CWI Tracts

Managing Editors

J.W. de Bakker (CWI, Amsterdam)
M. Hazewinkel (CWI, Amsterdam)
J.K. Lenstra (CWI, Amsterdam)

Editorial Board

W. Albers (Maastricht)
P.C. Baayen (Amsterdam)
R.T. Boute (Nijmegen)
E.M. de Jager (Amsterdam)
M.A. Kaashoek (Amsterdam)
M.S. Keane (Delft)
J.P.C. Kleijnen (Tilburg)
H. Kwakernaak (Enschede)
J. van Leeuwen (Utrecht)
P.W.H. Lemmens (Utrecht)
M. van der Put (Groningen)
M. Rem (Eindhoven)
A.H.G. Rinnooy Kan (Rotterdam)
M.N. Spijker (Leiden)

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The CWI is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

CWI Tract 19

Foundations and applications of Montague grammar
Part 1: Philosophy, framework, computer science

T.M.V. Janssen



Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

1980 Mathematics Subject Classification: 08A99, 03G15, 68F05, 03B15, 68F20.
1982 CR Categories: F.3.2, I.2.7, F.3.1, F.3.0, F.4.3.
ISBN 90 6196 292 7

Copyright © 1986, Mathematisch Centrum, Amsterdam
Printed in the Netherlands

PREFACE

The present volume is one of the two tracts which are based on my dissertation 'Foundations and applications of Montague grammar'. Volume 1 consists of the chapters 1,2,3 and 10 of that dissertation, and volume 2 of the chapters 4-9. Only minor corrections are made in the text. I would like to thank here again everyone who I acknowledged in my dissertation, in particular my promotor P. van Emde Boas, co-promotor R. Bartsch, and coreferent J. van Benthem. For attending me on several (printing-)errors in my dissertation I thank Martin van de Berg, Cor Baayen, Biep Durieux, Joe Goguen, Fred Landman and Michael Moortgat, but in particular Herman Hendriks, who suggested hundreds of corrections. The illustrations are made by Tobias Baanders.

The two volumes present an interdisciplinary study between mathematics, philosophy, computer science, logic and linguistics. No knowledge of specific results in these fields is presupposed, although occasionally terminology or results from them are mentioned. Throughout the text it is assumed that the reader is acquainted with fundamental principles of logic, in particular of model theory, and that he is used to a mathematical kind of argumentation. The contents of the volumes have a lineair structure: first the approach is motivated, next the theory is developed, and finally it is applied. Volume 1 contains an application to programming languages, whereas volume 2 is devoted completely to the consequences of the approach for natural languages.

The volumes deal with many facets of syntax and semantics, discussing rather different kinds of subjects from this interdisciplinary field. They range from abstract universal algebra to linguistic observations, from the history of philosophy to formal language theory, and from idealized computers to human psychology. Hence not all readers might be interested to read everything. Readers only interested in applications to computer science might restrict them selves to volume 1, but then they will miss many arguments in volume 2 which are taken from computer science. Readers only interested in applications to natural language might read chapters 1-3 of volume 1, and all of volume 2, but they will miss several remarks about the connection between the study of the semantics of programming languages and of the semantics of natural languages. Readers familiar with Montague grammar, and mainly interested in practical consequences of the approach, might read chapters 1 and 2 in volume 1 and chapters 6-10 in volume 2, but they will

miss new arguments and results concerning many aspects of Montague grammar.

Each chapter starts with an abstract. Units like theorems etc. are numbered (eg 2.3 Theorem). Such a unit ends where the next numbered unit starts, or where the end of the unit is announced (2.3 end). References to collected works are made by naming the first editor. Page numbers given in the text refer to the reprint last mentioned in the list of references, except in case of some of Frege's publications (when the reprint gives the original numbering).

CONTENTS

I.	The principle of compositionality of meaning	1
	1. An attractive principle	2
	2. Frege and the principle	5
	2.1. Introduction	5
	2.2. Grundlagen	6
	2.3. Sinn und Bedeutung	8
	2.4. The principle	9
	2.5. Conclusion	11
	3. Towards a formalization	11
	4. An algebraic framework	17
	5. Meanings	28
	5.1. Introduction	28
	5.2. Natural language	29
	5.3. Programming Language	29
	5.4. Predicate Logic	30
	5.5. Strategy	34
	5.6. Substitutional Interpretation	34
	6. Motivation	35
II.	The algebraic framework	41
	1. Introduction	42
	2. Algebras and subalgebras	43
	3. Algebras for syntax	50
	4. Polynomials	56
	5. Term algebras	61
	6. Homomorphisms	67
	7. A safe deriver	75
	8. Montague grammar	81
	9. Discussion	90
III.	Intensional logic	95
	1. Two facets	96
	1.1. Introduction	96
	1.2. Model-part I	96
	1.3. Model-part II	98
	1.4. Laws	98
	1.5. Method	99
	2. Two-sorted type theory	100
	3. The interpretation of Ty2	103
	4. Properties of Ty2	106
	5. Intensional Logic	113
	6. Properties of IL	117
	7. Extension and intension	123

IV Montague grammar and programming languages 127
1. Assignment statements 128
1.1. Introduction 128
1.2. Simple assignments 129
1.3. Other assignments 131
2. Semantics of programs 133
2.1. Why? 133
2.2. How? 135
3. Predicate transformers 137
3.1. Floyd's forward predicate transformer 137
3.2. Hoare's backward predicate transformer 139
3.3. Problems with Floyd's rule 139
3.4. Predicate transformers as meanings 141
4. Semantical Considerations 144
4.1. The model 144
4.2. The logic 148
4.3. Theorems 150
5. First fragment 152
5.1. The rules 152
5.2. Examples 154
6. Pointers and arrays 156
6.1. Pointers 156
6.2. Arrays 158
7. Second fragment 161
7.1. The rules 161
7.2. The postulates 164
7.3. A model 166
8. Correctness and completeness 168
8.1. State transition semantics 168
8.2. Strongest postconditions 169
8.3. Completeness 172
9. The backward approach 176
9.1. Problems with Hoare's rule 176
9.2. Backward predicate transformers 177
9.3. Weakest preconditions 178
9.4. Strongest and weakest 179
9.5. Correctness proof 182
10. Mutual relevance 185

Appendix Safe and polynomial 189

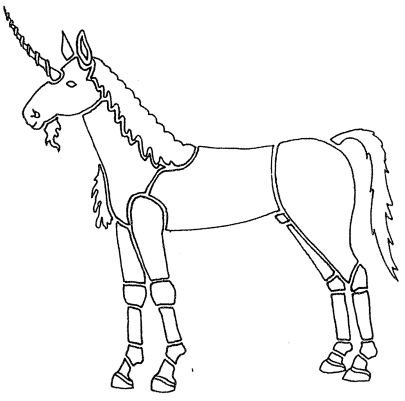
Index of names 193
References 197

CHAPTER I

THE PRINCIPLE OF COMPOSITIONALITY OF MEANING

ABSTRACT

This chapter deals with various aspects of the principle of compositionality of meaning. The role of the principle in the literature is investigated, and the relation of the principle to Frege's works is discussed. A formalization of the principle is outlined, and several arguments are given in support of this formalization.



1. AN ATTRACTIVE PRINCIPLE

The starting point of the investigations in this book is the principle of compositionality of meaning. This principle says:

The meaning of a compound expression
is built up from the meanings of its parts.

This is an attractive principle which pops up at a diversity of places in the literature. The principle can be applied to a variety of languages: natural, logical and programming languages. I would not know of a competing principle. In this section the attractiveness of the principle will be illustrated by means of many quotations.

In the philosophical literature the principle is well known, and generally attributed to the mathematician and philosopher Gottlob Frege. An example is the following quotation. It gives a formulation of the principle which is about the same as the formulation given above. THOMASON (1974,p.55) says:

> *Sentences [..] such as 'The square root of two is irrational', and 'Two is even', [..] ought to be substitutable salva veritate in all contexts obeying Frege's principle that the meaning of a phrase is a function of the meanings of its parts.*

Another illustration of the fame of the principle is given by DAVIDSON (1967, p.306):

> *If we want a theory that gives the meaning (as distinct from reference) of each sentence, we must start with the meaning (as distinct from reference) of the parts.*

Next he says:

> *Up to here we have been following Frege's footsteps; thanks to him the path is well known and even well worn.*

Popper mentions a version of the principle which applies to whole theories (POPPER 1976, p.22):

> *[..] the meaning of a theory [..] is a function of the meanings of the words in which the theory is formulated.*

Thereafter he says (ibid. p.22):

> *This view of the meaning of a theory seems almost obvious; it is widely held, and often unconsciously taken for granted.*

(For completeness of information: there is, according to Popper, hardly any truth in the principle). Concerning the origin of the principle, Popper remarks in a footnote (ibid. p.198):

> *Not even Gottlob Frege states it quite explicitly, though this doctrine is certainly implicit in his 'Sinn und Bedeutung', and he even produces there arguments in its support.*

In the field of semantics of natural languages, the principle is found implicitly in the works of Katz and Fodor concerning the treatment of semantics in transformational grammars. An explicit statement of the principle is KATZ (1966, p.152):

> *The hypothesis on which we will base our model of the semantic component is that the process by which a speaker interprets each of the infinitely many sentences is a compositional process in which the meaning of any syntactically compound constituent of a sentence is obtained as a function of the meanings of the parts of the constituent.*

Katz does not attribute the principle to Frege; his motivation is of a technical nature (ibid. p.152):

> *Accordingly, we again face the task of formulating an hypothesis about the nature of a finite mechanism with an infinite output.*

The principle is mentioned explicitly in important work on the semantics of natural languages by logicians, and it is related there with Frege. Cresswell develops a mathematical framework for dealing with semantics, and having presented his framework he says (CRESSWELL 1973, p.19):

> *These rules reflect an important general principle which we shall discuss later under the name 'Frege's principle', that the meaning of the whole sentence is a function of the meanings of it parts.*

For another logician, Montague, the principle seems to be a line of conduct (MONTAGUE 1970a, p.217):

> *Like Frege, we seek to do this [..] in such a way that [..] the assignment to a compound will be a function of the entities assigned to its components [..].*

The principle is implicitly followed by all logic textbooks when they define, for instance, the truth value of $\underline{p} \wedge \underline{q}$ as a function of the truth-values of $\underline{p}$ and of $\underline{q}$. In logic the enormous technical advantages of treating semantics in accordance with the principle are demonstrated frequently. For instance, one may use the power of induction: theorems with a semantic content can be proven by using induction on the construction of the expression under consideration. Logic textbooks usually do not say much about the motivation for their approach or about the principles of logic. In any case, I have not succeeded in finding a quotation in logic textbooks concerning the background of their compositional approach. Therefore, here is one from another source. In a remark concerning the work of Montague, Partee says the following about the role of compositionality in logic (PARTEE 1975, p.203):

> *A central working premise of Montague's theory [..] is that the syntactic rules that determine how a sentence is built up out of smaller syntactic parts should correspond one-to-one with the semantic rules that tell how the meaning of a sentence is a function of the meanings*

> *of its parts. This idea is not new in either linguistics or philosophy; in philosophy it has its basis in the work of Frege, Tarski, and Carnap, and it is standard in the treatment of formalized languages* [..].

Since almost all semantic work in mathematical logic is based upon Tarski, mathematical logic is indirectly based upon this principle of compositionality of meaning.

In the field of semantics of programming languages compositionality is implicit in most of the publications, but it is mentioned explicitly only by few authors. In a standard work for the approach called 'denotational semantics', the author says (STOY 1977, pp.12-13):

> *We give 'semantic valuation functions' which map syntactic constructs in the program to the abstract values (numbers, truth values, functions etc.) which they denote. These valuation functions are usually recursively defined: the value denoted by a construct is specified in terms of the values denoted by its syntactic subcomponents* [..].

It becomes clear that this aspect is a basic principle of this approach from a remark of Tennent in a discussion of some proposals concerning the semantics of procedures. Tennent states about a certain proposal the following (NEUHOLD 1978,p.163).

> *Your first two semantics are not 'denotational' in the sense of Scott/Strachey/Milner because the meaning of the procedure call construct is not defined in terms of the meanings of its components; they are thus partly operational in nature.*

Milner explicitly mentions compositionality as basic principle (MILNER 1975, p.167):

> *If we accept that any abstract semantics should give a way of composing the meanings of parts into the meaning of the whole* [..].

As motivation for this approach, he gives a very practical argument (ibid. p.158):

> *The designer of a computing system should be able to think of his system as a composite of behaviours, in order that he may factor his design problem into smaller problems* [..].

Mazurkiewics mentions naturalness as a motivation for following the principle (MAZURKIEWICS 1975, p.75).

> *One of the most natural methods of assigning meanings to programs is to define the meaning of the whole program by the meanings of its constituents* [..].

We observe that the principle arises in connection with semantics in many fields. In the philosophical literature the principle is almost always attributed to Frege, whereas in the fields of programming and natural language semantics this is not the case. Authors in these fields give a practical motivation for obeying the principle: one whishes to deal with an

infinity of possibilities in some reasonable, practical, understandable, and (therefore) finite way.

2. FREGE AND THE PRINCIPLE

2.1. Introduction

In the previous section we observed that several philosophers attribute the principle of compositionality to Frege. But it is not made clear what the relationship is of the principle to Frege, and especially on what grounds the principle is attributed to him.

In his standard work on Frege, Dummett devotes a chapter to 'Some theses of Frege on sense and reference'. The first thesis he considers is (DUMMETT 1973, p.152):

> *The sense of a complex is compounded out of the senses of the constituents.*

Since sense is about the same as meaning (this will be explained later), the thesis expresses the principle of compositionality of meaning. Unfortunately, Dummett does not relate this thesis to statements in the work of Frege, so it remains unclear on what writings the claim is based that it is a thesis of Frege. The authors quoted in the previous section who attribute the principle to Frege, do not refer to his writings either.

In the previous section we met a remark by Popper stating that the principle is not explicit in Frege's work, but that it is certainly implicit. The connection with Frege is, according to Cresswell, even looser. He says (CRESSWELL 1973, p.75):

> *For historical reasons we call this Frege's principle. This name must not be taken to imply that the principle is explicitly stated in Frege.*

And in a footnote he adds to this:

> *The ascription to Frege is more a tribute to the general tenor of his views on the analysis of language.*

However, Creswell does not explain these remarks any further. So we have to conclude that the literature gives no decisive answer to the question what the relationship is of the principle to Frege. I will try to answer the question by considering Frege's publications and investigating what he explicitly says about this subject.

2.2. Grundlagen

The study of Frege's publications brings us to the point of terminology. Frege has introduced some notions associated with meaning, but his terminology is not the same in all his papers. Dummett says about 'Die Grundlagen der Arithmetik' (FREGE 1884) the following (DUMMETT 1973, p.193):

> *When Frege wrote 'Grundlagen', he had not yet formulated his distinction between sense and reference, and so it is quite possible that the words 'Bedeutung' and 'bedeuten', as they occur in the various statements [..] have the more general senses of 'meaning' and 'mean' [..].*

This means that in 'Grundlagen' we have to look for Frege's remarks concerning the 'Bedeutung' of parts. He is quite decided on the role of their Bedeutung (FREGE 1884, p.XXII):

> *Als Grundsätze habe ich in dieser Untersuchung folgende festgehalten: [..] nach der Bedeutung der Wörter muss in Satzzusammenhänge, nicht in ihrer Vereinzelung gefragt werden [..]*

He also says (FREGE 1884, p.73):

> *Nur im Zusammenhange eines Satzes bedeuten die Wörter etwas.*

Remarks like these ones are repeated, heavily underlined, several times in 'Grundlagen' (e.g. on p.71 and p.116). The idea expressed by them is sometimes called the principle of contextuality. Contextuality seems to be in conflict with compositionality for the following reason. The principle of compositionality requires that words in isolation have a meaning, since otherwise there is nothing from which the meaning of a compound expression can be built. A principle of contextuality would deny that words in isolation have a meaning.

Dummett discusses the remarks from 'Grundlagen', and he provides an interpretation in which they are not in conflict with compositionality (DUMMETT 1973, pp.192-196). A summary of his interpretation is as follows. The statements express that it has no significance to consider first the meaning of a word in isolation, and next some unrelated other question. Speaking about the meaning of a word makes only significance as preparation for considering the meaning of a sentence. The meaning of a word is determined by the role it plays in the meaning of the sentence.

Following Dummett's interpretation, the remarks from Grundlagen have not to be considered as being in conflict with the principle of compositionality. It is quite well possible to build the meaning of a sentence from the meanings of its parts, and to base at the same time the judgements about the meanings of these parts on the role they play in the sentences in which they may occur. As a matter of fact, this approach is often

followed (for instance in the field of Montague grammar). In this way a bridge is laid between compositionality and contextuality. But even with his interpretation, the statements formulated in Grundlagen cannot be considered as propagating compositionality: nothing is said about building meanings of sentences from meanings of words.

Dummett's interpretation weakens the statements from 'Grundlagen' considerably. Unfortunately, Dummett hardly explains on which grounds he thinks that his interpretation coincides with Frege's intentions when writing 'Grundlagen'. He provides, for instance, no references to writings of Frege. I tried to do so, but did not find passages supporting Dummett's opinion. Dummett makes the remark that statements like the ones from 'Grundlagen' make no subsequent appearence in Frege's works. This is probably correct with respect to Frege's published works, but I found some statements which are close to those 'Grundlagen' in Frege's correspondence and in his posthumous writings. They do not express the whole context principle, but repeat the relevant aspect: that expressions outside the context of a sentence have no meaning. In a letter to E.V. Huntington, probably dating from 1902, Frege says the following (GABRIEL 1976, p.90).

> *Solche Zeichenverbindungen wie* "a+b", "f(a,b)" *bedeuten also nichts, und haben für sich allein keinen Sinn* [..]

In 'Einleitung in die Logik', dating from 1906, he says (HERMES 1969, p.204):

> *Durch Zerlegung der singulären Gedanken erhält man Bestandteile der abgeschlossenen und der ungesättigten Art, die freilich abgesondert nicht vorkommen.*

In an earlier paper (from 1880), called 'Booles rechnende Logik und die Begriffsschrift', he compares the situation with the behaviour of atoms (HERMES 1969, p.19).

> *Ich möchte dies mit dem Verhalten der Atome vergleichen, von denen man annimmt, dass nie eins allein vorkommt, sondern nur in einer Verbindung mit andern, die es nur verlässt, um sofort in eine andere einzugehen.*

The formulation of the statements from 'Grundlagen' is evidently in conflict with the principle of compositionality. From our investigations it appears that related remarks occur in other writings of Frege. This shows that the formulation used in 'Grundlagen' is not just an accidental, and maybe unfelicitous expression of his thoughts. For this reason, and for the lack of evidence for Dummett's interpretation, I am not convinced that Frege's clear statements have to be understood in a weakened way. I think that they should be understood as they are formulated. Therefore I conclude

that in the days of 'Grundlagen' Frege probably would have rejected the principle of compositionality, and, in any case, the formulation we use.

2.3. Sinn und Bedeutung

In 'Ueber Sinn und Bedeutung' (FREGE 1892) the notions 'Sinn' and 'Bedeutung' are introduced. Frege uses these two already existing German words to name two notions he wished to discriminate. The subtle differences in the original meaning of these two words do not cover the different use Frege makes of them. For instance, it is very difficult to account for their differences in meaning in a translation. Frege himself has been confronted with these problems as appears from a letter to Peano (GABRIEL 1976, p.196):

> [..] *Sie* [..] *sagen, dass zwei deutschen Wörtern, die ich verschieden gebrauche, dasselbe italienische nach den Wörterbüchern entspreche. Am nächsten scheint mir dem Worte 'Sinn' das italienische 'senso' und dem Worte 'Bedeutung' das italienische 'significazione' zu kommen.*

Concerning the terminology DUMMETT (1973, p.84) gives the following information. The term 'Bedeutung' has come to be conventionally translated as 'reference'. Since 'Bedeutung' is simply the German word for 'meaning', one cannot render 'Bedeutung' as it occurs in Frege by 'meaning', without a special warning. The word 'reference' does not belie Frege's intention, though it gives it a much more explicit expression. Concerning 'Sinn', which is always translated 'sense', Dummett says that to the sense of a word or expression only those features of meaning belong which are relevant to the truth-value of some sentence in which it may occur. Differences in meaning which are not relevant in this way, are relegated by Frege to the 'tone' of the word or expression. In this way Dummett has given an indication what Frege intended with sense. It is not possible to be more precise about the meaning of 'Sinn'. As van Heyenoort says (Van HEYENOORT 1977, p.93):

> *As for the 'Sinn' Frege gives examples, but never presents a precise definition.*

And Thiel states (THIEL 1965, p.165):

> *What Frege understood as the 'sense' of an expression is a problem that is so difficult that one generally weakens it to the question of when in Frege's semantics two expressions are identical in sense (synonymous).*

I will not try to give a definition; it suffices for our purposes to conclude that the notion 'Sinn' is very close to the notion 'meaning'. Therefore we have to investigate Frege's publications after 1892 to see what he says about the compositionality of Sinn. What he says about compositionality of 'Bedeutung' is a different story (as illustration: he explicitly

rejected that in 1919 (HERMES 1969, p.275), but this is not the case for compositionality of 'Sinn', as will appear in the sequel).

In 'Ueber Sinn und Bedeutung', I found one remark concerning the relation between the senses of parts and the sense of the whole sentence. Frege discusses the question whether a sentence has a reference, and, as an example, he considers the sentence *Odysseus wurde tief schlafend in Ithaka ans Land gesetzt*. Frege says that if someone considers this sentence as true or false, he assigns the name *Odysseus* a reference (Bedeutung). Next he says (FREGE 1892, p.33).

> *Nun wäre aber das Vordringen bis zur Bedeutung des Namens überflüssig: man könnte sich mit dem Sinne begnügen, wenn man beim Gedanken stehenbleiben wollte. Käme es nur auf den Sinn des Satzes, den Gedanken, an, so wäre es unnötig sich um die Bedeutung eines Satzteils zu kümmern; für den Sinn des Satzes kann ja nur der Sinn, nicht die Bedeutung dieses Teils in Betracht kommen.*

So Frege states that there is a connection between the sense of the whole sentence, and the senses of the parts. He does, however, not say anything about a compositional way of building the sense of the sentence. More in particular, the quotation is neither in conflict with the compositionality principle, nor with the statements from 'Grundlagen'. Therefore I agree with BARTSCH (1978), who says that, in 'Ueber Sinn und Bedeutung', Frege does not speak, as is often supposed, about the contribution of the senses of parts to the senses of the compound expression.

2.4. The principle

Up till now we have not found any statement expressing the principle of compositionality. But there are such fragments. The most impressive one is from 'Logik in der Mathematik', an unpublished manuscript from 1914 (HERMES 1969, p.243).

