}^{A}(\text{tinf})$.

This completes the proof. □

3.3.5. APPLICATIONS AND REMARKS

The union theorem can directly be applied to classes consisting of total functions or programs. The reason is that equality of two sets of programs (functions) remains valid if both sides are intersected with a fixed third class (in this case the class of total programs (functions)).

For the complexity classes with partial names as defined by E.L. ROBERTSON [Rb 71] a union theorem makes no sense. If the domain condition

$$\varphi_j \in F'_t \ \underline{\mathit{iff}}\ \varphi_j \in F_t \ \underline{\mathit{and}}\ \mathcal{D}\varphi_j \supset \mathcal{D}t$$

is enforced, then the sequence of names defined in <u>example 3.3.2</u> yields a "counterexample" to a union theorem for the classes F'_t.

<u>EXAMPLE 3.3.20</u>: [The union theorem does not hold if domain conditions are enforced]. Let

$$F'_t = \{\varphi_i \mid \varphi_i \in F_t \ \underline{\mathit{and}}\ \mathcal{D}\varphi_i \supset \mathcal{D}t\}.$$

Let

$$t_i = \lambda x[\underline{\mathit{if}}\ x \le i\ \underline{\mathit{then}}\ \underline{\mathit{loop}}\ \underline{\mathit{else}}\ 0\ \underline{\mathit{fi}}].$$

Suppose moreover that $t_i = \varphi_{\sigma(i)}$ with $\Lambda\varphi_{\sigma(i)} = \Lambda\Phi_{\sigma(i)}$ and suppose that all other programs have run-times larger than zero for almost all their arguments. Then for no function tinf the equality $\bigcup_i F'_{t_i} = F'_{tinf}$ is correct.

<u>PROOF</u>. One has $F'_{t_i} = \{\varphi_{\sigma(0)}, \varphi_{\sigma(1)}, \ldots, \varphi_{\sigma(i)}\}$ and therefore $\bigcup_i F'_{t_i} = \{\varphi_{\sigma(i)} \mid i \in \mathbb{N}\}$. Suppose now that $\bigcup_i F'_{t_i} = F'_{tinf}$. If $\mathcal{D}tinf = \emptyset$ then $F'_{tinf} = \mathcal{P} \ne \bigcup_i F'_{t_i}$. If $\mathcal{D}tinf \ne \emptyset$ then let z be the least element of $\mathcal{D}tinf$; now $\varphi_{\sigma(z)} \notin F'_{tinf}$ so again $\bigcup_i F'_{t_i} \ne F'_{tinf}$. □

The general union theorem on abstract resource-bound classes yields several other union theorems by selecting as acceptance relation one of the specific examples given in 3.1.3.

It should be noted that the union theorem for complexity classes modulo sets of exceptional points can be found already in the thesis of L. BASS [Ba 70]. He uses the trick to encode the set of exceptional points in the guess-value of the program and not, as is done in our proof, in the the index of the program.

If the class E of sets of exceptional points is not assumed to be recursively presentable the union theorem becomes invalid. L. BASS [Ba 70] describes a sequence of classes $C^{E}_{t_i}$ with $(t_i)_i$ non-decreasing and total, and E not recursively presentable, such that $\bigcup_i C^{E}_{t_i} \ne C^{E}_{tinf}$ for each total function tinf.

If the class E is recursively presentable then we can eliminate this class E completely by replacing the names of the classes by other names (which in general are partial); moreover this renaming can be done uniformly.

THEOREM 3.3.21. Let $E = (E_i)_i$ be a recursively presentable class of sets of exceptional points. Then there exists a transformation of programs σ so that $F^{E}_{\varphi_i} = F_{\varphi_{\sigma(i)}}$ for each index i.

PROOF. Since E is recursively presentable there exists a recursive function e such that $e(i,x) = \underline{if}\ x \in E_i\ \underline{then}\ \infty\ \underline{else}\ 1\ \underline{fi}$.

We define the transformation of programs τ by:

$$\varphi_{\tau(i,j)}(x) \Leftarrow \varphi_j(x) * max\{e(k,x) \mid k \le i\}.$$

Hence

$$\varphi_{\tau(i,j)}(x) = \underline{if}\ x \in \bigcup_{k\le i} E_k\ \underline{then}\ \infty\ \underline{else}\ \varphi_j(x)\ \underline{fi}.$$

Although we have not stated this explicitly in the formulation of the union theorem one derives easily, using the algorithm given to compute tinf, that the program for tinf depends uniformly on the program for the sequence $(t_i)_i$. We consider the sequence $(\varphi_{\tau(i,j)})_i$ to be a sequence of programs which depends uniformly on the index j. Since this sequence satisfies for each value of j the monotonicity condition $\varphi_{\tau(i,j)} \le \varphi_{\tau(i+1,j)}$ there exists a sequence of names $(tinf_j)_j$ which satisfy:

$$\bigcup_i F_{\varphi_{\tau(i,j)}} = F_{tinf_j}.$$

Moreover $tinf_j$ depends uniformly on j; there exists a transformation of programs σ so that $(\varphi_{\sigma(j)}) = tinf_j$. We claim that σ is the transformation we need.

From the union theorem it follows that:

$$\begin{aligned}
\varphi_k \in F_{\varphi_{\sigma(j)}}\ & \underline{iff}\ \exists i[\varphi_k \in F_{\varphi_{\tau(i,j)}}] \\
& \underline{iff}\ \exists i\overset{\infty}{\forall} x[\varphi_{\tau(i,j)}(x) < \infty\ \underline{imp}\ \Phi_k(x) \le \varphi_{\tau(i,j)}(x)] \\
& \underline{iff}\ \exists i\overset{\infty}{\forall} x[x \notin \bigcup_{1\le i} E_1\ \underline{and}\ \varphi_j(x) < \infty\ \underline{imp}\ \Phi_k(x) < \varphi_j(x)] \\
& \underline{iff}\ \exists i\exists F[\#F < \infty\ \underline{and}\ \forall x[x \notin \bigcup_{1\le i} E_1 \cup F\ \underline{and}\ \varphi_j(x) < \infty\ \underline{imp}\ \Phi_k(x) < \varphi_j(x)]] \\
& \underline{iff}\ \exists 1\forall x[x \notin E_1\ \underline{and}\ \varphi_j(x) < \infty\ \underline{imp}\ \Phi_k(x) < \varphi_j(x)] \\
& \underline{iff}\ \varphi_k \in F^{E}_{\varphi_j}.
\end{aligned}$$

These equivalences follow from the fact that $\mathcal{E}$ is closed under finite unions and contains all finite sets. One concludes that $F_{\varphi_{\sigma(j)}} = F^{\mathcal{E}}_{\varphi_j}$.

This completes the proof. □

An application of the union theorem for weak classes with partial names is the proof given below that the family of all programs with bounded range is an honesty class (provided that no computation terminates in zero steps).

DEFINITION 3.3.22. A program φ_i is a program with bounded range, if there exists a number k so that $\forall x[\varphi_i(x) < \infty \ \underline{imp}\ \varphi_i(x) \le k]$.

ASSERTION 3.3.23. Suppose that $\Phi_i(x) > 0$ for each i and x. Then the set of programs with bounded range is an honesty class of programs.

PROOF. Let

$$R_k = \lambda x,y[\underline{if}\ y \le k\ \underline{then}\ \underline{loop}\ \underline{else}\ 0\ \underline{fi}].$$

Now G_{R_k} is the set of all programs φ_i so that

$$\overset{\infty}{\forall} x[\varphi_i(x) < \infty\ \underline{imp}\ \varphi_i(x) \le k].$$

Hence the programs in G_{R_k} have a bounded range. Conversely, suppose that φ_i has a bounded range. Then there exists a k so that $\varphi_i(x) < \infty$ implies $\varphi_i(x) \le k$ and consequently $\varphi_i \in G_{R_k}$.

The conclusion is that $\bigcup_k G_{R_k}$ is precisely the class of all programs with bounded range. However, this union is also an honesty class, as follows from our union theorem. □

Considering the name Rinf for $\bigcup_k G_{R_k}$ which is computed by our union algorithm in section 3.3.3 we can make the following observations:

(i) $\forall x,y[\mathrm{Rinf}(x,y) = \infty\ \underline{or}\ \mathrm{Rinf}(x,y) = 0]$

since a value of Rinf is also a value of one of the R_k.

(ii) $\forall y \overset{\infty}{\forall} x[\mathrm{Rinf}(x,y) = \infty]$.

Otherwise a program with bounded range, which is not contained within G_{Rinf} could be constructed by diagonalization.

(iii) If f has infinite range then $\mathcal{D}$Rinf contains infinitely many elements from the graph of f.
Otherwise f would be contained in H_{Rinf}.

These peculiar properties of the domain of Rinf can also be expressed as follows:

(iv) If the (non-recursive) function k is defined by:

$$k(x) = \mu y[\exists z \geq x[\text{Rinf}(z,y) = 0]]$$

then k has unbounded range, but k increases slower than any partial recursive function with unbounded range.

A set with analogous properties can also be constructed straightforwardly; for example one might enumerate the graphs of all φ_i simultaneously, accepting from the graph of φ_k at most k points $<x_0,y_0>,\ldots,<x_n,y_n>$ so that $y_0 < y_1 < \ldots < y_n$.

At the same time this example shows that the concept of an honesty class with a partial name has little to do with the original concept of a function whose complexity is bounded by its size.

CHAPTER 3.4

THE NAMING THEOREM AND THE MEYER-McCREIGHT ALGORITHM

3.4.0. INTRODUCTION

This chapter discusses the MEYER-McCREIGHT naming theorem [MMC 69] and some related topics. This theorem states the existence of a transformation of programs σ so that the sequence $(\varphi_{\sigma(i)})_i$ is a measured set, and so that for each index i, the function φ_i and $\varphi_{\sigma(i)}$ are names of the same complexity class. The transformation σ is also called an honesty procedure on P, since the members of the measured set form a honest family of functions.

Although in the original proof only the restriction of σ to R is considered, and σ is defined in such a way that $\varphi_{\sigma(i)}$ is total whenever φ_i is total, (a so-called honesty procedure on R), the proof yields the result for P as well. The first Blum axiom is not used, and therefore a straightforward translation for strong abstract resource-bound classes can be proved. See section 1 of this chapter.

Section 2 contains the proof that an analogous uniform honesty procedure for weak abstract resource-bound classes is impossible. More specifically, we show that for every transformation of programs σ such that $(\varphi_{\sigma(i)})_i$ is a measured set, an index e can be found so that $H_W^{Cpl}(\varphi_e) \neq H_W^{Cpl}(\varphi_{\sigma(e)})$. A similar result is given for honesty classes and finally the construction of a total φ_e with the property that the above inequality holds proves the non-existence of honesty proceudres on R for weak classes as well.

The observations made in an informal correctness proof in section 1 are used in section 3 to construct a number of modified MEYER-McCREIGHT algorithms which prove some properties of the system of names of strong classes. We show that it is possible to rename a class $F_S(\varphi_i)$ by two names $\varphi_{\sigma_1(i)}$ and $\varphi_{\sigma_2(i)}$ having disjoint domains. We next prove a generalization of the union theorem where the condition of monotonicity of the sequence of names is repalced by the condition that the corresponding classes of indices themselves form a monotonic sequence of classes.

In section 4 we consider the problem whether a given set of indices can be represented as a strong abstract resource-bound class in a given acceptance relation. We prove that a suitable modification of the MEYER-McCREIGHT algorithm acts as a closure operator; it computes a name t so that a given class X is entirely contained within $F(t)$ and so that the class $F(t)$ is minimal with this property. If X is already a strong abstract resource-bound class named by some unknown name then the two classes are equal.

Section 5 contains a further analysis of the possibilities to save something of the MEYER-McCREIGHT algorithm for weak classes. By equipping the algorithm with a so-called "wizard" we loose measuredness but the renaming properties are preserved. As applications we find a generalized union theorem for weak classes with total names, and an intersection theorem.

3.4.1. THE MEYER-McCREIGHT ALGORITHM; AN INFORMAL DESCRIPTION AND CORRECTNESS PROOF

A formulation of the naming theorem in the context of abstract resource-bound classes reads:

THEOREM 3.4.1. For every acceptance relation A there exists a measured transformation of programs σ such that for each index i the equality $F^A_S(\varphi_i) = F^A_S(\varphi_{\sigma(i)})$ holds.

A straightforward corollary is a naming theorem for the classes $G^A_S(t)$ and $H^A_S(t)$.

DEFINITION 3.4.2. A transformation σ satisfying the condition of theorem 3.4.1 is called an *honesty procedure on* P. If σ maps R into R, σ is called an *honesty procedure on* R.

All honesty procedures which are known up to now are variants of the original MEYER-McCREIGHT algorithm [MCM 69] or the simplification of this algorithm as given by R. MOLL [Mo 72]. We call these algorithms therefore MEYER-McCREIGHT algorithms, which is abbreviated to MMC-algorithms.

In the sequel of this section we explain the design of the MMC-algorithm which is formally defined in the appendix. The essential ideas needed to prove the correctness are derivable from the design alone, and hence we disregard all implementation problems which might disturb our argumentation.

Our MMC-algorithm is described as a program to compute $\varphi_{\sigma(i)} = t'$ by enumeration of the graph of t'; since this program depends uniformly on the index i, this implicitly defines the transformation σ.

To enforce that the sequence $(\varphi_{\sigma(i)})_i$ is a measured set we design this enumeration in such a way that for increasing y pairs <x,y> are tested to see whether they belong to the graph of t' or not; moreover, we prevent a pair <x,y> being accepted as member of the graph of t' in case it was rejected at an earlier occasion. In other words: t' is computed by a least number operator application on a certain total but dynamically changing condition uniformly depending on i, j, x, and y.

Since it is never certain that the computation of t'(x) converges, the algorithm executes a dovetailed computation of t'. Traditionally one uses a stagewise description, where, at each stage, for one or more pairs <x,y> the equality t'(x) = y is tested (and in most cases rejected). This implies that in the case where t'(x) diverges in the course of time an infinite number of attempts to compute t'(x) will be elaborated, none of which is going to succeed.

From the point of view of the user of the algorithm, a description, where all values t'(x) are enumerated in parallel, using a least number operator on some continuously changing condition, gives a better impression on what the algorithm is intended to do.

The difficult part of the design of an MMC-algorithm is to make the equality $F(t) = F(t')$ true. This is done by enforcing the following equivalence:

j violates infinitely often against t *iff*
j violates infinitely often against t'

To enforce this relation one tries to create a violation by j against t' (by taking a "low" value for t'(x)) when a violation by j against t(x) has been discovered. However, t' should be larger than the run-times of the indices which either have committed no violations yet, or have been punished already for all the violations committed by them up to the current moment.

It is not hard to see that these requests on the indices currently under consideration may contradict each other. To resolve this problem we introduce a priority system. Each index j is given a priority status consisting of a Boolean b(j) and a priority number p(j). The Boolean b(j) represents whether j has still an "unpunished violation" on its "crime record" (this is encoded by b(j) = *false*). We say j is "on the black list" or shortly "j is black" to describe b(j) = *false*. The remaining indices form the "white list". The number p(j) places j in the so-called "priority queue". This priority queue consists of the indices introduced up to the current moment, ordered by increasing priority number. A higher value of p(j) means a lower priority.

Except for the introduction of an index in the algorithm, at which occasion this index is placed at the tail of the priority queue (with a value for b(j) which is irrelevant at this occasion), the indices are submitted to the following acts of "social regulation":

a) if a white index is detected to violate a newly enumerated value of t(x) it is moved as a black index to the tail of the priority queue.
b) If a black index is detected to violate a newly created value of t'(x) it is moved as a white index to the tail of the priority queue.

The values of t'(x) are defined in such a way that "justice" as described above is served in the most effective way. We want to "punish" a black index j_0 by creating a violation by j_0 against t'(x) but at the same time we should respect the demands of the white indices j, having higher priority than j_0 (the requests of indices which are white and have lower priority may be refuted; our algorithm is an implementation of corruption). More formally, t'(x) is defined to be:

"the least z such that there exists a black index j_0 for which $A(j_0,x,z) \neq$ *true* whereas for all white indices j having higher priority $A(j,x,z) =$ *true*".

(The priority of) this index j_0 above is called (if present) *the index (priority) used at* x.

Since it is quite possible that such a value z does not exist we protect ourselves against the resulting infinite computations by "dovetailing" the algorithm which computes this minimal z and the corresponding index j_0.

The diagram below explains the problem and the algorithm used to solve it. Horizontally we draw the successive indices in the priority queue, the priority decreasing from let to right. In the vertical direction we draw the run-times of these indices at x whereas the "colour" of each index is indicated by halflines going up for white indices (meaning that z should be at least as large as the corresponding run-time) and going down for black indices (meaning that z should be less than this run-time).

In the situation represented $z = \alpha_{j_4}(x)$ and $j_0 = j_7$ is a solution to the problem.

If at some point in time index j_4 should be moved to the tail of the priority queue, the solution would become $z = \alpha_{j_2}(x)$ and $j_0 = j_5$. This solution yields a lower value of z.

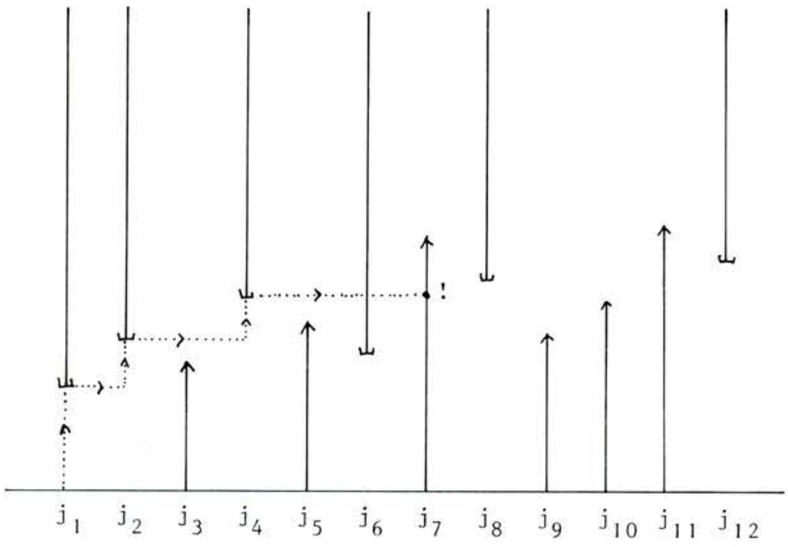

!
j_1 j_2 j_3 j_4 j_5 j_6 j_7 j_8 j_9 j_{10} j_{11} j_{12}

Diagram 3.4.3

As the example shows, care must be taken that by dovetailing no "revisions" occur in the sense that solutions which are rejected, are accepted at subsequent consideration. We will return to this problem in a moment.

The dotted line in the diagram represents the algorithm which we will use in the sequel to solve the problem.

Starting at the first index with the value z = 0 (or at a second try z = low, where low is the first value not yet tried) we walk through the diagram guided by the following instructions:

if we are considering a white index j for which $A(j,x,z) \neq$ *true* then we increase z by one and we try again

otherwise

if we are considering an index j with $A(j,x,z) =$ *true* then we proceed to the next index in the priority queue if such an index is present (otherwise report failure) otherwise

if we have a (black) index with A(j,x,z) ≠ _true_ then we have found the solution.

The algorithm is implemented below in the subroutine *searchtime* which is used in our informal description of the MMC-algorithm. The meaning of the parameters is the following:

x is the argument we are trying to compute t'(x) for;
low is the first value not yet tried as t'(x);
high is the largest value which will be tried as t'(x) at this run (to prevent infinite computations);
prior is a parameter which represents the priority queue [*)] used in the computation.

If the computation of searchtime terminates succesfully it has as output the value of z in the variable *val* and the index used at x in the variable *candidate*.

```
proc searchtime = (int x,low,high,¢ prior queue ¢ prior) void:
begin int val := low, candidate := ¢ first index in prior ¢;
   while val ≤ high do
      if ¢ b(candidate) and A(candidate,x,val) ≠ true ¢
            then val +:= 1
      elif ¢ A(candiate,x,val) = true ¢
              then ¢ if still candidate available
                     then candidate := next candidate in prior
                     else goto failure
                     fi ¢
      else ¢ t'(x) := val; index used at x := candidate; report success ¢
      fi             od;
   failure: ¢ report falure to calling program ¢
end ¢ searchtime ¢;
```

In the actual program the expressions _¢_ ... _¢_ are replaced by ments and expressions formally computing the values suggested by the clause in between the underlined comment symbols.

*) In an actual implementation it is handy if *searchtime* has access to *prior* as a linear list, whereas the rest of the program treates *prior* as an increasing array of priority-status values. We postpone the discussion on this interface till the appendix.

It is not hard to verify that in this algorithm at each occasion where candidate gets a new value, val equals the maximum of low and the maximal run-time of a white index with a higher priority than the index which is the new value of candidate. Moreover val is at least as large as the maximal run-time of a black index with higher priority.

If the new candidate happens to be a black index with run-time larger than the current value of val then the algorithm terminates successfully. In other words, the algorithm will not proceed in the priority queue beyond the first black index having a run-time at x larger than low and larger than the run-times of all the (white) indices having higher priority.

In the description of searchtime we gave already the beginning of an implementation of an MMC-algorithm. Our motivation is that the absense of this or a similar implementation of the "definition" of t'(x) as given before is the main reason why the known MMC-algorithms in the literature are hard to understand [MCM 69] [HH 71].

To complete the design we should explain how to prevent the acceptance of solutions being rejected at an earlier occasion. Two strategies are known in the literature.

The first strategy, which is used in the original MMC-algorithm of MEYER and McCREIGHT [MCM 69], uses for each computational try for t'(x) as priority queue status the historical contents of the priority queue at the end of *stage* n, where n in fact equals the value of the argument x. Therefore the problem which searchtime must solve, depends only on x and not on the stage during which searchtime is called.

The second strategy is used in the simplified MMC-algorithm of R. MOLL [Mo 73]. In this algorithm prior represents the current state of the priority queue; however, by using a non-zero value for low each value z is tried at at most one occasion, and therefore no revisions occur (in fact low = = high is used and at each call of searchtime only one value z is tried).

Having explained the design globally, we can now give an informal description of a complete MMC-algorithm. A stagewise description is used, without being to explicit on the amount of work executed during the different sections of a stage. This is motivated by our opinion that this choice should be made for an actual implementation. We must however be sure that (1), (2) and (4) are organized in such a way that:

(i) all indices j are introduced,

(ii) all converging values of t(x) are actually enumerated,

(iii) computations of t'(x) are tried until t'(x) has converged; moreover the value of high used in these computations grows to infinity where as at no moment low is larger (or, if no protection against revisions is used, smaller) than the first value z not yet tried at x,

(iv) the priority queue status used for the computation of t'(x) should represent the actual status of the priority queue in the computation at some stage in between *stage* x and the current stage.

(These conditions are needed to make the argumentation in our "correctness-proof" valid.)

The MMC-algorithm is initialized by clearing the lists of enumerated values of t and t' and the priority queue. Next *stage* 1 is executed.

stage k consists of the following 6 sections:

(1) Introduction of a finite number of indices j by placing them at the tail of the priority queue (their colours b(j) are irrelevant).
(2) Using a universal enumerator universal(i,x,z) we search for some time whether we can find one or more new values of t(x).
(3) If we have generated in (2) new values of t(x), we use them to test for all currently white indices j whether A(j,x,t(x)) = *true* or not. The indices which are detected to violate t are moved to the tail of the priority queue and become black indices.
(4) Using a devise against unwanted revisions of earlier rejections we execute a number of calls of searchtime, hoping to find new values of t'.
(5) If in (4) new values of t'(x) have been created, we use them to test for all currently black indices whether A(j,x,t'(x)) = *true* or not. The indices which are detected to violate t' are moved to the tail of the priority queue and become white indices.
(6) Proceed to *stage* k+1.

Before giving a "correctness-proof" we like to isolate sections (2) and (3) from their surroundings. In these two sections the tests are performed which discriminate between the indices which are contained in F(t) and those indices which are not. This discrimination should be understood in the following way: if an index j is a member of F(j) then it will be moved on the black list by (2) and (3) only a finite number of times; if however, $j \notin F(t)$ then (2) and (3) will remove j from the white list at a future stage, each time j happens to become white.

The sections (2) and (3) together are called the *discriminator* of the MMC-algorithm.

A further important observation concerns the index j_0 and the priority used at x. From the definition it is clear that this priority is the priority of a black index. Moreover by definition $A(j_0,x,t'(x)) \neq$ *true*. Hence, if the index j_0 still is a black index at the moment t'(x) is defined, it will be removed from the black list and its black priority will be deleted. It is however also possible (by the fact that the priority-queue status used to compute t'(x) was a "historical snapshot") that the black priority of j_0 was deleted already a long time ago. In both cases, however, the black priority of j_0 is a "mortal" one. From this we conclude that a certain priority cannot be used at infinitely many arguments x and that consequently for each k the number of arguments x at which a priority $\leq k$ is used is finite.

We now turn to our correctness proof for the MMC-algorithm. We devide the set of all indices in three classes, depending on the behaviour of their priority status $\langle p(j),b(j)\rangle$.

Case 1. p(j) does not stabilize.

In this case j is transferred infinitely many times from the white list to the black list (for violating t) and transferred back afterwards (for violating t'). In our design we test each index j only once for each argument against t respectively t'. Hence the number of violations by j against both t and t' is infinite: $j \notin F(t)$ and $j \notin F(t')$.

Case 2. p(j) stabilizes with b(j) = *true* (j is white-stable).

In this case j violates t at at most finitely many arguments, and therefore $j \in F(t)$. Let k be the value at which p(j) stabilizes. For almost all x the priority used at x is larger than k, and this means that for almost all x, by definition of t'(x), we have $A(j,x,t'(x)) =$ *true*. Therefore $j \in F(t')$ also.

Case 3. p(j) stabilizes with b(j) = *false* (j is black-stable).

In this case it is almost trivial that $j \in F(t')$. The tricky part is to show that $j \in F(t)$. We derive this from the following assertion.

ASSERTION 3.4.4. If p(j) becomes stable with b(j) = *false* and with p(j) = k, then the run-time of j is bounded almost everywhere by the maximum of the run-times of those indices i which are white-stable and stabilize at priorities $p(i) \le k$.

This assertion can be derived by considering the computations of searchtime as represented by diagrams like diagram 3.4.3 for x going to infinity.

For sufficiently large x the priority queue status used to compute t'(x) will represent a situation where p(j) together with all priorities $p(i) \le p(j)$ have become stable priorities. This means that the head of the priority queue upto j has become stable. For each x and at each occasion the priority queue used to compute t'(x) starts with the same sequence of indices having the same colours.

Now suppose that for such an x the run-time of j is larger than the (finite!) maximum of the run-times of the white stable indices with higher priorities. Then our search algorithm cannot move in the priority queue beyond j without finding j to be a solution. It can however not terminate before reaching j since this would mean that a priority $p(i) \le p(j)$ is used (which contradicts the fact that p(i) is stable). It is also not possible that our search algorithm diverges with val going to infinity on the initial segment of the priority queue since we assumed that the maximum of the white run-times on this segment is a finite value. Hence we must accept that index j is accepted as solution, and this contradicts the fact that j has a black stable priority. This proves assertion 3.4.4. □

From assertion 3.4.4 one easily derives that $j \in F(t)$. Let $\{j_1,\ldots,j_l\}$ be the finite set of white indices which stabilize with $p(j_n) \le p(j)$. Now by assertion 3.4.4 one has:

$$\overset{\infty}{\forall}x[\alpha_j(x) \le \underline{max}(\alpha_{j_1}(x),\ldots,\alpha_{j_l}(x))].$$

Since j_n is a white-stable index we have $j_n \in F(t)$; hence

$$\overset{\infty}{\forall}x[\alpha_{j_n}(x) \le t(x)] \qquad (1 \le n \le l).$$

Therefore:

$$\overset{\infty}{\forall}x[\underline{max}(\alpha_{j_1}(x),\ldots,\alpha_{j_1}(x)) \le t(x)]$$

and consequently

$$\overset{\infty}{\forall}x[\alpha_j(x) \le t(x)].$$

Hence $j \in F(t)$; this completes the proof. □

To complete this section we emphasize that assertion 3.4.4 does not depend on the actual structure of the discriminator in the MMC-algorithm, but represents a property of this algorithm in a much more general shape. This observation will be a crucial argument in section 3 and 4 of this chapter.

A final observation is that the argumentation given above is highly non-constructive. Given a concrete MMC-algorithm it is possible to give constructive proofs as well (see for example [EB 71] and the proof of the BASS-YOUNG irregularity theorem [Ba 70] [BY 73]).

3.4.2. THE NON-RENAMEABILITY OF WEAK ABSTRACT RESOURCE-BOUND CLASSES

{*The Lieutenant of the Tower of Barad-dûr he was, and his name is remembered in no tale; for he himself had forgotten it, ...*
J.R.R. Tolkien. The Lord of the Rings}

To prove the negative result that no naming theorem for weak abstract resource-bound classes exists, we cannot simply show that the MEYER-McCREIGHT algorithm we sketched in section 1 fails to rename these classes, because we could imagine that some weird but still unknown construction could be correct. We must show that each construction which does part of the job, fails in some other part.

In this section we prove assertions of the following type:

Let σ be a measured transformation of programs. Then there exists an index so that φ_e and $\varphi_{\sigma(e)}$ are names of different classes.

The so-called honesty procedures σ which are constructed in the proof of the naming theorem are known to show irregularities of the following type:

If φ_i is a program with an extremely large run-time Φ_i then $\varphi_{\sigma(i)}$ becomes a function with unreasonable large values at infinitely many arguments.

This phenomenon was first described by L. BASS [Ba 70, BY 73] for the original honesty procedure described in [MCM 69]. More recently A. MEYER and R. MOLL [MMo 72] have shown that these irregularities occur for each measured transformation of programs which maps the set of functions with finite domain into itself, a property shared by all honesty procedures, since these functions are precisely the names of the class $C_\varepsilon = \mathcal{P}$. Their proof is a simple application of the recursion theorem. Their much stronger result reads:

FACT 3.4.5. Let σ be a measured transformation of programs which is a honesty procedure on $\mathcal{P}$. Then there exists for each total f and each total t an index e of a program computing f so that:

$$\liminf_{x\to\infty} (\#\{y \le x \mid \varphi_{\sigma(e)}(y) \le t(y)\})/x = 0.$$

These irregularities can be explained intuitively as follows. Consider a machine into which is fed the graph of the program φ_i and which computes the relation $\varphi_{\sigma(i)}(x) = y$. If $\mathcal{D}\varphi_i$ is finite, then the answer onto the question "is $\varphi_{\sigma(i)}(x) = y$?" will be almost always negative. Moreover, since σ is measured these answers must be produced in a finite amount of time regardless of the speed at which the graph of φ_i is introduced into the machine. Consequently, if we make φ_i so expensive that the machine is lured into believing that we are feeding it a function with finite domain it will have decided that $\varphi_{\sigma(i)}(x) \neq y$ for the small values of y before receiving a new input.

Essential in this argumentation and also in the formal proofs is the assumption that σ should work correctly both for total and partial φ_i. If this assumption is weakened in the sense that the honesty procedure may run astray on partial functions the irregularities may be suppressed; cf. [Mo 73].

Our negative results are based on a similar pathology which is formulated in the lemma below:

LEMMA 3.4.6. [Mirror lemma]. Let σ be a measured transformation of programs. Let t be a total function and let u be a partial function satisfying $u > t$. Then there exists a program φ_e so that

$$\forall x[\varphi_e(x) = 0 \ \underline{or}\ \varphi_e(x) = u(x)] \ \ \underline{and}\ \ \forall x[\varphi_e(x) = 0 \ \underline{iff}\ \varphi_{\sigma(e)}(x) > t(x)].$$

The program φ_e is "reflected in t" by σ; $0 = \varphi_e(x) \le t(x)$ implies $\varphi_{\sigma(e)}(x) > t(x)$ and $u(x) = \varphi_e(x) > t(x)$ implies $\varphi_{\sigma(e)}(x) \le t(x)$.

Moreover, since $\varphi_{\sigma(e)}(x)$ *le* $t(x)$ is a decidable relation, one can decide also whether $\varphi_e(x) = 0$. Consequently if u is the empty function then $\mathcal{D}\varphi_e$ is a recursive set.

PROOF. Let τ be the transformation defined by

$$\varphi_{\tau(i)}(x) \Leftarrow \underline{if}\ \varphi_{\sigma(i)}(x)\ \underline{gt}\ t(x)\ \underline{then}\ 0\ \underline{else}\ u(x)\ \underline{fi}.$$

Since σ is a measured transformation this is a well-defined transformation of programs. As a consequence from the recursion theorem there exists a fixed point φ_e such that $\varphi_e = \varphi_{\tau(e)}$; therefore

$$\varphi_e(x) = \underline{if}\ \varphi_{\sigma(e)}(x)\ \underline{gt}\ t(x)\ \underline{then}\ 0\ \underline{else}\ u(x)\ \underline{fi}.$$

This program φ_e has the properties claimed by the lemma. □

Our proofs are based upon the diagonalization constructions for honesty classes which we mentioned in chapter 2.2.

In particular we use the following lemmas:

LEMMA 3.4.7. There exist transformations κ and θ satisfying the following conditions: Let $\pi_1\mathcal{D}\varphi_i^2$ be infinite, then:

(i) $\varphi_{\kappa(i)}$ is a total 1-1 function with $\mathcal{R}\varphi_{\kappa(i)} \subseteq \mathcal{D}\varphi_i$

(ii) $\pi_1\varphi_{\kappa(i)}(x) < \pi_1\varphi_{\kappa(i)}(x+1)$

(iii) $\mathcal{D}\varphi_{\theta(i)}(x) = \pi_1\mathcal{R}\varphi_{\kappa(i)}$

(iv) $\pi_2\varphi_{\kappa(i)}(x) = \varphi_{\theta(i)}(\pi_1\varphi_{\kappa(i)}(x))$.

This lemma (which is included in lemma 2.2.6 in chapter 2.2) shows that from an index for a program φ_i^2 with $\pi_1\mathcal{D}\varphi_i^2$ infinite one gets uniformly both a program and a graph-enumerator for a partial function, with infinite domain whose graph is contained within $\mathcal{D}\varphi_i^2$.

LEMMA 3.4.8. Define the transformation δ by

$$\varphi_{\delta(i,j,k)}(x) \Leftarrow (\underline{int}\ z = \mu w[\varphi_k(w) = x];\ \underline{if}\ \Phi_{\pi_1 z}(x) \le \varphi_i(x)\ \underline{then}\ \underline{loop}\ \underline{else}\ \varphi_j(x)\ \underline{fi}).$$

Then for indices i, j, and k such that φ_k is a total 1-1 function enumerating a subset of $\mathcal{D}\varphi_i \cap \mathcal{D}\varphi_j$ we have $\varphi_{\delta(i,j,k)} \notin C^W_{\varphi_i}$, $\varphi_{\delta(i,j,k)} \sqsubseteq \varphi_j$, $\mathcal{D}\varphi_{\delta(i,j,k)} \subseteq \mathcal{R}\varphi_k$.

The proof of these lemmas is left to the reader.

Our first negative result is the non-existence of a honesty procedure on P for weak complexity classes.

<u>THEOREM 3.4.9</u>. Let σ be a measured transformation of programs. Then there exists an index e such that $C^W_{\varphi_e} \neq C^W_{\varphi_{\sigma(e)}}$.

<u>PROOF</u>. Let t be a total function such that $C^W_t \supsetneq C^W_{zero}$. Take u = ε. Application of the mirror lemma yields an index e such that

$$\varphi_e(x) = \underline{if}\ \varphi_{\sigma(e)}(x)\ \underline{gt}\ t(x)\ \underline{then}\ 0\ \underline{else}\ \underline{loop}\ \underline{fi}.$$

We claim that $C^W_{\varphi_e} \neq C^W_{\varphi_{\sigma(e)}}$. We consider three cases:

(1) $\mathcal{D}\varphi_e$ is finite. Now $C^W_{\varphi_e} = C^W_\varepsilon = P$, whereas $C^W_{\varphi_{\sigma(e)}} \subseteq C^W_t \neq P$.

(2) $\mathbb{N} \setminus \mathcal{D}\varphi_e$ is finite. In this case $C^W_{\varphi_e} = C^W_{zero}$, whereas $C^W_{zero} \subsetneq C^W_t \subseteq C^W_{\varphi_{\sigma(e)}}$. Again the classes are distinct.

(3) Both $\mathcal{D}\varphi_e$ and $\mathbb{N} \setminus \mathcal{D}\varphi_e$ are infinite. Since $\mathcal{D}\varphi_e$ is recursive, there exists a total 1-1 φ_k such that $\mathcal{R}\varphi_k = \mathbb{N} \setminus \mathcal{D}\varphi_e$. Let i be an index for t and let φ_j be some arbitrary total function. Consider the function f = $\varphi_{\delta(i,j,k)}$ as given by lemma 3.4.8. Since $\mathcal{D}f \subseteq \mathbb{N} \setminus \mathcal{D}\varphi_e$ one has trivially $f \in C^W_{\varphi_e}$; at the same time for $x \in \mathcal{D}f$ one has $t(x) \geq \varphi_{\sigma(e)}(x)$. Consequently $f \notin C^W_{\varphi_i} = C^W_t$ implies $f \notin C^W_{\varphi_{\sigma(e)}}$. This proves $C^W_{\varphi_e} \neq C^W_{\varphi_{\sigma(e)}}$. □

Our second negative result treats the case of honesty classes. The proof is similar but the diagonalization is more complicated.

<u>THEOREM 3.4.10</u>. Let σ be a measured transformation of programs from P^2 into P^2. Then there exists an index e so that $H_{\varphi^2_e} \neq H_{\varphi^2_{\sigma(e)}}$.

<u>PROOF</u>. Using the two dimensional analogue of the mirror lemma with u the empty function $\varepsilon^2 = \lambda x,y[\underline{loop}]$ and for t a total function R so that $H_R \neq H_{zero2}$, we find an index e for which we have:

$$\varphi^2_e(x,y) = \underline{if}\ \varphi^2_{\sigma(e)}(x,y)\ \underline{gt}\ R(x,y)\ \underline{then}\ 0\ \underline{else}\ \underline{loop}\ \underline{fi}.$$

Let $S = \varphi^2_e$, $S' = \varphi^2_{\sigma(e)}$. We consider two cases:

(1) $\overset{\infty}{\forall} x \forall y[S(x,y) = 0]$. This leads to $H_S = H_{zero2} \subsetneq H_R \subseteq H_{S'}$.

(2) $\overset{\infty}{\exists} x \exists y[S(x,y) = \infty]$.

Since $\mathcal{D}S$ is recursive we can enumerate $\mathbb{N}^2 \setminus \mathcal{D}S$. Consequently the function T defined by:

$$T = \lambda x,y[\underline{if} \ \underline{¢} \ \langle x,y\rangle \in \mathcal{D}S \ \underline{¢} \ \underline{then} \ \underline{loop} \ \underline{else} \ R(x,y) \ \underline{fi}]$$

is a partial recursive function. By application of lemma 3.4.7 we find a function φ_k enumerating the graph of a partial function φ_j whose graph is contained within $\mathcal{D}T$. Moreover if $\varphi_{k'} = \lambda x[\pi_1 \varphi_k(x)]$ then $\varphi_{k'}$ is total and increasing.

Let $\varphi_i = T \square \varphi_j$. We consider the function $f = \varphi_{\delta(i,j,k')}$. Since $f \sqsubseteq \varphi_j$ and the graph of φ_j is contained in $\mathbb{N}^2 \setminus \mathcal{D}S$ we know that $f \in H_S$. At the same time by construction $f \notin C^W_{\varphi_i}$. This implies that for each index n for f we have

$$\overset{\infty}{\exists} x[\varphi_i(x) = R(x,f(x)) < \Phi_n(x) < \infty]$$

and since for these arguments x we have also $S'(x,f(x)) \leq R(x,f(x))$ this proves $f \notin H_S$.

This shows that the classes H_S and $H_{S'}$ are distinct. $\square$

The preceding results are based on the use of partial funcitons. It is a reasonable question to ask whether partial functions are essential. More particularly one may ask the following questions:

(1) Is it essential that σ is a honesty procedure on P? *)

(2) Do the negative results remain valid if only the total functions in the classes are considered? *)

(3) Does there exist a non-uniform renaming procedure, i.e. does there exist a measured set containing names for all honesty classes?

The first question can be settled completely. We can "uniformize" our proofs in such way that the negative results extend to total honesty procedures as well.

*) Questions (1) and (2) were suggested by A. MEYER.

The second question makes no sense for weak complexity classes since $C_t^W \cap R = C_t \cap R$. Consequently, the naming theorem itself yields a positive result for the classes $C_t^W \cap R$. Although we have no answer to this problem for general honesty classes we can prove that for the modified honesty classes the negative result remains valid: the classes $H_r^\wedge$ cannot be renamed uniformly by a measured set of names.

The third problem is still unsolved.

The results on total honesty procedures are based on a uniformized version of the mirror lemma:

EXERCISE 3.4.11. Let σ be a measured transformation of programs; let t be total. Then there exists a transformation ρ such that

$$\varphi_{\rho(j)}(x) = \underline{if}\ \varphi_{\sigma(\rho(j))}(x)\ \underline{le}\ t(x)\ \underline{then}\ t(x) + \varphi_j(x) + 1\ \underline{else}\ 0\ \underline{fi}$$

Moreover, the relation $\varphi_{\rho(j)}(x) \neq 0$ is recursive in j and x.

THEOREM 3.4.12. Let σ be a measured transformation of programs. Then there exists an index e of a total function such that $C^W_{\varphi_e} \neq C^W_{\varphi_{\sigma(e)}}$.