> *Die Leistungen der Sprache sind wunderbar. Mittels weniger Laute und Lautverbindungen ist sie imstande, ungeheuer viele Gedanken auszudrücken, und zwar auch solche, die noch nie vorher von einem Menschen gefasst und ausgedrückt worden sind. Wodurch werden diese Leistungen möglich? Dadurch, dass die Gedanken aus Gedankenbausteinen aufgebaut werden. Und diese Bausteine entsprechen Lautgruppen, aus denen der Satz aufgebaut wird, der den Gedanken ausdrückt, sodass dem Aufbau des Satzes aus Satzteilen der Aufbau des Gedankens aus Gedankenteilen entspricht. Und den Gedankenteil kann man den Sinn des entsprechendes Satzteils nennen, so wie man den Gedanken als Sinn des Satzes aufassen wird.*

This fragment expresses the compositionality principle. However, the fragment is not presented as a fragment expressing a basic principle. It is used as argument in a discussion, and does not get any special attention.

The quotation from 'Logik in der Mathematik', presented above, is considered very remarkable by the editors of Frege's posthumous works. They have added the following footnote in which they call attention to other statements of Frege which seem to conflict with the quotation in consideration (HERMES 1969, p.243)

> *An anderen Stellen schränkt Frege diesen Gedanken-Atomismus allerdings in dem Sinne ein, dass man sich die Gedanketeile nicht als van den Gedanken, in denen sie vorkommen, unabhängige Bausteine vorstellen dürfe.*

They give two references to such statements in Frege's posthumous writings (i.e. the book they are editors of). One is from 'Booles rechnende Logik..' (1880), the other from 'Einleitung in die Logik' (1906). I have quoted these fragments in the discussion of 'Grundlagen'. In this way the editors suggest that the fragment from 'Logik in der Mathematik' is a slip of the pen, and a rather incomplete formulation of Frege's opinion concerning these matters.

The fragment under discussion does, however, not stand on its own. Almost the same fragment can be found in 'Gedankenfüge' (FREGE 1923). I present the fragment here in its English translation from 'Compound thoughts' by Geach and Stoothoff (p.55).

> *It is astonishing what language can do. With a few syllables it can express an incalculable number of thoughts, so that even a thought grasped by a terrestrial being for the very first time can be put into a form of words which will be understood by someone to whom the thought is entirely new. This would be impossible, were we not able to distinguish parts in the thought corresponding to the parts of a sentence, so that the structure of the sentence serves as an image of the structure of the thought.*

Moreover, in a letter to Jourdain, written about 1914, Frege says (GABRIEL 1976, p.127):

> *Die Möglichkeit für uns, Sätze zu verstehen, die wir noch nie gehört haben, beruht offenbar darauf, dass wir den Sinn eines Satzes aufbauen aus Teilen, die den Wörtern entsprechen.*

It is a remarkable fact that all quotations propagating compositionality are written after 1910: 'Gedankenfüge' (1923), 'Logik in der Mathematik' (1914), letter to Jourdain (1914). I have not succeeded in finding such quotations in earlier papers. But the statements which seem to conflict with compositionality are from an earlier period: 'Booles rechnende Logik ....'(1880), 'Grundlagen' (1884), letter to Huntington (1902), 'Einleitung in die Logik' (1906). This shows that, say after 1910, Frege has written about these matters in a completely different way than before. From this I conclude that his opinion concerning these matters changed. On the other hand, Frege never put forward the idea of compositionality as a principle.

It was rather an argument, although an important one, in his discussions. I would therefore not conclude to a break in his thoughts; rather it seems me to be a shift in conception concerning a detail.

In the light of this change, the following information appears relevant. In 1902 Frege received a letter from Russell in which the discovery was mentioned of the famous contradiction in naive set theory, and, in particular, in the theory of classes in Frege's 'Grundgesetze'. About the influence of this discovery on Frege, Dummett says the following (DUMMETT 1973, p.657):

> *It thus seems highly probable that Frege came quickly to regard his whole programme of deriving arithmetic from logic as having failed. Such a supposition is not only probable in itself: it is in complete harmony with what we know of his subsequent carreer. The fourth period of his life may be regarded as running from 1905 to 1913, and it was almost entirely unproductive.*

For this reason I consider it as very likely that in this period Frege was not concerned with issues related to compositionality. Then it is understandable that after this period he writes in a different way about the detail of compositionality (recall that it never was a principle, but just an argument).

2.5. Conclusion

My conclusions are as follows. Before 1910, and in any case especially in the years when he wrote his most important and influential works, Frege would probably have rejected the compositionality principle, in any case the formulation we use nowadays. After 1910 his opinion appears to have changed, and he would probably have accepted the principle, in any case the basic idea expressed in it. However, Frege never put forward such an idea as a basic principle, it is rather an argument in his discussions. Therefore, calling the compositionality principle 'Frege's principle' is above all, honouring his contributions to the study of semantics. But it is also an expression of his final opinion on these matters.

3. TOWARDS A FORMALIZATION

In this section, I will give the motivation for a formalized version of the compositionality principle. It is *not* my purpose to formalize what Frege or other authors might have intended to say when uttering something like the principle. I rather take the principle in the given formulation

as a starting point and proceed along the following line; the (formalized version of) the principle should have as much content as possible. This means that the principle should make it possible to derive interesting consequences about those grammars which are in accordance with the principle, and at the same time it should be sufficiently abstract and universal to be applicable to a wide variety of languages. From the formalization it must be possible to obtain necessary and sufficient conditions for a grammar to be in agreement with the compositionality principle.

Consider a language L which is to be interpreted in some domain D of meanings. The kind of objects D consists of depends on the language under consideration, and the use one wishes to make of the semantics. In this section such aspects are left unspecified. Defining the semantics of a language consists in defining a suitable relation between expressions in L and semantic objects in D. Then the compositionality principle says something about the way in which this relation between L and D has to be defined properly.

In the formulation of the principle given in section 1, we encounter the phrase 'its parts'. Clearly we should not allow the expressions of L to be split in some random way. In the light of the standard priority conventions, the expression $y+8$ is not to be considered as a *part* of the expression $7.y + 8.x$; so the meaning of $7.y + 8.x$ does not have to be built up from the meaning of $y + 8$. It would also be pointless to try to build the meaning of some compound expression directly from the meanings of its atomic symbols (the terminal symbols of the alphabet used to represent the language). Since distinct expressions consist of distinct strings of symbols, there is always some dependence of the meanings of the basic symbols. Consequently such an interpretation would trivialize the principle. Another trivialization results by taking all expressions of the language to be 'basic', and interpreting them individually. The principle is interesting only in case the 'parts' are not trivial parts. Traditionally, the true decomposition of an expression into parts is described in the syntax for the language. Thus a language, the semantics of which is defined in accordance with the principle, should have a syntax which clearly expresses what the parts of the compound expressions are.

Let the language L, together with the set of expressions we wish to consider as parts, be denoted by $\underline{E}$. In order to give the principle a non-trivial content, we assume that the syntax of the language consists of rules of the following form:

If one has expressions $E_1,\ldots,E_n$ *then one can build the compound expression* $S_j(E_1,\ldots,E_n)$.

Here S_j is some operation on expressions, and $S_j(E_1,\ldots,E_n)$ denotes the result of application of S_j to the arguments $E_1,\ldots,E_n$. If the rules have the above format, we define the notion 'parts of' as follows.

If expression E *is built by a rule* S_j *from arguments* $E_1,\ldots,E_n$, *then the parts of* E *are the expressions* $E_1,\ldots,E_n$.

It often is the case that a rule does not apply to all expressions in E, and that certain groups of expressions behave the same in this respect. Therefore the set of expressions is divided into subsets. The names of these subsets are called *types* or *sorts* in logic, *categories* in linguistics, and *types* or *modes* in programming languages. Often the name of a set and the set itself are identified and I will follow this practice. Instead of speaking about elements of the subset of a certain type, I will speak about the elements of a certain type, etc. The use of names for subsets allows us to specify in each rule from which category its arguments have to be taken, and to which category the resulting expression belongs. Thus, a *syntactic rule* S_j has the following form:

If one has expressions $E_1,\ldots,E_n$ *of the categories* $C_1,\ldots,C_n$ *respectively, then one can form the expression* $S_j(E_1,\ldots,E_n)$ *of category* C_{n+1}.

An *equivalent formulation* is:

Rule S_j *is a function from* $C_1\times\ldots\times C_n$ *to* C_{n+1};
i.e. $S_j: C_1 \times \ldots \times C_n \rightarrow C_{n+1}$.

Suppose that a certain rule S_i is defined as follows:

$$S_i: C_1 \times C_2 \rightarrow C_3, \quad \text{where } S_i(E_1,E_2) = E_1E_2.$$

This means that S_i concatenates its arguments. Then our interpretation of the principle says that the meaning of E_1E_2 has to be built up from the meanings of E_1 and E_2. A case like this constitutes the most elementary version of the principle. A compound expression is divided into real subexpressions, and the meaning of the compound expression is built up from the meanings of these subexpressions. In such a case the formalization coincides with the simplest, most intuitive conception of the principle: parts are visible as parts. But in some situations one might wish to consider as part an expression which is not visible as a part. We will meet several examples in later chapters. One example is the phenomenon of discontinuous constituents. The phrase *take away* is not a subphrase of *take the apple away*; it is not visible as a part. Nevertheless, one might here

wish to consider it as a unit which contributes to the meaning of *take the apple away*, i.e. as a part in the sense of the principle. The above definition of 'part' gives the possibility to do so. If the phrase *take the apple away* is produced by means of a rule which takes as arguments the phrases *the apple* and *take away*, then *take away* is indeed a part in the sense of the definition. The definition generalizes the principle for rules which are not just a concatenation operation, and consequently the parts need not be visible in the expression itself.

There are no restrictions on the possible effect of the rules S_j. They may concatenate, insert, permute, delete, or alter (sub)expressions of their arguments in an arbitrary way. A rule may even introduce symbols which do not occur in its arguments. Such symbols are called *syncategorematic symbols*. These are not considered as parts of the resulting expression in the sense of the principle, and therefore they do not contribute a meaning from which the meaning of the compound can be formed. I will assume in general that the rules are total (i.e. they are defined for all expressions of the required categories). In chapter 6 partial rules will be discussed.

The abstraction just illustrated implies that we have lost the most intuitive conception of the principle. But it is not unlikely that several authors who mention Frege's principle only have the most intuitive version in mind. In order to avoid confusion, I will call the more abstract version not 'Frege's principle', but 'the compositionality principle'. As for the simple rules, where only concatenation is used, our interpretation of the principle coincides with the most intuitive interpretation. In more complex cases, where it might not be intuitively clear what the parts are, our interpretation can be applied as well. If one wishes to stick to the most intuitive interpretation of the principle, one must use only concatenation rules. In that way one would restrict considerably the applicability of grammars satisfying the framework (see chapter 2, section 5).

So far we have not considered the possibility of ambiguities. It is not excluded that some expression E can be obtained both as $E = S_i(E_1,\ldots,E_n)$ and as $E = S_j(E'_1,\ldots,E'_m)$. In practice such ambiguities frequently arise. In a programming language (e.g. in Algol 68), the procedure identifier *random* can be used to denote the process of randomly selecting a number, as well as to denote the number thus obtained. The information needed to decide which interpretation is intended, is present in the production tree of the program, where the expression *random* is either of type 'real' or not. In natural languages ambiguities arise even among expressions of the same category.

Consider for instance the sentence *John runs or walks and talks*. Its meaning depends on whether *talks* is combined with *walks*, or with *runs or walks*. Also here, the information needed to solve the ambiguity is hidden in the production tree. In the light of such ambiguities, we cannot speak in general of the meaning of some expression, but only of its meaning with respect to a certain derivational history. If we want to apply the compositionality principle to some language with ambiguities, we should not apply it to the language itself, but to the corresponding language of derivational histories.

In computer science it is generally accepted that the derivational histories form the real input for the semantical interpretation. SCHWARTZ (1972, p.2) states:

> *We have sufficient confidence in our understanding of syntactic analysis to be willing to make the outcome of syntactic analysis, namely the syntax tree representation of the program, into a standard starting point for our thinking on program semantics. Therefore we may take the semantic problem to be that of associating a value* [..] *with each abstract program, i.e. parse tree.*

In the field of semantics of natural languages, it is also common practice not to take the expressions of the language themselves as input to the semantical interpretation, but structured versions of them. KATZ & FODOR (1963, p.503) write:

> *Fig. 6 shows the input to a semantic theory to be a sentence* S *together with a structural description consisting of the* n *derivations of* S, $d_1, d_2, \ldots, d_n$, *one for each of the* n *ways that* S *is grammatically ambiguous.*

The book of Katz and Fodor is one of the early publications about the position of semantics in transformational grammars. There has been a lot of discussion in that field concerning the part of the derivational history which may actually be used for the semantic interpretation. In the so-called 'standard theory' only a small part of the information is used: the 'deep structure'. In the 'extended standard theory', one also uses the information which 'transformations' are applied, and what the final outcome, the 'surface structure', is. In the most recent proposals, the view on syntax and its relation with semantics is rather different.

As a matter of fact, neither Schwartz, nor Katz and Fodor use the same (semantic) framework we have. They are quoted here to illustrate that the idea of using information from the derivational history as input to the semantic component is not unusual.

Let us now turn to the phrase 'composed from the meanings of its parts'. Consider again a rule S_i which allows us to form the compound

expression $S_i(E_1,\ldots,E_n)$ from the expressions $E_1,\ldots,E_n$. Assume moreover that the meanings of the E_k are the semantical objects D_k. According to the principle, the information we are allowed to use for building the meaning of the compound expression consists in the meanings of the parts of the expression and the information which rule was applied. As usual, 'rule' is intended to take into account the order of its arguments. So the meaning of a compound expression is determined by an n-tuple of meanings (of its parts) and the information of the identity of the rule. This is in fact the only information which may be used. If one would be allowed to use other information (e.g. syntactic information concerning the parts), the principle would not express the whole truth, and not provide a sufficient condition. Thus the principle would tend to become a hollow phrase.

As argued for above, I interpret the compositionality principle as stating that the meaning of a compound expression is determined *completely* by the meanings of its parts and the information which syntactic rule is used. This means that for each syntactic rule there is a function on meanings which yields the meaning of a compound expression when it is applied to the meanings of the parts of that expression. So for each syntactic rule there is a corresponding semantic operation. Such a correspondence is not unusual; it can be found everywhere in mathematics and computer science. If one encounters for instance a definition of the style 'the function f*g is defined by performing the following calculations using f and g...', then one sees in fact a syntactic and a semantic rule. The syntactic rule introduces the operator * between functions and states that the result is again a function, whereas the semantic rule tells us how the function should be evaluated

If we use the freedom allowed by the principle at most, we may associate with each syntactic rule S_i a distinct semantic operation T_i. So the most general description of the situation is as follows. The meaning of an expression formed by application of S_i to $(E_1,\ldots,E_n)$ can be obtained by application of operator T_i to $(D_1,\ldots,D_n)$, where D_j is the meaning of E_j. These semantic operators T_i may be partially defined functions on the set $\underline{D}$ of meanings, since T_i has to be defined only for those tuples from $\underline{D}$ which may arise as argument of T_i. These are those tuples which can arise as meanings of arguments of the syntactic rule S_i which corresponds with T_i. In this way the set $\underline{E}$ leaves a trace in the set $\underline{D}$. The set of meanings of the expressions of some category forms a subset of $\underline{D}$ which becomes the set of possible arguments for some semantic rule. Thus the domain $\underline{D}$ obtains a structure which is closely related to the structure of the syntactic domain

E. Our *formalization* of the compositionality principle may *at this stage* be summarized as follows.

> *Let* $S_i: C_1 \times \ldots \times C_n \rightarrow C_{n+1}$ *be a syntactic rule, and* $M: E \rightarrow D$ *be a function which assigns a meaning to an expression with given derivational history. Then there is a function* $T_i: M(C_1) \times \ldots \times M(C_n) \rightarrow M(C_{n+1})$ *such that* $M(S_i(E_1,\ldots,E_n)) = T_i(M(E_1),\ldots,M(E_n))$.

In the process of formalizing the principle of compositionality we have now obtained a special framework. The form of the syntactic rules and the use of sorts give the syntax the structure of, what is called, a 'many-sorted algebra'. The correspondence between syntax and semantics implicates that the semantic domain is a many-sorted algebra of the same kind as the syntactic algebra. The meaning assignment is not based upon the syntactic algebra itself, but on the associated algebra of derivational histories. The principle of compositionality requires that meaning assignment is a homomorphism from that algebra to the semantic algebra. Note that the principle, which is formulated as a principle for semantics, has important consequences not only for the semantics, but also for the syntax.

The approach described here, is closely related to the framework developed by the logician Richard Montague for the treatment of syntax and semantics of natural languages (MONTAGUE 1970b). It is also closely related to the approach propagated by the group called 'Adj' for the treatment of syntax and semantics of programming languages (ADJ 1977, 1979). Consequently frameworks related to the one described here can be found in the publications of authors following Adj (for references see ADJ 1979), or following Montague (for references see DOWTY, WALL & PETERS 1981, or the present book). The conclusion that the principle of compositionality requires an algebraic approach is also given by MAZURKIEWICS (1975) and MILNER (1975), without, however, developing some framework. The observation that there is a close relationship between the frameworks of Adj and Montague, was independently made by MARKUSZ & SZOTS (1981), ANDREKA & SAIN (1981), and Van EMDE BOAS & JANSSEN (1979).

4. AN ALGEBRAIC FRAMEWORK

In this section I will develop the framework sketched in section 3, and arguments concerning the practical use of the framework will influence this further development. The mathematical theory of the framework will be investigated in chapter 2.

The central notions in our formalization of the principle of

compositionality are 'many-sorted algebra' and 'homomorphism'. An algebra consists of some set (the elements of the algebra), and a set of operations defined on those elements. A many-sorted algebra is a generalization of this. It consists of a non-empty set S of sorts (types, modes, or categories), for each sort $s \in S$ a set A_s of elements of that sort (A_s is called the carrier of sort s), and a collection $(F_\gamma)_{\gamma\in\Gamma}$ of operations which are mappings from cartesian products of specified carriers to a specified carrier. So in order to determine a many-sorted algebra, one has to determine a 'sorted' family of sets and a collection of operators. This should explain the following definition.

4.1. DEFINITION. A *many-sorted algebra* A is a pair $\langle (A_s)_{s\in S}, \underline{F}\rangle$, where
1. S is a non-empty set, its elements are called the sorts of A.
2. A_s is a set (for each $s\in S$), the carrier of sort s.
3. $\underline{F}$ is a collection of operators defined on certain n-tuples of sets A_s, where $n>0$.

4.1. END.

Structures of this kind have been defined by several authors, using different names; the name 'many-sorted algebra' is borrowed from ADJ(1977). Notice that in the above definition there are hardly any restrictions on the sets and operators. The carriers may be non-disjunct, the operators may perform any action, and the sets involved (except for S) may be empty.

In order to illustrate the notion 'many-sorted algebra', I will present three examples in an informal way. I assume that these examples are familiar, and I will, therefore, not describe them in detail. The main interest of these examples is that they illustrate the notion of a many-sorted algebra. The examples are of a divergent nature, thus illustrating the generality of this notion.

4.2. EXAMPLE: *Real numbers*.

Let us consider the set of real numbers as consisting of two sorts. *Neg* and *Pos*. The carrier R_{Neg} of sort Neg consists of the negative real numbers, the carrier R_{Pos} of sort Pos of the positive real numbers, zero included. An example of an operation is *sqrt*: $R_{Pos} \rightarrow R_{Pos}$, where *sqrt* yields the square root of a positive number. For R_{neg} there is no corresponding operation. Since we consider (in this example) the real numbers as a two-sorted algebra, there are two operations for squaring a number. One for squaring a positive number (*sqpos:* $R_{Pos} \rightarrow R_{Pos}$) and one for squaring a

negative number (*sqneg:* $R_{Neg} \rightarrow R_{Pos}$). Since these two operations are closely related, we may use the same symbol for both operations: $(\ )^2$.

4.3. EXAMPLE: *Monadic Predicate Logic*

Sorts are *Atom*, *Pred* and *Form*. The carrier A_{Atom} of sort Atom consists of the symbols $a_1, a_2, \ldots$ and the carrier A_{Pred} of the sort Pred consists of the predicate letters $P_1, P_2, \ldots$. The carrier A_{Form} consists of formulas like $P_1(a_1)$, $\neg P_1(a_1)$, and $P_1(a_2) \vee P_2(a_3)$. Two examples of operators are as follows.

1. The operation *Apl*: $A_{Pred} \times A_{Atom} \rightarrow A_{Form}$. Apl assigns to predicate P and atom a the formula where P is applied to a; viz. $P(a)$.
2. The operation *Disj*: $A_{Form} \times A_{Form} \rightarrow A_{Form}$. Disj assigns to two formulas ϕ and ψ their disjunction $\phi \vee \psi$.

Notice that (in the present algebraization) the brackets (,), and the disjunction symbol $\vee$ are syncategorematic symbols.

4.4. EXAMPLE: *English*

Examples of sorts are *Sentence*, *Verb phrase*, and *Noun phrase*. The carrier of sort Sentence consists of the analysis trees of English sentences, and the carriers of other sorts of trees for expressions of other sorts. An example of an operator is T_{Neg}: Sentence $\rightarrow$ Sentence. The operator T_{Neg} assigns to an analysis tree of an English sentence the analysis tree of the negated version of that sentence. An explicit and complete description of this algebra I cannot provide. This example is mentioned to illustrate that complex objects like trees can be elements of an algebra.
4.4. END.

As explained in the previous section, we do not assign meanings to the elements of the syntactic algebra itself, but to the derivational histories associated with that algebra. These derivational histories form an algebra: if expressions E_1 and E_2 can be combined to expression E_3, then the derivational histories of E_1 and E_2 can be combined to a derivational history of E_3. So the derivational histories constitute a (many-sorted) set in which certain operations are defined. Hence it is an algebra. The nature of the operations of this algebra will become evident when we consider below representations of derivational histories.

Suppose that a certain derivational history consists of first an application of operator S_1 to basic expressions E_1 and E_2, and of next an

application of S_2 to E_3 and the result of the first step. A description like this of a derivational history is not suited to be used in practice (e.g. because of its verbosity). Therefore formal representations for derivational histories will be used (certain trees or certain mathematical expressions).

In Montague grammar one usually finds trees as representation of a derivational history. The history described above is represented in figure 1. Variants of such trees are used as well. Often the names of the rules are not mentioned (e.g. S_1,S_2), but their indices (viz. 1,2). Sometimes the rule, or its index is not mentioned, but the category of the resulting expression. Even the resulting expressions are sometimes left out, especially when the rules are concatenation rules (figure 2). The relation between the representations of derivational histories and the expressions of the languages is obvious. In figure 1 one has to take the expression labelling the root of the tree, and in figure 2 one has to perform the mentioned operations. (For this kind of trees, it usually amounts to a concatenation of the expressions mentioned at the leaves (i.e. end-nodes)).