PROOF. Take a total function t such that $C^W_{zero} \neq C^W_t$. Let ρ be the transformation from lemma 3.4.11. Since $\varphi_{\rho(j)}(x) \neq 0$ is decidable the function k defined by:

$$k = \lambda j,x[\underline{¢}\ \#\{z \leq x \mid \varphi_{\rho(j)} \neq 0\}\ \underline{¢}]$$

is a total recursive function.

We define a transformation of programs τ by:

$$\varphi_{\tau(j)}(x) \Leftarrow \underline{if}\ \varphi_{\rho(j)}(x) = 0\ \underline{then}\ \underline{loop}$$
$$\underline{elif}\ \Phi_{\pi_1 k(j,x)}(x) \leq t(x)\ \underline{then}\ \varphi_{\pi_1 k(i,x)}(x) + 1\ \underline{else}\ 0\ \underline{fi}$$

Since $\mathcal{D}\varphi_{\tau(j)} = \{x \mid \varphi_{\rho(j)}(x) \neq 0\}$ is recursive the following function m is total:

$$m = \lambda x[max\{\underline{if}\ \varphi_{\rho(j)}(x) = 0\ \underline{then}\ 0\ \underline{else}\ \Phi_{\tau(j)}(x)\ \underline{fi} \mid j \leq x\}]$$

The definition of τ ensures that whenever $\mathcal{D}\varphi_{\tau(j)}$ is infinite $\varphi_{\tau(j)} \notin C_t^W$. By the definition of m, $\varphi_{\tau(j)} \in C_m^W$ for each j. Let j_0 be an index of m and let

$e = \rho(j_0)$. If we take $u = \varphi_e$, $u' = \varphi_{\sigma(e)}$ then we have

$$u(x) = \underline{\mathit{if}}\ u'(x)\ \underline{\mathit{le}}\ t(x)\ \underline{\mathit{then}}\ t(x) + m(x) + 1\ \underline{\mathit{else}}\ 0\ \underline{\mathit{fi}}$$

and

$$\mathcal{D}\varphi_{\tau(j_0)} = \{x \mid u'(x)\ \underline{\mathit{le}}\ t(x)\}.$$

Since m is total, so is u. If $u(x) = 0$ almost everywhere, then $u'(x) > t(x)$ almost everywhere and then

$$C^W_u = C^W_{zero} \subsetneq C^W_t \subseteq C^W_{u'}.$$

If $u(x) \neq 0$ infinitely often then $\varphi_{\tau(j_0)} \notin C^W_t$ and hence $\varphi_{\tau(j_0)} \notin C^W_{u'}$, whereas $\varphi_{\tau(j_0)} \in C^W_u$ by construction.

This proves that $C^W_{\varphi_e} \neq C^W_{\varphi_{\sigma(e)}}$. □

THEOREM 3.4.13. Let σ be a measured transformation of programs. Then there exists an index e of a total function such that $H_{\varphi^2_e} = H_{\varphi^2_{\sigma(e)}}$.

PROOF. The proof combines ideas from the preceding proofs.

Let $H_{zero2} \subsetneq H_R$, R total. By 3.4.11 there exists a transformation ρ satisfying:

$$\varphi^2_{\rho(j)}(x,y) = \underline{\mathit{if}}\ \varphi^2_{\sigma(\rho(j))}(x,y)\ \underline{\mathit{le}}\ R(x,y)\ \underline{\mathit{then}}\ R(x,y) + \varphi^2_j(x,y) + 1\ \underline{\mathit{else}}\ 0\ \underline{\mathit{fi}}.$$

Now for total φ^2_j, $\varphi^2_{\rho(j)}$ is total and moreover the relation $\varphi^2_{\rho(j)}(x,y) = 0$ is recursive.

Using lemma 3.4.7 we can find transformations κ and θ such that $\varphi_{\kappa(j)}$ enumerates the graph of a partial function $\varphi_{\theta(j)}$, which graph is entirely contained in the set $L_j = \{\langle x,y\rangle \mid \varphi^2_{\rho(j)}(x,y) \neq 0\}$. Given that $\pi_1 L_j$ is infinite $\varphi_{\kappa(j)}$ will be a total increasing function. In this case we derive from the recursiveness of L_j that it is decidable whether $\langle x,y\rangle \in \mathcal{R}\varphi_{\kappa(j)}$ (or equivalently whether $\varphi_{\theta(j)}(x) = y$); consequently both κ and θ are measured transformations.

We define the transformations η and δ' by:

$$\varphi_{\eta(j)}(x) \Leftarrow (R\square\varphi_{\theta(j)})(x) \quad \text{and} \quad \delta' = \lambda j[\delta(\eta(j),\theta(j),\kappa(j))].$$

where δ is defined as in lemma 3.4.8. Consequently $\varphi_{\delta'(j)} \sqsubseteq \varphi_{\theta(j)}$ and $\varphi_{\delta'(j)} \notin H_R$ whenever $\varphi_{\kappa(j)}$ is total.

The following equivalence can be derived:

$$\varphi_{\delta'(j)}(x) = y \ \underline{\mathit{iff}}\ \exists w \leq x [\varphi_{\kappa(j)}(w)\ \underline{\mathit{eq}}\ \langle x,y \rangle\ \underline{\mathit{and}}\ \Phi_{\pi_1 w}(x) > R(x,y)].$$

This equivalence shows that δ' is a measured transformation as well. By the equivalence of measured and honest sets (th. 2.3.8) there exists a total function S such that $(\varphi_{\delta(j)})_j \subseteq H_S$. Let j_0 be an index for S, and let $e = \rho(j_0)$. Writing T resp. T' for φ^2_e resp. $\varphi^2_{\sigma(e)}$ we conclude that T is total. Moreover the case that $\varphi_{\kappa(j_0)}$ is a finite function leads to the inequality $H_T = H_{\mathit{zero2}} \subsetneq H_R \subseteq H_{T'}$, whereas in the alternative case the construction of δ' implies $\varphi_{\delta'(j_0)} \in H_T \setminus H_{T'}$. Hence e satisfies the condition of the theorem. $\square$

These theorems yield a satisfying answer to question (1): there exist no honesty procedures on $\mathcal{R}$ for weak complexity classes or honesty classes. We next consider the second question.

All our diagonalization procedures used upto this point constructed expensive functions by deleting values from some given function. This way we produce partial diagonalization functions. If we want a total function we must provide also finite "escape values". Our aim was to define $\varphi_{\delta(j)}$ in such a way that $\varphi_{\delta(j)}(x) = y$ only when $\varphi_{\rho(j)}(x,y)$ was large (and consequently $\varphi_{\sigma(\rho(j))}(x,y)$ was small). Therefore we need for each x at least two values of y such that $\varphi_{\rho(j)}(x,y) \neq 0$.

Up to now we are unable to solve this difficulty for ordinary honesty classes. Using modified honesty classes the problem however disappears; if $r(x) \neq 0$ for infinitely many x and if $R(x,y) = r(\underline{\mathit{max}}(x,y))$ then $R(x,y) \neq 0$ whenever $x \leq y$ and $r(y) \neq 0$.

<u>THEOREM 3.4.14</u>. For each measured transformation σ there exists an index e (of a total function) such that $H^{\wedge}_{\varphi_e} \cap \mathcal{R} \neq H^{\wedge}_{\varphi_{\sigma(e)}} \cap \mathcal{R}$.

<u>PROOF</u>. We describe the diagonalization procedure for tha case that φ_e is obtained by application of the mirror lemma using $u = \varepsilon$, leaving the modification yielding a total φ_e to the reader.

Suppose $H^{\wedge}_{\mathit{zero}} \neq H^{\wedge}_t$ and let φ_e satisfy the relation

$$\varphi_e(x) = \underline{\mathit{if}}\ \varphi_{\sigma(e)}(x)\ \underline{\mathit{le}}\ t(x)\ \underline{\mathit{then}}\ \underline{\mathit{loop}}\ \underline{\mathit{else}}\ 0\ \underline{\mathit{fi}}.$$

Let $R(x,y) = \varphi_e(\underline{\mathit{max}}(x,y))$, $R'(x,y) = \varphi_{\sigma(e)}(\underline{\mathit{max}}(x,y))$. The case that φ_e is cofinite leads to the inclusions $H^{\wedge}_{\varphi_e} = H^{\wedge}_{\mathit{zero}} \subsetneqq H^{\wedge}_t \subseteq H^{\wedge}_{\varphi_{\sigma(e)}}$.

Otherwise there exist infinitely many x such that $\varphi_e(x) \neq 0$; these x form a recursive set. Let y_1 and y_2 be defined by

$$y_1 = \lambda x[\mu z[z \geq x\ \underline{\mathit{and}}\ \varphi_e(z) \neq 0]]$$

and

$$y_2 = \lambda x[\mu z[z > y_1(x)\ \underline{\mathit{and}}\ \varphi_e(z) \neq 0]].$$

Then y_1 and y_2 are total and for each x, $R(x,y_1(x)) = R(x,y_2(x)) = \infty$. Define f by:

$$f = \lambda x[\underline{\mathit{if}}\ \varphi_{\pi_1 x}(x) \leq t(y_1(x))\ \underline{\mathit{and}}\ \varphi_{\pi_1 x}(x) = y_1(x)$$
$$\underline{\mathit{then}}\ y_2(x)\ \underline{\mathit{else}}\ y_1(x)\ \underline{\mathit{fi}}].$$

Then as before $f \notin H_{R'}$, and $f \in H_R$; moreover, f is total. This proves that $H^{\wedge}_{\varphi_e} \cap R \neq H^{\wedge}_{\varphi_{\sigma(e)}} \cap R$. $\square$

The third question on the existence of non-uniform honesty procedures remains unsolved. An idea which is used as a short cut in the proof of theorem 3.4.13 suggests a way to approach the solution. Let $(R_i)_i$ be a measured set of names of honesty classes. Then there exists a transformation δ such that $\varphi_{\delta(i)} \notin H_{R_i}$ (unless $H_{R_i} = P$). It is not very difficult to construct such a transformation.

However, if we are able to define δ in such a way that δ becomes a measured transformation of programs then we are done. In this case $(\varphi_{\delta(i)})_i$ is a measured set which is contained in H_R for some total R. This honesty class H_R clearly has no name in the sequence $(R_i)_i$. We therefore specialize our third problem to:

<u>UNSOLVED PROBLEM 3.4.15</u>. Let $(R_i)_i$ be a measured sequence of functions in two variables. Does there exist a measured transformation δ such that, for $H_{R_i} \neq P$, $\varphi_{\delta(i)} \notin H_{R_i}$?

One may weaken the condition by asking $\varphi_{\delta(i)} \notin H_{R_i}$ only for total functions R_i. A positive answer to this question should show the non-existence of a measured set containing total names for all honesty classes named by a total function.

3.4.3. ALTERNATIVE MEYER-McCREIGHT ALGORITHMS FOR STRONG CLASSES

This section contains two modifications of the MEYER-McCREIGHT algorithm. The first modification is designed to produce two names for a single class which have disjoint domains. The second modification proves a generalized union theorem for strong classes. Finally we discuss the problem whether this generalization allows us to name classes which are not already named by the original union theorem.

A result of R. MOLL [MMo 72] states that for each strong class $F(t)$ a name t' can be constructed with a domain having asymptotical density zero. His proof uses a modified MMC algorithm which manipulates the priorities in such a way that t'(x) is seldom defined. It is not very amazing that there are arguments enough in the complement of $\mathcal{D}t'$ to define a second name t" for $F(t)$. In order to prevent that t' uses all the "good" arguments for its own we define t' and t" simultaneously.

We shall see afterwards that we even may assume that $\mathcal{D}t'$ and $\mathcal{D}t''$ are subsets of $\mathcal{D}t$ - this however for the price of loosing the property of measuredness. We indicate further how one may prove a generalization where a single name is splitted into an infinite number of names with disjoint domains, each naming the original class.

<u>THEOREM 3.4.15</u>. There exists a measured transformation of programs from P into $P \times P$ mapping each index i onto a pair of indices $<\sigma(i),\tau(i)>$ so that

(i) $\forall i[F(\varphi_i) = F(\varphi_{\sigma(i)}) = F(\varphi_{\tau(i)})]$

(ii) $\forall i[\mathcal{D}\varphi_{\sigma(i)} \cup \mathcal{D}\varphi_{\tau(i)} = \emptyset]$.

By measuredness of this transformation we mean that both sequences $(\varphi_{\sigma(i)})_i$ and $(\varphi_{\tau(i)})_i$ are measured sets.

<u>PROOF</u>. We construct a modified MMC-algorithm which enumerates $\varphi_{\sigma(i)}$ and $\varphi_{\tau(i)}$ simultaneously, using an enumerator for φ_i as input.

Let $t = \varphi_i$, $t' = \varphi_{\tau(i)}$ and $t'' = \varphi_{\tau(i)}$.

The indices j are manipulated as before. If a violation by j against t is discovered j is placed on the black list. To be transferred back to the white list a violation by j against both t' and t" must be created, and these violations will have to occur at two distinct arguments.

To represent the intermediate situation where a violation by j against t is halfway being punished we introduce two more colours red and blue. The procedure searchtime works as before, but identifies all non-white indices as black ones. If it yields a positive solution the index used at x choosen whether the value found by searchtime is going to be a value for t' or for t". This way any index which is used twice becomes white again. □

The algorithm below is a modification of the MMC-algorithm by R. MOLL; next is a subroutine which computes the next free priority number, as suggested by the program:

```
        proc next = int: p +:= 1;
```

which operates on a global variable p which is initialized at zero. *defined* is a boolean operator testing whether some variable is initialized or not. The algorithm is a stagewise algorithm; below we describe *stage* n.

stage n:

```
1. Introduce index n: p[n] := next; b[n] := white;
2. Computation of t: x := $\pi_1 n$; y := $\pi_2 n$;
     if $\Phi_i(x) \neq y$ then goto 4 else z := t[x] := $\varphi_i(x)$ fi
3. Discriminator against t:
     for j $\leq$ n do if b[j] = white and A(j,x,z) $\neq$ true
                    then (b[j] := black; p[j] := next) fi od;
4. if defined t'[x] or defined t"[x] then goto stage n+1 fi;
5. searchtime(x,y,y,prior); if failure then goto stage n+1 fi;
   ¢ if success occurred in searchtime cand is the index used at x and val
     is the value found; we now choose which bound is going to collect this
     value ¢
6. if b[cand] = black or b[cand] = blue
     then (choice := 1; t'[x] := y)
     else (choice := 2; t"[x] := y) fi;
```

7. Discriminator against t' and t":

```
for j ≤ n do if b[j] ≠ white and A(j,x,z) ≠ true then
    if b[j] = black then b[j] := if choice = 1 then red else blue fi
    elif b[j] = red and choice = 2 then (b[j] := white; p[j] := next)
    elif b[j] = blue and choice = 1 then (b[j] := white; p[j] := next)
    fi
    fi od;
```

8. goto stage n+1;

It is clear from the description that t' and t" have disjoint domains. To prove that t, t' and t" all name the same class we consider the possible behaviours of p[j] and b[j].

Case 1. p[j] is unstable.

Moving back and forwards j violates t, t' and t" all at infinitely many arguments. Hence $j \notin F(t)$, $j \notin F(t')$ and $j \notin F(t")$.

Case 2. p[j] is white-stable.

In this case $j \in F(t)$ is trivial like before. To show that $j \in F(t')$ and $j \in F(t")$ we must prove that the priority used at x is almost everywhere larger than lim p[j]. In the original MMC-algorithm this is clear since a priority used at x is a mortal priority, but this assertion is no longer true for our modification. However, an index which is used twice liberates himself by choosing the right bound to violate against, and with this liberation its priority dies. Hence each priority is used at most twice. (Since we have implemented an R. MOLL MMC-algorithm there is no historical backlog between the priority status used by searchtime and the actual priority status.)

Therefore the assertion that the priority used at x is almost everywhere larger then lim p[j] remains valid. Consequently $j \in F(t')$ and $j \in F(t")$.

Case 3. p[j] is black-stable.

Now $j \in F(t')$ and $j \in F(t")$ are trivial. Moreover by assertion 3.4.4 the run-time α_j is bounded almost everywhere by the maximum of the run-runtimes of a finite set of white-stable indices which run-times are bounded in their turn by t almost everywhere. Hence $j \in F(t)$.

Case 4. p[j] is blue-stable.

This situation arises only if the index j, after having reached its stable priority value k on the black list is used at most once, for if it is used at a moment where its colour is already blue then we take choice = 1 and b[j] becomes again white and the blue priority is deleted. Furthermore no violation against t' is created after b[j] has become blue for the last time. Hence $j \in F(t')$.

Because searchtime makes no difference between black and blue indices $j \in F(t)$ can be derived from assertion 3.4.4.

To prove that $j \in F(t'')$ we need a stronger assertion about the working of searchtime than assertion 3.4.4. We use that val is not only larger than all white run-times left of candidate, but larger than the non-white run-times left of candidate also, since otherwise such a run-time should have yielded a solution. Since the priority used at x is almost everywhere larger than lim p[j] one derives $\alpha_j(x) \leq t'(x)$ and $\alpha_j(x) \leq t''(x)$ almost everywhere. This shows that $j \in F(t'')$.

Case 5. p[j] is red-stable.

This case is analogous to case 4.

In each of the five case we have $j \in F(t)$ *iff* $j \in F(t')$ *iff* $j \in F(t'')$. This proves that the three classes are equal.

Finally the measuredness follows straightforward from the program. □

Using Theorem 3.4.15 we may split a single name into an infinite sequence of names by defining $t_k = \varphi_{\rho(i,k)} = \varphi_{\tau(\sigma^{(k)}(i))}$. To derive that these names have disjoint domains we need an inclusion of the type $\mathcal{D}\varphi_{\sigma(i)} \subseteq \mathcal{D}\varphi_i$ and $\mathcal{D}\varphi_{\tau(i)} \subseteq \mathcal{D}\varphi_i$, but these inclusions are not derivable from the above algorithm.

In order to enforce such inclusions we modify the program so that calls of searchtime are suppressed until t[x] is enumerated, but by doing so the condition of measuredness is lost. Then we may also withdraw all precautions against unwanted revisions.

PROPOSITION 3.4.16. There exists a transformation of programs from P into $P \times P$ which maps each index i onto a pair of indices $<\sigma(i),\tau(i)>$ so that:

(i) $\forall i[F(\varphi_i) = F(\varphi_{\sigma(i)}) = F(\varphi_{\tau(i)})$

(ii) $\forall i[\mathcal{D}\varphi_{\sigma(i)} \cap \mathcal{D}\varphi_{\tau(i)} = \emptyset, \mathcal{D}\varphi_{\sigma(i)} \cup \mathcal{D}\varphi_{\tau(i)} \subset \mathcal{D}\varphi_i]$.

PROOF. Replace in the above algorithm the instructions after 5 by

```
5': if defined t[x] then searchtime (x,0,y,prior) else goto stage n+1 fi;
    if failure then goto stage n+1 fi;
```

The correctness proof above must be relativized to the domain of t; for cases 1 and 2 this makes no difference, whereas for cases 3, 4 and 5 the following relativization of assertion 3.4.4 may be used:

FACT 3.4.17. If an index j has a stable non-white priority, then its run-time α_j is bounded for almost all x in the domain of t by the maximum of the run-times of (white) stable indices with higher priority.

The remainder of the proof is left to the reader. □

COROLLARY 3.4.18. For each t there exists an infinite sequence of names t_i for $F(t)$ with disjoint domains.

A nicer way to prove this corollary is the construction of an MMC-algorithm which enumerates infinitely many names for $F(t)$ instead of two. In this algorithm a black index j on its way back to the white list must violate the first p[j] names being constructed (at p[j] distinct arguments) before it becomes white again. We need an infinite collection of intermediate colours. The choice which name is going to collect a fresh value at x found by searchtime is made by the index used at x, which selects the lowest bound which he must violate and which he has not yet violated. The details are left to the reader.

In contrast to the sequence found in 3.4.18 the sequence constructed by an algorithm as suggested above will be a measured set.

Our next subject is the generalized union theorem. This theorem states that the union of an increasing sequence of strong classes of indices is again a strong class with a name which is computable from programs for the sequence of names. In contrast to the situation of the classical union theorem, which was discussed in chapter 3.3, this union theorem does not hold for classes of functions. The reason is that many indices with far distinct run-times may be mapped by *fun* onto the same functions. Consequently we cannot derive from the fact that the sequence of classes of functions is an increasing sequence that these classes correspond to an increasing sequence of classes of programs. A counterexample is given below.

EXAMPLE 3.4.19. [The union of an increasing sequence of complexity classes is not in general again a complexity class].

Let $\mathbb{N}$ be written as the disjoint union of an infinite sequence of infinite recursive sets A_i. $\mathbb{N} = \bigcup_i A_i$, $A_i \cap A_j = \emptyset$ for $i \neq j$.