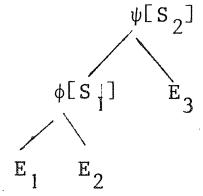


Figure 1. Representation of a derivational history

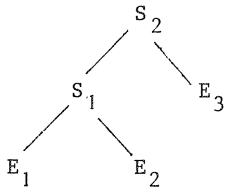


Figure 2. Another representation of the same derivational history

An alternative representation originates from the field of algebra. Derivational histories are represented by a compound expressions, consisting of basic expressions, symbols for the operators, and brackets. The derivational history from figure 1 is represented by the expression:

$S_2(S_1(E_1,E_2),E_3)$.

Such expressions are called terms. The algebra of terms corresponding with algebra A is called the term algebra T_A. From a term one obtaines an expression of the actual language by evaluating the term, i.e. by application of the operators (corresponding with the operator symbols) to the mentioned arguments. The sorts of the term algebra T_A are identical to the sorts of A, the operators are concatenation operators on terms. Note that all these

different representations mathematically are equivalent.

MONTAGUE (1970b) introduced the name 'disambiguated language' for the algebra of derivational histories. The relation between the disambiguated language and the language under consideration (he calls it R) is completely arbitrary in his approach. The only information he provides is that it is a binary relation with domain included in the disambiguated language (MONTAGUE 1970b, p.226). From an algebraic viewpoint this arbitrariness is very unnatural, and therefore I restrict this relation in the way described above (evaluating the term, or taking the expression mentioned at the root). This is a restriction on the framework, but not on the class of languages that can be described by the framework (see chapter 2).

The tree representations are the most suggestive representations, and they are most suitable to show complex derivational histories. The term representations take less space and are suitable for simple histories and in theoretical discussions. In the first chapters I will mainly use terms, in later chapters trees. According to the framework we have to speak about the meaning of an expression relative to a derivational history. In practice one often is sloppy and speaks about the meaning of an expression (when the history is clear from the context, or when there is only one).

After this description of the notion of a many-sorted algebra, I will introduce the other central notion in our formalization of the principle of compositionality: the notion 'homomorphism'. It is a special kind of mapping between algebras, and therefore first mappings are introduced.

4.5. DEFINITION. By a *mapping* m from an algebra A to an algebra B is understood a mapping from the carriers of A to the carriers of B. Thus:

$$m: \bigcup_{s \in S_A} A_s \rightarrow \bigcup_{s \in S_B} B_s .$$

4.5. END.

A mapping is called a *homomorphism* if it respects the structures of the algebras involved. This is only possible if the two algebras have a similar structure. By this is understood that there is a one-one correspondence between the sorts in the one algebra and in the other algebra, and between the operators in the one algebra and in the other algebra. The latter means that if an operator is defined for certain sorts in the one algebra, then the corresponding operator is defined for the corresponding sorts in the other algebra. This should describe the essential aspects of the technical

notion of 'similarity' of two algebras; a formal definition will be given in chapter 2. Then the definition of a homomorphism given below, will be adapted accordingly, and the slight differences with the definitions in the literature (Montague, Adj) will be discussed.

4.6. DEFINITION. Let $A = \langle (A_s)_{s \in S}, \underline{F} \rangle$ and $B = \langle (B_t)_{t \in T}, \underline{G} \rangle$ be similar algebras. A mapping h from A to B is called a *homomorphism* if the following two conditions are satisfied

1. h respects the sorts, i.e. $h(A_s) \subset B_t$, where t is the sort of B which corresponds to sort s of A.
2. h respects the operators, i.e. $h(F(a_1,\ldots,a_n)) = G(h(a_1),\ldots,h(a_n))$ where $G \in \underline{G}$ is the operator of B which corresponds to $F \in \underline{F}$.

4.6. END

Now that the notions of a many sorted algebra and of a homomorphism are introduced, I will present two detailed examples.

4.7. EXAMPLE: *Fragment of English.*

Syntactic Algebra

The syntactic algebra E consists of some English words and sentences

I. Sorts

$S_E = \{Sent, Subj, Verb\}$

II. Carriers

$E_{Subj} = \{John, Mary, Bill\}$

$E_{Verb} = \{runs, talks\}$

$E_{Sent} = \{John\ runs,\ Mary\ runs,\ Bill\ runs,\ John\ talks,\ Mary\ talks,\ Bill\ talks\}$

III. Operations

$C: E_{Subj} \times E_{Verb} \rightarrow E_{Sent}$
defined by $C(\alpha,\beta) = \alpha\beta$

So C(*John, runs*) is obtained by concatenating *John* and *runs*, thus yielding *John runs*.

Semantic Algebra

The semantic Algebra M consists of model-theoretic entities, such as truth values and functions.

I. Sorts

$S_M = \{e, t, <e,t>\}$

So there are three sorts: two sorts being simple symbols (e ~ entity, t ~ truthvalue), and the compound symbol <e,t> (function from e to t).

II. Carriers

$M_t = \{\underline{true}, \underline{false}\}$ The carrier M_t consists of two elements, the truthvalues $\underline{true}$ and $\underline{false}$.

$M_e = \{e_1, e_2, e_3\}$ The set M_e consists of three elements: e_1, e_2 and e_3.

$M_{<e,t>} = (M_t)^{M_e}$ The carrier $M_{<e,t>}$ consists of all functions from M_e to M_t. This set has 8 elements.

III. Operations

There is one operation in M: the operation F of function application.

$F: M_e \times M_{<e,t>} \rightarrow M_t$,

where $F(\alpha,\beta)$ is the result of application of β to argument α.

The algebras E and M are similar. The correspondence of sorts is Subj ~ e, Verb ~ <e,t>, Sent ~ t, and operation C corresponds to F. Although the algebras E and M are similar, they are not the same. For instance, the number of elements in E_{verb} differs from the number of elements in $E_{<e,t>}$.

There are a lot of homorphisms from T_E (the derivational histories in E), to M. An example is as follows.

Let h be defined by

$h(\textit{John}) = e_1$, $h(\textit{Bill}) = e_2$, $h(\textit{Mary}) = e_3$

$h(\textit{runs})$ is the function f_1 which has value $\underline{true}$ for all $e \in M_e$

$h(\textit{talks})$ is the function f_2 which has value $\underline{false}$ for all $e \in M_e$

Furthermore we define h for the compound terms.

$h(C(\textit{John},\textit{runs})) = h(C(\textit{Mary},\textit{runs})) = h(C(\textit{Bill},\textit{runs})) = \underline{true}$

$h(C(\textit{John},\textit{talks})) = h(C(\textit{Mary},\textit{talks})) = h(C(\textit{Bill},\textit{talks})) = \underline{false}$

The function h, thus defined, is a homomorphism because

1. $h(T_{E,Subj}) \subset M_e$, $h(T_{E,Verb}) \subset M_{<e,t>}$, $h(T_{E,sent}) \subset M_E$

2. $h(C(\alpha,\beta)) = F(h(\alpha),h(\beta))$ for all subjects α and verbs β.

It is easy to define other homomorphisms from T_E to M. Notice that once h is defined for $T_{E,Subj}$ and for $T_{E,Verb}$, then there is no choice left for the definition of h for $T_{E,Sent}$ (provided that we want h to be a homomorphism).

4.8. EXAMPLE: *Number denotations*

Syntactic Algebra

The algebra *Den* of natural number denotations is defined as follows

I. Sorts

S_{Den} = {digit,num}

II. Carriers

D_{digit} = {*0,1,2,3,4,5,6,7,8,9*}

D_{num} = {*0,1,2,3,...10,11,...01,02,..010,..001,..007,.....*}

So D_{num} is the set of all number denotations, including denotations with leading zero's. Notice that $D_{digit} \subset D_{num}$.

III. Operators

There is one operation.

$C: D_{num} \times D_{digit} \rightarrow D_{num}$

where C is defined by $C(\alpha,\beta) = \alpha\beta$.

So C concatenates a number with a digit.

Semantic Algebra

The algebra *Nat* of natural numbers is defined as follows

I. Sorts

S_{Nat} = {d,n}.

II. Carriers

N_d consists of the natural numbers up to nine (zero and nine included)

N_n consists of all natural numbers.

III. Operations

There is one operation:

$F: N_n \times N_d \rightarrow N_n$

where F is defined as multiplication of the element from N_n by ten, followed by addition of the element from N_d.

A natural homomorphism h from T_{Den} to *Nat* is the mapping which associates with the derivational history of a digit or number denotation the corresponding number. Then h(C(*0,7*)) and h(*7*) are both mapped onto the number seven. That this h is a homomorphism follows from the fact that F describes the semantic effect of C, e.g. h(C(*2,7*)) = F(h(*2*),h(*7*)).

4.8. END

Syntax is an algebra, semantics is an algebra, and meaning assignment is a homomorphism; that is the aim of our enterprise. But much work has to be done in order to proceed in this way. Consider the two examples given above. The carriers were defined by specifying all their elements, the homomorphisms were defined by specifying the image of each element, and the operations in the semantic algebra were described by means of full English sentences. For larger, more complicated algebras this approach will be very impractical. Therefore a lot of technical tools will have to be introduced before we can deal with an interesting fragment of natural language or programming language. Consider again the first example (i.e. 4.7). The semantic operation corresponding to the concatenation of a *Subj* and a *Verb* was described as the application of the function corresponding to the verb to the element corresponding to the subject. One would like to use standard notation from logic and write something like *Verb(Subj)*. Thus one is tempted to use some already known language in order to describe a semantic operation. This is precisely the method we will employ. If we wish to define the meaning of some fragment of a natural language, or of a programming language, we will not describe the semantic operations in the meta-language (for instance a mathematical dialect of English), but use some formal language, the meaning of which has already been defined somehow: we will use some formal or logical language. Thus the meaning of an expression is defined in two steps: by translating first, and next interpreting, see figure 3.

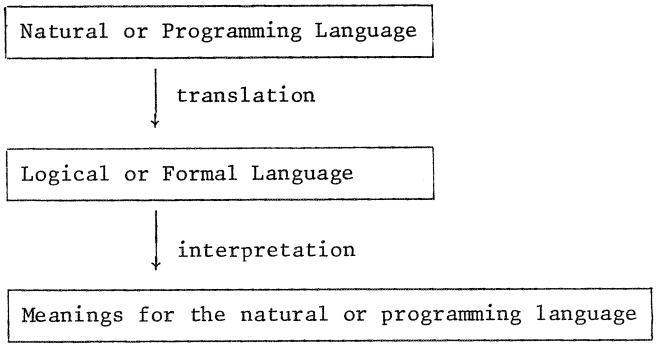


Figure 3 Meaning assignment in two steps.

Figure 3 illustrates that the semantics of the fragment of English is defined in a process with two stages. But is this approach in accordance with our algebraic aim? Is the mapping *from* the term algebra corresponding

with the syntax of English *to* the algebra of meanings indeed a homomorphism? The answer is that we have to obey certain restrictions, in order to be sure that the two-stage process indeed determines a homomorphism. The translation should be a homomorphism from the term algebra for English to the logical language and the interpretation of the logical language should be a homomorphism as well. Then, as is expressed in the theorem below, the composition of these two mappings is a homomorphism.

4.9. THEOREM. *Let* A,B, *and* C *be similar algebras, and* $h: A \to B$ *and* $g: B \to C$ *homomorphisms. Define the mapping* $h \circ g: A \to C$ *by* $(h \circ g)(a) = g(h(a))$. *Then* $h \circ g$ *is a homomorphism.*

PROOF.

1. $(h \circ g)(A_s) \subset g(h(A_s)) \subset g(B_{s'}) \subset C_{s''}$, where s' and s'' are the sorts in B and C corresponding with s.
2. Let G_γ, H_γ be the operators in B and C corresponding with F_γ. Then
   $(h \circ g)(F_\gamma(a_1,\ldots,a_n)) = g(h(F_\gamma(a_1,\ldots,a_n))) = g(G_\gamma(h(a_1),\ldots,h(a_n))) =$
   $= H_\gamma(g(h(a_1),\ldots,g(h(a_n))) = H_\gamma((h \circ g)(a_1),\ldots,(h \circ g)(a_n))$

4.9. END.

The semantical language does not always contain basic operators which correspond to the operators in the syntax. In the example concerning natural number denotations there is no basic arithmetical operator which corresponds to the syntactic operation of concatenation with a digit. I described the semantic operator by means of the phrase 'multiplication of the element from N_n with ten; followed by an addition with the element of N'_d. One is tempted to indicate this operation not with this compound phrase, but with something like '10 × number + digit'. One wishes to use a compound expression from the language of arithmetic for the semantic operation which corresponds to the concatenation operation, i.e. to build new operations from old ones.

The situation I have just described, is the one which almost always arises in practice. One wishes to define the semantics of some language. The set of semantic objects has some 'natural' structure of its own, and a 'natural' semantical language which reflects this structure. So this 'natural' semantical language has not the same algebraic structure as the language for which we wish to describe the semantics. Therefore we use the semantical language (usually some kind of formal or logical language) to build a new algebra, called a derived algebra. We make new operations by

forming compound expressions which correspond with the syntactic operations of the language for which we wish to describe the semantics. This situation is presented in figure 4; the closed arrows denote mappings, the dotted arrows indicate the construction of a new algebra by means of the introduction of new operations (built from old ones).

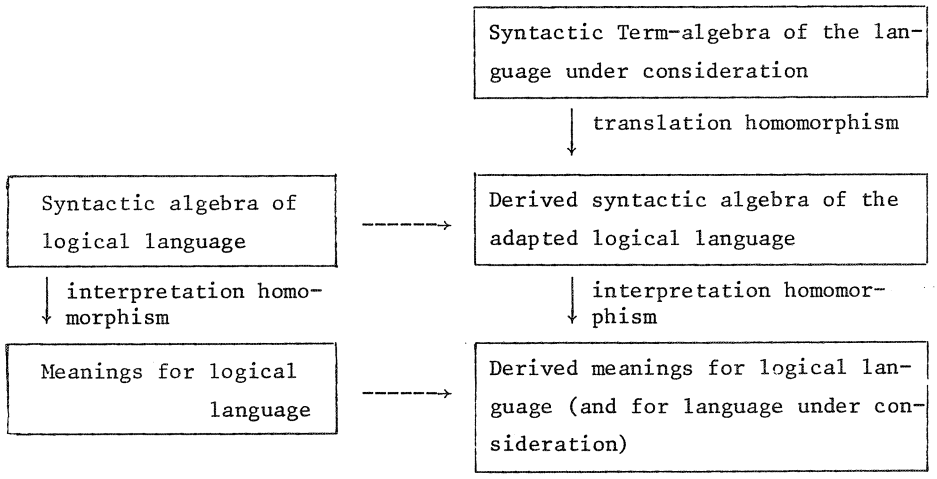


Figure 4. Meaning assignment using derived algebras

In this way, we have derived a new syntactic algebra from the syntactic algebra of the logical language. The syntactic algebra of which we wish to define the semantics is translated into this derived algebra. Now the question arises whether this approach is in accordance with our aim of defining some homomorphism from the syntactic algebra to the collection of meanings. The theorem that will be mentioned below guarantees that under certain conditions this is the case. The interpretation of the logical language has to be a homomorphism, and the method by which we obtain the derived algebra is restricted to the introduction of new operators by composition of old operators. Such operators are called polynomials; for a formal description see chapter 2. If these conditions are satisfied, then the interpretation homomorphism for the logical language is also an interpretation homomorphism of the derived algebra (when restricted to this algebra). Composition of this interpretation homomorphism with the translation homomorphism gives the desired homomorphism from the language under consideration to its meanings. The theorem is based upon MONTAGUE (1970b), for its proof see chapter 2.

4.10. THEOREM. *Let* A *and* B *be similar algebras and* h: A → B *a homomorphism onto* B. *Let* A' *be an algebra obtained from* A *by means of introduction of polynomially defined operators over* A.
Then there is a unique algebra B' *such that* h *is a homomorphism from* A' *onto* B'.
4.10. END

Finally I wish to make some remarks about the translation into some logical language. As I explained when introducing this intermediate step, it is used as a tool for defining the homomorphism from the syntactic algebra to the semantic one. If we would appreciate complicated definitions in the meta language, we might omit the level of a translation. It plays no essential role in the system, it is there for convenience only. If convenient, we may replace a translation by another translation which gets the same interpretation. We might even use another logical language. So in a Montague grammar there is nothing which deserves the name of *the* logical form of an expression. The obtained translation is just one representation of a semantical object, and might freely be interchanged with some other representation. KEENAN & FALTZ (1978), in criticizing the logical form obtained in a Montague grammar, criticize a notion which does not exist in Montague grammar.

5. MEANINGS

5.1. Introduction

In this section some consequences are considered of the requirement of a homomorphic mapping from the syntactic term algebra to the semantic algebra. These consequences are considered for three kinds of language: natural languages, programming languages and logical languages. It will appear that the requirement of associating a single meaning with each expression of the language helps us, in all three cases, to find a suitable formalization of the notion of meaning. Furthermore, an example will be considered of an approach where the requirement of a homomorphic relation between syntax and semantics is *not* obeyed.

5.2. Natural Language

Consider the phrase *the queen of Holland*, and assume that it is used to denote some person (and not an institution). Which person is denoted, depends on the moment of time one is speaking about. This information can usually be derived from the linguistic context in which the expression occurs. In

(1) *The queen of Holland is married to Prince Claus.*

Queen Beatrix is meant, since she is the present queen. But in

(2) *In 1910 the queen of Holland was married to Prince Hendrik.*

Queen Wilhelmina is meant, since she was the queen in the year mentioned. So one is tempted to say that the meaning of the phrase *the queen of Holland* varies with the time one is speaking about. Such an opinion is, however, not in accordance with our algebraic (compositional) framework. The approach which leads to a single meaning for the phrase under discussion is to incorporate the source of variation into the notion of meaning. In this way we arrive at the conception that the meaning of the phrase *the queen of Holland* is a function from moments of time to persons. For other expressions (and probably also for this one) there are more factors of influence (place of utterance, speaker,..). Such factors are called indices; a function with the indices as domain is called an intension. So the meaning of an expression is formalized by an intension: our framework leads to an intensional conception of meaning for natural language. For a more detailed discussion concerning this conception, see LEWIS 1970. A logical language for dealing with intensions is the language of 'intensional logic'. This language will be considered in detail in chapter 3.

5.3. Programming Language

Consider the expression $x+1$. This kind of expressions occurs in almost every programming language. It is used to denote some number. Which number is denoted depends on the internal situation in the computer at the moment of consideration. For instance, in case the internal situation of the computer associates with x the value seven, then $x+1$ denotes the number eight. So one is tempted to say that the meaning of $x+1$ varies. But this is not in accordance with the framework. As in example 1, the conflict is resolved by incorporating the source of variation into the notion of meaning. As the meaning of an expression like $x+1$ we take a function from computer states to numbers. On the basis of this conception a compositional treatment

can be given of meanings of computer languages (See chapter 10). States of the computer can be considered as an example of an index, so also in this case we use an intensional approach to meaning. In the publications of Adj a related conception of the meaning of such expressions is given, although without calling it an intension (see e.g. ADJ 1977, 1979).

Interesting in the light of the present approach is a discussion in PRATT 1979. Pratt discusses two notions of meaning: a static notion (an expression obtains once and for all a meaning), and a dynamic notion (the meaning of an expression varies). He argues that (what he takes as) a static notion of meaning has no practical purpose because we frequently use expres-pression obtains once and for all a meaning), and a dynamic notion (the of time. Therefore he develops a special logic for the treatment of semantics of programming languages, called 'dynamic logic'. But on the basis of our framework, we have to take a 'static' notion of meaning. By means of intensions we can incorporate all dynamics into such a framework. Pratt's dynamic meanings might be considered as a non-static version of intensional logic.

5.4. Predicate Logic

It is probably not immediately clear how predicate logic fits into the algebraic framework. PRATT (1979,p.55) even says that 'there is no function F such that the meaning of $\forall xp$ can be specified with a constraint of the form $\mu(\forall xp) = F(\mu(p))$'. In our algebraic approach we have to provide for such a meaning function μ and operator F.

Let us consider the standard (Tarskian) way of interpreting logic. It roughly proceeds as follows. Let @ be a model and g be an @-assignment. The interpretation in @ of a formula ϕ with respect to g, denoted ϕ^g, is then recursively defined. One clause of this definition is as follows (here 1 denotes the truth value for truth).

$[\phi \wedge \psi]^g$ is 1, if ϕ^g is 1 and ψ^g is 1.

This suggest that the meaning of $\phi \wedge \psi$ is a truth value, which is obtained out of the truth values for ϕ and for ψ. Another clause of the standard way of interpretation is not compatible with this idea.

$[\exists x\phi(x)]^g$ is 1, if there is a $g' \underset{x}{\sim} g$ such that $[\phi(x)]^{g'}$ is 1.

(Here $g' \underset{x}{\sim} g$ means that g' is the same assignment as g except for the possible difference that $g'(x) \neq g(x)$).

This clause shows that the concept of meaning being a truth value is too simple for our algebraic framework. One cannot obtain the truth value of

$\exists x \phi(x)$ (for a certain value of g) out of the truth value of $\phi(x)$ (for the same g). If we wish to treat predicate logic in our framework, we have to find a more sophisticated notion of meaning for it.

Note that there is not a single truth value in the semantic domain which corresponds with $\phi(x)$. Its interpretation depends on the interpretation of x, and in general on the interpretation of the free variables in ϕ, and therefore on g. In analogy with the previous examples, we incorporate the variable assignment into the conception of meaning. The meaning of a formula is a function from variable assignments to truthvalues, namely the function which yields 1 for an assignment in case the expression is true for that assignment. With this conception, we can easily build the meaning of $\phi \wedge \psi$ out of the meaning of ϕ and of ψ: a function which yields 1 for an assignment iff both the meanings of ϕ and of ψ yield 1 for that assignment. The formulation becomes simpler by adopting a different view of the same situation. A function from assignments to truthvalues can be considered as the characteristic function of a set of assignments. Using this, we may formulate an alternative definition: the meaning of a formula is a set of variable assignments (namely those for which the formula gets the truth value 1). Let M denote the meaning assignment function. Then we have:

$$M(\phi \wedge \psi) = M(\phi) \cap M(\psi).$$

For the other connectives there are related set theoretical operations. Thus this part of the semantic domain gets the structure of a Boolean algebra.

For quantified formulas we have the following formulation.

$$M(\exists x \phi) = \{h \mid h \underset{x}{\sim} g \text{ and } g \in M(\phi)\}.$$

Let C_x denote the semantical operation described at the right hand side of the = sign, i.e. C_x is the operation 'extend the set of assignments with all x variants'. The syntactic operation of writing $\exists x$ in front of a formula now has a semantic interpretation: namely apply C_x to the meaning of ϕ.

$$M(\exists x \phi) = M(\exists x)(M(\phi)) = C_x M(\phi).$$

In this algebraization there are infinitely many operations which introduce the existential quantifier. One might wish to go one step further and produce $\exists x$ from $\exists$ and x. This would require that given the meaning of a variable (being a function from assignments to values) we are able to determine of which variable it is a meaning. This is not an attractive algebraic operation, and therefore this last step is not made. I conclude that we have obtained a compositional interpretation of predicate logic: a homomorphism to some semantic algebra. One might say that it shows how we have to look

at the Tarskian interpretation of logic in order to give it a compositional perspective.