Let

$$g_i(x) = \underline{\textit{if}}\ x \in A_i\ \underline{\textit{then}}\ 1\ \underline{\textit{else}}\ 0\ \underline{\textit{fi}}.$$

We construct a complexity measure which contains among others the following programs ψ_{ij} with corresponding run-times Ψ_{ij}:

$$\begin{array}{lll}
\psi_{ij} = \lambda x[j], & \Psi_{ij} = g_i & \text{for } 0 < j \leq i, \\
\psi_{ij} = \lambda x[x], & \Psi_{ij} = g_i + g_j & \text{for } 0 < i < j, \\
\psi_{ij} = \lambda x[\infty], & \Psi_{ij} = \lambda x[\infty] & \text{for } 0 = i < j, \\
\psi_{ij} = \lambda x[i], & \Psi_{ij} = \lambda x[x] & \text{for } 0 = j < i, \\
\psi_{ij} = \lambda x[x], & \Psi_{ij} = \lambda x[x] & \text{for } 0 = i = j.
\end{array}$$

All other programs have run-times larger than $\lambda x[x+1]$ at all arguments. We claim that $C_{g_i} \subsetneq C_{g_{i+1}}$, whereas $C = \bigcup_i C_{g_i}$ is not a complexity class.

PROOF. From the definitions one concludes that F_{g_i} consists of the programs ψ_{ij} with $0 < j \leq i$. Therefore C_{g_i} consists of the constant functions with values $1,2,\ldots,i$. This shows that $C_{g_i} \subsetneq C_{g_{i+1}}$. Note however, that $F_{g_i} \cap F_{g_{i+1}} = \emptyset$.

Now suppose that $C_t = C = \bigcup_i C_{g_i}$. If $t(x) \geq x$ almost everywhere then F_t contains the program ψ_{00}, and C_t contains therefore the non-constant function $\lambda x[x]$ which is not contained in C. Hence $t(x) \leq x$ for infinitely many x.

We conclude that F_t consists of programs from the sequence $(\psi_{ij})_{ij}$. Let A be the set of arguments x with $t(x) > 0$.

If A contains (modulo a finite set) none of the sets A_j then F_t is empty and consequently $C_t = \emptyset \neq C$. If A contains (modulo a finite set) precisely one of the sets A_j (lets say A_i) then $F_t = F_{g_i}$ and again C_t and C are distinct.

If A however contains (modulo a finite set) two of the sets A_j (say A_i and A_l with $i < l$) then F_t contains among others the program ψ_{il} which computes the non-constant function $\lambda x[x]$. Therefore in this case the classes C_t and C are also different.

This completes our proof. □

THEOREM 3.4.20. Let A be an acceptance relation and let $(t_i)_i$ be a sequence of (partial) functions, so that $F_S^A(t_i) \subset F_S^A(t_{i+1})$. Then there exists a function tinf so that $\bigcup_i F_S^A(t_i) = F_S^A(\text{tinf})$.

If *prog* is defined for A and if *prog* i = φ_i then the theorem holds also for the classes $G_S^A(t_i)$.

PROOF. The function tinf is computed by a modified MMC-algorithm. The items manipulated are not the indices j themselves, but pairs consisting of an index j and a bound-index i of a bound t_i which j want to respect. The item <j,i> is placed on the black list if it is discovered that j violates t_i and <j,i> is placed back on the white list after creation of a violation by j against tinf.

It may happen that a pair <j,i> becomes white-stable whereas the same index, combined with a "wrong" bound i' violates its bound infinitely often. In this situation the pair <j,i'> is doomed to become black-stable.

We assume that the functions t_i are computed by the program φ_t^2 The algorithm is a stagewise algorithm; we describe *stage* n.

```
stage n:

1. Introduce item n: p[n] := next; b[n] := true;

2. Computation of the $t_i$: discriminator: x := $\pi_1 n$; y := $\pi_2 n$;
      for k ≤ x do if $\Phi_t^2(k,x)$ = y then
          begin z := $\varphi_t^2(k,x)$;
              for m ≤ n do if b[m] = true and $\pi_2 m$ = k and
                                 $A(\pi_1 m, x, z) \neq$ true
                           then (p[m] := next; b[m] := false) fi od
          end             fi od;

3. if defined tinf[x] then goto stage n+1 fi;

4. searchtime (x,y,y,prior); ¢ the run-time of an item as considered by
                               searchtime is the run-time of its index ¢
   if failure then goto stage n+1 fi;

5. Discriminator against tinf: tinf[x] := y;
      for m ≤ n do if b[m] = false and $A(\pi_1 m, x, y) \neq$ true
                   then (b[m] := true; p[m] := next) fi od;

6. goto stage n+1;
```

To prove that $F(\mathrm{tinf}) = \bigcup_i F(t_i)$ we consider the possible behaviours of the items. Let $j = \pi_1 n$, and $i = \pi_2 n$.

Case 1. p[n] is unstable.

From the unstability of p[n] one derives that j violates both t_i and tinf at infinitely many arguments. Hence $j \notin F(t_i)$ and $j \notin F(\mathrm{tinf})$.

Case 2. p[n] is white-stable.

The white stability of n implies $j \in F(t_i)$. Moreover by the usual argument that the priority used at x is almost everywhere larger than lim p[n] one derives that $j \in F(\mathrm{tinf})$ also.

Case 3. p[n] is black-stable.

In this case $j \in F(\mathrm{tinf})$ by the black stability of n. Assertion 3.4.4 can be applied and yields that the run-time of item n i.e. α_j is bounded almost everywhere by the run-times of finitely many white-stable items. Let these finitely many white stable items bounding the run-times of item n be denoted by $n_1,\ldots,n_k$. Let $j_m = \pi_1 n_m$ and let $i_m = \pi_2 n_m$.

Since the n_m are white-stable we have $j_m \in F(t_{i_m})$ for $1 \le m \le k$. Now the condition of monotonicity of classes is applied. Let i_0 be the maximum of $i_1,\ldots,i_k$. Then $j_m \in F(t_{i_0})$ for $1 \le m \le k$ and consequently:

$$\overset{\infty}{\forall} x[\alpha_{j_m}(x) \le t_{i_0}(x)].$$

This implies

$$\alpha_j(x) \le \underline{max}(\alpha_{j_1}(x),\ldots,\alpha_{j_k}(x)) \le t_{i_0}(x) \quad \text{for almost all } x$$

Consequently $j \in F(t_{i_0}) \subset \bigcup_i F(t_i)$.

If $j \in F(\mathrm{tinf})$ and $i \in \mathbb{N}$ then we have for item $n = \langle j,i\rangle$ either case 2 or case 3 and consequently $j \in \bigcup_i F(t_i)$. Conversely if $j \in F(t_i)$ then we have for $n = \langle j,i\rangle$ also either case 2 or case 3 and therefore $j \in F(\mathrm{tinf})$.

This completes the proof. □

Although 3.4.20 as a theorem looks stronger than the union theorem in chapter 3.2 for strong classes, it is not clear whether it is an essential generalization or not. It could be that each sequence of strong classes

$(F(t_i))_i$ satisfying $F(t_i) \subset F(t_{i+1})$ can be renamed by a sequence of names which is monotonic in the sense of chapter 3.3.

Upto now this is an open question. We can show by an example that it is not generally possible to rename a class $F(t) \supset F(u)$ by a name t' with $t' \geq u$. If t and u both are total we can rename u by $u' = \underline{min}(t,u)$ but this construction does not work for infinite sequences.

EXAMPLE 3.4.21. [$F(t) \supset F(u)$ cannot generally be renamed by a name $t' \geq u$]. Let $(\gamma_i)_i$ be the measured set defined by:

$$\gamma_0 = \lambda x[\underline{if}\ \underline{even}\ x\ \underline{then}\ 2\ \underline{else}\ 1\ \underline{fi}]$$
$$\gamma_1 = \lambda x[\underline{if}\ \underline{even}\ x\ \underline{then}\ 3\ \underline{else}\ 0\ \underline{fi}]$$
$$\gamma_k = \lambda x[4] \qquad for\ k \geq 2.$$

If Γ is the corresponding acceptance relation, and if t and u are defined by:

$$u(x) = 1; \quad t = \lambda x[\underline{if}\ \underline{even}\ x\ \underline{then}\ 3\ \underline{else}\ 0\ \underline{fi}],$$

then $F^\Gamma(u) = \emptyset$ and $F^\Gamma(t) = \{1\}$, hence $F^\Gamma(u) \subset F^\Gamma(t)$.

Suppose that $F^\Gamma(t) = F^\Gamma(t')$ with $t' \geq u$. Since $1 \in F^\Gamma(t)$ we have $t'(x) \geq 3$ for almost all even x, whereas $t' \geq u$ implies $t'(x) \geq 1$ for all odd x. But now $0 \in F^\Gamma(t')$ also which is a contradiction. □

The transformation of sequences $(t_i)_i \Rightarrow (u_i)_i$ defined below is an infinite version of the operation of taking the minimum to rename the smallest class.

$$u_i(x) = \underline{if}\ x < i\ \underline{then}\ \max_{j \leq x}\{t_j(x)\}\ \underline{else}\ \min_{i \leq k \leq x}\{t_j(x)\}\ \underline{fi}.$$

This transformation maps a sequence of total functions onto a non-decreasing sequence of total functions. It does not yield however

$$\forall i[F(t_i) \subset F(t_{i+1})]\ \underline{imp}\ \forall i[F(t_i) = F(u_i)].$$

The reason for this failure is shown in the example below; it is not possible to prevent the addition of infinitely many meaningless restrictions forming together an unwanted serious restriction.

EXAMPLE 3.4.22. Let α_0 be a total run-time and suppose $F(zero) = \emptyset$. Let

$$t_i(x) = \underline{if}\ x \neq i\ \underline{then}\ \alpha_0(x)\ \underline{else}\ 0\ \underline{fi}.$$

Now the transformed sequence u_i becomes:

$$u_i(x) = \underline{if}\ x < i\ \underline{then}\ \alpha_0(x)\ \underline{else}\ 0\ \underline{fi}.$$

Hence $0 \in F(t_i) = F(t_j)$ whereas $F(u_i) = F(zero) = \emptyset$.

An approach which looks more fruitful is to rename the complete sequence of classes by an increasing sequence of names, not necessarily preserving the individual classes but preserving the union of the classes. Such a construction is possible for total t_i.

PROPOSITION 3.4.23. Let $(t_i)_i$ be a sequence of total functions so that $F(t_i) \subset F(t_{i+1})$. Then there exists a sequence of total functions $(u_i)_i$ so that:

(i) $\forall i \forall x[u_{i+1}(x) \geq u_i(x)]$,
(ii) $\bigcup_i F(u_i) = \bigcup_i F(t_i)$.

PROOF. Define

$$u_k(x) = max\{\alpha_i(x) \mid i \leq k\ \underline{and}$$
$$\exists n \leq k[\alpha_i(x)\ \underline{le}\ t_n(x)\ \underline{and}\ \forall y[(k \leq y\ \underline{and}\ y < x)\ \underline{imp}\ \alpha_i(y)\ \underline{le}\ t_n(y)]]\}$$

For $x \leq k$ this definition simply yields

$$u_k(x) = max\{\alpha_i(x) \mid i \leq k\ \underline{and}\ \alpha_i(x)\ \underline{le}\ \max_{j\leq k}\{t_j(x)\}\}.$$

For $x > k$ the extra condition that a fixed t_n bounds α_i over the complete interval $[k,x]$ starts its influence.

First suppose that $j \in F(t_n)$. Then there exists an argument b so that $x \geq b$ implies $\alpha_j(x) \leq t_n(x)$. If $k \geq \underline{max}(j,n,b)$ then $u_k(x) \geq \alpha_j(x)$ for $x \geq k$, since $j,n \leq k$ and $\alpha_j(x)\ \underline{le}\ t_n(x)$ for $x \in [k,x]$. Consequently $j \in F(u_k)$.

Conversely, suppose that $j \in F(u_k)$ then α_j is bounded almost everywhere by the maximum of the run-times of a finite set of indices, which are all contained within $\bigcup_{i\leq k} F(t_i) = F(t_k)$; for if α_m is bounded over arbitrary long intervals $[k,x]$ by a function t_i with $i \leq k$ then α_m is bounded also by a fixed t_i with $i \leq k$ over the infinite interval $[k,\infty)$. But now one concludes that $j \in F(t_k)$ by the same argumentation we used before.

This shows that

$$\bigcup_i F(t_i) = \bigcup_i F(u_i).$$

The monotonicity of the sequence $(u_i)_i$ is derived from the fact that the conditions on the run-times $\alpha_j(x)$ used in the maximalization for $u_{k+1}(x)$ are weaker than the corresponding conditions for $u_k(x)$; therefore the maximum does not become smaller. □

3.4.4. THE MEYER-McCREIGHT ALGORITHM AS A CLOSURE OPERATOR

In this section we consider the MMC-algorithm with a "general" discriminator. Although we do not intend to present a discussion on relativised complexity theory, it should be noted that the results in this section do not assume computability of this discriminator. If it is a non-recursive discriminator, then the function computed by our MMC-algorithm is non-recursive also.

In section 1 we considered the MMC-algorithm which computes a new name for a strong class, given by an old name. Forgetting the fact that our intention was that the new name should be measured, the MMC-algorithm is an inefficient procedure. In fact we use the old name t to generate an infinite sequence of "boolean procedures" $b_j(x)$ (the tests on violations perpetrated by the indices) so that $j \in F(t)$ *iff* $b_j(x) =$ *true* for almost all x. Using these test-results the priority statusses are manipulated in such a way that the equivalence $j \in F(t')$ *iff* $b_j(x) =$ *true* for almost all x holds also.

Now suppose that the class $F(t)$ is not given by the name t but by an oracle for the indices contained in $F(t)$. One may ask whether it is still possible to compute a name t' so that $F(t) = F(t')$. The answer is positive To compute t' one simply replaces the test-results reporting on the violations by the answers given by the oracle; it is not difficult to verify that the modified MMC-algorithm indeed computes a function t' satisfying $F(t) = F(t')$.

The same modification is possible if not an oracle for $F(t)$ is given but a so-called *general discriminator* for $F(t)$, a concept which is defined below:

DEFINITION 3.4.24. Let X be an arbitrary set of indices. A *general discriminator for* X, is a total boolean function $B(i,x)$ so that the following equivalence holds:

$$i \in X \text{ \underline{iff} } \overset{\infty}{\forall} x[B(i,x) = \underline{true}].$$

Using this terminology we can reformulate a technical result in Chapter 1.4 (lemma 1.4.10) by the following proposition:

PROPOSITION 3.4.25. Let X be a Σ_2-class of indices, then there exists a recursive general discriminator for X.

PROOF. Let P be a total recursive predicate so that

$$j \in X \text{ \underline{iff} } \exists b \forall a[P(j,a,b)].$$

We define $B(j,x)$ by:

$$B(j,x) = \exists y \le x[\forall z \le x[P(j,z,y)] \text{ \underline{and} \underline{not} } \exists w<y[\forall z<x[P(j,z,w)] \text{ \underline{and} \underline{not} } P(j,x,w)]].$$

Then $j \in X$ *iff* $\overset{\infty}{\forall} x[B(j,x)]$ (see chapter 1.4). □

The converse implication holds also: if B is a general discriminator for X and if B is a recursive function, then X is a Σ_2-class.

Let X be a class of indices and let B be a general discriminator for X. Then we construct the following stagewise algorithm, called the MEYER-McCREIGHT algorithm *based on* B.

```
stage n:
1. Introduce index n: p[n] := next; b[n] := true;
2. Discriminator:
       for y ≤ n do if b[y] and not B(y,n) then
                          (p[y] := next; b[y] := false) fi od;
3. x := $\pi_1$n; y := $\pi_2$n; if defined t'[x] then goto stage n+1 fi;
4. searchtime (x,y,y,prior); if failure then goto stage n+1 fi;
```

```
5. Discriminator against t': t'[x] := y;
     for z ≤ n do if not b[z] and A(z,x,y) ≠ true then
                         (p[z] := next; b[z] := true) fi od;
6. goto stage n+1;
```

Dropping the prime which is superfluous since there is no danger for confusion we denote the function computed by this program by t. The relation between X and $F(t)$ is expressed by the following theorem:

THEOREM 3.4.26: Let B be a general discriminator for a class of indices and let t be the function computed by the MMC-algorithm based on B. Then:

(i) $X \subset F(t)$.
(ii) $X \subset F(u)$ *imp* $F(t) \subset F(u)$.
(iii) $X = F(t)$ *iff* there exists a function u (not necessary recursive) so that $X = F(u)$.

PROOF.

(i) Consider the behaviour of the priorities during execution of the MMC-algorithm based on B.

Case 1. p[j] is unstable.

In this case $j \notin X$ since $B(j,x)$ = *false* for infinitely many x; $j \notin F(t)$ by the same argumentation as before.

Case 2. p[j] is white-stable.

In this case $j \in X$ since at the moment that p[j] gets its ultimate value, only finitely many values of $B(j,x)$ are computed; for all x for which $B(j,x)$ is computed afterwards this value is *true*. The proof that $j \in F(t)$ remains unchanged.

Case 3. p[j] is black-stable.

Now $j \in F(t)$ as before but we cannot prove that $j \in X$.

In each case we have $j \in X$ *imp* $j \in F(t)$ which proves $X \subset F(t)$.

(ii) From assertion 3.4.4 we conclude that the run-time of a black-stable index j is bounded almost everywhere by the maximum of finitely many run-times of white indices having stable priorities larger than the stable priority of j.

Denote this set of white-stable indices with higher priority by $\{j_1,\dots,j_k\}$. Then we have:

$$\{j_1,\dots,j_k\} \subset X \subset F(u) \ \underline{\mathit{imp}}$$

$$\forall l \overset{\infty}{\forall} x[\alpha_{j_l}(x) \le u(x)] \ \underline{\mathit{imp}}$$

$$\overset{\infty}{\forall} x[\ \max_{1\le l\le k} \{\alpha_{j_l}(x)\} \le u(x)].$$

Combining this with $\overset{\infty}{\forall} x[\alpha_j(x) \le \max_{1\le l\le k} \{\alpha_{j_l}(x)\}]$ (assertion 3.4.4) we find

$$\overset{\infty}{\forall} x[\alpha_j(x) \le u(x)]$$

hence $j \in F(u)$.

Note that u does not need to be recursive in order to make the proof correct.

(iii) The only if side is trivial since $X = F(t)$ implies the existence of a name u so that $X = F(u)$. Conversely suppose that $X = F(u)$ for some unknown name u. Then $F(t) \subset F(u)$ by (ii). Combining (i) and (ii) this yields $F(t) = X$. □

The MMC-algorithm based on B computes a name of a strong ARBC which contains X, and the equality holds if and only if X is already a strong ARBC with an unknown, not necessarily recursive, name. The above proof however yields further information on the indices contained in $F(t) \setminus X$.

<u>COROLLARY 3.4.27</u>. $j \in F(t)$ <u>*iff*</u> there exist finitely many indices $j_1,\dots,j_k$ which are contained in X so that

$$\overset{\infty}{\forall} x[\alpha_j(x) \le \max_{1\le l\le k} \{\alpha_{j_l}(x)\}].$$

<u>PROOF</u>. If $j \in X$ then the assertion is trivial whereas for indices $j \in F(t) \setminus X$ the assertion is nothing else but assertion 3.4.4, using the fact that j must be a black-stable index.

Conversely assuming that the run-time α_j is bounded almost everywhere by the maximum of finitely many run-times of indices which are contained in X and therefore also are contained in $F(t)$ we derive that $j \in F(t)$ by the same argumentation as before. □

The preceding results yield also the following corollary:

<u>COROLLARY 3.4.28</u>. $F(t) = \cap\{F(u) \mid X \subseteq F(u)\}$.

<u>PROOF</u>. Since $X \subseteq F(t)$ the inclusion $\cap\{F(u) \mid X \subseteq F(u)\} \subseteq F(t)$ is trivial. The converse inclusion follows directly from 3.4.26 (ii). □

Note that the preceding results show that the class $F(t)$ does not depend on the discriminator B used for X. The results show moreover that every Σ_2-class of indices is contained in a minimal strong class (without minimality one has always $X \subseteq F(\varepsilon)$).

<u>COROLLARY 3.4.29</u>. Let X be some Σ_2-class of indices. Then there exists a function t such that $X \subseteq F(t)$ and such that $F(t) \subseteq F(u)$ for each u with $X \subseteq F(u)$.

<u>PROOF</u>. By 3.4.25 there exists a recursive general discriminator for X, and therefore the MMC-algorithm based on this discriminator yields a recursive name for a class $F(t)$ which has the claimed properties by 3.4.26. □

As a fourth corollary we prove that the inclusion $X \subset F(t)$ preserves monotonicity:

<u>COROLLARY 3.4.30</u>. If X_1 and X_2 are two sets of indices determined by discriminators B_1 and B_2 and if the functions computed by the MMC-algorithms based on B_1 respectively B_2 are denoted by t_1 and t_2 then $X_1 \subset X_2$ implies $F(t_1) \subset F(t_2)$.

<u>PROOF</u>. $X_1 \subset X_2$ implies $\forall u[F(u) \supset X_2$ *imp* $F(u) \supset X_1]$. Hence

$$F(t_1) = \cap\{F(u) \mid F(u) \supset X_1\} \subset \cap\{F(u) \mid F(u) \supset X_2\} = F(t_2). \quad \square$$

Writing $\bar{X}$ for $F(t)$ we conclude that the transformation $X \to \bar{X}$ is a closure operator (the MMC-closure operator):

$X \subset \bar{X}$ by 3.4.26,

$\bar{X} = \bar{\bar{X}}$ by 3.4.26,

$X_1 \subset X_2$ *imp* $\bar{X}_1 \subset \bar{X}_2$ by 3.4.30.