The view on the semantics of predicate logic presented here is not new. Some logic books are based on this approach in which the meaning of a quantified formula is a set of assignments (MONK 1976, p.196, KREISEL & KRIVINE 1976, p.17). The investigations on the algebraic structure of predicate logic constitute a special branch of logic: the theory of cylindric algebras. It requires a shift of terminology to see that the kinds of structures studied there is the same as those introduced here. An assignment can be considered as an infinite tuple of elements in the model: the first element of the tuple is the value for the first variable, etcetera. Thus an assignment can be considered as a point in an infinite dimensional space. So if ϕ holds for a set of assignments, then ϕ is interpreted as the set of corresponding points in this universe. The operator C_x applied to a point p causes that all points are added which differ from p only in their x-coordinate. Geometrically speaking, a single point extends to an infinite stick. If C_x is applied to a set consisting of a circle area, then this is extended to a cylinder. Because of this effect, C_x is called a cylindrification operator, and in particular, the x-th cylindrification. (see fig.5) The algebraic structure obtained in connection with predicate logic is called a cylindric set-algebra. These algebras and their connection with logic are studied in the theory of cylindric algebras (see HENKIN, MONK & TARSKI 1971).

The original motivation for studying cylindric algebras was a technical one. Cylindric algebras were introduced to make the application of the powerful tools of algebra possible in studying logics, as can be read in HENKIN, MONK & TARSKI (1971, p.1):

> *This theory [..] was originally designed to provide an apparatus for an algebraic study of first order, predicate logic.*

New in the above discussion was the motivation which led us towards cylindric algebras. In my opinion, the compositional approach gives rise to a more direct introduction to this field than the existing one. Moreover, on the basis of the approach given above, it is not too difficult to find algebras for other order logics, such as intensional logic.

It cannot be said that the theory of cylindric algebras itself is a flourishing branch of logic nowadays. But the use of algebra is widespread in model theory (i.e. the branch of logic which deals with interpretations). Often one uses the terminology and techniques from universal algebra, as is evidenced by the amount of universal algebra in 'Model theory' by CHANG &

KEISLER (1973), and by the amount of model theory in 'Universal algebra' by GRAETZER (1968). Results from one field are proven using methods from the other field in Van BENTHEM (1979b). Algebraic interpretations of several non-classical logics are given by RASIOWA (1974). Important results concerning modal logics are obtained, using algebraic techniques, by Blok (e.g. BLOK 1980).

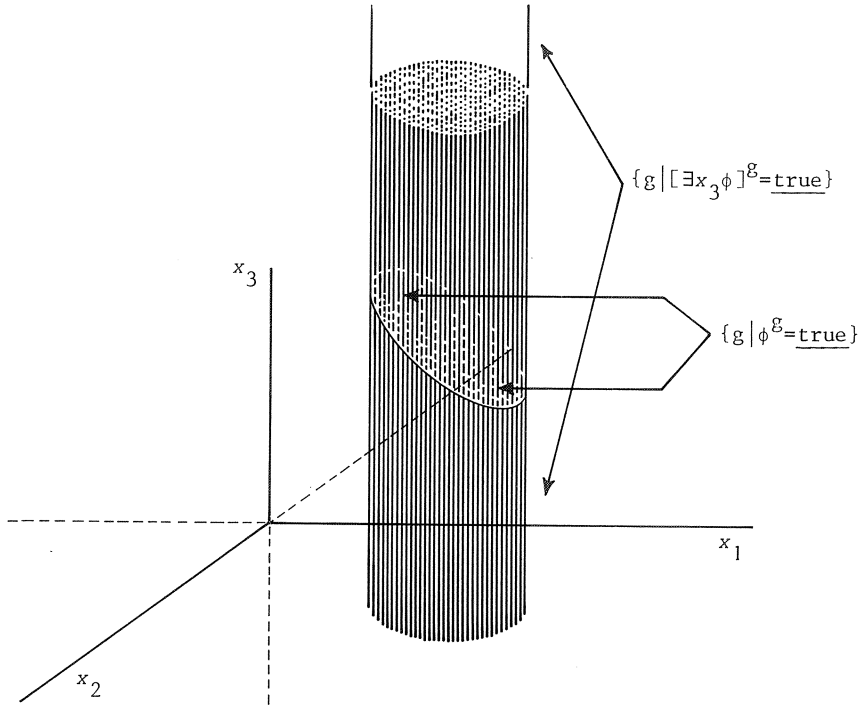


Figure 5. A cylindrification

The present discussion should not be understood as claiming that the only legitimate way of studying (predicate) logic is by means of (cylindric) algebras. There are a lot of topics concerning logic that can be studied, and each has a natural viewpoint. For instance, if one is studying deduction systems, a syntactic point of view is the natural approach. One should take that view which is the best for one's current aims. What I claim is that, if one is studying semantics, then there has to be an algebraic

interpretation existing in the background, and one should take care that this interpretation is not violated by what one is doing.

5.5. Strategy

In all three examples discussed above, we followed the strategy of first investigating what a meaning should do, and then defining such an entity as the formalized notion of meaning which does that and which satisfies the compositionality principle. In all examples such an entity was obtained by giving the notion of meaning a sufficient degree of abstraction. By proceeding in this way (first investigating, then defining) we follow the advice of LEWIS (1970,p.5)

> *In order to say what a meaning is, we may first ask what a meaning does and then find something that does that.*

5.6. Substitutional Interpretation

Next I will discuss an approach to the semantics of predicate logic which is not compositional with respect to the interpretation of quantifiers. For the interpretation of $\exists x[\phi(x)]$ an alternative has been proposed which is called the 'substitutional interpretation'. It says:

> $\exists x\ \phi(x)$ *is true iff there is some substitution* a *for* x *such that* $\phi(a)$ *is true.*

Whether this definition is semantically equivalent to the Tarskian definition depends, of course, on whether the logical language contains a name for every element of the semantic domain or not. A definition like the above one can be found in two rather divergent branches of logic: in philosophical logic, and in proof theory.

In *philosphical logic* the substitutional interpretation has been put forward by R. Marcus (e.g. MARCUS 1962). Her motivation was of an ontological nature. Consider sentence (3).

(3) *Pegasus is a winged horse.*

According to standard logic, (4) is a consequence of (3), and Marcus accepts this consequence.

(4) $\exists x$(*x is a winged horse*).

She argues, however, that one might believe (3), without believing (5).

(5) *There exists at least one thing which is a winged horse.*

This opinion has as a consequence that the quantification used in (4) cannot be considered as an existential quantification in the ontological sense. The substitutional interpretation of quantifiers allows her to have (4) as

a consequence of (3), without being forced to accept (5) as a consequence.

KRIPKE (1976) discusses this approach in a more formal way. As syntax for the logic he gives the traditional syntax: $\exists x\phi(x)$ is produced from $\phi(x)$ by placing $\exists x$ in front of it. According to such a grammar $\phi(a)$ certainly is not a part of $\exists x\phi(x)$. This means that the substitution interpretation is not a compositional interpretation (this was noticed by Tarski, as appears from a footnote in PARTEE (1973,p.74)).

In *proof theory* the substitutional interpretation is given e.g. in SCHUETTE 1977. In his syntax he constructs $\forall x\phi(x)$ from $\phi(a)$, where a is arbitrary. So the formula $\forall x\phi(x)$ is syntactically rather ambiguous: It has as many derivations as there are expressions of the form $\phi(a)$. Given one such production, it is impossible to define the interpretation of $\forall x\phi(x)$ on the basis of the interpretation of the formula $\phi(a)$ from which $\forall x\phi(x)$ was built in the parse under consideration. It may be the case that $\forall x\phi(x)$ is false, and $\phi(a)$ is true for some a, but false for another one. So we see that the truth value of $\forall x\phi(x)$ cannot depend on the truth value of $\phi(a)$ for any single a. Hence in this case the substitutional interpretation does not satisfy the compositionality principle.

If one wishes to define the semantics in a compositional way, and to follow at the same time the substitutional interpretation of quantifiers, then the syntax has to contain an infinitistic rule which says that *all* expressions of the form $\phi(a)$ are part of $\forall x\phi(x)$. Such an infinitistic rule has not been proposed by authors which follow the substitutional interpretation.

6. MOTIVATION

In this section I will give several arguments for accepting the compositionality principle and the formalization given for it. I will give three kinds of arguments. The first kind is very general and argues for working within some mathematically defined framework. The second kind of arguments lists benefits of working with the present framework, and is based upon the properties of the framework. The third kind concerns the principle itself. As a matter of fact, this entire book is intended as a support for the algebraic formalization of the compositionality principle, and many of the arguments will be worked out in the remainder of this book.

Regarding the *first* kind of *arguments*: it is very *useful* to work with-

in some mathematically well defined framework. Such a standard framework gives rise to a language in which one can formulate observations, relations and generalizations. It is a point of departure for formulating extensions, restrictions and deviations. If one has no standard framework, then whenever one considers a new proposal, one has to start anew in obtaining intuitions concerning properties of the system, and to check whether old knowledge still holds. It is then difficult to see whether the proposals within some framework are in accordance with those in other frameworks, and whether they can be combined into a coherent treatment. If one wishes to design a computer program for Montague grammars, then one has to design for each proposed extension or variant a completely new program, unless all proposals fit into a single framework. This experience was my original motivation for the whole research presented in this book. But the final result is independent of this motivation: only at a few places programming considerations are mentioned (viz. here and in chapters 7 and 8).

The *second* kind of *arguments* is based upon the *quality* of the framework.

a) *Elegance*

The framework presented here is mathematically rather elegant. This is apparent especially from the fact that it is based upon two simple mathematical notions: many-sorted algebra and homomorphism. The important tool of a logical language is combined in an elegant way with these algebraic notions. One should, however, not confuse the notion of 'elegant' with 'elementary' or 'easy to understand'. That the system is elegant, is due to its abstractness, and this abstractness might be a source of difficulties in understanding the system. The insight obtained from the abstract view on the framework led to an answer to a question of PARTEE 1973 concerning restrictions on relative clause formation, see chapter 9 or JANSSEN 1981a. It also led to an application in a rather different direction by providing a semantics for Dik's functional grammar, see JANSSEN 1981b.

b) *Generality*

The framework can be applied to a wide variety of languages: natural, programming and logical languages. See chapter 3 for an application to logic, chapter 10 for an application to programming languages, and the other chapters of this book for applications to natural languages.

c) *Restrictiveness*

The framework gives rise to rather strong restrictions concerning the organization of syntax and semantics, and their mutual relation. The use

of polynomial operators especially constitutes a concrete, practical restriction. For a discussion of several deviations from the present framework, see chapters 5 and 6.

d) *Comprehensibility*

The argument given by Milner (see section 1) for designers of computing systems can be generalized to: 'if someone describes the semantics of some language, he should be able to think of the description as a composite of descriptions, in order that he may factor a semantic problem into smaller problems'. And what is said here for the designer of a system, holds at least as much for someone trying to understand the system. This property of the system is employed in the presentation of the fragment in chapter 4.

e) *Power*

The recursive definitions used in the framework allow us to apply the technique of induction. Statements concerning structures and expressions can be proved by using induction to the complexity of the elements involved. Especially in chapters 2 and 3 this power is employed.

f) *Heuristic tool*

A most valuable argument in favor of the principle and its formalization is its benefit for the practice of describing semantics of languages. Examples of this benefit, however, would require a detailed knowledge of certain proposals. Therefore some quotations have to suffice.

ADJ 1979 (p.85) say about the algebraic approach:

> *The belief that the ideas presented here are key, comes from our experience over the last eight years in developing and applying these concepts.*

Furthermore they say (op.cit.p.88):

> *When one becomes familiar with such concepts (and the results concerning them) they provide a guide as to what one should look for, and as to how to formulate one's definitions and results.*

Van EMDE BOAS & JANSSEN 1979 (p.112) claim:

> *It will turn out that quite often some complicated description in a semantic treatment actually hides a deviation from the principle. Confronted with such a violation the principle sometimes suggests an alternative approach to the problematic situation which does obey the principle and solves the problem easier than thought to be possible. Such cases establish the value of the principle as a heuristic tool.*

Both papers contain a lot of evidence for their claims. I will present several examples supporting them: concerning programming languages in chapter 10, and concerning natural languages in the other chapters.

The *last* kind of *arguments* concerns the *principle itself*.

g) *No alternative*

An important argument in favor of the principle is that there is no competing principle. Authors not working in accordance with the principle do not, as far as I know, put forward an alternative general principle with a mathematical formalization. The principles one finds in the literature are language-specific, or specific for a certain theory of languages, but never principles concerning a framework.

h) *Widespread*

As demonstrated in section 1, the principle of compositionality is widespread in sciences dealing with semantics; it arises in philosophy, linguistics, logic and computer science.

i) *Psychology*

An argument sometimes put forward is that the principle reflects something of the way in which human beings understand natural language. The principle explains how it is possible that a human being, with his finite brain, can understand a potentially infinite set of sentences. Or to say it in Frege's words (as translated by Geach & Stoothof (FREGE 1923, p.35)):

> [..] *even a thought grasped by a terrestrial being for the first time can be put into a form of words which will be understood by someone to whom the thought is entirely new. This would be impossible, were we not able to distinguish parts in the thought corresponding to the parts of a sentence* [..].

The last two arguments I do not consider as very strong. As for argument h), I think that the principle is so popular because it is so vague. There are many undefined words in the formulation of the principle, so that everybody can find his own interpretation in it. As for argument i), we know so little about the process in the human brain associated with learning or understanding natural language, that arguments concerning psychological relevance are no more than speculations. I would not like to have the mathematical attractiveness of the framework disturbed by further speculations of this nature. The most valuable arguments are, in my opinion, those concerning the elegance and power of the framework, its heuristic value, and the lack of a mathematically well defined alternative. So I adhere to the principle for the technical qualities of its formalization.

An argument *not* found above is the *truth* of the principle: a statement like 'The semantics of English is compositional'. Such an argument would not be convincing since it is circular. In section 5, I gave examples which illustrated that the principle, and especially the requirement of similarity, may lead us to a certain conception of meaning. And in section 3 I gave a

definition of the notion 'parts' which made it possible to have 'abstract parts'. So there is a large freedom: we may choose what the parts are of an expression, and what the meanings are of those parts. In such a situation it is not surprising that there is some choice which gives rise to a compositional treatment of the semantics. In the next chapter I will prove that it is possible within this framework to generate every recursively enumerable language, and to relate with every sentence any meaning we would like. If someone wishes to doubt the principle, this only seems possible if he has some judgements at forehand about what the parts of an expression are, and what their meanings are. In the light of the power and flexibility of the framework, it cannot be refuted by pointing out in some language a phenomenon which requires a non-compositional treatment. I expect that problematic cases can always be dealt with by means of another organization of the syntax, resulting in more abstract parts, or by means of a more abstract conception of meaning. The principle only has to be abandoned if it leads too often to unnecessarily complicated treatments.

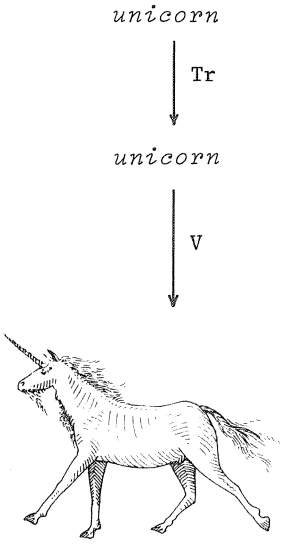
As appears from this discussion, the principle of compositionality is not a principle about languages. It is a principle concerning the organization of grammars dealing both with syntax and semantics. The arguments given above for adhering to the principle, are not based on phenomena in languages, but on properties of grammars satisfying the framework. If one is not pleased with the power of the grammars, one might formulate severe restrictions within the framework. In the light of the examples to be given in chapter 5, it seems that the framework as it is, gives, from a practical viewpoint, already more than enough restrictions.

CHAPTER II

THE ALGEBRAIC FRAMEWORK

ABSTRACT

In this chapter a formal framework is defined for the description of the syntax and semantics of languages. The theory of many-sorted algebra which is needed for this framework is explained, and special attention is paid to the motivation and mathematical justification of the framework. The framework is a synthesis of the approaches of Montague and Adj and it constitutes a formalization of the principle of compositionality of meaning.



1. INTRODUCTION

The aim of this chapter is to present a mathematical description of a framework for the description of syntax and semantics of a language. The framework is a formalization of the principle of compositionality of meaning. The framework is based upon universal algebra: a branch of mathematics which is concerned with the general theory of algebraic structures (the standard work in this field is GRAETZER 1968). Universal algebra deals with the general structures we need, and it provides a language which allows us to speak with precision about such abstract structures. The most important contribution of universal algebra to this book consists of the concepts it provides. I will hardly use any deep mathematical results from universal algebra, but mainly rather elementary notions such as 'subalgebra', 'homomorphism' and 'polynomial' (here generalized to the case of many sorted algebras).

The framework I will present is designed with two predecessors in mind: 'Universal Grammar' (MONTAGUE 1970b), and 'Initial algebra semantics' (ADJ 1977). Montague did not use many sorted algebras, although it is the natural mathematical notion for his purposes. The group Adj was not primarily interested in developing a general framework, but in its practical applications. The present framework is based upon the ideas of Montague, and on the techniques of Adj, and as such it is new. In a few cases, a definition or theorem concerning this framework deviates considerably from what can be found in the literature. The present framework is developed for practical purposes, and I constantly kept PTQ (MONTAGUE 1973) and its successors in mind. As often happens in applying mathematics, the available theory was not applicable in its original form. I had to invent definitions myself, with the literature as a source of analogous notions (this point is also made in Van BENTHEM 1979a,p.17). In the presentation much attention is paid to the motivation of the definitions: if one understands why definitions are the way they are, then it is possible to predict what happens when the conditions in the definitions are violated. The insights developed in this chapter will also be useful in the discussion of several deviations from the framework (see chapter 5). My aim is to give a comprehensible description of an elegant, very abstract mathematical system. In one respect this attempt probably has not been successful: the description of how to obtain new algebras out of old ones. There is no general theory which I could use here, and I had to apply 'ad hoc' methods (see sections 6 and 7).

2. ALGEBRAS AND SUBALGEBRAS

In chapter 1 it was explained that the key notions in our formalization of the compositionality principle are the notions 'many-sorted algebra' and 'homomorphism'. For several reasons these definitions have to be refined. The definition of 'many sorted algebra' is given below; the definition of 'homomorphism' will be given in section 6.

2.1. DEFINITION. A *many-sorted algebra* of *signature* (S,Γ,τ) is a pair $\langle (A_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ such that

a) S is a non-empty set; its elements are called *sorts*.

b) $(A_s)_{s\in S}$ is an indexed family of sets. The set A_s is called the *carrier* of sort s.

c) Γ is a set; its elements are called *operator indices*.

d) τ is a function such that

$\tau\colon \Gamma \to \bigcup_n S^n \times S$ where $n \in \mathbb{N}$ and $n > 0$.

Thus the function τ assigns to each operator index γ a pair $\langle w,s\rangle$, where s is a sort, and $w = \langle s_1,\ldots,s_n\rangle$ is an n-tuple of sorts. Such a pair denotes the type of the operator with index γ. Therefore the pair is called an *operator type*, and the function τ is called a *type assigning function*.

e) $(F_\gamma)_{\gamma\in\Gamma}$ is an indexed family of *operators* such that if $\tau(\gamma) = \langle\langle s_1,\ldots,s_n\rangle,s_{n+1}\rangle$ then $F_\gamma\colon A_{s_1}\times\ldots\times A_{s_n} \to A_{s_{n+1}}$

2.1. END

The definition given above is due to J. Zucker (pers.comm.); it is very close to the one given in ADJ 1977. The main difference is that we have no restrictions on the carriers: they may have an overlap, be included in each other or some may be equal. Another, minor, difference is that we have no nullary operators (i.e. it is not allowed in clause d) that n=0). For a motivation and discussion of these differences, see sections 8 and 9. Structures like many sorted algebras are introduced, under different names, in BIRKHOFF & LIPSON 1970 (heterogeneous algebras) and HIGGINS 1963 (algebras with a scheme of operators).

The different components of the above definition are illustrated in the following example

2.2. EXAMPLE. I describe an algebra E consisting of some English words and sentences.

a) The set of sorts S = {Sent, Term, Verb}

b) The carriers are

E_{Term} = {*John*, *Mary*}

E_{Verb} = {*run*}

E_{Sent} = {*John runs*, *Mary runs*}

c) The set of operator indices $\Gamma = \{1\}$

d) The type assigning function τ is defined by $\tau(1) = \langle\langle Term, Verb\rangle, Sent\rangle$.

e) The set of operators is $\{F_1\}$. This operator consists of first adding an *s* to its second argument, followed by a concatenation of its first argument with its thus changed second argument. So

$F_1(\alpha,\beta) = \alpha\ \beta s$ (e.g. F_1(*John*, *run*) = *John runs*)

2.2. END

In this example I have followed the definition in order to illustrate the definition. It is, however, not the most efficient way to represent the required information. There are several conventions which facilitate these matters. The sorts may be used to denote the carriers as well: we will write $a \in s$ instead of $a \in A_s$. We often will write A when $(A_s)_{s\in S}$ or $\bigcup_{s\in S}(A_s)$ is meant, but we will use A for the algebra itself as well. By a Σ-algebra we understand an algebra with signature Σ. We will avoid to mention S,Γ and τ when they become clear from the context (or are arbitrary). These conventions are employed in the example which will be given below. A final remark about the notation in MONTAGUE 1970b. There an algebra is denoted as $\langle A_s, F_\gamma\rangle_{s\in S,\gamma\in\Gamma}$. I agree with LINK & VARGA (1975) that this is not a correct notation for what is intended: an algebra is not a collection of pairs, but a pair consisting of two collections (the carriers and the operators).

2.3. EXAMPLE. The algebra <A,F> is defined as follows.

Its sorts are

Nat = {0,1,2,3..} (the natural numbers)

Bool = {true,false} (the truth values)

Its operators are

$F_<$: Nat × Nat → Bool where $F(\alpha,\beta) = \begin{cases} \text{true} & \text{if } \alpha < \beta \\ \text{false} & \text{otherwise} \end{cases}$

$F_>$: Nat × Nat → Bool where $F(\alpha,\beta) = \begin{cases} \text{true} & \text{if } \alpha > \beta \\ \text{false} & \text{otherwise} \end{cases}$

So $\langle A_s, F_\gamma\rangle$ is a two sorted algebra with two operators of the same type. S,Γ and τ are implicitly defined by the above description.

2.3. END

It is useful to have some methods to define a new algebra out of an old one. An important method is by means of subalgebras. A subalgebra is, roughly, a collection of subsets of the carriers of an algebra which are closed under the operations of the original algebra. The theorems and definitions which follow, are generalizations of those for the one-sorted case in GRAETZER 1968.

2.4. DEFINITION. Let $A = \langle (A_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ be an algebra. A *subalgebra* of A is an algebra

$$B = \langle (B_s)_{s\in S}, (F'_\gamma)_{\gamma\in\Gamma}\rangle$$

such that

1) for each $s \in S$ it holds that $B_s \subset A_s$
2) for each $\gamma \in \Gamma$ it holds that $F'_\gamma = F_\gamma \restriction B$. (i.e. F'_γ is the restriction of F_γ to B).

2.4. END

Note that from the requirement that B is an algebra, it immediately follows that $F'_\gamma(b_1,\ldots,b_n) \in B$. In the sequel we will not distinguish operators of the original algebra and operators of its subalgebras (e.g. we will not use primes to distinguish them). The next example illustrates this.

2.5. EXAMPLES. Let E be the algebra from example 2.2. Hence E is defined by:

$$E = \langle (E_s)_{s\in\{Term,Verb,Sent\}}, \{F_1\}\rangle$$

where $E_{Term} = \{John, Mary\}$, $E_{Verb} = \{run\}$,

$E_{Sent} = \{John\ runs,\ Mary\ runs\}$

and $F_1 : E_{Term} \times E_{Verb} \to E_{Sent}$

is defined by $F_{(\alpha,\beta)} = \alpha\ \beta s$.

Some examples of subalgebras are:

I. E itself is a subalgebra of E.

II. Let $B_{Term} = \{John\}$, $B_{Verb} = \{run\}$, and $B_{Sent} = \{John\ runs\}$. Then $B = \langle (B_s)_{s\in S}, \{F_1\}\rangle$ is a subalgebra of E.

III. Let $C_{Term} = \{Mary\}$, $C_{Verb} = \{run\}$, and $C_{Sent} = \emptyset$. Then $C = \langle (C_s)_{s\in S}, \{F_1\}\rangle$ is *not* a subalgebra of E.

IV. Let D_{Term} = {*John*}, D_{Verb} = $\emptyset$, and D_{Sent} = {*Mary runs*}.
Then $D = \langle (D_s)_{s\in S}, \{F_1\}\rangle$ is a subalgebra of E, although a rather strange one.

2.5. END

A sorted collection of subsets of an algebra may be contained in several subalgebras. There is a smallest one among them, namely the intersection of these subalgebras. This is proven in the next theorem, this probably corresponds with the mysterious 'theorem due to Perry Smith' mentioned in MONTAGUE 1973 (p.253).

2.6. THEOREM. *Let* $\langle (B_s^{(i)})_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle_{i\in I}$ *be a collection subalgebras of the algebra* $\langle (A_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$. *For each s we define,* $C_s = \cap_{i\in I}(B_s^{(i)})$. *Then* $\langle C_s, F_\gamma\rangle$ *is a subalgebra of* A.

PROOF. We have to prove that $\langle C_s, F_\gamma\rangle$ is closed under the operations F_γ.
Let $\tau(\gamma) = \langle\langle s_1,\ldots,s_k\rangle, s_{k+1}\rangle$ and $c_1 \in C_{s_1}, \ldots, c_k \in C_{s_k}$.
Then for all $i \in I$: $c_1 \in B_{s_1}^{(i)}, \ldots, c_k \in B_{s_k}^{(i)}$.
So for all i: $F_\gamma(c_1,\ldots,c_k) \in B_{s_{k+1}}^{(i)}$, and consequently
$F_\gamma(c_1,\ldots,c_k) \in \cap_{i\in I} B_{s_{k+1}}^{(i)} = C_{s_{k+1}}$.

2.6. END

We will often be interested in the smallest algebra containing a given collection of subsets. Then the following terminology is used.

2.7. DEFINITIONS. Let $\langle A, \underline{F}\rangle$ be an algebra, and H a sorted collection of subsets of A. The smallest subalgebra of A containing H is called the *subalgebra generated by* H. This algebra is denoted by $\langle [H], \underline{F}\rangle$, and its elements are denoted by [H]. A sorted collection H of subsets of an algebra $\langle A, \underline{F}\rangle$ is called a *generating set* if $\langle [H], \underline{F}\rangle = \langle A, \underline{F}\rangle$. The elements of the sets in H are called *generators*.

2.7. END

Theorem 2.6. characterizes the algebra $\langle [G], \underline{F}\rangle$ as the intersection of all subalgebras containing G. Another characterization will be given in section 4.

An important consequence of theorem 2.6 is that it allows us to use the power of induction. A property P can be proved to hold for all elements of an algebra $\langle A, \underline{F}\rangle$ by proving that:

1) Property P holds for a generating set G of A
2) The set B = {a ∈ A | P(a)} is closed under all $F_\gamma \in \underline{F}$.
(i.e. if $b_1,\ldots,b_n \in B$ and F_γ is defined for them, then $F_\gamma(b_1,\ldots,b_n) \in B$).

From 2) it follows that <B,$\underline{F}$> is a subalgebra of <A,$\underline{F}$>. From 1) it follows that G ⊂ B, hence <[G], $\underline{F}$> is a subalgebra of <B,$\underline{F}$>. Since A = <[G],$\underline{F}$> it follows that A = <B,$\underline{F}$>. So for all a ∈ A property P holds.

Theorem 2.6 provides us an easier way to present subalgebras than the method used in example 2.5. The theorem shows that it is sufficient to give a set of generators.

2.8. EXAMPLES. The subalgebra mentioned in example 2.5, case II, can be denoted as:

$$\langle[\{John\}_{\text{Term}}, \{run\}_{\text{Verb}}], \{F_1\}\rangle$$

where F_1: Term × Verb → Sent is defined by $F_1(\alpha,\beta) = \alpha\ \beta s$.
Note that the sorts of the generators are mentioned in the subscripts.

The subalgebra mentioned in example 2.5, case III, can be denoted as:

$$\langle[\{John\}_{\text{Term}}, \{Mary\ runs\}_{\text{Sent}}]\rangle.$$

This algebra is 'generated' in the formal sense; it is however intuitively strange to have a compound expression (*Mary runs*) as generator.
2.8 END

If the 'super' algebra within which we define a subalgebra is clear from the context, we need not to mention this algebra explicitly. This gives a simplification of the presentation of the subalgebra. Such a situation arises when we wish to define some language, i.e. a subset of all strings over some alphabet. In this situation one may conclude from the generators what the elements of the alphabet are, and the 'super' algebra is the algebra with as carrier all finite strings over this alphabet. An example is given below.

2.9. EXAMPLE. We define an algebra N; the carrier of this algebra consists of denotations for numbers. Leading zero's are not accepted, so *700* is an element of N, whereas *007* is not. This algebra is described by:

$$N = \langle [\{0,1,\ldots,9\}_{Num}], \{F\} \rangle$$

where F: Num × Num → Num is defined by

$$F(\alpha,\beta) = \begin{cases} \beta & \text{if } \alpha = 0 \\ \alpha\beta & \text{otherwise.} \end{cases}$$

Now it is implicitly assumed that N determines a subalgebra of

$$A = \langle \{0,1,\ldots,9\}^*_{Num}, \{F\} \rangle$$

where F is as just defined, and $\{0,1,\ldots,9\}^*$ is the set of all strings formed of symbols in the set $\{0,1,\ldots,9\}$.

The difference between A and N is that A contains all strings (*007* included), whereas this is not the case for N. Notice that N is highly ambiguous in the sense that its elements can be obtained from the generators in several ways

e.g. $F(1,7) = 17$ but also $F(0,F(1,7)) = 17$.

2.9. END

The above example concerns an algebra with only one sort. In 2.2 and 2.5 we defined algebras with several sorts, and when we consider a subalgebra of such algebras, we can also avoid writing explicitly the 'super' algebra. In that case the most simple algebra we may take as the 'super' algebra, is the one in which all carriers consist of all possible finite strings.
An example as illustration:

2.10 EXAMPLE. We define an algebra M for number denotations, in which leading zero's are accepted, and in which each element can be obtained from the generators in only one way.

$$\underline{M} = \langle [\{0,1,\ldots,9\}_{dig}], \{F_1, F_2\} \rangle$$

where F_1: dig → num is defined by $F_1(\alpha) = \alpha$
and F_2: num × dig → num is defined by $F_2(\alpha,\beta) = \alpha\beta$.

So F_1 says that all digits are number denotations, and F_2 says that one obtains a new denotation by concatenating the old denotation with a digit (in this order).

e.g. $F_2(F_1(7),1) = 71$ and $F_2(F_2(F_1(0),0),7) = 007$.

The implicit 'super' algebra is

$$<\{\{0,1,\ldots,9\}^*_{dig},\{0,1,\ldots,9\}^*_{num}\},\{F_1,F_2\}>$$

2.11. EXAMPLE. Another algebra for number denotations is one which differs from the above one in only one respect. Digits are concatenated with numbers for obtaining new numbers (and not in the opposite order as in 2.10).

$\underline{M}' = <[\{0,1,\ldots,9\}_{dig}],\{F_1,F_3\}>$
where F_1: dig → num defined by $F_1(\alpha) = \alpha$
and F_3: num × dig → num defined by $F_3(\alpha,\beta) = \beta\alpha$.

2.11.END

In the examples 2.8/2.11 subalgebras are defined by mentioning a generating set. In all these examples this was a special set: one which was minimal in the sense that none of the generators can be obtained from the other generators. The following terminology can be used to describe this situation.

2.12. DEFINITIONS. A collection B of generators of algebra <A,F> is called A-*independent* if for all b ∈ B holds that b ∉ <[B-{b}],F>.
An algebra <A,F> is called *finitely generated* if A = <[B],F> where B is some finite A-independent generating set of A.
An algebra is called *infinitely generated* if it is not finitely generated.
A collection of generators G is called *the generating set* of the algebra if the algebra is generated by that set and if all generating collections contain G as subcollection.
2.12. END

3. ALGEBRAS FOR SYNTAX

In most examples we have considered, the carriers consisted of strings of symbols. Such algebras can be used to describe languages, and in fact we did so in the previous section. The set we were interested in was the carrier of a certain sort. This should explain the following definition (the epithet 'general' will be dropped in a more restrictive variant).

3.1. DEFINITION. A *general algebraic grammar* G is a pair <A,s>, where A is a many-sorted algebra, s is a sort of A, and all carriers of A consist of strings over some alphabet. The sort s is called the *distinguished sort*, and the carrier of sort s is called the *language generated* by G, denoted L(G).

3.2. EXAMPLE. Let E be the algebra $\langle[\{\textit{Mary}, \textit{John}\}_{\text{Term}}, \{\textit{run}\}_{\text{Verb}}], \{F_1\}\rangle$, where F_1: Term × Verb → Sent is defined by $F_1(\alpha,\beta) = \alpha\ \beta s$.
Then the general algebraic grammar <E,Sent> generates the language {*Mary runs*, *John runs*}; and the general algebraic grammar <E,Verb> generates the language {*run*}.
3.2. END

First a warning. In the French literature one finds the notion 'grammaire algebraique'. This notion has nothing to do with the algebraic grammars we will consider here: 'grammaire algebraique' means the same as 'context free grammar'. In the definition above, I have not used the name algebraic grammar because I will use it for a subclass of the general algebraic grammars. Most general algebraic grammars are not interesting because they do not provide the information needed to generate expressions of the language. This is illustrated by the trivial proof of the statement that there is for any language L (even for non-recursively enumerable ones) a general algebraic grammar generating L. Take the grammar which has as algebra the one with no operators, one sort and L as carrier of that sort. This is an uninteresting grammar. It is not sufficient to add the requirement 'finitely generated'; this is illustrated by the following example.

3.3. EXAMPLE. Let L be some nonempty language over some finite vocabulary V. Let w be an arbitrary element of L. Consider now algebra A

$$A = \langle [V_{s_1}], \{F_1, F_2\} \rangle$$

where $F_1: s_1 \times s_1 \to s_1$ defined by $F_1(\alpha,\beta) = \alpha\beta$
and $F_2: s_1 \to s_2$ defined by $F_2(\alpha) = \begin{cases} \alpha & \text{if } \alpha \in L \\ w & \text{otherwise.} \end{cases}$

So F_1 generates all strings over V, whereas F_2 selects those strings which belong to L. The definition of an algebra requires that F_2 be a function, so that F_2 delivers some element of sort s_2 even in case its argument is not in L. For this purpose we use the expression w from L. Now $\langle A, s_2 \rangle$ is a finitely generated algebraic grammar with generated language L.
3.3. END

In case L is empty, then we may take a grammar with an empty generating set: $\langle\langle [\emptyset_{s_1}], \{F_1, F_2\} \rangle, s_1 \rangle$.

The crux of the above example lies in the operation F_2. There is no algorithm which for every argument yields its image under F_2. So the general algebraic grammar does not provide us with the information which allows us to generate expressions of the language. The absurdity of such a grammar becomes evident if we replace F_2 by the function F_3:

$F_3(\alpha) = \alpha$ if α is an English sentence, and $F_3(\alpha) = w$ otherwise.

The above example shows that for certain generalized algebraic grammars there exists no algorithm which produces the expressions of the language defined by the grammar. The example also illustrates that such grammars are uninteresting for practical purposes. Therefore we will restrict our attention to those algebraic grammars for which there is an algorithm for producing the expressions of the grammar. For this purpose I require that the operators of the grammars be recursive (the notion 'recursive' is the formal counterpart of the intuitive notion 'constructive', see e.g. ROGERS 1967). But this requirement is not sufficient: a grammar might have only recursive operators, whereas the definition of the set of operators is not recursive. Then we would not know of an arbitrary operator whether it is an operator of the grammar, i.e. we do not effectively have the tools to produce the expressions of the language of that grammar, although the tools themselves are recursive. Therefore I also require that there exists some recursive function which decides whether any given operator belongs

to the set of operators of the grammar, in other words, that the set of operators be recursive. For similar reasons it is required that the sets of sorts and generators be recursive. In section 5 it will be proven that for such grammars there indeed exists an algorithm generating the expressions of the language defined by the grammar. A more liberal notion is 'enumerable algebraic grammar'. Also for these grammars there exists a generating algorithm, but they have some unattractive properties (e.g. it is undecidable whether a given derivational history is one from a given grammar. Formal definitions concerning recursivity and algebras are given in e.g. RABIN 1960, but the intuitive explication given above is sufficient for our purposes.

3.4. DEFINITION. An *algebraic grammar* is a general algebraic grammar such that
1. its set of operators and its set of sorts are recursive, and it has a recursive generating set
2. all its operators are recursive.

3.5. DEFINITION. An *enumerable algebraic grammar* is a general algebraic grammar such that
1. its set of operators and its set of sorts are recursively enumerable, and it has a recursively enumerable generating set
2. all its operators are recursive.

3.6. DEFINITION. A *finite algebraic grammar* is an algebraic grammar such that
1. its set of operators, and its set of sorts are finite, and it has a finite generating set
2. all its operators are recursive.

3.6. END

I have formally described what kind of language definition device we will use. Next it will be investigated whether our device restricts the class of languages which can be dealt with. The theorem below is of great theoretical impact. It says that, even when using finite algebraic grammars we can deal with the same class of languages as can be generated by the most powerful language definition devices (Turing machines, Chomsky type 0 languages, van Wijngaarden grammars, recursive functions). This means that the requirement of using an algebraic grammar, which was one of the consequences of the compositionality principle, is a restriction only on

the organisation of the syntax, but not on the class of languages which can be described by means of an algebraic grammar. The theorem, however, is, from a practical point of view not useful because it does not help us in any way to find a grammar for a given language; this appears from the fact that the proof neglects all insights one might have about the structure of the language: the sorts in the proof have nothing to do with intrinsic properties of the language under consideration.

3.7. THEOREM. *For each recursively enumerable language over some finite alphabeth there exists a finite algebraic grammar that generates the same language.*

PROOF. Let G be a type-0 grammar. So, following the definition in HOPCROFT & ULLMAN (1979, p. 79) we have

$$G = (V_N, V_T, P, S)$$

where V_N, V_T, P, and S are respectively the non-terminal symbols of the grammar, the terminal symbols, the production rules and the start symbol. The set P consists of a finite list of rules $p_1, \ldots, p_n$ of the form $\mu \rightarrow \nu$ where $\mu \in (V_N \cup V_T)^+$ and $\nu \in (V_N \cup V_T)^*$; so μ is a nonempty string of symbols over $V_N \cup V_T$, and ν is a possibly empty string over this set.

We have to prove that there is a finite algebraic grammar A such that $L(A) = L(G)$. I distinguish two cases. I) $L(G) = \emptyset$ and II) $L(G) \neq \emptyset$. In case I we take an algebra with an empty set of generators, and that gives us a finite algebraic grammar. In case II we know that there is at least one expression in L(G). Let $e \in L(G)$. Then the finite algebraic grammar A for L(G) is defined below.

There are three sorts in A:

In : the sort which contains the only generator of the algebra: the symbol S.

Mid: the sort of intermediate expressions

Out: the sort of resulting expressions; i.e. the sort of the generated language.

The operations of the algebra will simulate derivations of G. The symbol \$ is used to focus our attention on that part of the string on which an operator of algebra A is applied which simulates some rule of G. If α is some string, we understand by α' the result of deleting from α all occurrences of \$.

The algebra A is defined as follows:

$$A = \langle\langle [S_{In}], (F_\gamma)_{\gamma\in\Gamma}\rangle, Out\rangle$$
$$\text{where } \Gamma = \{1,2,3,4\} \cup P.$$

The operators are defined as follows:

F_1: In → Mid

$F_1(\alpha) = \$\alpha$

F_2: Mid → Mid

$F_2(\alpha_1\$v\alpha_2) = \alpha_1 v\$\alpha_2$ where $\alpha_1,\alpha_2 \in (V_N \cup V_T)^*$ and $v \in V_N \cup V_T$

$F_2(\alpha) = \alpha$ if α is not of the form $\alpha_1\$v\alpha_2$

F_3: Mid → Mid

$F_3(\alpha_1 v\$\alpha_2) = \alpha_1\$v\alpha_2$ where $\alpha_1,\alpha_2 \in (V_N \cup V_T)^*$ and $v \in V_N \cup V_T$

$F_3(\alpha) = \alpha$ if α is not of the form $\alpha_1 v\$\alpha_2$

F_4: Mid → Out

$F_4(\alpha) = \alpha'$ if $\alpha \in (V_T \cup \{\$\})^*$, so F_4 deletes the occurrences in α of \$

$F_4(\alpha) = e$ if $\alpha \notin (V_T \cup \{\$\})^*$; remember that $e \in L(G)$.

F_{p_i}: Mid → Mid

$F_{p_i}(\alpha_1\$\mu\alpha_2) = \alpha_1\$\nu\alpha_2$ where p_i is $\mu \to \nu$

$F_{p_i}(\alpha) = \alpha$ if α is not of the form just mentioned.

Note that F_{p_i} is a function since the \$-mark indicates to which expression p_i is applied.

The proof that L(G) is generated by this grammar follows from the two lemmas below. But first some definitions.

W is the set of all finite strings over $V_N \cup V_T \cup \{\$\}$ in which at most one \$ symbol occurs

$W_\$$ is the subset of W of strings in which precisely one \$ symbol occurs

$\alpha \overset{}{\underset{A}{\Rightarrow}} \beta$ iff $\beta = F_\gamma(\alpha)$ for some $\gamma \in \Gamma$

$\alpha \overset{}{\underset{G}{\Rightarrow}} \beta$ iff $\alpha = \delta\mu\varepsilon$, $\beta = \delta\nu\varepsilon$ and $\mu \to \nu \in P$, where $\delta, \varepsilon \in (V_N \cup V_T)^*$

$\overset{*}{\underset{A}{\Rightarrow}}$ is the transitive and reflexive closure of $\overset{}{\underset{A}{\Rightarrow}}$

$\overset{*}{\underset{G}{\Rightarrow}}$ is the transitive and reflexive closure of $\overset{}{\underset{G}{\Rightarrow}}$

Recall that we defined α' as the result of deleting all \$ marks from α.

LEMMA. $L(G) \subset L(A)$.

PROOF. First we prove that $\alpha' \underset{G}{\Rightarrow} \beta'$ implies $\alpha \overset{*}{\underset{A}{\Rightarrow}} \beta$ for all $\alpha,\beta \in W_{\$}$.
Consider the following three cases:

1. $\alpha' = \beta'$ and $\alpha = \beta$. Then $\alpha \overset{*}{\underset{A}{\Rightarrow}} \beta$.
2. $\alpha' = \beta'$ and $\alpha \neq \beta$. Then α contains a \$ in a different position than in β. By repeated application of F_2 or F_3 the \$ sign can be moved to that position.
3. $\alpha' = \delta\mu\varepsilon$ and $\beta' = \delta\nu\varepsilon$ and $\mu \to \nu \in p_i$, where $\delta, \varepsilon \in (V_N \cup V_T)^*$
   So $\alpha \overset{*}{\underset{A}{\Rightarrow}} \delta\$\mu\varepsilon$ (using F_2 or F_3); $\delta\$\mu\varepsilon \underset{A}{\Rightarrow} \delta\$\nu\varepsilon$ (using F_{p_i}) and $\delta\$\nu\varepsilon \overset{*}{\underset{A}{\Rightarrow}} \beta$ (using F_3 or F_2.
   So $\alpha \overset{*}{\underset{A}{\Rightarrow}} \beta$.

Suppose now that $w \in L(G)$, so $S \overset{*}{\underset{G}{\Rightarrow}} w$. Hence $(\$S)' \overset{*}{\underset{G}{\Rightarrow}} (\$w)'$. Repeated application of the argumentation given above shows that $\$S \overset{*}{\underset{A}{\Rightarrow}} \$w$. Since $S \underset{A}{\Rightarrow} \$S$ and $\$w \underset{A}{\Rightarrow} w$, it follows that $w \in L(A)$.

LEMMA. $L(A) \subset L(G)$.

PROOF. We first prove that $\alpha \underset{A}{\Rightarrow} \beta$ implies $\alpha' \overset{*}{\underset{G}{\Rightarrow}} \beta'$ for all $\alpha,\beta \in W\backslash\{e\}$.
Consider the following five cases

1. $\beta = F_1(\alpha)$. Then $\alpha = S$, $\beta = \$S$ so $\alpha' = \beta'$ hence $\alpha' \overset{*}{\underset{G}{\Rightarrow}} \beta'$
2. $\beta = F_2(\alpha)$. Then $\alpha' = \beta'$.
3. $\beta = F_3(\alpha)$. Then $\alpha' = \beta'$.
4. $\beta = F_4(\alpha)$. Since $\beta \neq e$ we have $\alpha' = \beta'$.
5. $\beta = F_p(\alpha)$ for $p = \mu \to \nu$. Then either $\alpha = \beta$, or $\alpha = \delta\$\mu\varepsilon$ and $\beta = \delta\$\nu\varepsilon$. So $\alpha' \underset{G}{\Rightarrow} \beta'$.