This closure operator is however no Kuratowski closure operator for a topology, since $\overline{X_1 \cup X_2}$ is not in general equal to $\bar{X}_1 \cup \bar{X}_2$. Take for example two complexity classes whose union is not a complexity class [MC 69].

Another application yields the following result:

PROPOSITION 3.4.31. The intersection of two strong classes is again a strong class.

PROOF. In fact this is nothing but a trivial property of closure operators. Strong classes are Σ_2-classes which are identical to their MMC-closure. The intersection of two Σ_2-classes X and Y is again a Σ_2-class. Moreover if $X = \bar{X}$ and $Y = \bar{Y}$ we have:

$$\bar{X} \cap \bar{Y} = X \cap Y \subset \overline{X \cap Y} \subset \bar{X} \cap \bar{Y}$$

hence $X \cap Y = \overline{X \cap Y}$
This shows that $X \cap Y$ is a Σ_2-class which is identical to its MMC-closure. □

Note that for total t and u one trivially has $F(t) \cap F(u) = F(\underline{min}(t,u))$. The function *min*(t,u) is computable also in the case where t and u are selected from a measured set. Consequently 3.4.31 follows directly from the naming theorem.

To complete this section we consider the intersection of a sequence of strong classes. It is known that there exist examples of decreasing sequences (t_i) of total recursive functions, so that $\bigcap_i F(t_i) \neq F(t)$ for each recursive t [Ba 70]. Using 3.4.28 we can prove that this intersection is a class $F(t)$ for a Δ_4 function t. The non-existence of an intersection theorem means therefore only that the intersection is not named by a recursive name, and not (as is sometimes the case with the union of two classes) that the intersection is not nameable.

PROPOSITION 3.4.32. Let $(t_i)_i$ be a sequence of partial recursive functions. Then there exists a Δ_4 function t so that $\bigcap_i F(t_i) = F(t)$.

PROOF. Let $X = \bigcap_i F(t_i)$ and let B be a discriminator for X. By 3.4.28 the function t computed by the MMC-algorithm based on B is a name for the set $X = \bigcap\{F(u) \mid X \subset F(u)\}$.

Since $X \subset F(t_i)$ for each i we conclude that $F(t) \subset \bigcap_i F(t_i) = X$. The converse inclusion follows straightforwards from 3.4.26. This proves that $X = F(t) = \bigcap_i F(t_i)$.

By repeating the original proof one shows that the function t computed by the MMC-algorithm based on B is "measured modulo B" i.e. the graph of t is a set which is recursive relative B. To determine the arithmetical complexity of B we remark that the set X is a Π_3-set: we have $j \in X$ *iff* $\forall k[j \in F(t_k)]$; since $j \in F(t_k)$ is a Σ_2-relation we conclude that X is a Π_3-set. Furthermore it is trivial that any Π_3-set possesses a Π_3-discriminator B. Since t is Δ_1 relative B we conclude that t is a Δ_4-function. □

The above bounds are not sharp; it is not difficult to construct a Σ_2-discriminator B for X. The existence of non-recursive names for infinite intersections is proved also in the thesis of E.L. ROBERTSON [Rb 70].

3.4.5. A MEYER-McCREIGHT ALGORITHM FOR WEAK CLASSES

{*A magician uses deception, deceit and fakery to hoodwink the people, while a wizard is a master of the secrets of the universe.*
Parker & Hart. The wizard of ID}

In section 3.4.2 we proved that weak classes cannot be uniformly renamed, so the title of this section announces a non-existent algorithm. The reader should expect therefore not to much.

The proofs in section 3.4.2 are all based on the assumption that we are considering a measured transformation of programs. In order to escape the prohibiting effect of these results we may drop this condition of measuredness. One should beware however for trivialities like "The identity transformation renames both weak and strong classes".

In the preceding section the MMC-algorithm has been considered as a method to compute a name for a set X defined by means of a general discriminator, provided that the set X can be represented as an ARBC.

Now our weak classes are Σ_2-classes of indices also, and hence recursive discriminators for a weak class exist. Moreover we will construct in the sequel of this section a modification of the routine searchtime which creates weak violations instead of strong ones. So the subroutines for a weak MMC-algorithm are available, and it seems that we only have to wait for a clever programmer to combine these parts into a complete algorithm.

To the opinion of the author it is precisely this combination which is forbidden by the negative results in section 2. In synchronizing the discriminator with the modified searchtime we must prevent someway that searchtime starts looking for solutions at extremely high values, because of the discriminator being unable to determine which white indices i at x are "good" bound-respecting ones, which ones are the "bad" violators at x, and which ones are the "ugly" bystanders (with $A(i,x,z)$ = *void* for each z).

To eliminate this uncertainty we must teach the algorithm to separate the violators from the bystanders, something which can only be done on the base of prejudice.

To be more precise, we use in our modified routine which plays the role fulfilled by searchtime in the ordinary MMC-algorithm, and which will be called *weak searchtime* hereafter, a subroutine called *bias*. The subroutine *bias* predicts whether some white index in the priority queue is well-behaved, and should be respected, or whether the index is a bystander and should be disregarded. The routine *bias* is called the *wizard* of the resulting weak MEYER-McCREIGHT algorithm.

First we describe the routine *weak searchtime*. In the ordinary MMC-algorithm the routine searchtime terminates successfully if it has located a black index, whose run-time at x exceeds the run-times of the white indices with higher priority. However, in order to be sure that we will create a weak violation, we must be certain that this black index has indeed a finite run-time at x. Therefore weak searchtime marks all possible candidates (black indices j which are encountered at a moment where val $\leq \alpha_j(x)$), and we continue by increasing val, and pushing cand through the priority queue, until one of our marked candidates is found to have a run-time $\alpha_j(x) = \text{val} + 1$. At this moment weak searchtime termiantes successfully, and nominates this index j to be the index used at x.

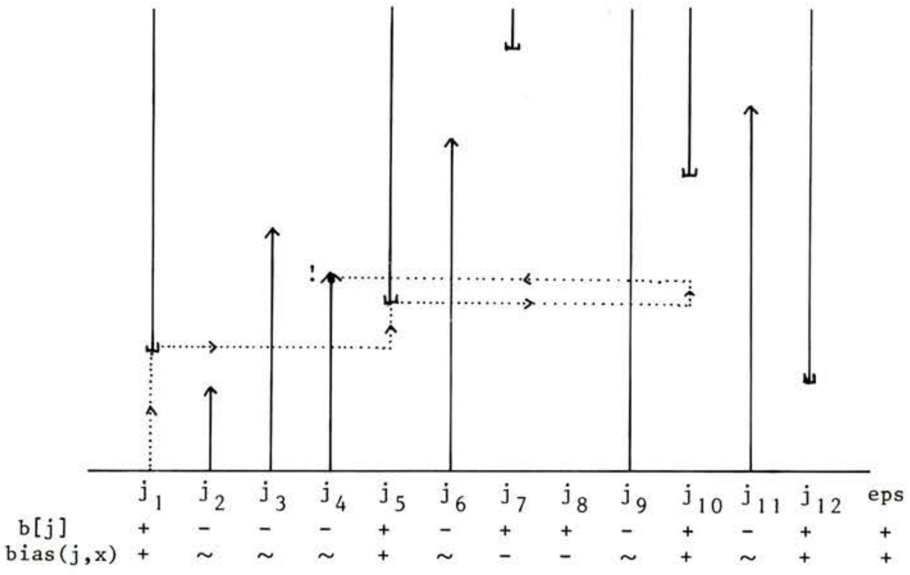
To prevent abnormal termination of weak searchtime because of exhaustion of the priority queue, we insert at the tail of the priority queue a white index with infinite run-time.

If the current candidate is a white index, weak searchtime uses its subroutine *bias* to determine whether it should increase val in order to respect this index, or whether it should forget about this index and proceed to the next candidate. If val has to be increased we must look for a possible marked black candidate which might have run-time equal to val + 1. These black candidates are stored in a linear list *earlier cand*; the task involved in increasing val is performed by the local subroutine *increase val*.

The routine weak searchtime is described by the following program.

```
proc weak searchtime = (int x,low,high,¢ priorqueue ¢ prior,
                                          proc(int,int) bool bias) void:

    begin proc increaseval = (ref int val,llint earlier cand) void:
                begin val +:= 1;
                   for j over earlier cand do
                   if ¢ A(j,x,val+1) = true ¢ then cand :=j; goto success fi
                                          od
                end;
          ¢ ... remaining declarations etc. ... ¢

    start weak searchtime: val := low; cand := ¢ first index of prior ¢;
       ¢ initialize earlier cand to be an empty list, insert the closing
         item "eps" with α_eps(x) = ∞, b[eps] = true, and bias(eps,x) = true
         at the tail of the priority queue ¢

    steps: while val ≤ high do
              if b[cand] then
                 if bias(cand,x) and ¢ A(cand,x,val) ≠ true ¢
                    then increaseval(val,earlier cand)
                    else cand := ¢ next item in prior ¢
                 fi
              elif ¢ A(cand,x,val) = true ¢ then cand := ¢ next item in prior ¢
              elif ¢ A(cand,x,val+1) = true ¢ then goto success
              else attach int(earlier cand,cand); cand := ¢next item in prior ¢
              fi                  od;

    failure: ¢ report failure to calling program ¢.
    success: ¢ report success to calling program with t(x) = val and
                index cand to become the index used at x ¢

    end ¢ weak searchtime ¢;
```

The computation of weaksearchtime is illustrated in diagram 3.4.33. At the moment of successful termination as represented in this diagram the list earlier cand contains the indices j_3, j_4, j_6 and j_9.



	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	eps
b[j]	+	−	−	−	+	−	+	+	−	+	−	+	+
bias(j,x)	+	~	~	~	+	~	−	−	~	+	~	+	+

Diagram 3.4.33.

Note that not all the properties of seachtime are preserved. For example it is no longer true that val is larger than all white run-times left of cand; this holds only for the white indices j satisfying bias(j,x) = *true*. Moreover val is not larger than all black run-times left of candidate; this holds only for the black indices which are not contained in the list earlier cand.

As before the routine weak searchtime will be used in a dovetailed manner by our weak MMC-algorithm. One could ask for whether the list earlier cand must be saved from one call of weak searchtime at the argument x to the subsequent call at the same argument. This is unnecessary, since the candidates on earlier cand are precisely the black indices with run-time greater than val. One should beware for "hidden" increases of val in between two calls of weak searchtime at the same argument. Moreover, since we are not interested in computing a measured function, we may forget the precautions against revisions of earlier rejections; hence we can take low = 0 throughout our weak MMC-algorithm.

The weak MMC-algorithm based on the general discriminator B and the wizard bias is decribed by the following program:

```
stage n:

1. Introduction: b[n] := true; p[n] := next;

2. Discriminator:

     for m ≤ n do
          if b[m] and not B(m,x)
          then p[m] := next; b[m] := false
          fi    od;

3. $x := \pi_1 n$; $y := \pi_2 n$;
   if defined t[x] then goto 5 fi;

4. weak searchtime(x,0,y,prior,bias);
   if success then t[x] := y fi;

5. Discriminator against t:

     for m ≤ n do
          if not b[m] then
             for x ≤ n do
                  if defined t[x] and not test against output [m,x]
                  then if ¢ A(m,x,t[x]) = true ¢
                          then test against output [m,x] := true;
                          elif ¢ A(m,x,n) = true ¢
                          then test against output [m,x] := true;
                                p[m] := next; b[m] := true
                          fi
                  fi    od
          fi    od ¢ discriminator against t ¢;

6. goto stage n+1;
```

Note that a major part of the algorithm consists of section 5: *discriminator against t*; we must look for weak violations against t, which violations have to be enumerated. Consequently we must beware for multiple tests at a single argument; the bookkeeping array *test against output* is introduced for this purpose.

Before investigating what the computed function t stands for, we allow ourselves one more generalization. Upto now we have tacitly assumed that the wizard *bias* used by searchtime is a total function. In our applications we need a wizard which is partial; however, if for a given argument x, bias(i,x) converges for some index i, then bias(i,x) should converge for all indices i; i.e. $\mathcal{D}$bias = $\pi_2^{-1}A$ for some recursively enumerable subset $A \subseteq \mathbb{N}$.

To prevent that bias(i,x) is called for at arguments outside $\mathcal{D}$bias, the calls of weak searchtime are executed conditionally; section 4 is replaced by:

```
4': if ¢ bias(0,x) converges within n steps ¢ then
    begin weak searchtime(x,0,y,prior,bias);
          if success then t[x] := y fi end
    fi;
```

It is clear that by this modification only terminating calls of bias are issued, and that the domain of the computed function t satisfies $\mathcal{D}t \subseteq A$.

In order to investigate the relation between the Σ_2-class X discriminated by B and the weak class $F_\omega(t)$, we consider (as usual) the behaviour of the priorities of the indices.

If p[j] is instable then $j \notin X$ and $j \notin F_\omega(t)$.

If p[j] is white-stable then $j \in X$. In order that $j \in F_\omega(t)$ it is necessary that the finite run-times $\alpha_j(x)$ for $x \in A$ are almost everywhere respected. Since the priority used at x grows unboundedly (this property of the MMC-algorithm is not lost by our modifications) the only possible cause of not respecting a finite run-time $\alpha_j(x)$ with $x \in A$ and x sufficiently large, can be an incorrect prediction by the wizard: bias(j,x) = *false* for an argument x where $\alpha_j(x) < \infty$.

Therefore, in order to deal correctly with white-stable indices j the wizard should not overlook more than finitely many finite run-times $\alpha_j(x)$ with $x \in A$.

If p[j] is black-stable then $j \in F_\omega(t)$. Inspired by the result on strong classes we should not expect anything more than that $j \in F_\omega(u)$ for each u such that $X \subset F_\omega(u)$. In order to be able to derive such a relation we need an assertion like assertion 3.4.4. This assertion is formulated below:

ASSERTION 3.4.34. If p[j] becomes black-stable, then for almost all $x \in A$ either $\alpha_j(x) = \infty$ or otherwise $\alpha_j(x)$ is bounded by the maximum of the finite set of run-times $\alpha_i(x)$ of those white-stable indices with higher priority which satisfy at x the condition bias(i,x) = *true*.

If at a certain argument x one of the run-times $\alpha_i(x)$ included in this finite collection actually diverges the assertion becomes trivial for this x. Consequently this indicates a second condition which the wizard must satisfy in order to deal correctly with black-stable indices; for white-stable indices j the wizard should not declare to be finite more than a finite number of runtimes $\alpha_j(x)$ with $x \in A$ which are actually diverging.

We now have found two conditions on the wizard, in order that the weak MMC-algorithm behaves correctly. Clearly the property of an index to be white-stable is algorithm dependent; however, white-stable indices are always members of X. Combining these observations we arrive at the following definition:

DEFINITION 3.4.35. The wizard bias is called *justified for the class* X *on* A if $\mathcal{D}$bias $= \pi_2^{-1}A$ and the following condition holds:

$$\forall i \overset{\infty}{\forall} x[i \in X \ \underline{and} \ x \in A \ \underline{imp}(\alpha_i(x) < \infty \ \underline{iff} \ \text{bias}(i,x))].$$

THEOREM 3.4.36. Let B be a general discriminator for the Σ_2-class X. Let bias be a wizard which is justified for X on $A \subseteq \mathbb{N}$. Then the function t computed by the weak MEYER-McCREIGHT algorithm based on B and bias satisfies the following conditions:

(i) $X \subseteq F_\omega(t)$

(ii) if $\mathcal{D}u \subset A$ and $X \subset F_\omega(u)$ then $F_\omega(t) \subset F_\omega(u)$.

PROOF. The condition that bias is justified for X on A includes both conditions which we recognized to be necessary in order that the MMC-algorithm behaves correctly.

The instable indices present no problem at all. White-stable indices j are member of X and, because of the fact that bias is justified, their finite run-times $\alpha_j(x)$ with $x \in A$ are respected.

Since black-stable indices are contained automatically in $F_\omega(t)$ the first assertion $X \subset F_\omega(t)$ is proved.

To derive the second assertion we use assertion 3.4.34. Let $\mathcal{D}u \subset A$ and $X \subset F_{\mathcal{W}}(u)$. Assume that $j \in F_{\mathcal{W}}(t)$. Then j is a stable index. If j is white-stable then $j \in X$ and we are done. If j is black-stable then for almost all $x \in A$ the run-time $\alpha_j(x)$ either diverges, or $\alpha_j(x)$ is bounded by the maximum of the finite set of run-times $\alpha_i(x)$ of the white-stable indices with higher priority, satisfying the condition bias(i,x) = *true*. Since bias is justified for X on A and since white-stable indices are contained in X bias(i,x) = *true* is almost everywhere equivalent to $\alpha_i(x) < \infty$. Since $X \subseteq F_{\mathcal{W}}(u)$ this implies also $\alpha_i(x) < u(x)$ (for almost all x). From this we derive:

$$\overset{\infty}{\forall} x[x \in A \; \underline{and} \; \alpha_j(x) < \infty \; \underline{imp} \; \alpha_j(x) \leq u(x)]$$

and since $\mathcal{D}u \subseteq A$ this implies $j \in F_{\mathcal{W}}(u)$. □

We complete this section by presenting some applications of the weak MMC-algorithm. The first application is a triviality.

TRIVIALITY 3.4.37. There exists a transformation σ such that for each index i, $\mathcal{D}\varphi_i = \mathcal{D}\varphi_{\sigma(i)}$; $\varphi_{\sigma(i)} \leq \varphi_i$ and $F_{\mathcal{W}}(\varphi_i) = F_{\mathcal{W}}(\varphi_{\sigma(i)})$.

PROOF. For B we take the natural discriminator for $F_{\mathcal{W}}(\varphi_i)$ whereas bias is defined by:

$$\text{bias} = \lambda j,x[A(j,x,\varphi_i(x)) = \underline{true}].$$

To enforce the domain condition and the inequality we include in our weak MMC-algorithm an escape-value mechanism: whenever $\varphi_i(x) = y$ has converged and weak searchtime(x,0,z,prior,bias) fails to provide a solution when called with $z \geq y$, we set $t[x] = y$. It is left to the reader to show that this escape-value mechanism does not disturbe the correctness of our preceding arguments.

It is clear that bias as defined above is justified for $F_{\mathcal{W}}(\varphi_i)$ on $\mathcal{D}\varphi_i$. The result now follows by 3.4.36. □

The transformation σ from 3.4.37 may be used to eliminate unnecessarily large values from φ_i.

Our next application yields a generalized union theorem for weak classes:

<u>THEOREM 3.4.38</u>. Let $(t_i)_i$ be a sequence of total functions such that $\mathcal{F}_\omega(t_i) \subset \mathcal{F}_\omega(t_{i+1})$. Then there exists a function tinf such that $\bigcup_i \mathcal{F}_\omega(t_i) = \mathcal{F}_\omega(\text{tinf})$.

<u>PROOF</u>. Similarly to the proof of 3.4.20 we design a weak MMC-algorithm which operates on items consisting of an index and a bound-index. The discriminator B tests for the pair $\langle i,k\rangle$ whether index i respects the bound t_k.

The run-time of an item $\langle i,k\rangle$ is the run-time α_i of its constituent index.

The wizard bias is defined by

$$\text{bias}(\langle i,k\rangle,x) \ \underline{\mathit{iff}}\ A(i,x,t_k(x)) = \underline{\mathit{true}}.$$

Note that by the assumption that the t_k are total, bias is a total function.

Let tinf be the function computed by the weak-MMC-algorithm.

The class of pairs X discriminated by B contains all pairs $\langle i,k\rangle$ with $i \in \mathcal{F}_\omega(t_k)$. Moreover "$\langle i,k\rangle \in \mathcal{F}_\omega(\text{tinf})$" provided $i \in \mathcal{F}_\omega(\text{tinf})$. Clearly bias is justified for X on $\mathbb{N}$.

By 3.4.36(i) we conclude "$X \subset \mathcal{F}_\omega(\text{tinf})$" which implies $\bigcup_i \mathcal{F}_\omega(t_k) \subset \mathcal{F}_\omega(\text{tinf})$.

To prove the converse inclusion 3.4.36(ii) is not strong enough but by using assertion 3.4.34 and by repeating the argumentation from the proof of 3.4.20 this inclusion is easily derived. This completes the proof. □

Looking backwards one may ask whether this theorem could be proven also by renaming the sequence of classes by a non-decreasing sequence of names. For total t_i such a renaming leads to the same problems as was the case with the strong classes. A weak analogue of proposition 3.4.23 is valid.

<u>PROPOSITION 3.4.39</u>. Let $(t_i)_i$ be a sequence of total functions so that for all i, $\mathcal{F}_\omega(t_i) \subset \mathcal{F}_\omega(t_{i+1})$. Then there exists a non-decreasing sequence of total functions $(u_i)_i$ so that $\bigcup_i \mathcal{F}_\omega(t_i) = \bigcup_i \mathcal{F}_\omega(u_i)$.