Suppose now that $w \in L(A)$, so $S \overset{*}{\underset{A}{\Rightarrow}} w$. By repeated application of the above argumentation we find that $S \overset{*}{\underset{G}{\Rightarrow}} w$. Hence $L(A)\backslash\{e\} \subset L(G)\backslash\{e\}$. Since $e \in L(G)$ it follows that $L(A) \subset L(G)$.
LEMMA END

From the above two lemmas it follows that $L(A) = L(G)$.
3.7. END

Note that the proof of this theorem does not provide an algorithm for making a finite algebraic grammar for a given type-0 grammar G. The decision whether we are in case I ($L(G) = \emptyset$), or in case II ($L(G) \neq \emptyset$), is not an effective decision because there exists no algorithm which decides whether a given type-0 grammar produces an empty language (HOPCROFT & ULLMAN 1979, p. 218 and p. 189). This non-constructive aspect of the proof

is unavoidable, as will be proved in the next section.

We aim at a kind of grammar which produces only recursively enumerable languages. The theorem has as a consequence that in all cases a finite grammar is sufficient for the syntax. Nevertheless, we will, in the following chapters, frequently use infinite grammars. Such a decision is motivated mainly by semantic considerations.

4. POLYNOMIALS

In this section a method will be presented for describing new operators: polynomials. I will first present an example that is based upon high school algebra. Consider the polynomial *7y+1*. This polynomial is a compound symbol that defines a certain function. The value of this function for a given argument is obtained by substituting the argument in the polynomial for the variable *y* and calculating the outcome. So its value for argument *2* is *7.2+1*, being *15*, and for argument *1* it is *8*. From the basic operations of multiplication and addition we have built in this way a new operation. The fact that the polynomial contains a multiplication operation is not evident from the notation; it might be emphasized by writing the polynomial as *7.y+1*. In less familiar algebras the operation symbols are not written between their arguments, but in front. Using this function-argument notation the polynomial gets the form *+(.(7,y),1)*. Functions with several arguments are obtained from polynomials with several variables. An example is the polynomial $7y_1+5y_2$, or equivalently $+(.(7,y_1),.(5,y_2))$. It represents a function which has for $y_1=1$ and $y_2=2$ the value *17*. In order to let the polynomial denote a unique function on pairs of integers, we need a convention which determines what the first argument is and what the second. The convention is that this corresponds with the indices of the variables. For the last example this means that the value for the pair *(0,4)* is *20* and not *28*.

The notions discussed above are defined abstractly for the one sorted algebras in GRAETZER 1968. Below I will generalize them to the case of many-sorted algebras. The definitions are somewhat more complicated than in the one sorted case because it is not evident what the first argument and what the second argument is of a polynomial like $P_1(y_1)$. The definition is based upon a suggestion of Jim Thatcher (pers.comm.).

4.1. DEFINITIONS. Let $\underline{A} = \langle (A_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ be a many sorted algebra. For each $s \in S$ we introduce a set VAR_s consisting of countably many variables:

$$VAR_s = \{x_{1,s}, x_{2,s}, \dots\} \qquad VAR = \bigcup_{s\in S} VAR_s.$$

For each element a of A we introduce a symbol $\underline{a}$

$$CON^A_s = \{\bar{a} \mid a \in A_s\} \qquad CON^A = \bigcup_{s\in S} CON^A_s.$$

For each operator F_γ of type $\langle w,s\rangle \in S^n \times S$ we introduce a symbol $\bar{F}_\gamma$

$$Op^A_{\langle w,s\rangle} = \{\bar{F}_\gamma \mid F_\gamma \in (F_\gamma)_{\gamma\in\Gamma} \text{ and } \tau(\gamma) = \langle w,s\rangle\}$$

$$Op^A = \bigcup_{\langle w,s\rangle \in S^n\times S} \left(Op^A_{\langle w,s\rangle}\right).$$

Let $w \in S^n$ be $\langle s_1,\dots,s_n\rangle$. Then we define

$$X^w = \{x_{1,s_1}, x_{2,s_2}, \dots, x_{n,s_n}\}.$$

4.2. DEFINITIONS. Let A be a many-sorted algebra. By $POL^A_{\langle w,s\rangle}$ we understand the set of *polynomial symbols* over A of type $\langle w,s\rangle$. These sets are inductively defined as follows:

I. If $x_{j,s} \in X^w$, then $x_{j,s} \in POL^A_{\langle w,s\rangle}$

II. If $\bar{c}_s \in CON^A_s$, then $\bar{c}_s \in POL^A_{\langle w,s\rangle}$

III. If $\bar{F}_\gamma \in Op^A_{\langle\langle s_1,\dots,s_k\rangle s_{k+1}\rangle}$ and $p_1 \in POL^A_{\langle w,s_1\rangle}, \dots, p_k \in POL^A_{\langle w,s_k\rangle}$

then $\bar{F}_\gamma(p_1,\dots,p_k) \in POL^A_{\langle w,s_{k+1}\rangle}$.

The set POL^A of polynomial symbols over A is defined by $POL^A = \{POL^A_{\langle w,s\rangle} \mid w \in S^n, s \in S\}$. The symbols $\bar{c}_s$ are called the parameters of the polynomial symbols.

4.2. END

Definition 4.2 differs from the standard definition by clause II. The clause is required here since our definition of an algebra does not allow for nullary operators (they are used in the same way, to denote a specific element of the algebra). In the sequel I will omit the bar when no confusion

is likely; so I will write c and F_γ instead of $\bar{c}$ and $\bar{F}_\gamma$. Furthermore the superscript A will often be omitted.

A measure for the complexity of a polynomial symbol is its height. (Other names for the same notion are complexity or depth).

4.3. DEFINITION. The *height* h of a polynomial symbol p is defined by the following clauses

I. $h(x) = 0$ if x is a variable

II. $h(c) = 0$ if c is a constant

III. $h(F_\gamma(p_1,\ldots,p_k)) = 1 + \max(h(p_1),\ldots,h(p_k))$.

4.3. END

A polynomial symbol $p \in POL^A_{<w,s>}$ determines uniquely a (polynomial) operator p_A of type <w,s> in the following way.

4.4. DEFINITION. Suppose $w = <s_1,\ldots,s_k>$ and $a_1 \in A_{s_1}\ldots a_{s_k} \in A_{s_k}$. Then $p_A(a_1,\ldots,a_k)$ is defined by

I. if $p = x_{j,s}$ then $p_A(a_1,\ldots,a_k) = a_j$.

II. If $p = c_s$ then $p_A(a_1,\ldots,a_k) = c_{s,A}$

III. if $p = \bar{F}_\gamma(p_1,\ldots,p_m)$
then $p_A(a_1,\ldots,a_k) = \bar{F}_{\gamma,A}(p_{1,A}(a_1,\ldots,a_k)\ldots,p_{m,A}(a_1,\ldots,a_k))$

4.4. END

The interpretation of a polynomial does not depend on arguments of which the corresponding variable does not occur in the polynomial.

4.5. THEOREM. *If* $x_{i,s_i} \in X^w$ *does not occur in* $p_{<w,s>}$ *then for all* a_{s_i} *and* b_{s_i} *from* A_{s_i} *we have* $p(a_{s_1},\ldots,a_{s_i},\ldots,a_{s_n}) = p(a_{s_1},\ldots,b_{s_i},\ldots,a_{s_n})$.

PROOF. By induction on the height of p.

4.5. END

The following theorem says that the polynomially definable operations give rise to a new algebra over the elements of the old algebra. The operators of the new algebra are the (interpretations of) the polynomial symbols.

4.6. THEOREM. *Let* $A = <(A_s)_{s\in S},(F_\gamma)_{\gamma\in\Gamma}>$ *be an algebra and* $(G_\delta)_{\delta\in\Delta}$ *the collection polynomial symbols over* A. *Then* $B = <(A_s)_{s\in S},(G_{\delta,A})_{\delta\in\Delta}>$ *is an algebra.*

PROOF. Each G_δ defines a function, and $(A_s)_{s\in S}$ is closed under such functions since

I. The polynomials of the form $x_{i,s}$ yield one of the arguments as result.

II. The polynomials of the form c_s yield an element of A_s as result.

III. The collection $(A_s)_{s\in S}$ is closed under the operations F_γ.

4.6. END

Note that for each operator F_γ from A there is a corresponding polynomial symbol G in B such that $F_\gamma = G_A$. Let $\tau(\gamma) = \langle\langle w_1,\ldots,w_n\rangle,w_{n+1}\rangle$. Then the polynomial symbol corresponding to F_γ is $\bar{F}_\gamma(x_{1,w_1},x_{2,w_2},\ldots,x_{n,w_n})$.

4.7. EXAMPLE *Formulas from propositional logic*

In this example several algebras are presented, of which the last one is an algebra defining formulas of propositional logic.

Let $V = \{p,q,r\} \cup \{\neg,\vee,\wedge,\rightarrow,(,)\}$.

Consider

$A = \langle[V_t],C\rangle$, where C is the two-place concatenation operator; so $C(p,\rightarrow) = p\rightarrow$. Let α,β,γ be $x_{1,s},x_{2,s},x_{3,s}$ respectively. Then we define

$$A' = \langle[V_t],\{C_2,C_3\}\rangle$$

where C_2 and C_3 are the 2-place and 3-place concatenation operators:

$$C_2 = C(\alpha,\beta)_{\langle\langle t,t\rangle,t\rangle} \text{ and } C_3 = C(C(\alpha,\beta),\gamma)_{\langle\langle t,t,t\rangle,t\rangle}.$$

Out of this algebra we define a new one:

$$B = \langle[\{p,q,r\}_t],\{R_\wedge,R_\vee,R_\rightarrow,R_\neg\}\rangle$$

where the R's are polynomial symbols over A':

$$R_\neg = C_2(\neg,C_3((,\alpha,))) \qquad \text{so } R_\neg(\alpha) = \neg(\alpha)$$

$$R_\wedge = C_3(C_3((,\alpha,)),\wedge,C_3((,\beta,))) \qquad \text{so } R_\wedge(\alpha,\beta) = (\alpha)\wedge(\beta)$$

and analogously for $R_\vee$ and $R_\rightarrow$.

The expressions of B are the formulas from propositional logic with proposition letters p,q and r. One observes that B is step by step defined out

of a very simple algebra: the algebra of all strings over the vocabulary with concatenation as operator. Only on the final level does the algebra provide interesting information concerning the structures of logical expressions. On the final level we may define the meanings of the formulas. Usually one will present only the final algebra, the step by step construction out of the basic algebra is omitted, and the concatenation operators C_2 and C_3 are written as concatenations. An algebra like B will in the sequel be defined as follows:

$$B = \langle [\{p,q,r\}_t], \{\neg(\alpha), (\alpha)\wedge(\beta), (\alpha)\vee(\beta), (\alpha)\rightarrow(\beta)\}\rangle.$$

4.8. EXAMPLE: *Non-polynomially defined operators*

In example 2.9 we have met an operator which is not a polynomial one. I repeat the relevant aspects of that example. The algebra considered there is one of strings of digits:

$$N = \langle [\{0,1,\ldots,9\}_{Num}], \{F\}\rangle$$

where $F: Num \times Num \rightarrow Num$ is defined by $F(\alpha,\beta) = \begin{cases}\beta & \text{if } \alpha \equiv 0\\ \alpha\beta & \text{otherwise.}\end{cases}$

The operator F is not defined using some polynomial symbol, i.e. it is not a polynomial operator. By using the if-then-else construction, well known from programming languages, we obtain something of the format of a polynomial:

$$F = \textit{if } \alpha = 0 \textit{ then } \beta \textit{ else } \alpha\beta.$$

This is a convenient way to write the definition in one line, and I will use that notation in the sequel. One might be tempted to think that it becomes a polynomial if one rewrites it in the function argument notation:

$$F = \textit{if-then-else } (\alpha=0, \beta, \alpha\beta).$$

This is, however, not the case. The *if-then-else* operator requires as first

argument a truthvalue. So an algebra over which *if-then-else* can be an operator, has to contain the sort of truth values, and operations yielding truth values (e.g. the two-place predicate =). Since the algebra N of number denotations does not contain these, F cannot be a polynomial operator over N. Nevertheless, F is a fully legitimately defined operator in N.

Some other examples of non-polynomial operators are

G_1: take the reversed sequence of symbols, so $G_1(792) = 297$

G_2: take the digits in even position and concatenate them, so $G(2345) = 35$

G_3: substitute 7 for each occurrence of 3, so $G_3(3723) = 7727$.

4.8. END

5. TERM ALGEBRAS

In this section the notion 'term algebra' will be introduced. The carriers of a term algebra consist of polynomial symbols which can be considered as representations of the productions of a generated algebra. Term algebras play an important role in the formalization of the compositionality principle. In chapter 1, section 3, it was explained that the meaning of an expression depends on its derivational history. A term algebra represents derivational histories, therefore the meanings of the elements of $A = \langle[(B_s)_{s\in S}],(F_\gamma)_{\gamma\in\Gamma}\rangle$ will be defined on the elements of the corresponding term algebra. Another important aspect of the notion term algebra is that it allows us to describe generated algebras in a way that is more constructive than the description given in section 2 (there they are defined by means of the intersection of a - possibly infinite - number of algebras). The new description will be used to obtain an algorithm generating the elements of an algebra, thus justifying the name 'generated algebra'.

Two arguments are mentioned above for considering generated algebras: semantic interpretation and syntactic production. This means that, in this context, we do not deal with algebras as such, but with algebras with a specified set of generators. Therefore we introduce the notion of a Σ,X-algebra, being a Σ-algebra with as collection of generators the sorted collection X. The term algebra $T_{\Sigma,X}$ consists of polynomial symbols which contain no variables and which have only parameters that correspond with elements in X.

5.1. DEFINITIONS. A Σ,X-*algebra* A is a Σ-algebra such that $A = \langle[X],\underline{F}\rangle$. Let us assume that $X = (X_s)_{s\in S}$ and $\underline{F} = (F_\gamma)_{\gamma\in\Gamma}$. Then we define the *term*

algebra $T_{\Sigma,X}$ as the algebra:

$$\langle (T_{\Sigma,X,A,s})_{s\in S}, (F^T_\gamma)_{\gamma\in\Gamma}\rangle$$

where

I. $T_{\Sigma,X,A,s} = \{p \in POL^A_{<w,s>} \mid$ p contains no variables and for all constants $\bar{c}$ in p holds $c \in X\}$

and

II. If $\tau(\gamma) = \langle\langle s_1,\ldots,s_n\rangle,s_{n+1}\rangle$ and $t_1 \in T_{\Sigma,X,A,s_1},\ldots,t_n \in T_{\Sigma,X,A,s_n}$ then $F^T_\gamma(t_1,\ldots,t_n) = \bar{F}_\gamma(t_1,\ldots,t_n)$.

5.1. END

In the sequel we will often simplify the notation for the term algebra. We will attach to T a subscript which identifies the intended term algebra sufficiently. For instance, if in the context the algebra A is given with a specified collection of generators, we may write T_A.

5.2. EXAMPLE. Consider the algebra from example 2.11:

$$M = \langle [\{0,1,\ldots,9\}_{dig}], \{F_1,F_2\}\rangle.$$

Then examples of elements in the term algebra T_M are $0,1$, $F_1(0)$, $F_2(F_1(0),1)$, $F_2(F_2(F_1(1),2),3))$.

5.2. END

The above example shows that each element of T_M represents a way of producing an element of M from the generators by means of successive application of the operators. The following theorem says that all elements of an algebra can be obtained from expressions in the corresponding term algebra, and that only elements of the algebra are obtained in this way (for the definition of t_A, see def.4.4).

5.3. THEOREM. *Let* $A = \langle [(B_s)_{s\in S}], (F_\gamma)_{\gamma\in\Gamma}\rangle$ *be an algebra. Then* $a \in A_s$ *iff there is some* $t \in T_A$ *such that* $t_A = a$.

PROOF. Let $K_s = \{a \in A_s \mid$ there is some $t \in T_A$ such that $t_A = a\}$.

Since $\{b_{i,s} \mid b_{i,s,A} \in B_s\} \subset T_{A,s}$, we have $B_s \subset K_s$. Hence $(K_s)_{s\in S}$ contains all generators of A.

Next we prove that $K = \langle (K_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ is a subalgebra of A. It suffices to show that $(K_s)_{s\in S}$ is closed under $(F_\gamma)_{\gamma\in\Gamma}$. Let $\tau(\gamma) = \langle\langle s_1,\ldots,s_n\rangle, s_{n+1}\rangle$ and let $a_1 \in K_{s_1},\ldots,a_n \in K_{s_n}$. By definition of K_s we know that there are $t_1 \in T_{A,s_1},\ldots,t_n \in T_{A,s_n}$ such that $t_{1,A} = a_1,\ldots,t_{n,A} = a_n$. Define t_{n+1} as $\bar{F}_\gamma(t_1,\ldots,t_n)$. Then $t_{n+1,A} = \bar{F}_{\gamma,A}(a_1,\ldots,a_n)$, so $t_{n+1} \in K_{s_{n+1}}$. Hence K is a subalgebra of A.

Since $\langle [(B_s)_{s\in S}], (F_\gamma)_{\gamma\in\Gamma}\rangle$ is the smallest algebra containing B_s, it follows that K = A, and in particular $K_s = A_s$.

5.3. END

This theorem gives the justification for the algorithm used in the next theorem.

5.4. THEOREM. *Let* A *be an enumerable algebraic grammar. Then there is an algorithm that produces for each sort* s *of* A *the elements of* A_s.

PROOF. Since the grammar is enumerable, there is an algorithm that produces the operators of A, and an algorithm that produces the generators of A. Let Alg_{op} and Alg_{gen} be two such algorithms. The algorithm generating the sets A_s uses these two algorithms.

The algorithm that produces for each sort s of A the elements of A_s can be considered as consisting of a sequence of stages, numbered 1,2,... . Stage N is described as follows.

Perform the first N steps of the algorithm Alg_{op}, thus obtaining a sorted collection of operators, called F_N. Perform the first N steps of the algorithm Alg_{gen}, thus obtaining a sorted collection of generators, called $B_N^{(0)}$. Since we performed a finite number of steps of Alg_{op} and Alg_{gen}, there are finitely many elements in F_N and $B_N^{(0)}$. So for an $f \in F_N$ there are finitely many possible arguments in $B_N^{(0)}$. Perform all these applications of operators in F_N to arguments in $B_N^{(0)}$. In this way finitely many elements are produced. By addition of these elements to $B_N^{(0)}$ we obtain $B_N^{(1)}$. Next we apply each $f \in F_N$ to all possible arguments in $B_N^{(1)}$, add the new elements to $B_N^{(1)}$, etc. This process is repeated until we have obtained $B_N^{(N)}$. This completes the description of stage N, next stage N+1 has to be performed. Notice that N is used three times as a bound: for Alg_{op}, for Alg_{gen} and for the height of the produced polynomials.

The algorithm is rather inefficient: in stage N+1 all elements are produced again which were already produced in stage N. A more efficient

algorithm might be designed as a variant of the above algorithm. Our aim, however, was to prove the existence of a generating algorithm, and not to design an efficient one. From the description of the algorithm it should be clear that only elements of the algebra are produced.

It remains to be proven that the algorithm produces all elements of A. Theorem 5.3 says that for each $a \in A$ there is a term t in the corresponding term algebra such that $t_A = a$. For each term t there is a stage in the above algorithm in which t_A is produced for the first time. This appears from considering the following two cases.

I. t is a generator:
Since Alg_{gen} produces all generators of A, there is a number N_t such that after N_t steps t_A is produced.

II. $t = F_\gamma(t_1,\ldots,t_n)$.
Assume that $t_{1,A},\ldots,t_{n,A}$ are produced for the first time in stages $N_{t_1},\ldots,N_{t_n}$ respectively, and that F_γ is produced in stage N_{F_γ}. Then t_A is produced in stage $\max(N_{F_\gamma},N_{t_1},\ldots,N_{t_n})+1$.

5.4. END

Theorem 5.4 says that an enumerable grammar produces a recursively enumerable language. In theorem 3.7 it is proven that every recursively enumerable language over a finite alphabet can be produced by a finite algebraic grammar. So every enumerable algebraic grammar (and every algebraic grammar) over a finite alphabet can be 'replaced' by a finite algebraic grammar (this observation is due to Johan van Benthem). As has been said, our choice of a grammar depends also on semantic considerations, and these might lead us to the use of an enumerable grammar instead of a finite one.

The proof of theorem 3.7 contains a non-constructive step. This cannot be avoided, as follows from the next theorem.

5.5. THEOREM. *Let* A *be a finite* Σ,X*-grammar. Then for each sort* s *of* A *it is decidable whether* $A_s = \emptyset$.

PROOF. The algorithm proceeds as follows:
stage 1:
For all sorts s check whether there is a generator of sort s, i.e. whether $X_s = \emptyset$. If there are no generators at all, then all carriers are empty, and the algorithm halts here. If generators are found, then it follows that the corresponding sorts have non-empty carriers.

stage 2N:

For all operators F_γ we check whether they give us about new sorts the information that they are non-empty. Assume $\tau(\gamma) = \langle\langle s_1,\ldots,s_n\rangle,s_{n+1}\rangle$. If it was shown in a previous stage that $A_{s_1},\ldots,A_{s_n}$ are non-empty, then it follows that $A_{s_{n+1}}$ is non-empty as well.

stage 2N+1:

If with the results of the previous stage all carriers are shown to be non-empty, then the algorithm halts here. If in the previous stage no new sort was found with a non-empty carrier, then it follows that all remaining carriers are empty, and the algorithm halts here. If in the previous stage some new sort was found with a non-empty carrier, then go to stage 2N+2 (which is described above).

5.5. END

In theorem 3.7 it is stated that every recursively enumerable language over some finite alphabet can be produced by means of a finite algebraic grammar. The proof was based upon the construction of a finite algebraic grammar simulating a type-0 grammar. That construction was not effective: the construction depends on the question whether the type-0 grammar produces an empty language or not, which question is undecidable (HOPCROFT & ULLMAN 1979, p.281). Every constructive version of theorem 3.7 would reduce emptyness of type-0 languages to emptyness of algebraic grammars. Since the emptyness of algebraic grammars is decidable (th.5.5), such a reduction is impossible.

In chapter 1 an interpretation of Frege's principle was mentioned which I described as the 'most intuitive' interpretation. It says that the parts of a compound expression have to be visible parts of the compound expression and that a syntactic rule concatenates these parts. The following theorem concerns such grammars. It is shown that we get such grammars as a special case of our framework.

5.6. THEOREM. *Let* A *be a finite algebraic grammar with generating set* B. *Suppose that all operations* A *are of the form*

$$F_\gamma(a_1,\ldots,a_n) = w_{0,\gamma}a_1w_{1,\gamma}a_2w_{2,\gamma}\ldots a_nw_{n,\gamma}$$

where $w_{i,\gamma}$ *is some possibly empty string of symbols.*

Then L(A) *is a context free language.*

PROOF. We define a context free grammar G as follows:

The set V_N of non-terminal symbols of G consists of symbols corresponding with sorts of A:

$$V_N = \{\bar{s} \mid s \text{ is a sort of A}\}.$$

The start symbol of the grammar G is the symbol corresponding with the distinguished sort s (i.e. the sort such that $L(A) = A_s$).

The set V_T of terminal symbols of G consists of symbols corresponding with the generators of A:

$$V_T = B.$$

The collection of rules of G consists of two subcollections

$$R = \{\bar{s} \to b \mid b \in B_s\} \cup \{\bar{s}_{n+1} \to w_{0,\gamma}\bar{s}_1 w_{1,\gamma}\bar{s}_2 w_{2,\gamma}\ldots\bar{s}_n w_{n,\gamma} \mid \tau(\gamma) = \langle\langle s_1, s_2, \ldots, s_n\rangle, s_{n+1}\rangle\}.$$

It should be clear that L(G) = L(A). This means that L(A) is context free. 5.6. END

The above theorem could easily be generalized to the case that the arguments of an operation are not concatenated in the given order, but are permuted first. Theorem 5.6 shows that the most intuitive interpretation of Frege's principle (all parts have to be visible parts) is a special case of our framework. It shows moreover that with this interpretation one either has to accept an infinite number of operators, or to conclude that the principle can only be applied to context free languages. A restriction to context free languages is not attractive because that would exclude a large class of interesting languages (e.g. ALGOL 68 and predicate logic), furthermore it has been claimed that natural languages are not context free (for a discussion see PULLUM & GAZDAR 1982). An attempt to use only context free rules for the treatment of natural language, but an infinite number of them, is made by GAZDAR (1982).

In our approach the generation of a context-free language is only a special case of the framework. The group Adj, which works in a similar framework, seems to have another opinion about context-freedom. They give

no explicit definition of the notion 'algebraic grammar' nor of its 'generated language', but the definition they implicitly use, seems similar to ours. They suggest, however, that by using algebraic grammars, one can obtain only context-free languages. Evidence for this is that they construct an algebraic grammar for a context-free language and next state that that is 'the most important and general example' (ADJ 1977, p.75). Another statement suggesting this arises when they discuss SCOTT & STRACHEY 1971. Those authors say (p.29):

> *Our language .. is no longer context free. But if we may say so, who cares? .. The last thing we want to be dogmatic about is language.*

As a reaction to this, they say (ADJ 1977, p.76)):

> *'But their semantics does depend on the context free character of the source language, because the meaning of a phrase is a function of the meanings of its constituent phrases'.*

So again they take for granted that an algebraic grammar generates a context-free language. The difference of opinion in these matters might be explained by the fact that they consider only a very special relation between the syntactic algebra and the corresponding term algebra (but see also the discussion in section 9).

6. HOMOMORPHISMS

A homomorphism from algebra A to B is a mapping from the carriers of A to the carriers of B such that the structure of A and B is respected. This is only possible if A and B have about the same structure, although it is not needed that A and B are identical or isomorph. For instance, it is not needed that the two algebras have the same sorts, but there has to be a one-one correspondence of the sorts. It is not necessary that the operators perform the same action, but there has to be a one-one correspondence between the operators such that if an operator in A is defined for certain sorts, then the corresponding operator in B is defined for the corresponding sorts in B. These considerations are expressed formally in the following definitions (they are due to J. Zucker, pers. comm.).

6.1. DEFINITION. Let A be an algebra with signature $\Sigma_A = (S_A, \Gamma_A, \tau_A)$, and B an algebra with signature $\Sigma_B = (S_B, \Gamma_B, \tau_B)$. Let $\sigma: S_A \to S_B$ and $\rho: \Gamma_A \to \Gamma_B$ be bijections (i.e. mappings which are one-one and onto). Then two algebras A and B are called (σ,ρ)-*similar* if the following holds:

$$\tau_A(\gamma) = \langle\langle s_1,\ldots,s_n\rangle,s_{n+1}\rangle \quad \text{if and only if}$$

$$\tau_B(\rho(\gamma)) = \langle\langle\sigma(s_1),\ldots,\sigma(s_n)\rangle,\sigma(s_{n+1})\rangle.$$

If σ and ρ are fixed in a certain context, we will omit them and say that the algebras A and B are *similar*.

6.2. DEFINITIONS. Let A and B be (σ,ρ)-similar algebras. By a (σ,ρ)-*homomorphism* h from A to B we understand a mapping $h: \bigcup_{s\in S_A}(A_s) \to \bigcup_{s\in S_B}(B_s)$ from the carriers of A to the carriers of B such that

1) $h(A_s) \subset B_{\sigma(s)}$.

2) If $\tau_A(\gamma) = \langle\langle s_1,\ldots,s_n\rangle,s_{n+1}\rangle$ and $a_1 \in A_{s_1},\ldots,a_n \in A_{s_n}$ then $h(F_\gamma(a_1,\ldots,a_n)) = F_{\rho(\gamma)}(h(a_1),\ldots,h(a_n))$.

The collection of (σ,ρ)-homomorphisms from A to B, where A and B are (σ,ρ)-similar algebras, is denoted $\mathrm{Hom}(A,B,\sigma,\rho)$. When σ and ρ are clear from the context, or are arbitrary (but fixed), then we will simply speak of a homomorphism h; the collection is then denoted by $\mathrm{Hom}(A,B)$.

In case h is surjective, it is called a *homomorphism onto*, or an *epimorphism*. The collection of epimorphisms is denoted $\mathrm{Epi}(a,B,\sigma,\rho)$, or simplified $\mathrm{Epi}(A,B)$. In case h is bijective (one-one and onto), it is called an *isomorphism* (note that in category theory this term is used with a different meaning).

6.2. END

The definition of 'homomorphism' given in 6.2 differs from the one given by Adj (see e.g. ADJ 1977). One difference is that our definition can be used in more circumstances: we do not require, for instance, that the collections of operator indices and sorts are identical. I prefer, in this respect, our definition for practical reasons. Sometimes algebras have 'natural' sorts, e.g. an algebra generating a language may have a carrier of the sort sentence, whereas a semantical algebra may have a sort of truth-values, or of propositions. Then one might wish to define a homomorphism between these two algebras, although the sorts are not identical. Our definition allows to do so directly, whereas according Adj's definition renaming of the sorts has to be done first. This difference in the definitions is, in theoretical respect, not important, and does not give rise to interesting theoretical consequences. In the following theoretical investigations

I will assume, for the ease of discussion, that similar algebras do have the same sorts and operator indices; then σ and ρ are assumed to be the identity mapping. A more fundamental difference of the definitions is that Adj defines a homomorphism as a sorted collection of mappings $(h_s)_{s\in S}$, where $h_s: A_s \to B_s$, and where these operations respect, in a certain sense, the structure of the algebras involved. Since, according to our definition of a many sorted algebra, the carriers need not be disjoint, it would under Adj's definition of homomorphism be possible for an element occurring in two carriers to have two different images under h. In section 9 it will be explained why Adj's definition is not suitable for us in this respect.

A homomorphism respects structure. Therefore it is not surprising that the homomorphic image of an algebra is a (similar) algebra. This is expressed in the following theorem.

6.3. THEOREM. *Let* $A = \langle\langle A_s\rangle_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ *and* $B = \langle(B_s)_{s\in S}, (G_\gamma)_{\gamma\in\Gamma}\rangle$ *be similar algebras, and* $h \in \mathrm{Hom}(A,B)$. *Then* $\langle(h(A_s))_{s\in S}, (G_\gamma)_{\gamma\in\Gamma}\rangle$ *is a subalgebra of* $\langle(B_s)_{s\in S}(G_\gamma)_{\gamma\in\Gamma}\rangle$.

PROOF. We prove the theorem by proving that the sets $h(A_s)_{s\in S}$ are closed under G_γ. Let $\tau(\gamma) = \langle\langle s_1, s_2, \ldots, s_n\rangle, s_{n+1}\rangle$ and let $b_i \in h(A_{s_i})$. This means that there are $a_i \in A_{s_i}$ such that $b_i = h(a_i)$. Consequently

$$G_\gamma(b_1,\ldots,b_n) = G_\gamma(h(a_1),\ldots,h(a_n)) = h(F_\gamma(a_1,\ldots,a_n)).$$

It is clear from the last expression that it denotes an element of $h(A_{s_{n+1}})$, so of $B_{s_{n+1}}$.
6.3. END

In chapter 1 we discussed the way in which the set E of expressions of the language should be related to the set D of semantic objects. We concluded that, in order to obey the compositionality principle, the syntax has to be a many sorted algebra and that the meaning of an expression has to be obtained in the following way. For each syntactic operator F_γ, there is an operator G_γ on D, where G_γ is defined for the images of the arguments of F_γ. For the mapping M which yields the corresponding meaning it is required that:

$$M(F_\gamma(e_1,\ldots,e_k) = G_\gamma(M(e_1),\ldots,M(e_k)).$$

We concluded that these requirements have the consequence that D gets the same structure as the syntactic algebra. More formally this is stated in the following theorem.

6.4. THEOREM. *Let* $E = \langle (E_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ *be an algebra,* D *a set and* M *a mapping from* E *to* D. *Let* $(G_\gamma)_{\gamma\in\Gamma}$ *be operators defined on the subsets* $M(E_s)$ *of* D.
Suppose

$$M(F_\gamma(e_1,\ldots,e_k)) = G_\gamma(M(e_1),\ldots,M(e_k))$$

for all $\gamma \in \Gamma$ *and for all arguments* $e_1,\ldots,e_k$ *for which* F_γ *is defined. Then* $D' = \langle (M(E_s))_{s\in S}, (G_\gamma)_{\gamma\in\Gamma}\rangle$ *is an algebra similar to* E.

PROOF.

I. $(D'_s)_{s\in S}$ is a collection of sets closed under the operations G_γ since $G_\gamma(m_1,\ldots,m_k) = G_\gamma(M(e_1),\ldots,M(e_k)) = M(F_\gamma(e_1,\ldots,e_k)) \in D'_{s_{k+1}}$.

II. D is similar to E since the sorts are the same, the operator indices are the same and
if $F_\gamma : E_{s_1} \times E_{s_2} \times \ldots E_{s_n} \to E_{s_{n+1}}$
then $G : D'_{s_1} \times D'_{s_2} \times \ldots D'_{s_n} \to D'_{s_{n+1}}$.

III. M is a mapping onto $U_s D'_s$ satisfying the conditions for homomorphisms.

6.4. END

Having introduced the notion 'homomorphism', we may formalize the compositionality principle as follows: the syntax is a many sorted algebra A, the semantic domain is a similar algebra M, and the meaning assignment is a homomorphism from the term algebra T_A to M.

A first consequence of this formalization is the following theorem concerning the replacement of expressions with the same meaning.

6.5. THEOREM. *Let* $e,e' \in A_s$, *with* $M(e) = M(e')$. *Suppose* $F_\gamma(\ldots,e,\ldots)$ *to be defined. Then* $M(F_\gamma(..,e,..)) = M(F_\gamma(..,e',..))$.

PROOF. $M(F_\gamma(..,e,..)) = G_\gamma(..,M(e),..) = G_\gamma(..,M(e'),..) = M(F_\gamma(..,e',..))$. The equalities hold since M is a homomorphism and F_γ is defined for all elements of A_s.

6.5. END

The theorem states that in case two expressions of the same category have the same meaning, they can be interchanged in all contexts without changing the resulting meaning. The reverse is not true, interchangeable in all contexts without changing the meaning does not imply that the meanings are identical, since the language might be too poor to provide for contexts where the difference becomes visible.

The above theorem is related to the well known principle of Leibniz concerning substitutions (GERHARDT, 1890, p.228).

Eadem sunt, quorum unum potest substitui alteri, salva veritate.

This principle is sloppy formulated: it confuses the thing itself with the name refering to it (CHURCH 1956, p.300, QUINE 1960, p.116). It should be read as saying that two expressions refer to the same object if and only if in all contexts the expressions can be interchanged without changing the truthvalue. Let us generalize the principle to all expressions, instead of only referring ones, thus reading 'Eadem sunt' as 'have the same meaning'. Then the above theorem gives us a formalisation of one direction of Leibniz' principle. The other direction can then be considered as a restriction on the selection of a semantical domain. The semantical domain may only give rise to differences in meaning that are expressible in the language.

An important, although very elementary, property concerning homomorphisms is that the compositions of two homomorphisms h and g is a homomorphism again. As defined in chapter 1, the composition which consists in first applying h and next g is denoted $h \circ g$. This has as a consequence that $(h \circ g)(x) = g(h(x))$, note that the order is reversed here. For this reason one sometimes defines $h \circ g$ as first applying g and next h. Adj follows the standard definition, but in order to avoid the change of the order, they have, in some of their papers the convention to write the argument in front of the operator (so $(x)(h \circ g) = ((x)h)g$!). I will use the standard definition ($h \circ g$ means first applying h). The announced theorem concerning composition of homomorphisms is as follows.

6.6. THEOREM. *Let* A,B *and* C *be similar algebras and let* $h \in \mathrm{Hom}(A,B)$ *and* $g \in \mathrm{Hom}(B,C)$. *Then* $h \circ g \in \mathrm{Hom}(A,C)$.

PROOF. Let F_γ, G_γ and H_γ denote operators in A,B and C respectively. Then

$$h \circ g(F_\gamma(a_1,\ldots,a_k)) = g(h(G_\gamma(a_1,\ldots,a_k))) = g(G_\gamma(h(a_1),\ldots,h(a_k))) =$$
$$H_\gamma(g(h(a_1)),\ldots,g(h(a_k))) = H_\gamma(h \circ g(a_1),\ldots,h \circ g(a_k)).$$

6.6. END

In general, it is not necessary to define a homomorphism by stating its values for all possible arguments, since (as I will now show) in the same way as a subalgebra is completely determined by its generators, a homomorphism is completely determined by its values on these generators.

6.7. THEOREM. *Let* $h,g \in \mathrm{Hom}(\langle [A],(F_\gamma)_{\gamma\in\Gamma}\rangle, \langle B,(G_\gamma)_{\gamma\in\Gamma}\rangle)$.
Suppose that $h(a) = g(a)$ *holds for all generators* $a \in A$. *Then* $h(e) = g(e)$ *holds for all elements* e *of* $\langle [A],(F_\gamma)_{\gamma\in\Gamma}\rangle$.

PROOF. Let $K = \{a \in [A] \mid h(a) = g(a)\}$. Now K is closed under the operations $(F_\gamma)_{\gamma\in\Gamma}$:

Let $k_1,\ldots,k_n \in K$. Then:
$h(F_\gamma(k_1,\ldots,k_n)) = F_\gamma(h(k_1),\ldots,h(k_n)) =$
$F_\gamma(g(k_1),\ldots,h(k_n) = g(F_\gamma(k_1,\ldots,k_n))$.

So K is a subalgebra with $A \subset K$. Since [A] is the smallest subalgebra with this property, it follows that $K = [A]$.

6.7. END

Suppose that we have defined a mapping from the generators of algebra A to those of algebra B. Then the above theorem says that there is at most one extension of this mapping to a homomorphism. But not in all cases such an extension exists. Suppose that in A two different operators yield for different arguments the same result (i.e. $F_i(a_1,\ldots,a_n) = F_j(a'_1,\ldots,a'_m)$), whereas this is not the case in B for the corresponding operators. Then there is no homomorphism from A to B. But for the algebras we will work with, (viz. termalgebras) this situation cannot arise. In a termalgebra an operator (e.g.S_1) leaves the corresponding symbol as a trace in the resulting expression (e.g. the symbol S_1 in $S_1(E_1,E_2)$). Hence in a termalgebra different operators always yield different results. Therefore we may define a meaning assigning homomorphism by providing 1. meanings for the generators of the syntactic algebra and 2. semantic operators corresponding to the syntactic operators.

6.8. EXAMPLE. Let M be as in example 2.10, so

$M = \langle [\{0,1,2,\ldots,9\}_{dig}], \{F_1,F_2\}\rangle$
where F_1: dig → num $\qquad F_1(\alpha) = \alpha$
and F_2: num × dig → num $\qquad F_2(\alpha,\beta) = \alpha\beta$.

This algebra produces strings of symbols. The meaning of such a string has to be some natural number. Let us denote natural numbers by symbols such as 7, 70 etc. The reader should be aware of the fact that there is (in this example) a great difference between strings such as *1* and *7* for which e.g. concatenation is defined, but not addition or multiplication, and numbers such as 1 and 7 for which addition and multiplication are defined, but not concatenation. Another difference is that *7*, *07*, and *007* are distinct strings all corresponding to the same number 7. The meaning algebra N corresponding to M, consists of numbers and is defined as follows.

$$N = \langle [\{0,1,\ldots,9\}_{dig}], \{G_1, G_2\} \rangle$$
$$\text{where } G_1\colon \text{dig} \to \text{num} \qquad \text{defined by } G_1(\alpha) = \alpha$$
$$\text{and} \quad G_2\colon \text{num} \times \text{dig} \to \text{num} \quad \text{defined by } G_2(\alpha,\beta) = 10 \times \alpha + \beta.$$

The meaning homomorphism h is defined by $h(0) = 0 \ldots h(9) = 9$.
So

$$h(F_1(1)) = G_1(h(1)) = G_1(1) = 1$$

and

$$h(F_2(F_1(0),7)) = G_2(G_1(0),7) = 10 \times 0 + 7 = 7.$$

6.9. EXAMPLE. In example 2.11 we considered an algebra which was the same as the above one, with the difference that the digits are written in front of the numbers:

$$F_3\colon \text{num} \times \text{dig} \to \text{num} \qquad \text{defined by } F_3(\alpha,\beta) = \beta\alpha.$$

In this situation it is impossible to find a semantic operation G_3 corresponding with F_3. For suppose there were such an operation G_3. Then, since e.g. $h(7) = h(007) = 7$, we would have that on the one hand $G_3(7,h(2)) = G_3(h(7),h(2)) = h(F_3(7,2)) = h(27) = 27$, but on the other hand $G_3(7,h(2)) = G_3(h(007),h(2)) = h(F_3(007,2)) = h(2007) = 2007$, which is a contradiction. So whereas in example 6.8 it was rather easy to find a semantic operation, it here is impossible since on the level of semantics there is no difference between the meanings of *7* and *007*.
6.9. END

The last example is a formal illustration of a statement of Montague's concerning the syntax of natural languages (MONTAGUE 1970b, p.223, fn.2):

> *It is to be expected, then, that the aim of syntax can be realized in many different ways, only some of which would provide a suitable basis for semantics.*

Next I will prove a theorem that is important from a theoretical point of view. The theorem implies that the framework allows for assigning any meaning to any language. This means that working in accordance with the framework gives rise to no restriction: neither on the produced languages, nor on the assigned meanings. Notice that there is no conflict between the theorem and the example above. The example shows that not every syntax can be used, whereas the theorem states that there is at least one syntax. The theorem is, however, not useful from a practical point of view, since it is based upon the syntax developed in theorem 3.7: the construction does not reflect the structure of the language. The proof of the theorem just says that if you know what the intended meanings of the expressions of the language are, then this knowledge defines some algebraic operation.

6.10. THEOREM. *Let* L *be a recursively enumerable language over a finite alphabet, and* M *a set of meanings for elements of* L. *Let* $f: L \to M$ *be a function. Then there is a finite algebraic grammar* A *and an algebra* B *such that*

1) $L(A) = L$.

2) A *and* B *are similar.*

3) *There is an* $h \in Epi(A,B)$ *such that* $h(w) = f(w)$ *for all* $w \in L$.

PROOF. For the case $L = \emptyset$ the theorem is trivial. For the case $L \neq \emptyset$ consider the algebraic grammar defined in theorem 3.7.

Let A be the algebraic grammar obtained in this way for L.

Recall that the last operation of this algebra gives for some strings as output the same string, but with \$ deleted. For other strings it gives a special string as output. The semantic algebra will differ from the syntactic one only in this last operation: the function f will be incorporated.

More formally, let A be the syntactic algebra from theorem 3.7. So $A = \langle\langle [S_{in}], (F_\gamma)_{\gamma \in PU\{1,2,3,4,5\}}\rangle, in\rangle$, where $S_A = \{in, mid, out\}$. This algebra is transformed into a semantic algebra B as follows:

$$B = \langle (B_s)_{s \in S}, (G_\gamma)_{\gamma \in PU\{1,2,3,4,5\}}\rangle$$

where $B_{in} = A_{in}$, $B_{mid} = A_{mid}$ and $B_{out} = M$ and

$$G_\gamma = F_\gamma \quad \text{if} \quad \gamma \neq 5,$$

and

$$G_5 = F_5 \circ f.$$

The homomorphism h is defined by

$$h(w) = \begin{cases} w & \text{for } w \in B_{in} \cup B_{mid} \\ f(w) & \text{for } w \in B_{out}. \end{cases}$$

6.10. END

7. A SAFE DERIVER

In chapter 1, section 4, I have sketched the framework in which we will work, and I will repeat here some relevant aspects. The syntax of the language of which we wish to define the semantics is an algebra A, and the function which assigns meanings is an homomorphism defined on T_A. In order to define this homomorphism we use a logical algebra L which is interpreted by homomorphism h in model M. From the algebra L a new algebra L' is defined, using deriver D, where L' is similar with A. The interpretation h for L should determine uniquely an interpretation h' for L'. This situation is represented in figure 1. We will return to this framework in section 8.

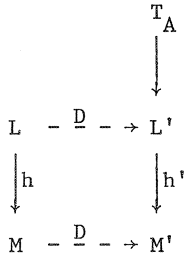


Figure 1. The framework

In this and in the next section I will investigate some methods for building new algebras out of old ones, in such a way that an interpretation homomorphism defined on the old algebra determines a unique inter-

pretation for the new algebra. Such a method will be called safe. We will meet several examples of methods to obtain new algebras from old ones; as neutral name for such methods I will use deriver.

7.1. DEFINITION. A deriver is called *safe* if for algebras A and B and all $h \in Epi(A,B)$ there is a unique algebra B' such that for the restriction h' of h to D(A) it holds that $h' \in Epi(D(A),B')$.
7.1. END

The requirement that h' is an epimorphism is important. If we would not require this, B' would in most cases not be unique. An extreme example arises when D(A) is an empty algebra. Then there are infinitely many algebras B' such that $h' \in Hom(D(A),B')$, but only one such that $h' \in Epi(D(A),B')$.

In this section I will consider the aspect of the introduction of new operators. MONTAGUE (1970b) claims that polynomially defined operators are safe. A proof that for many-sorted algebras polynomial extensions are safe, will be given below.

7.2. DEFINITION. Let $A = \langle A,F\rangle$ be an algebra and G a collection of operator symbols such that for all $g \in G$ there are $s_1,\ldots,s_n,s_{n+1} \in S_A$ such that $g_A : A_{s_1} \times \ldots \times A_{s_n} \to A_{s_{n+1}}$. Then the algebra $\langle A,F\cup G\rangle$ is denoted $Addop_G(\langle A,F\rangle)$.