<u>PROOF</u>. Define u_k by:

$$\begin{aligned} u_k = \lambda x[\underline{\mathit{if}}\ x \le k\ &\underline{\mathit{then}}\ \mathit{max}\{\alpha_j(x) \mid j \le k\ \underline{\mathit{and}}\ \alpha_j(x)\ \underline{\mathit{le}}\ \max_{i\le k}\{t_i(x)\}\} \\ &\underline{\mathit{else}}\ \mathit{max}\{\alpha_j(x) \mid j \le j\ \underline{\mathit{and}}\ \alpha_j(x)\ \underline{\mathit{le}}\ \max_{i\le k}\{t_i(x)\}\ \underline{\mathit{and}} \\ &\exists i{\le}k[\forall y{\in}[k,x][\alpha_j(y)\ \underline{\mathit{le}}\ t_i(y)\ \underline{\mathit{or}}\ \alpha_j(y)\ \underline{\mathit{gt}}\ x]]\}\ \underline{\mathit{fi}}\ ]. \end{aligned}$$

The proof that $\bigcup_i F_{\mathcal{W}}(t_i) = \bigcup_i F_{\mathcal{W}}(u_i)$ is analogous to the proof of proposition 3.4.23. If $j \in F_{\mathcal{W}}(t_n)$ then there exists an $m \geq j,n$ so that for all $x \geq m$, $\alpha_j(x) = \infty$ or $\alpha_j(x) \leq t_n(x)$. From this one derives that for all $x \geq m$ the run-time $\alpha_j(x)$ is used in the maximalization to compute $u_m(x)$ whenever $\alpha_j(x)$ is finite, and therefore $j \in F_{\mathcal{W}}(u_m)$.

Conversely the run-times $\alpha_l(x)$ which are used in the maximalization to compute $u_m(x)$ are the run-times of precisely those indices $l \leq m$ which do not violate weakly at least one of the bounds t_i with $i \leq m$ over arbitrary long intervals $[m,x]$. Hence for sufficiently large x only indices l contained in $\bigcup_{i\leq m} F_{\mathcal{W}}(t_i) = F_{\mathcal{W}}(t_m)$ contribute to the maximalization. Since only finite run-times contribute to the value of $u_m(x)$ we conclude that for sufficiently large x, $u_m(x) \leq t_m(x)$ and consequently $F_{\mathcal{W}}(u_m) \subset F_{\mathcal{W}}(t_m)$.

Finally the fact that $u_m(x) \leq u_{m+1}(x)$ is derived from the fact that more run-times $\alpha_l(x)$ are used in the maximalization for $u_{m+1}(x)$ than for $u_m(x)$.

This completes the proof. □

Proposition 3.4.39 reduces 3.4.38 to the union theorem for weak classes (3.3.19).

There remains an open question whether a generalized union theorem for weak classes with partial names exists or not. Starting with the original union theorem, we have investigated generalizations in three independent directions (*partial names*, *weak classes*, and *monotonicity of classes* instead of monotonicity of names). These generalizations and their pairwise combinations now are proved, but the triple combination remains unsolved. It should be quite amazing if this combination should be false.

Our final application concerns the intersection of two weak classes. If $\underline{\mathit{min}}(t,u)$ is a computable function then one has $F_{\mathcal{W}}(\underline{\mathit{min}}(t,u)) = F_{\mathcal{W}}(t) \cap F_{\mathcal{W}}(u)$. Consequently the intersection of two weak classes with total (or measured) names is again a weak class. Since the weak classes cannot be renamed by a measured set of names we cannot reduce the general case to this special case.

<u>PROPOSITION 3.4.40</u>. Let t and u be partial functions then there exists a partial function v such that $F_{\mathcal{W}}(t) \cap F_{\mathcal{W}}(u) = F_{\mathcal{W}}(v)$.

<u>PROOF</u>. Take for B a discriminator for the Σ_2-class $X = F_{\mathcal{W}}(t) \cap F_{\mathcal{W}}(u)$. A wizard for X is constructed as follows: Let i and j be indices for t and u.

Define the function w by:

$$w = \lambda x[\underline{if}\ \Phi_i(x) \le \Phi_j(x)\ \underline{then}\ t(x)\ \underline{else}\ u(x)\ \underline{fi}].$$

Clearly $\mathcal{D}w = \mathcal{D}t \cup \mathcal{D}u$. We define bias by:

$$bias = \lambda i,x[A(i,x,w(x)) = \underline{true}].$$

Now bias is justified for X on $\mathcal{D}w$. Let v be the function computed by the weak MMC-algorithm based on B and bias. By 3.4.36 we have:

(i) $X \subset F_{\omega}(v)$ and

(ii) if $\mathcal{D}f \subset \mathcal{D}w$ and $X \subset F_{\omega}(f)$ then $F_{\omega}(v) \subset F_{\omega}(f)$.

Taking f = u and f = t in (ii) proves $F_{\omega}(v) \subset X$. □

The following generalization of 3.4.40 is left to the reader:

<u>EXERCISE 3.4.41</u>. Let $(t_i)_i$ be a sequence of functions then there exists a (non-recursive) function tint such that $\bigcap_i F_{\omega}(t_i) = F_{\omega}(tint)$.

APPENDIX

ALGORITHMS

{*Wahre Worte sind nicht wohlklingend.*
Wohlklingende Worte sind nicht wahr.
Ein guter Mensch streitet nicht mit Worten.
Wer mit Worten streitet, ist kein guter Mensch.
Der Weise weiss nicht vieles;
Wer vieles weiss, ist nicht weise.
Lao Tse, das Buch vom Tao, VII 81,
ed. Lin Yutang}

A0. INTRODUCTION

This appendix contains some algorithms which were described more or less formally in the preceding parts of this treatise. We use the language defined in §1.1.2, the extensions defined in §1.1.3 and the procedures given in §1.1.4.

In order to get some useful programs we have to disregard several scope restrictions in the definition of ALGOL 68.

Consider the following program:

```
begin
mode fun = proc(int) int;
mode operator = proc(fun) fun;
fun f := λx[0];
operator gamma = (fun h) fun: (int x) int: h(x)+1;
   fun g := gamma(f);
         pr illegal since the environ of "gamma(f)" is newer than that of
         g pr
      f := gamma(f)
         pr illegal as above; moreover what is happening to the relation
         g = gamma(f)? pr
end
```

In the above example our interpretation is that g posesses after the assignation *g := gamma(f)* the routine λx[1], and so does f after the assignation *f := gamma(f)*. Hence g = gamma(f) is no longer true.

A1. THE MYHILL ISOMORPHISM ALGORITHM (chapter 1.4)

If f and g are two injective total recursive functions, and if A and B are two subsets of $\mathbb{N}$ such that $A \leq_1 B$ by f and $B \leq_1 A$ by g then myhill (f,g) yields a recursive permutation s such that $A \equiv B$ by s.

```
proc myhill = (proc(int) int f,g) proc(int) int:
    (int x) int:
    (flex [0:0] int x list, y list;
     flex [0:0] bool x def, y def;
         x def[0] := y def[0] := false;
     while not look up bool(x def,x) do
        int i := μk[not look up bool(x def,k)];
        int j := f(i);
            while look up bool(y def,j) do j := f(y list[j]) od;
            insert int(x list,i,j); insert int(y list,j,i);
            insert bool(x def,i,true); insert bool(y def,j,true);
            j := μk[not look up bool(y def,k)];
            i := g(j);
            while look up bool(x def,i) do i := g(x list[i]) od;
            insert int(x list,i,j); insert int(y list,j,i);
            insert bool (x def,i,true); insert bool(y def,j,true)
                                          od;
        x list[x]
    ) ¢ myhill ¢;
```

DISCUSSION

Values of s and s^{-1} are stored in x list and y list. The boolean arrays x def and y def designate whether s(x) or $s^{-1}(y)$ is defined or not. During execution of the program a loop is executed during which first s(x) is defined for the lowest x for which s(x) was not defined before; afterwards the lowest value y is found for which $s^{-1}(y)$ is not yet defined and a corresponding x is found such that s(x) = y is a legal extension of s. Execution of the loop is terminated when s(x) is found to be defined for the requested argument x.

During execution of the algorithm we preserve correctness of the assertions:

(a) $\overset{\infty}{\forall}$x[x def[x] = *false*]; $\overset{\infty}{\forall}$y[y def[y] = *false*];

(b) x def[x] = *true* *imp* ∃y[x list[x] = y *and* y def[y] = *true* *and* y list[y] = x *and* (x ∈ A *iff* y ∈ B)];

y def[y] = *true* *imp* $\exists$x[y list[y] = x *and* x def[x] = *true* *and*
x list[x] = y *and* (x $\in$ A *iff* y $\in$ B)].

(c) x def[x1] = x def[x2] = *true* *and* x1 $\neq$ x2 *imp* x list[x1] $\neq$ x list[x2];
y def[y1] = y def[y2] = *true* *and* y1 $\neq$ y2 *imp* y list[y1] $\neq$ y list[y2].

Because of (a) computation of i always succeeds. Hence j can be computed as well. When i and j are known execution of a while loop is initiated. Let i_0 and j_0 be the values of i and j when the execution of the while loop begins. If y def[j_k] = *true* then let i_{k+1} = y list[j_k] and $j_{k+1} = f(i_{k+1})$. By (a) the while loop will terminate unless the sequences $(i_k)_k$ and $(j_k)_k$ become periodic. Since by our assumptions on f and g and (c) both f and the partial function whose values are stored in y list are 1 - 1 this occurs only if $i_k = i_0$ for some k > 0, which, by (b) contradicts the fact that x def[i_0] = *false*. This shows that the while loop must terminate by detecting y def[j_k] = *false*. Next s is extended by letting $s(i_0) = j_k$, (and $s^{-1}(j_k) = i_0$). By (b) and the assumptions on f and g we have $i_0 \in A$ *iff* $j_0 \in B$ *iff* $i_1 \in A$ *iff* .... *iff* $j_k \in B$. Hence the extension of s with the pair $<i_0,j_k>$ preserves the validity of (a), (b) and (c).

During the second half of the main loop the roles of x and y are interchanged in order to enforce that s becomes a surjection.

A2. ENUMERATION OF A Σ_2-PRESENTABLE CLASS X USING A WAY-OUT STRATEGY (§2.3.2)

Let A be a Σ_2-class and let B be a general discriminator for A (i.e. x $\in$ A *iff* $\overset{\infty}{\forall}$x[B(i,x)]). Assume moreover that $X = \{ \varphi_i \mid i \in A\}$.

Finite functions are represented by tables for their values; i.e. an array of the mode *ref flex*[ ] *struct(int val,bool def)*. The default value of a value of the mode *(int val,bool def)* equals *(0,false)*.

The way-out strategy is represented by a routine *proc way-out = = (table t) int : ¢ some element i such that φ_i is an extension of the finite function encoded in t ¢;*

For example, the way-out strategy which extends each finite function by the constant value zero is defined by:

```
mode tablit = struct(int val,bool def); pr the default value of a tablit
                                              equals (0,false) pr
mode table = ref flex[ ] tablit;
proc way out = (table t) int:
     index(int x) int:
          (tablit z = look up tablit(t,x);
               (def of z | val of z | 0));
```

The enumeration of the class X is performed by the transformation τ: for each i and j $\varphi_{\tau(i,j)}$ will be a member of X and for each $f \in X$ there exist indices i and j such that $f = \varphi_{\tau(i,j)}$.

```
     proc(int i,x) bool B = ~;
     proc(table t) int way out = ~;

$\varphi_{\tau(i,j)}(x) \Leftarrow$
          (flex[0:0] tablit results;
           int prog := i;
           bool original prog := true;
           for z while not def of look up tablit(results,x) do
             if original prog and z > j and not B(i,z)
                then prog := way out(results); original prog := false fi;
             if not def of look up tablit(results,$\pi_1$z) and $\Phi_{prog}(\pi_1 z) \le z$
                then insert tablit(results,$\pi_1$z,($\varphi_{prog}(\pi_1 z)$,true)) fi
                                                                  od;
           val of look up tablit(results,x));
```

DISCUSSION

This enumeration technique was described in §2.3.2. The way-out strategy is used at most once (because of the boolean original prog). The integral variable prog is initialized at the candidate index i. If the way-out strategy is not used then the function φ_i is enumerated by a dove-tailed computation; otherwise a safe extension of some subfunction in φ_i is enumerated.

Only finite values of φ_{prog} are stored in results. Note that the computation runs independent of the argument x up to the point where $\varphi_{prog}(x)$ is enumerated.

A3. THE OPERATOR GAP ALGORITHM FOR WEAK CLASSES (chapter 3.2)

```
begin
  ¢    The algorithm described in §3.2.3 is a non-terminating algorithm
    which enumerates the graph of t. Clearly this program may be used to
    compute t(x) by inserting into it a suitable exit, which is activated
    if t(x) is enumerated.
         Our formal description follows the informal one in §3.2.3 except
    that the instruction "proceed to stage k+1" is implemented by a recur-
    sive call of the procedure stage.          ¢

mode fun = proc(int) int;
mode operator = proc(fun) fun;
mode acrel = proc(int,int,int) bool;

fun lower bound = ~;
operator gamma = ~;
acrel acceptance = ~;
flex[0:0] int table;
  ¢ it is assumed that lower bound is total, that gamma is total effective
    and satisfies gamma(f) ≥ f and that acceptance computes A(i,x,z) = true
    for some acceptance relation A              ¢

proc extend = (int x,low,fun t,ref[ ] fun tj) void:
  begin int ub = ⌈tj; (ub < x | error);
         tj[0] := (int z) int:
              if z ≤ low then t(z)
              else max(t(z∸1),lower bound(z)+1)
              fi;
  for j from 1 to x do
         tj[j] := (int z) int:
              if z ≤ low then t(z)
              else max(tj[j∸1](z),tj[j](z∸1),gamma(tj[j∸1])(z))
              fi
                          od
  end ¢ extend ¢;

pr the above procedure is illegal in ALGOL 68 since the environ of the
   routine denotations in the above routine text is newer than the environ
   of the names to whom they are assigned pr
```

```
proc support = (int x,fun f,operator g) int:
     begin int ub := 0;
           fun f star = (int i) int: ((i > ub | ub := i); f(i));
               g(f star)(x);
               ub
     end ¢ support ¢;

pr we assume that the operator g works by issuing calls of the function it
   works on, the result being independent of the actual computation in-
   voked by it but dependent only on the value which is delivered pr

proc suponint = (int x,y,fun f,operator g) int:
     begin (x > y | error);
       int out := y+1;
           for z from x to y do
           int p = support(z,f,g); (p > out | out := p)
                                   od;
           out
     end ¢ suponint ¢;

proc domains = (int low, [ ] fun tj) [ ] int:
     begin up = ⌈tj; if ⌊tj > 0 then error fi;
           [0:up] int yj; int last yj := low + 1;
           for j from up by -1 to 0 do
               last yj := yj[j] := suponint(0,last yj,tj[j],gamma)
                                           od;
                    pr this downward loop is assumed to be legitimate, al-
                       though it uses the monadic operator - pr
               yj
     end ¢ domains ¢;

pr domains is the procedure which computes the pointers vj,l in part 2 of
   the informal algorithm pr
```

```
proc entergap = (int nof, low,up,fun bottom,roof) bool:
     ¢ tests whether index nof enters the local gap-section determined by
       bottom and roof over the interval [low+1,up]; the gap-section is as-
       sumed to be closed ¢

     begin bool safe := true;
           for z from low + 1 to up while safe do
            safe ∧:= acceptance(nof,z,bottom(z)) = acceptance(nof,z,roof(z))
                                                       od;
           safe
     end ¢ entergap ¢;

proc unsafegap = (int nof,k,low,[ ]int up,[ ]fun bottom,roof) result:
     ¢ seeks the highest entered gap-section if such a gap-section exists;
       otherwise unsafegap yields true  ¢

     begin if k > ⌈up or k > ⌈bottom or k > ⌈roof then error fi;
           bool untouched := true; int p := k+1;
                for j from k by -1 to 0 while untouched do
                if untouched ∧:= entergap(nof,low,up[j],bottom[j],roof[j])
                then p := p∸1 fi
                                                               od;
                (p=0 | true | p∸1)
     end ¢ unsafegap ¢;

proc stage = (int stage number,last defined,[ ]fun local extension ¢ = tj ¢,
               [ ]int locub ¢ = zk ¢) void:

     begin int st = stage number, y0 = last defined;
           [0:st][0:st+1] int next stage locub;
           [0:st][0:st+1] fun next stage extension;
           [0:st] fun giant;

           for j from 0 to st do

           extend(st+1,locub[j],local extension[j],next stage extension[j]);
           next stage locub[j] := domains(locub[j],next stage extension[j]);
           ¢ informal description 1 and 2 ¢
```

```
        giant[j] := (int z) int:
             if z ≤ y0 or z > locub[j] then skip
             elif j < st and z ≤ locub[j+1]
                then local extension[j+1](z)
             else int great := local extension[j](z)+1;
                for k to st+1 do
                int p = gamma(next stage extension[j,k])(z);
                   (p > great | great := p)
                              od;
                great
             fi
        ¢ informal description 3 ¢
                          od ¢ end for j-loop ¢;
        [0:st] bool safegap;
        for j to st do safegap[j] := true od;
        for nof to st∸1 do
            case unsafegap(nof,st,y0,locub,local extension,giant)
                in bool : skip,
                   int mis : safegap[mis] := false
                out error
            esac        od;
                ¢ informal description 4 and 5 ¢
        int select := 0;
            while not safegap[select]do select +:= 1 od;
            if select > st then error fi;
        ¢ informal description 6 ¢
        for x from y0+1 to locub[select]do
        insert(table,x,local extension[select](x)) od;
        pr if locub[select] ≥ argument looked for then exit fi pr
        stage(sn+1,locub[select],next stage extension[select],
             next stage locub[select])
        ¢ this recursive call takes care of 7 in the informal descrip-
          tion ¢
end ¢ stage ¢;
```

```
start of the algorithm:
      [0:1] fun extensions;
      [0:1] int upper bounds;
      extend(1,0,zero,extensions);
      upper bounds := domains(0,extensions);
      ¢ initialization from informal description ¢
      stage(1,0,extensions,upper bounds)
end ¢ operator-gap algorithm ¢
```

A4. THE UNION ALGORITHM (chapter 3.3)

The union algorithm as described in chapter 3.3 clearly is a sequential implementation of a more general algorithm involving a great deal of synchronization and parallelism. Below we present first a sequential and next a parallel implementation.

```
begin
¢ declarations common to both implementations ¢
mode tablit = struct(int val,bool def);
mode table = ref flex[ ]tablit;
mode sr = struct(int ind,arg,bnd,val);
mode vr = sr;
        pr the default value of a suspect report equals (0,0,0,0) pr
mode acrel = proc(int,int,int) bool;
flex[0:0]tablit tinf;
acrel acceptance = skip;
int index of sequence = skip; ¢ the index t such that $\varphi_t^2(i,x) = t_i(x)$ ¢
flex[0:0]int tcomp, guess;
llsr suspect list := (skip,nil);
llvr violation queue := (skip,nil);
int max comp, max test, max guess, max arg, max ind, max bnd, stage number;
```

```
¢ sequential implementation ¢
proc sequential union algorithm = void: (
proc stage = (int n) void:
begin int t = index of sequence;

l0: max comp := max test := max guess := max arg := max ind := max bnd := n;
l1: insert int(guess,max ind,max guess);
    insert int(tcomp,max ind,0);
l2: for z to max arg do
    int new comp := tcomp[z]; bool new := false;
    for i from new comp to min(z,max bnd) do
        if $\Phi_t^2(i,z)$ ≤ max comp then new comp := i; new := true fi
                                                     od;
    if new then
    for i from tcomp[z]to new comp do
        int val = $\varphi_t^2(i,z)$;
            for j to max ind do
            if guess[j] = i and not acceptance(z,j,val)
               then attach sr(suspect list,(j,z,i,val))
            fi
                                      od
                                            od
    fi
                          od;
l3: for item over suspect list do
    if acceptance(ind of item,arg of item,max test)
    then attach vr(violation queue,item)
    fi                             od;
    for item over violation queue do
    for item 1 over suspect list do
        if ind of item = ind of item 1 then
           delete sr(suspect list,item1)
        fi                            od;
        guess[ind of item] := max guess + 1
                                        od;
```

```
l4: for z to max arg do
      if not def of look up tablit(tinf,z) then
        int viol bnd := max bnd + 1; val := 0;
        for item over violation queue do
            if arg of item = z and bnd of item < viol bnd
            then viol bnd := bnd of item; val := val of item
            fi                           od;
        if viol bnd ≤ max bnd then
        insert tablit(tinf,z,(val,true))
        fi
      fi            od;
l5: clear vr(violation queue); stage(n+1)
end ¢ stage ¢;

start of sequential implementation: stage(0)
¢ end of sequential union algorithm ¢ );
¢ parallel implementation of union algorithm.
    There are three independent sections. The driver adjusts max arg, max
  bnd, max ind and max guess. The enumerator computes values of t_i(x) and
  uses these values to test indices in order to place possible violators
  on the suspect list. The inventor tries to find on the suspect list a
  weak violator and using these weak violators tinf is defined. The array
  guess and the suspect list are shared by the enumerator and the inventor;
  consequently these structures can only be read or written within "criti-
  cal sections". The driver is only activated at a time where both the
  enumerator and the inventor have completed a full turn. ¢
proc parallel union algorithm = void: (
proc something = int: skip + 1;
proc waste time = void:
    ¢ e.g. executes Lucas' test in order to find a new Mersenne prime ¢;
sema inventor sem = level 1, enumerator sem = level 1, guessem = level 1,
                        suspect sem = level 1;
int t = index of sequence;
```

```
proc driver = void:
    do down enumerator sem; down inventor sem;
       max bnd +:= something;
       max arg +:= something; extend int(tcomp,max arg);
                   extend tablit(tinf,max arg);
       int mi = max ind; maxind +:= something; extend int(guess,max ind);
       max guess +:= something;
       int mai = max(max ind,max arg);
       while max guess < mai do max guess +:= something od;
       for i from mi+1 to max ind do guess[i] := max guess od;
       up enumerator sem; up inventor sem;
       waste time
    od ¢ end driver ¢;

proc enumerator = void:
    do down enumerator sem;
       for z to max arg do
       new comp := tcomp[z]; bool new := false;
       for i from new comp to min(z,max bnd) do
           if $\Phi_t^2(i,z) \le$ max comp then new comp := i; new := true fi
                                                   od;
       if new then for i from tcomp[z] to new comp do
          int val = $\varphi_t^2(i,z)$;
          for j to max ind do
          down guessem; int g = guess[j]; up guessem;
               if g = i and not acceptance(z,j,val)
                  then down suspect sem;
                       attach sr(suspect list,(j,z,i,val));
                       up suspect sem
               fi          od                        od
       fi            od;
       max comp +:= something; up enumerator sem
    od ¢ end enumerator ¢;
```

```
proc inventor = void:
    do down inventor sem;
       down suspect sem;
       for item over suspect list do
           if acceptance(ind of item,arg of item,max test)
           then attach vr(violation queue,item)
           fi                    od;
       up suspect sem;

       for item over violation queue do
       down suspect sem;
            for item 1 over suspect list do
                if ind of item = ind of item 1
                then delete sr(suspect list,item 1)
            fi                            od;
       up suspect sem;
       down guessem;
            guess[ind of item] := max guess + 1;
       up guessem
                                     od;

       for z to max arg do
           if not def of tinf[z]
           then int viol bnd := max bnd + 1, val := 0;
                for item over violation queue do
                if arg of item = z and bnd of item < viol bnd
                then viol bnd := bnd of item; val := val of item
                fi                          od;
                if viol bnd ≤ max bnd then tinf[z] := (val,true) fi
           fi           od;
       clear vr(violation queue);
       max test +:= something;
       up inventor sem
    od ¢ end inventor ¢;
start of parallel implementation:
par begin driver,enumerator,inventor end
¢ end of parallel union algorithm ¢ );
pr at this place one of the two union algorithms must be called pr skip
end ¢ union algorithm ¢
```

A5. THE MEYER-McCREIGHT ALGORITHM (chapter 3.4)

In the discussion of the MEYER-McCREIGHT algorithm we indicated a number of variants of such an algorithm. Below we present again a sequential and a parallel implementation, both of which use the strategy against unwanted revisions developed by R. MOLL. Both MMC-algorithms are based upon some general discriminator B; for the classical case, where membership of F(t) is tested, a procedure is given.

As we indicated in §3.4.1 there must be designed some interface between the MMC-algorithm and its subroutine searchtime with respect to the representation of the priority queue. The possible representations are

(i) an infinite array of pairs consisting of a priority number and a boolean

(ii) a queue of pairs consisting of an index and a boolean

(iii) a double representation consisting of both (i) and (ii).

The first representation is easy for the MMC-algorithm as a whole whereas the second one makes it easier to write the subroutine searchtime. Choosing a double administration does not solve the problem since every manipulation on one structure must be repeated for the other structure.

In our implementation we have used the representation by an infinite array. The routine searchtime is equipped with a number or procedures yielding the first or next element in the priority queue.

```
begin ¢ declarations common to both MMC-algorithms ¢
mode priostat = struct(int prio,bool stat);
mode acrel = proc(int,int,int) bool;
mode discr = proc(int,int) bool;
mode quit = struct(int ind,bool stat);
mode candval = struct(int cand,val);
mode quiter = union(quit,bool);
mode candvaler = union(candval,bool);
mode tablit = struct(int val,bool def);

acrel acceptance = skip;
discr B = skip;
int index name = skip; int t = index name;

discr strong class = (int i,x) bool:
($\Phi_t(\pi_1 x)=\pi_2 x$ | acceptance($i,\pi_1 x,\varphi_t(\pi_1 x)$) | true);
¢ tests membership $i \in F(\varphi_t)$ ¢
```

```
flex[0:0]priostat prior;
pr the default value of a priorstat value equals (0,true) pr
int priorcounter := 0;
proc next = int: priorcounter +:= 1;
flex[0:0]tablit t prime;
¢ the procedures first and next take care of the interface inbetween search-
  time and the MMC-algorithm ¢
proc first = ([ ] priostat prior) quiter:
    if ⌈ prior < ⌊ prior then false
    else int cand := ⌊ prior; int height := prio of prior[cand];
         bool colour := stat of prior[cand];
         for i from cand+1 to ⌈ prior do
             if int now = prio of prior[i]; height > now
             then height := now; cand := i; color := stat of prior[i]
             fi                         od;
    (cand,colour)
    fi ¢ end first ¢;

proc next = ([ ]priostat prior,quit this) quiter:
    begin int height; bool not found := true;
          for i from ⌊ prior to ⌈ prior while not found do
              if(i,stat of prior[i]) = this
              then height := prio of prior[i]; not found := false
              fi                                        od;
          if not found then error fi;
          int next height,cand; bool colour,found := false;
          for i from ⌊ prior to ⌈ prior do
          if int p = prio of prior[i]; p > height then
             if not found
             then found := true; next height := p;
                  cand := i; colour := stat of prior[i]
             elif next height > p
             then next height := p; cand := i; colour := stat of prior[i]
             fi
          fi                            od;
    if found then(cand,colour) else found fi
    end ¢ next ¢;
```

```
proc searchtime = (int arg, low,high, [ ]priorstat prior) candvaler:
     begin int val := low; int cand; quit candidate;
           case first(prior)
           in quit yes: candidate := yes
           out goto failure
           esac;
           while val ≤ high do
           if stat of candidate and not acceptance(ind of candidate,arg,val)
           then val +:= 1
           elif acceptance(ind of candidate,arg,val)
           then case next(prior,candidate)
                in quit yes: candidate := yes
                out goto failure
                esac
           else ¢ solution found ¢ goto success
           fi                od;
           sucess: (ind of candidate,val).
           failure: false
     end ¢ searchtime ¢;
```

```
proc sequential MMC algorithm = void: (
proc stage = (int n,discr B) void:
     begin extend priostat(prior,n); prior[n] := (next,skip);
           extend tablit(t prime,n);
           for ind to n do
               if not B(ind,n) then prior[ind] := (next,false)
               fi       od;

           if not def of t prime[$\pi_1 n$]then
           case searchtime($\pi_1 n$,$\pi_2 n$,$\pi_2 n$,prior)
           in candval good:
              (int test := val of good; t prime[$\pi_1 n$] := (test,true);
              for i to n do
                  if not stat of prior[i]and not acceptance(i,$\pi_1 n$,test)
                  then prior[i] := (next,true)
                  fi     od  )
               pr if def of t prime[wanted]
                  then terminate(val of t prime[wanted])
                  fi pr
           esac
           fi;
           stage(n+1,B)
     end ¢ stage ¢;
¢ to start a MMC algorithm to compute a new name for F($\varphi_t$) ¢
     stage(1,strong class)
¢ end sequential MMC-algorithm ¢ );
```

```
proc parallel MMC algorithm = void: (
¢ parallel implementation of MMC-algorithm.
     There are two independently operating sections. The discriminator
tests indices and moves violators to the black list. Also the task of in-
troducing new indices is given to the discriminator. The incriminator issues
calls of searchtime, and tries to invent values for t-prime. Both sections
share the priority queue as a common data structure. This queue is protected
using a semaphore prior sem ¢
proc something = int: skip;
sema prior sem = level 1;
proc discriminator = void:
     do
     for ind to max ind do
         if not B(ind,max ind)
         then down prior sem;
                   prior[ind] := (next,false);
              up prior sem
         fi              od;
         max ind +:= 1;
         down prior sem;
              insert priorstat(prior,max ind,(next,skip));
         up prior sem
     od ¢ end discriminator ¢;
proc incriminator = void:
     do int arg = π_1 trial;
        if def of t prime[arg] then skip
        else int low = last tried[arg]; int high = low + something;
                       last tried[arg] := high + 1
             candval happy;
             down prior sem;
             case searchtime(arg,low,up,prior)
             in candval good: happy := good; up prior sem
             out up prior sem; failure
             esac;
```

```
            int test = val of happy;
            t prime[arg] := (test,true);
            down prior sem; int up := ⌈ prior; up prior sem;
            for i to up do
            down prior sem;
                 if not stat of prior[i]
                    and not acceptance(i,arg,test)
                 then prior[i] := (next,true)
                 fi;
            up prior sem od
        fi;
failure: trial +:= 1;
         extend tablit(t prime,trial);
         extend int(last tried,trial)
     od ¢ end incriminator ¢;
start: int max ind := 0, trial := 0;
       flex[0:0]int last tried;
       par begin discriminator,incriminator end
           ¢ end parallel MMC-algorithm ¢ );

       pr at this place one of the MMC-algorithms must be called pr skip
end ¢ MMC-algorithms ¢
```

A6. THE WEAK MEYER-McCREIGHT ALGORITHM (Chapter 3.4)

For this algorithm we present a parallel implementation. The algorithm is based upon the acceptance relation "acceptance", the discriminator "B" and the wizard "bias". The algorithm consists of three independent sections. The discriminator has the same function as in A5. The incriminator uses the subroutine weak searchtime in order to generate new values of t'. Testing of indices against t' and making white indices out of black ones is performed by a new section called "the judge".

A number of mode declarations, which are equal to the corresponding ones in A5, have been omitted.

```
begin
proc(int,int)bias = skip; acrel acceptance = skip; discr B = skip;
int k = index λx[bias(0,x)] ¢ used to test convergence of bias(0,x) ¢;
¢ int eps = an index whose α-run-time equals ε ¢
int prior counter := 0; proc next = int: prior counter +:= 1;
int time := 0, max test := 0, trial := 0, max ind := 0;
proc something = int: skip;
proc current time = int: time +:= (something + 1);
sema prior sem = level 1, t prime sem = level 1;
flex[0:0]priorstat prior; flex[0:0]tablit t prime;
proc first = ([ ]priorstat prior)quitter: ¢ see A5 ¢;
proc next = ([ ]priorstat prior,quit this)quitter: ¢ see A5 ¢;
flex[0:0]bool test against output;
pr remember that the default value of a boolean value equals false pr


proc weak searchtime =
     (int x,low,high,[ ]priorstat prior,proc(int,int)bool bias)candvaler:
begin if $\Phi_k(x)$ > current time then come back another time fi;
       down prior sem; int up = ⌈ prior;
       [0:up+1]priorstat own prior; own prior[0:up] := prior;
       own prior[up+1] := (next,true);
       up prior sem;
       proc own bias = (int i)bool: if i = up+1 then true else bias(i,x)fi;
       proc own accept = (int i,z)bool: if i = up+1 then false else
                              acceptance(i,x,z)fi;
       ¢ by these redefinitions of prior, bias and acceptance the element
         eps is inserted at the tail of the priority queue ¢

       llint earlier cand := (up+1,nil);
       ¢ this element never gives a solution ¢
       proc increase val = void:
            (val +:= 1;
             for j over earlier cand do
                 if own accept(j,val+1) then cand := j; goto success fi
                                         od);
```

```
    int val := low; quit candidate;
    case first(own prior)
    in quit yes: candidate := yes
    out error ¢ own prior is not empty ¢
    esac;
    int cand := ind of candidate;
    while val ≤ high do
          if stat of candidate
          then if own bias(cand) and not own accept(cand,val)
                then increase val
                else case next(own prior,candidate)
                     in quit yes: candidate := yes
                     out error ¢ the last queue element has infinite
                                   α-run-time ¢
                     esac
                fi
          elif own accept(cand,val)
          then case next(own prior,candidate)
                in quit yes: candidate := yes
                out error ¢ see above ¢
                esac
          elif own accept(cand,val+1)
          then goto success
          else attach int(earlier cand,cand);
                case next(own prior,candidate)
                in quit yes: candidate := yes
                out error ¢ the last item in the queue is white ¢
                esac
          fi              od;
    failure: false.
    come back another time: true.
    success: (cand,val)
end ¢ weak searchtime ¢;
```

```
proc discriminator = void:
do for ind to max ind do
   if not B(ind,max ind)
   then down prior sem;
        prior[ind] := (next,false);
        up prior sem
   fi              od;
   max ind +:= 1;
   down prior sem;
   insert priorstat(prior,max ind,(next,skip));
   up prior sem
od ¢ end discriminator ¢;
proc incriminator = void:
do int arg = π1 trial;
   down t prime sem;
   if def of t prime[arg]
   then up t prime sem
   else up t prime sem;
        case weak searchtime(arg,0,trial,prior,bias)
        in candval good:
           (down t prime sem;
            t prime[arg] := (val of good,true);
            up t prime sem),
            bool mis: skip
        esac
   fi;
   trial +:= 1;
   down t prime sem;
   extend tablit(t prime,trial);
   up t prime sem
od ¢ end incriminator ¢;
```

```
proc judge = void:
do down prior sem; int upp = ⌈ prior; up prior sem;
   down t prime sem; int upt = ⌈ t prime; up t prime sem;
   if <upp,upt> > ⌈ test against output
   then extend bool(test against output, <upp,upt>)
   fi;
   for ind to upp do
   for arg to upt do
   if test against output[<ind,arg>]
   then skip ¢ test was already executed ¢
   else tablit targ; priorstat prind;
        down prior sem; prind := prior[ind]; up prior sem;
        down t prime sem; targ := t prime[arg]; up t prime sem;
        if not def of targ
        then skip ¢ test value not yet available ¢
        elif stat of prind
        then skip ¢ white indices are not tested ¢
        else int val targ = val of targ;
             if acceptance(ind,arg,val targ)
             then test against output[<ind,arg>] := true
             elif acceptance(ind,arg,max test)
             then test against output[<ind,arg>] := true
                  down prior sem;
                  prior[ind] := (next,true);
                  up prior sem
             fi
        fi
   fi
                  od od;
   max test +:= (something + 1)
od ¢ judge ¢;
start of the weak MMC algorithm:
par begin discriminator,incriminator,judge end
end ¢ weak MMC-algorithm ¢
```

REFERENCES

[Al 73] ALTON, D., *Speed-ups and embeddability in Computational Complexity*. Dept. of Comp. Sci., University of Iowa, Iowa City, TR 73-01, 1973.

[Al 73.b] ALTON, D., *Non existence of program optimizers in an abstract setting*. Dept. of Comp. Sci., University of Iowa, Iowa City, TR 73-08, 1973.

[Au 70] AUSIELLO, G., *On the bounds on the number of steps to compute functions*. Proc. 2nd ACM Symp. on the theory of computing, Northampton, Mass., (1970) 41-47.

[Ax 63] AXT, P., *Enumeration and the Grzegorczyk hierarchy*. Z. Math. Logik. Grondlagen Math. 9 (1963) 53-65.

[Ba 70] BASS, L., *Hierarchies based on computational complexity and irregularities of class determining measured sets*. Ph.D. Thesis, Purdue Univ., 1970.

[BCH 69] BORODIN, A., CONSTABLE R.L. & HOPCROFT, J.E., *Dense and nondense families of complexity classes*. Proc. 10th SWAT Symp., Waterloo, Ontario, (1969) 7-19.

[Bl 66] BLUM, M., *Recursive function theory and speed of computation*. Canad. Math. Bull. 9 (1966) 745-750.

[Bl 67] BLUM, M., *A machine-independent theory of the complexity of recursive functions*. J. Assoc. Comput. Mach. 14 (1967) 322-336.

[Bl 71] BLUM, M., *On effective procedures for speeding up algorithms*. J. Assoc. Comput. Mach. 18 (1971) 290-305.

[Bo 72] BORODIN, A., *Computational complexity and the existence of complexity gaps*. J. Assoc. Comput. Mach. 19 (1972) 158-174.

[BY 73] BASS, L. & YOUNG, P., *Ordinal hierarchies and naming complexity classes*. J. Assoc. Comput. Mach. 20 (1973) 668-686.

[Co 72] CONSTABLE, R.L., *The operator gap*. J. Assoc. Comput. Mach. 19 (1972) 175-183.

[Da 58] DAVIS, M.D., *Computability and unsolvability*. McGraw-Hill, New York, 1958.

[Da 73] DAVIS, M.D., *Hilberth tenth problem is unsolvable.* Amer. Math. Monthly 80 (1973) 233-269.

[EB 71] EMDE BOAS, P. van, *A note on the Meyer-McCreight naming theorem in the theory of computational complexity.* Report ZW 7/71, Math. Centrum, Amsterdam, 1971.

[EB 72] EMDE BOAS, P. van, *Gap and operator gap.* Report ZN 42/72, Math. Centrum, Amsterdam, 1972.

[EB 72.b] EMDE BOAS, P. van, *A comparison of the properties of complexity classes and honesty classes.* in: Automata Languages and Programming, M. Nivat (ed.), Proc. IRIA Symp. Rocquencourt, North Holl. Publ. Comp., 1973.

[EB 73] EMDE BOAS, P. van, *Mostowski's universal set algebra.* Report ZW 14/73, Math. Centrum, Amsterdam, 1973.

[EB 73.b] EMDE BOAS, P. van, *The non-renameability of honesty classes.* Preprint:Report ZW 18/73, Math. Centrum, Amsterdam, 1973. (To appear in Computing Arch. Elektron. Rechnen.)

[EB 74] EMDE BOAS, P. van, *Introduction to machine-independent complexity theory; Part I: Complexity measures, Part II: Resource-bound Classes, Part III: Abstract resource-bound classes.* Math. Centre Tracts 61 & 62 (in preparation), Amsterdam, 1974.

[EL 66] Reference Manual ELX8 Chapter A4 (Dutch) ed. Electrologica, 1966.

[En 73] ENGELER, E., *Introduction to the theory of computation.* Ac. Press, New York, 1973.

[FS 72] FELDMAN, J.A. & SHIELDS, P.C., *Total complexity and the inference of best programs.* Standford artificial intelligence project, Memo AIM-159, CSD rep. CS-253, 1972.

[Gz 53] GRZEGORCZYK, A., *Some classes of recursive functions.* Rozprawy Matematyczne 4, Warsaw, (1953) 1-45.

[HH 71] HARTMANIS, J. & HOPCROFT, J.E., *An overview of the theory of computational complexity.* J. Assoc. Comput. Mach. 18 (1972) 444-475.

[Hm 71] HARTMANIS, J., *Computational complexity of random access stored program machines,* Math. Systems Theory 5 (1971) 232-245.

[Hm 73] HARTMANIS, J., *On the problem of finding natural computational complexity measures*. Proc. Symp. and Summer School on Mathematical foundations of computer science, High Tatras, C.S.S.R., (1973) 95-104.

[Hm 73.b] HARTMANIS, J., *Computational complexity of formal translations*. Cornell Dept. of Comp. Sci., TR 73-192, 1973.

[HY 71] HELM, J.P. & YOUNG, P.R., *On size versus efficiency for programs admitting speed-up*. J. Symbolic Logic 36 (1971) 21-27.

[Kl 52] KLEENE, S.C., *Introduction to Metamathematics*. van Nostrand, Princeton, New Jersey, 1952.

[Kn 68] KNUTH, D.E., *The art of computer programming, Vol. 1. Fundamental algorithms*. Addison Wesley, Reading, Mass., 1968.

[Le 70] LEWIS, F.D., *Unsolvability considerations in computational complexity*. Ph.D. Thesis, Cornell University, 1970.

[Le 71] LEWIS, F.D., *The enumerability and invariance of complexity classes*. J. Comput. System Sci. 5 (1971) 286-303.

[LR 72] LANDWEBER, L.H. & ROBERTSON, E.L., *Recursive properties of abstract complexity classes*. J. Assoc. Comput. Mach. 19 (1972) 296-308.

[Ly 72] LYNCH, N.A., *Relativization of the theory of computational complexity*. Ph.D. Thesis, MIT, rep. MAC TR-99, Cambridge, Mass., 1972.

[MC 69] McCREIGHT, E.M., *Classes of computable functions defined by bounds on computation*. Ph.D. Thesis, Carnegy Mellon Univ., 1969.

[MC 69.b] McCREIGHT, E.M., *A note on complex recursive characteristic functions*. Rep. Comp. Sci. Dept., Carnegy Mellon Univ., 1969.

[MCM 69] McCREIGHT, E.M. & MEYER, A., *Classes of computable functions defined by bounds on computation*. ACM Symp. on the theory of computing, Marina del Rey, Cal., (1969) 79-88.

[MF 72] MEYER, A.R. & FISCHER, P.C., *Computational speed-up by effective operators*. J. Symbolic Logic 37 (1972) 55-68.

[MMC 69] MEYER, A.R. & McCREIGHT, E.M., *Properties of bounds on computation*. Proc. 3rd ann. Princeton Conf. on Information Sciences and Systems, (1969) 154-156.

[MMo 72] MEYER, A.R. & MOLL, R., *Honest bounds for complexity classes of recursive functions*. Proc. 13th SWAT Symp. Univ. of Maryland, 1972. (to appear in J. Symbolic Logic.)

[Mo 73] MOLL, R., *Complexity classes of recursive functions*. Ph.D. Thesis, MIT, report MAC TR-110, Cambridge, Mass., 1973.

[Mo 73.b] MOLL, R., *An operator embedding theorem for complexity classes of recursive functions*. MAC. Techn. Mem. 32, 1973.

[MT 72] MILLER, R.E. & J.W. THATCHER (eds.), *Complexity of computer computations*. Plenum Press, New York, 1972.

[Pa 67] PARIKH, R.J., *On non-uniqueness in transfinite progressions*. J. Indian Math. Soc. 31 (1967) 23-32.

[Pe 50] PETER, R., *Rekursive funktionen*. Akademiai Kiado, Budapest, 1950. (English translation, Academic Press, New York, 1967).

[Rb 70] ROBERTSON, E.L., *Properties of complexity classes and sets in abstract complexity theory*. Ph.D. Thesis, Univ. of Wisconsin, 1970.

[Rb 71] ROBERTSON, E.L., *Complexity classes of partial recursive functions*. 3rd ACM Symp. on the theory of computing, Shaker Heights, Ohio, (1971) 258-266.

[Rd 68] RITCHIE, D.M., *Program structure and computational complexity*. Ph.D. Thesis, Harvard Univ., 1968 (unpublished).

[Ro 58] ROGERS, H., *Gödel numberings of partial recursive functions*. J. Symbolic Logic 23 (1958) 331-341.

[Ro 67] ROGERS, H., *The theory of recursive functions and effective computability*. McGraw Hill, New York, 1967.

[Rr 63] RITCHIE, R.W., *Classes of predictably computable functions*. Trans. Amer. Math. Soc. 106 (1963) 139-173.

[Ru 73] RUSTIN, R. ed., *Computational complexity*. Courant Comp. Sci. Symp. 7, Algorithmic Press Inc., New York, 1973.

[SS 63] SHEPHERDSON, J.C. & STURGIS, H.E., *Computability of recursive functions*. J. Assoc. Comput. Mach. 10 (1963) 217-255.

[Sy 71] SYMES, D.M., *The extension of machine independent computational complexity theory to oracle machine computation, and to the computation of finite functions*. Ph.D. Thesis, Univ. of Waterloo, Ontario, CSRR 2057, 1971.

[Yo 71.a] YOUNG, P., *Speed-ups by changing the order in which sets are enumerated*. Math. Systems Theory 5 (1971) 148-156.

[Yo 71.b] YOUNG, P., *A note on dense and non-dense families of complexity classes*. Math. Systems Theory 5 (1971) 66-70.

[Yo 73] YOUNG, P., *Easy constructions in complexity theory: gap and speed-up theorems*. Proc. Amer. Math. Soc. 37 (1973) 555-563.

[Wea 74] WIJNGAARDEN, A. van, e.a. (eds.), *Revised Report on the Algorithmic Language ALGOL 68*. Math. Centre Tracts 50, Amsterdam, (to appear).

SUMMARY

We present a survey of the theory of resource-bound classes in abstract complexity theory within a machine-independent framework.

To be able to discuss algorithms without relying on the informal or semi-formalized descriptions found otherwise, we present a formalism to represent algorithms and expressions denoting computable functions. This formalism, described in chapter 1.1, consists of a high level programming language, extended by a number of primitives needed for discussing the basic concepts of an effective enumeration and a complexity measure. In order to accomodate the mathematical reader, a mathematical style of representation is defined, which makes it possible in a majority of situations to use the same informal expressions which have been traditionally used, by giving these expressions a formalized meaning. In this way a number of implicit ambiguities within the language of abstract complexity theory are eliminated.

After this introduction we present the basic concepts of an effective enumeration (1.2) and a complexity measure (1.3). To illustrate our formalism, and in order to make this publication self-contained, we present some basic facts from recursion theory (1.4) and some elementary results on complexity measures (1.5). The monotonicity lemma in 1.5, showing the existence of programs with increasing run-times for total functions, completely solves the problem whether sufficiently many increasing run-times exist. Part 1 is completed by mentioning the speed-up phenomenon.

The second part gives a survey of the known results on classes of computable functions defined by bounds on computations, the so-called resource-bound classes. The most important types of such classes are the complexity classes (where the bound on the run-time depends on the argument only) and the honesty classes (where the maximally allowable run-time depends both on the argument and on the computed value). In particular, the results on the behaviour of honesty classes compared with the known properties of complexity classes, are new.

Following the definitions of the classes (2.1) we discuss diagonalization techniques (2.2), enumerability properties (2.3) and set theoretical closure properties (2.4).

In part 3 the resource-bound classes mentioned above are reconsidered from a more general point of view. To explain the different behaviour of complexity classes and honesty classes, given their similarity of defini-

tion, we introduce the concept of an acceptance relation, together with a corresponding measured set of generalized run-times. It is argued that there exist two different ways in which some function, used as a name of a class, restricts membership in this class, depending on whether divergence of the run-time is felt to be in violation of the restriction (strong restriction) or not (weak restriction).

Relative to a given acceptance relation we define in this way for each partial recursive function a corresponding strong (weak) class; these classes will be called abstract resource-bound classes hereafter.

The difference between complexity classes and honesty classes originates from the fact that classes of the former type are examples of strong classes whereas the latter type consists of weak classes.

In the sequel of Part 3 we discuss those results on resource-bound classes which can easily be treated within the language of abstract resource-bound classes. Chapter 3.2 contains the proofs of the gap theorem and the operator-gap theorem for both strong and weak classes, yielding the operator-gap theorem for honesty classes as a corollary. In chapter 3.3 we discuss the union theorem, which is proved for both strong and weak classes, where, moreover, the functions in the increasing sequence of names are allowed to be partial (using the extra condition that the domains of the functions form a decreasing sequence of sets). Again the union theorem for honesty classes is a straightforward corollary.

In chapter 3.4 we discuss the MEYER-McCREIGHT naming theorem, which claims the existence of a measured transformation of programs renaming all complexity classes. This theorem, in combination with the compression theorem, yields a method to extend uniformly complexity classes by operating on programs for their names. This is interesting since the gap theorems show that no uniform extension by operation on the names themselves is possible. In contrast to the situation in the preceding two chapters, this theorem cannot be generalized for weak classes; instead we prove that each measured transformation fails to rename correctly at least one honesty class.

Section 3.4.3 contains a number of modifications of the MEYER-McCREIGHT algorithm which is used in the proof of the naming theorem. The first modification enables us to rename a strong class by two names having disjoint domains. The second modification yields a further generalization of the union theorem for strong classes, where the monotonicity of the sequence of names is replaced by the monotonicity of the sequence of classes of indices c.q. programs.

In section 3.4.4 we discuss how the MEYER-McCREIGHT algorithm can be used to compute a name for the smallest strong class containing a Σ_2-class which is given to us by means of some "almost everywhere"-condition. Although the computed name highly depends on the precise MEYER-McCREIGHT algorithm used, the strong class named by this function is characterized in terms of the given Σ_2-class.

The theory of these last two sections is generalized for weak classes in section 3.4.5. As follows from the negative results in 3.4.2, the MEYER-McCREIGHT algorithm behaves badly for weak classes; however, its renaming properties may be preserved by equipping it with a so-called "wizard" which guesses by means of prejudice which run-times are finite and which ones diverge. The weak MEYER-McCREIGHT algorithm, constructed this way, yields a further generalization of the union theorem for weak classes, and a proof that the intersection of two honesty classes of programs with partial names is again an honesty class; the latter result cannot be derived in the usual way by taking the minimum of the two names since this no longer needs to be a computable function.

In the Appendix we formally represent a number of the more complicated algorithms by means of programs written in the programming language described in chapter 1.1.

SAMENVATTING

Dit proefschrift geeft, in het kader der abstracte complexiteitstheorie, een overzicht van de theorie van de klassen van functies die gedefinieerd worden in termen van begrenzingen op de rekentijd, die we bij gebrek aan ingeburgerde Nederlandse terminologie "verbruiksklassen" zullen noemen. We werken op een machine-onafhankelijke basis.

Om te kunnen praten over algoritmen zonder te hoeven vervallen in het gebruikelijke, informele of half-formele taalgebruik, ontwikkelen we een formalisme om algoritmen, en expressies die een berekenbare functie voorstellen, te beschrijven. Dit formalisme is gebaseerd op een hogere orde programmeertaal, voorzien van enkele primitiva, nodig om de structuur van een effectieve enumeratie van recursieve functies, of een complexiteitsmaat, formuleerbaar te maken. Om het de wiskundige lezer echter niet al te moeilijk te maken, definieren we ook een wiskundige representatiestijl voor deze programmeertaal. Hiermee vangen we twee vliegen in een klap: we kunnen in een groot aantal gevallen volstaan met de expressies die ingeburgerd zijn voor simpele berekenbare functies, en bovendien krijgen dezelfde expressies een geformaliseerde betekenis. Op deze wijze verdwijnen bovendien een aantal impliciet aanwezige ambiguiteiten uit de taal der abstracte complexiteitstheorie.

De basisbegrippen van een effectieve enumeratie en een complexiteitsmaat worden gedefinieerd in hoofdstuk 1.2 resp. 1.3. Hoofdstuk 1.4 bevat (ten behoeve van de zelfstandigheid van dit proefschrift en ter illustratie van ons formalisme) enkele basisstellingen uit de recursietheorie, en in hoofdstuk 1.5 behandelen we de elementaire theorie der complexiteitsmaten. In dit laatste hoofdstuk lossen we het probleem op of er voldoende stijgende rekentijden bestaan, door aan te tonen, dat iedere totale functie zich laat berekenen met behulp van een programma met stijgende rekentijd. Een korte toelichting van de versnellings-stelling in hoofdstuk 1.6 besluit het eerste deel.

In het tweede gedeelte vertellen we wat er bekend is over verbruiksklassen. De belangrijkste typen verbruiksklassen zijn de zo te noemen complexiteitsklassen (waar de begrenzing op de rekentijd slechts van het argument afhangt) en de "gelijkmatigheidsklassen" (waar de begrenzing zowel bepaald wordt door het argument als door de berekende waarde). De hier gepresenteerde resultaten over gelijkmatigheidsklassen, in vergelijking met die over complexiteitsklassen, zijn nieuw.

Na de definities (2.1) volgen achtereenvolgens diagonalisatie-technieken (2.2), opsombaarheids-kwesties (2.3) en verzamelingstheoretische aspecten van de verbruiksklassen (2.4).

In deel 3 bekijken we de verbruiksklassen vanuit een algemener standpunt. Hoe is het mogelijk dat, ondanks een zichtbare analogie in definities van complexiteits- en gelijkmatigheidsklassen, de twee types een verschillend gedrag vertonen? Om dit te verklaren voeren we het begrip "acceptatie-relatie" in met een daarbij behorende "opvraagbare rij" van gegeneraliseerde rekentijden. We maken aannemelijk dat er twee verschillende manieren zijn waarop een functie op kan treden als naam van een klasse, afhankelijk of het divergeren van de rekentijd beschouwd wordt als een schending van de begrenzingsconditie (sterke begrenzing), of niet (zwakke begrenzing). Met betrekking tot een gegeven acceptatierelatie definieren we voor iedere partieel recursieve functie een bijbehorende sterke resp. zwakke klasse. Deze klassen noemen we in het vervolg abstracte verbruiksklassen.

Het verschil tussen complexiteits- en gelijkmatigheidsklassen is nu geheel en al te verklaren, door op te merken, dat eerstgenoemde klassen sterk begrensd zijn, terwijl de laatstgenoemde klassen een voorbeeld zijn van zwak begrensde klassen.

In de rest van deel 3 komen die resultaten uit de abstracte complexiteitstheorie aan de orde, die zich goed laten behandelen in de taal der abstracte verbruiksklassen. Hoofdstuk 3.2 bevat het bewijs van de "gatenstelling" en de "operator-gatenstelling" voor zowel de sterke als zwakke verbruiksklassen; geldigheid van de laatste stelling voor gelijkmatigheidsklassen is een direct gevolg.

In hoofdstuk 3.3 behandelen we de verenigingsstelling. Deze laat zich generaliseren tot zwakke klassen; bovendien mogen de functies in de stijgende rij namen partieel zijn, mits wordt aangenomen dat de bijbehorende rij definitie-gebieden een krimpende rij verzamelingen is. Ook deze stelling is nu voor gelijkmatigheidsklassen bewezen.

Hoofdstuk 3.4 handelt over de omnoemings-stelling van MEYER en McCREIGHT, die uitspreekt dat alle complexiteitsklassen op uniforme wijze kunnen worden voorzien van een opvraagbare rij namen. Deze stelling ontleent haar belang aan het feit dat zij, in samenwerking met de "compressie-stelling", de mogelijkheid geeft op uniforme wijze complexiteitsklassen te vergroten door middel van operaties op programma's voor de namen van deze klassen. Men dient hierbij te beseffen dat een zodanige vergroting door middel van operaties op de namen zelf op grond van de gatenstellingen onmogelijk is. In tegenstelling tot de voorafgaande stellingen laat deze stelling zich niet generalizeren voor zwakke klassen; in tegendeel: iedere transformatie van programma's die een opvraagbare rij namen geeft is foutief voor minstens één gelijkmatigheidsklasse (3.4.2).

Paragraaf 3.4.3 bevat een aantal varianten van de Algoritme van MEYER en McCREIGHT, die gebruikt wordt om de omnoemings-stelling te bewijzen. De eerste variant laat zien hoe een naam van een sterke verbruiksklasse zich laat transformeren tot een paar namen voor dezelfde klassen die disjuncte definitie-gebieden hebben. De tweede variant levert een generalisatie van de verenigings-stelling, waarbij de conditie, dat de namen een monotone rij vormen, is vervangen door de monotonie van de rij verbruiksklassen van rangnummers resp. programma's.

Paragraaf 3.4.4 laat zien hoe de MEYER-McCREIGHT algoritme ons in staat stelt een naam te berekenen voor de kleinste sterke verbruiksklasse die een Σ_2-klasse omvat, die gedefinieerd is m.b.v. een "bijna overal"-criterium. Hoewel de berekende naam afhangt van de gebruikte algoritme, laat de bijbehorende sterke klasse zich helemaal definieren in termen van de gegeven Σ_2-klasse.

De theorie uit de laatste twee paragrafen wordt gegeneraliseerd voor zwakke klassen in paragraaf 3.4.5. Het negatieve resultaat uit paragraaf 3.4.2 laat zien dat de MEYER-McCREIGHT algoritme niet werkt voor zwakke klassen. We kunnen echter, door de algoritme te voorzien van een "waarzegger" die (op grond van ingeworteld vooroordeel) gokt of een rekentijd al dan niet divergeert, een algoritme construeren, die de omnoemings-eigenschappen van de MEYER-McCREIGHT algoritme bewaart. Dit geeft aanleiding tot een verdere generalisatie van de verenigings-stelling voor zwakke klassen en een bewijs dat de doorsnede van twee gelijkmatigheidsklassen met partiele namen een gelijkmatigheidsklasse is; dit laatste laat zich niet bewijzen door (als gewoonlijk) het minimum van twee namen te trekken, omdat dit een niet-recursieve functie kan zijn.

In de Appendix geven we voor enkele der meer ingewikkelde algoritmen programma's, geschreven in onze programmeertaal uit hoofdstuk 1.1.

Bij het schrijven van deze samenvatting is de volgende terminologie in het Nederlands ingevoerd:

(Abstract) Resource-Bound class	(Abstracte) Verbruiksklasse
Honesty class	Gelijkmatigheidsklasse
Measured Set	Opvraagbare rij
Acceptance relation	Acceptatierelatie
Run-time	Rekentijd
Index	Rangnummer
Gap Theorem	Gatenstelling
Naming Theorem	Omnoemings-stelling

Een aantal andere nieuwe begrippen werd verkregen door woordelijke vertaling van het Engelse equivalent.

A COMPARISON OF THE PROPERTIES OF COMPLEXITY CLASSES AND HONESTY CLASSES

	Complexity classes	Honesty classes
Definition	$F_t = \{\varphi_i \mid \overset{\infty}{\forall} x[t(x) < \infty \Rightarrow \Phi_i(x) \le t(x)]\}$. $C_t = \{f \mid \exists i[i \in F_t \ \underline{\text{and}}\ \varphi_i = f]\}$.	$G_R = \{\varphi_i \mid \overset{\infty}{\forall} x[(\varphi_i(x) < \infty \ \underline{\text{and}}\ R(x,\varphi_i(x)) < \infty) \Rightarrow \varphi_i(x) \le R(x,\varphi_i(x))]\}$. $H_R = \{f \mid \exists i[i \in G_R \ \underline{\text{and}}\ \varphi_i = f]\}$.
Arithmetical Complexity	For sufficiently large total t, F_t is Σ_2-complete. For sufficiently large t, C_t is recursive presentable .	For all total R, G_R is Σ_2-complete. For all R, H_R is recursive presentable. If R is total, then H_R is presented by a measured set .
Weak Compression theorem	$\exists_K \quad \forall i[\varphi_i \text{ total} \Rightarrow C_{\varphi_i} \subsetneqq C_{\lambda x[K(x,\Phi_i(x))]}]$.	$\exists_K \quad \forall i[\varphi_i^2 \text{ total} \Rightarrow H_{\varphi_i^2} \subsetneqq H_{\lambda xy[K(x,y,\Phi_i^2(x,y))]}]$.
Gap theorem	For every total K with $K(x,y) \ge y$ there exists a total t with $F_t = F_{\lambda x[K(x,t(x))]}$ $\underline{\text{and}}$ $C_t = C_{\lambda x[K(x,t(x))]}$.	For every total K with $K(x,y,z) \ge z$ there exists a total with $G_R = G_{\lambda xy[K(x,y,R(x,y))]}$ $\underline{\text{and}}$ $H_R = H_{\lambda xy[K(x,y,R(x,y))]}$.
Operator-Gap theorem	For every total effective operator Γ with $\Gamma(t)(x) \ge t(x)$ there exists a total t with $F_t = F_{\Gamma(t)}$ $\underline{\text{and}}$ $C_t = C_{\Gamma(t)}$.	For every total effective operator Γ with $\Gamma(R)(x,y) \ge R(x,y)$ there exists a total R with $G_R = G_{\Gamma(R)}$ $\underline{\text{and}}$ $H_R = H_{\Gamma(R)}$.
Union theorem	For every sequence $(t_i)_i$ such that $t_i(x) \le t_{i+1}$ there exists a t_{inf} such that $\bigcup_i F_{t_i} = F_{t_{inf}}$ $\underline{\text{and}}$ $\bigcup_i C_{t_i} = C_{t_{inf}}$. For every sequence $(t_i)_i$ such that $F_{t_i} \subseteqq F_{t_{i+1}}$ there exists a t_{inf} such that $\bigcup_i F_{t_i} = F_{t_{inf}}$ $\underline{\text{and}}$ $\bigcup_i C_{t_i} = C_{t_{inf}}$.	For every sequence $(R_i)_i$ such that $R_i(x,y) \le R_{i+1}(x,y)$ there exists an R_{inf} such that $\bigcup_i G_{R_{inf}} = G_{R_{inf}}$ $\underline{\text{and}}$ $\bigcup_i H_{R_i} = H_{R_{inf}}$. For every sequence $(R_i)_i$ such that $G_{R_i} \subseteqq G_{R_{i+1}}$ and such that the R_i are total there exists an R_{inf} such that $\bigcup_i G_{R_i} = G_{R_{inf}}$ $\underline{\text{and}}$ $\bigcup_i H_{R_i} = H_{R_{inf}}$.
Naming theorem	There exists a transformation of programs σ such that $(\varphi_{\sigma(i)})_i$ is a measured set and $\forall i[F_{\varphi_i} = F_{\varphi_{\sigma(i)}}$ $\underline{\text{and}}$ $C_{\varphi_i} = C_{\varphi_{\sigma(i)}}]$.	For every transformation of programs σ such that $(\varphi^2_{\sigma(i)})_i$ is a measured set there exists a φ_e^2 such that $H_{\varphi_e^2} \ne H_{\varphi^2_{\sigma(e)}}$ $\underline{\text{and}}$ $G_{\varphi_e^2} \ne G_{\varphi^2_{\sigma(e)}}$.