7.3. THEOREM. *If* P *is a collection of polynomial symbols, then* $Addop_P$ *is safe.*

PROOF. Let $A = \langle (A_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ and $B = \langle (B_s)_{s\in S}, (G_\gamma)_{\gamma\in\Gamma}\rangle$ be similar algebras. Suppose that $h \in Epi(A,B)$ and $P \times POL^A$. We define $h: POL^A \to POL^B$ as follows:

I. Each operator symbol $\bar{F}_\gamma$ is replaced by a symbol $\bar{G}_\gamma$.
II. Each constant $\bar{a}$ is replaced by a constant $\bar{b}$, where $b = h(a)$.

Define $A' = Addop_P(A)$, $B' = Addop_{h(P)}(B)$.

We now prove that h is an epimorphism from A' onto B'. That h is surjective follows from the fact that $h \in Epi(A,B)$. Remains to show that $h \in Hom(A',B')$, so that for all new operators, i.e. for all $p \in P$, holds:

$$h(p_A,(a_1,\ldots,a_n)) = p_B,(h(a_1),\ldots,h(a_n)).$$

This is proved by induction on the complexity of p.

I. $p = x_{j,s}$

$$h(x_{j,s,A'}(a_1,\ldots,a_n)) = h(a_j) = x_{j,s,B'}(h(a_1),\ldots,h(a_n)).$$

II. $p = \bar{a}$

$$h(\bar{a}_A(a_1,\ldots,a_n)) = h(\bar{a}_A) = \bar{b}_B = \bar{b}_B(h(a_1),\ldots,h(a_n)).$$

III. $p = \bar{F}_\gamma(p_1,\ldots,p_m)$

$$h(p_{A'}(a_1,\ldots,a_n)) = h(F_\gamma(p_{1,A'},\ldots,p_{m,A'})(a_1,\ldots,a_n)) =$$
$$= h(F_\gamma(p_{1,A'}(a_1,\ldots,a_n)),\ldots,p_{m,A'}(a_1,\ldots,a_n)) =$$
$$= G_\gamma(h(p_{1,A'}(a_1,\ldots,a_n)),\ldots,h(p_{m,A'}(a_1,\ldots,a_n))) =$$
$$= G_\gamma(p_{1,B'},\ldots,p_{m,B'})(h(a_1),\ldots,h(a_n)) = p_{B'}(h(a_1),\ldots,h(a_n)).$$

Next we prove that B' is unique in the following sense: if $h \in \mathrm{Epi}(A',B')$ and $h \in \mathrm{Epi}(A',D)$, then $D = B'$. This follows from:

1. the carriers of B' and D are equal:

$$B'_s = \{h(a) \mid a \in A'_s\} = D_s$$

2. the operators of B' and D are identical:

Let $b_1 \in B_{s_1},\ldots,b_n \in B_{s_n}$. Then there are $a_1 \in A_{s_1},\ldots,a_n \in A_{s_n}$ such that $h(a_i) = b_i$. Hence $p_{B'}(b_1,\ldots,b_n) = p_{B'}(h(a_1),\ldots,h(a_n)) =$
$= h(p_{A'}(a_1,\ldots,a_n)) = p_D(h(a_1),\ldots,h(a_n)) = p_D(b_1,\ldots,b_n)$.

One observes that uniqueness is a direct consequence of existence.

7.3. END

Now the question arises whether the restriction to polynomially defined operations is necessary. We cannot generalize theorem 7.3 to operators which are defined in an arbitrary way, as is shown by the next example.

7.4. EXAMPLE. Consider the following algebra of strings of digits:

$$N = \langle \{0,1,\ldots,9\}_{dig}, \{0,\ldots,9\}^*_{num}, C\rangle$$
where $C : N_{dig} \times N_{num} \to N_{num}$ is defined by $C(\alpha,\beta) = \alpha\beta$.

With these strings we associate a somewhat unusual meaning: their length (it is not that unusual if one remembers the system I-one, II-two, III-three). So the semantic algebra M corresponding to N consists of natural numbers. Notice that in N we had digits (denoted *0,1*, etc.) with concatenation, but in M we have natural numbers (denoted 0,1 etc.), with the operation of addition. The interpretation homomorphism h from N to M is defined as follows

$$h(0) = h(1) = .. h(9) = 1.$$

The operation corresponding with C is of course addition of the lengths of the strings. So the semantic algebra M is defined as

$$M = \langle (\{1\}_{dig}, \mathbb{N}_{num}), + \rangle.$$

Now we extend N with a new operator D, defined as follows:

$$D: N_{dig} \times N_{num} \rightarrow N_{num}$$

$$\text{where } D(\alpha,\beta) = \begin{cases} \beta & \text{if } \alpha \text{ is the symbol 0} \\ \alpha\beta & \text{otherwise.} \end{cases}$$

This operator is not polynomially defined, and it cannot be defined polynomially because there are no truth values in the algebra N (see example 4.8). Let N' be the algebra obtained from N by adding the operator D. Is there a unique algebra M' such that $h \in Epi(N',M')$?

Suppose that there is such an algebra, called M', with as operator d, corresponding with D. What is then the value of d(1,1)?
On the one hand: $d(1,1) = d(h(3),h(7)) = h(D(3,7)) = h(37) = 2$.
On the other hand: $d(1,1) = d(h(0),h(7)) = h(D(0,7)) = h(7) = 1$.
This is a contradiction. So there is no such algebra M'. The source of this problem is that we make a distinction at the syntactic level which has no influence on the semantic level: the difference between *0* and the other digits.
7.4. END

This example has shown the dangers of using a non-polynomially defined operator. If one introduces an operator which is defined in some arbitrary

way, then there is the danger of disturbing the interpretation homomorphism. In practice the situation often arises that the meaning of some language is defined by translation into a logic. The addition to the logic of an operator which is not polynomially defined, could invoke the danger that there is no longer an associated semantics: a translation is defined, but there is no guarantee of an interpretation for the derived logical algebra (i.e. there is, in figure 1, no h'). In chapter 6 (part 2) we will meet several examples of proposals from the literature which are incorrect since there is not such an interpretation.

The following example shows that in some cases operators which are not polynomially definable nevertheless may respect homomorphic interpretations. So in theorem 7.3 the condition 'polynomially defined' is not a necessary condition.

7.5. EXAMPLE (W. Peremans, pers. comm.).

Consider the algebra of natural numbers with as only operation S, the successor operation ('add one'). So the algebra we consider is:

$$N = \langle \mathbb{N}, S \rangle$$

where $S: \mathbb{N} \to \mathbb{N}$ is defined as 'addition with one'. We extend N with the operator $\oplus$, defined by the equalities $n \oplus 0 = n$ and $n \oplus S(m) = S(n \oplus m)$. This means that $\oplus$ is the usual addition operator. This operator is not polynomially definable over N. One sees this as follows. All polynomial symbols over N are of the form $S(S(\ldots S(x))$. So a polynomial symbol which corresponds with an operator which takes two arguments, contains only one variable, and is therefore dependent on only one of its arguments. Consequently the two place operation of addition cannot be defined polynomially.

In spite of the fact that $\oplus$ is not a polynomially definable operator, a (variant of) theorem 7.3 holds. For every algebra M and every $h \in Epi(N,M)$ there is an unique M' such that $h \in Epi(AddOp_{\oplus}N,M')$. This is proved as follows. Let $S^n(0)$ denote the n-times repeated application of S to 0; so $S^n(0) = S(S(\ldots S(0)..))$. For all $n \in N$ we have $n = S^n(0)$. Since $h \in Epi(N,M)$ this means that for all $m \in M$ there is an n such that $m = T^n(h(0))$, where T is the operator in M which corresponds with S. We define an operator $*$ in M as follows:

Assume: $m_1 = T^{n_1}(h(0))$ and $m_2 = T^{n_2}(h(0))$. Then $m_1 * m_2 = {}_d T^{n_1+n_2}(h(0))$. This definition is independent of the choice of n_1 and n_2 as is shown as

follows:

Suppose that $m_1 = T^{n_1}(h(0)) = T^{n_3}(h(0))$, and that

$m_2 = T^{n_2}(h(0)) = T^{n_4}(h(0))$.

Then

$$T^{n_3+n_4}(h(0)) = T^{n_3}(T^{n_4}(h(0))) = T^{n_3}(T^{n_2}(h(0))) =$$
$$T^{n_3+n_2}(h(0)) = T^{n_2+n_3}(h(0)) = T^{n_2+n_1}(h(0)).$$

This shows that the definition of $*$ is a correct definition.

Now, let M' be $\text{AddOp}_*(M)$. Then $h \in \text{Epi}(N',M')$ since it is a surjective mapping and

$$h(n_1 \oplus n_2) = h(S^{n_1}(0) \oplus S^{n_2}(0)) = h(S^{n_1+n_2}(0)) =$$
$$T^{n_1+n_2}(h(0)) = T^{n_1}(h(0)) * T^{n_2}(h(0)) = h(S^{n_1}(0)) * h(S^{n_2}(0)) =$$
$$= h(n_1) * h(n_2).$$

To prove the unicity of M', we only have to prove the unicity of this definition of $*$. Suppose that $h \in \text{Epi}(\text{Add Op}_{\oplus}(N),M'')$, where the operation corresponding with $\oplus$ in M'' is $\circ$.

$m_1 \circ m_2 = T^{n_1}(h(0)) \circ T^{n_2}(h(0)) = h(S^{n_1}(0)) \circ h(S^{n_2}(0)) = h(S^{n_1}(0) \oplus S^{n_2}(0))$
$= h(S^{n_1+n_2}(0)) = T^{n_1+n_2}(h(0)) = m_1 * m_2$.

7.5. END

The characterization of operators which are safe is still an open question. But for a class of algebras which is relevant for us, such a characterization can be given. In the sequel we will always work with a logic which has as syntax a free algebra with infinitely many generators (all variables and constants are generators). For such algebras all safe operators are polynomially definable. Note that in example 7.5, were there was no polynomial definition for $\oplus$, there is a single generator (viz.0). Results related to the above one are given in Van BENTHEM (1979b); the proof of the above result is given in appendix 1 of this book. The notion of a 'safe deriver' is, related to the notions 'enrichment' and 'derivor' used in the theory of abstract data types, e.g. in ADJ 1978.

8. MONTAGUE GRAMMAR

The notion 'Montague grammar' is often used to indicate a class of grammars which resembles the grammar used in Montague's most influential publication PTQ (MONTAGUE 1973). It is, however, not always made clear what is to be understood by 'resembling' in this context. There are a lot of proposals which deviate from PTQ in important respects. Some proposals have a rather different syntax, other use a different logic or different models. The definition of 'Montague grammar' should make clear, which proposals are covered by this notion and which not.

In my opinion the essentail feature of a Montague grammar consists in its algebraic structure. The most pure (and most simple) definition would be that a Montague grammar consists in an algebraic grammar and a homomorphic interpretation. One always uses, in practice, some formal (logical) language as auxiliary language, and the language of which one wishes to describe the meanings is translated into this formal language. Thus the meaning assignment is performed indirectly. The aspect of translating into an auxiliary language is, in my opinion, unavoidable for practical reasons, and I therefore wish to incorporate this aspect in the definition of a Montague grammar. This decision includes (by suitable interpretation) grammars in which the interpretation is given directly. The most important example of that kind of grammar is the grammar in 'English as a formal language' (MONTAGUE 1970a). For such grammars the name simple Montague grammar seems suitable. These considerations should explain the following definitions.

8.1. DEFINITION. A *simple Montague grammar* consists of
1. an algebraic grammar A
2. an algebra M similar to A
3. a homomorphism $h \in Hom(A,M)$.

8.2. DEFINITION. A *Montague grammar* consists of
1. an algebraic grammar A (the 'syntactic algebra')
2. an algebraic grammar L (the 'logical algebra')
3. an algebra M similar to L (the 'semantic algebra')
4. a homomorphism $h \in Hom(L,M)$ (the 'interpretation of the logic')
5. an algebra D(L), similar to A derived from L, where D is a safe deriver.
6. a homomorphism $h \in Hom(T_A,D(L))$ (the 'translation').

8.2 END

Definition 8.2 is illustrated by figure 2 (cf. figure 1.)

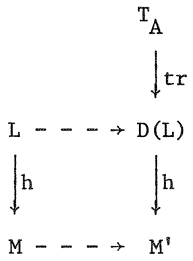


Figure 2. A Montague grammar

The logical language which we will use is just as in PTQ, the language of intensional logic. Its (algebraic) grammar L and its (homomorphic) interpretation will be considered in chapter 3. The grammar A of the PTQ-fragment and its translation D(L) will be presented in chapter 4. The deriver D that will be used can be considered as being built from more elementary ones. I have found it convenient to define *four* more elementary derivers, but other decisions are possible as well. The most important deriver is AddOp which has been discussed in section 7. The other three are introduced below: first an informal discussion; then a formal definition.

The first deriver I will discuss is Add Sorts. An application has the form $\text{AddSorts}_{T,\sigma}(A)$, where T is a collection of sorts, and σ: $T \to S_A$ a function. The effect of this deriver is that a new algebra is formed with as sorts $T \cup S_A$, and with as carrier for $\tau \in T$ the set $A_{\sigma(\tau)}$. So this deriver introduces new sorts without introducing new elements. This deriver will be used when we need to introduce several new sorts which get their elements from one single old sort. An example of this as follows. In a syntax for English, nouns like *man* and verbs like *run* will be in different categories because they have different syntactic properties. But semantically both expressions are considered as predicates of the same type. Therefore we have to build from one old carrier (of predicates) two carriers (of nouns and of verbs). We may remove from each of these two carriers the elements which are not necessary for the translation of English, but in

principle the carriers may still be non-disjoint (e.g. both may contain variables for predicates).

The second deriver I will discuss is DelOp. An application of this deriver has the form $DelOp_{\Delta}(A)$. Here is Δ a subset of the set of operator symbols of A. The effect of this deriver is that a new algebra is formed which differs from A in the respect that it does not have the operators mentioned in Δ. This deriver is needed for the following reason. The derived algebra D(L), see figure 2, should only have operators which correspond with operators in A. Not all operators of the logical algebra L will be operators of D(L). For instance, the introduction of the universal quantifier might not correspond to any of the operations of the grammar A for the natural or programming language under consideration. Therefore we need a deriver which removes operators.

The last deriver is SubAlg. An application of this deriver has the form $SubAlg_{H}(A)$. Here is H a sorted collection of elements of A. Its effect is that an algebra is formed which has the same operators as A, but which has H as a generating set. This deriver is used in the following kind of situation. The logical algebra L (see figure 2) has for each sort infinitely many generators. The grammar A might not have this property. For instance, the sentences of a natural language are all built up from smaller components, and hence there are no generators of the sort 'sentence'. SubAlg is then used to reduce the carriers of L to those elements which will be images of elements in A.

Below these three methods are defined, and their safeness is proven. It is not surprising that these methods are safe. Nevertheless the proofs are not elegant, but rather ad-hoc. This is probably due to the fact that there is hardly any theory about derivers of many-sorted algebras which I could use here. GOGUEN & BURSTALL (1978) present some category-theoretic considerations about derivers for many-sorted algebras in the sense of ADJ 1977. I have already mentioned the work of Van BENTHEM (1979b) concerning the introduction of new operators in one sorted algebras. That there is a need for a general theory appears, apart from the present context, in work in the field of abstract data types in the theory of programming languages, see e.g. EHRIG, KREOWSKI & PADAWITZ 1978 and ADJ 1978.

8.3. DEFINITION. Let $\sigma: T \to S$ arbitrary. Then $\sigma' = U_n (T \cup S)^n \times (T \cup S) \to U_n S^n \times S$ is defined by

1. $\sigma'(s) = s$ for $s \in S$

2. $\sigma'(t) = \sigma(t)$ for $t \in T$

3. $\sigma'(\langle\langle t_1,\ldots,t_n\rangle t_{n+1}\rangle) = \langle\langle\sigma'(t_1),\ldots,\sigma'(t_n)\rangle,\sigma'(t_{n+1})\rangle$.

In the sequel we will write σ for σ'.

8.4. THEOREM. *Let* $A = \langle(A_s)_{s\in S},(F_\gamma)_{\gamma\in\Gamma}\rangle$ *be a* Σ*-algebra. Let* T *be some set* $(S\cap T=\emptyset)$ *and* σ: $T \to S$ *some mapping. Then there is an algebra* $A' = \langle(A'_t)_{t\in T\cup S},G\rangle$ *where*

I. $A'_t = A_{\sigma'(t)}$

II. *The set of operators* G *is defined as follows: for all* $\gamma \in \Gamma$ *and all* $\langle w,u\rangle \in (T\cup S)^n\times(T\cup S)$ *with* $\sigma(\langle w,u\rangle) = \tau(\gamma)$ *we add a new operator* $g_{\gamma,\langle w,u\rangle}$ *of type* $\langle w,u\rangle$, *and define the effect of* $g_{\gamma,\langle w,u\rangle}$ *to be equal to* F_γ.

(*So the operator indices are the compound symbols* $\gamma,\langle w,u\rangle$).

PROOF. The elements of A and A' are equal. Since A is closed under F_γ, we have that A' is closed under $g_{\gamma,\langle w,u\rangle}$.

8.5. DEFINITION. The algebra introduced in 8.4 is denoted Add $\text{Sorts}_{\sigma,T}(A)$.

8.6. THEOREM. AddSorts *is safe.*

PROOF. Let $A = \langle(A_s)_{s\in S},(F_\gamma)_{\gamma\in\Gamma}\rangle$ and $B = \langle(B_s)_{s\in S},(G_\gamma)_{\gamma\in\Gamma}\rangle$ be similar algebras, and let $h \in \text{Epi}(A,B)$. Suppose σ: $T \to S$. Define $A' = \text{Add Sorts}_{\sigma,T}(A)$ and $B' = \text{Add Sorts}_{\sigma,T}(B)$. We now prove that $h \in \text{Epi}(A',B')$, and that B' is (up to isomorphism) the unique algebra with this property.

Since the elements in B' are the same as in B, the mapping h is surjective. Remains to show that h is a homomorphism. In analogy of theorem 8.4 we denote the operators introduced in $\text{AddSorts}_{\sigma,T}(B)$ by $g_{\gamma,\langle w,u\rangle,B}$, and those introduced in $\text{AddSorts}_{\sigma,T}(A)$ by $g_{\gamma,\langle w,u\rangle,A}$. Then

$$h(g_{\gamma,\langle w,u\rangle,A}(a_1,\ldots,a_n)) = h(F_\gamma(a_1,\ldots,a_n)) =$$

$$= G_\gamma(h(a_1),\ldots,h(a_n)) = g_{\gamma,\langle w,u\rangle,B}(h(a_1),\ldots,h(a_n)).$$

That B' is unique can be proved in the same way as we did in the proof of 7.3 (if there was another algebra, it should have the same carriers and operations).

8.7. THEOREM. *Let* $A = \langle (A_s)_{s\in S}, \underline{F}\rangle$ *be an algebra and let* $\Delta \subset \underline{F}$. *Then* $A = \langle (A_s)_{s\in S}, \underline{F}\backslash\Delta\rangle$ *is an algebra.*

PROOF. A is closed under all F_γ from $\underline{F}\backslash\Delta$.

8.8. DEFINITION. The algebra introduced in 8.7 is denoted

$$\mathrm{DelOp}_\Delta(A).$$

8.9. THEOREM. DelOp *is safe.*

PROOF. Let A and B be similar algebras and $h \in \mathrm{Epi}(A,B)$.
Define $A' = \mathrm{DelOp}_\Delta(A)$ and $B' = \mathrm{DelOp}_\Delta(B)$. We now prove that $h \in \mathrm{Epi}(A',B')$ and that B' is the unique algebra with this property.
Algebras A' and B' are similar since if F_γ is an operation of A' then G_γ is an operation of B'. That h is surjective on B' is evident. Since h respects all F_γ from $\underline{F}$ it also does for those in $\underline{F}\backslash\Delta$. That B' is unique is proved in the same way as in 7.3.

8.10. THEOREM. *Let* $A = \langle (A_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ *be an algebra and* $H = (H_s)_{s\in S}$ *a collection sets with* $H_s \subset A_s$. *Let* $B = \langle [(H_s)_{s\in S}], (F_\gamma)_{\gamma\in\Gamma}\rangle$. *Define* $T = \{s \mid s \in S$ *such that* $H_s \neq \emptyset\}$. *Then* $B' = \langle (B_t)_{t\in T}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ *is an algebra.*

PROOF. $(B_t)_{t\in T}$ is closed under F_γ.

8.11. DEFINITION. The algebra B' from theorem 8.10 is denoted $\mathrm{SubAlg}_H(A)$.

8.12. THEOREM. SubAlg *is safe.*

PROOF. Let $A = \langle (A_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ and $B = \langle (B_s)_{s\in S}, (G_\gamma)_{\gamma\in\Gamma}\rangle$ be similar algebras and $h \in \mathrm{Epi}(A,B)$. Suppose that $H = (H_s)_{s\in S}$ is a collection such that $H_s \subset A_s$. Define $A' = \mathrm{SubAlg}_H(A)$ and $B' = \mathrm{SubAlg}_{h(H)}(B)$. We now prove that for $\hat{h} = h\restriction A'$ holds that $\hat{h} \in \mathrm{Epi}(A',B')$, and that B' is the unique algebra with this property.
First we proof that $\hat{h}(A'_s) = B'_s$.

I. $D = \langle (\hat{h}(A'_s))_{s\in S}, (G_\gamma)_{\gamma\in\Gamma}\rangle$ is an algebra, see theorem 6.3. Since $H_s \subset A_s$ we have $\hat{h}(H_s) \subset h(A_s)$. So the generators of B' are in D, so $(B'_s) \subset \hat{h}(A'_s)$.

II. That $\hat{h}(A'_s) \subset (B'_s)$ is proved by induction:

First note that this is true for the generators of A'_s.
Suppose $\tau(\gamma) = \langle\langle s_1,\ldots,s_n\rangle, s_{n+1}\rangle$. Let $a_1 \in A'_{s_1},\ldots,a_n \in A'_{s_n}$, and assume that

$\hat{h}(a_1) \in B'_{s_1}, \ldots, \hat{h}(a_n) \in B'_{s_n}$. Then $\hat{h}(F_\gamma(a_1,\ldots,a_n)) = G_\gamma(\hat{h}(a_1),\ldots,\hat{h}(a_n))$ because $h \in Epi(A,B)$. Since B is an algebra we have $\hat{h}(F_\gamma(a_1,\ldots,a_n)) \in B'_{s_{n+1}}$.

From I and II it follows that $B' = \hat{h}(A'_s)$, hence $h \in Epi(A',B')$. That B' is unique can be proved in the same way as in 7.3.
8.1. END

Derivers like the ones defined above are not the only safe derivers. Taking a cartesian product or taking a projection from such a product are probably safe in some sense. Such derivers could be relevant for linguistic purposes. In the treatment of presuppositions (by KARTTUNEN & PETERS 1979) a phrase is connected with two formulas: one denoting its meaning, and one denoting its presuppositions. If two phrases are combined in the syntax to form a new phrase, then the two meanings are combined to form the meaning of the new phrase, and the presuppositions are combined to form a new presupposition. This situation fits into the framework if the new semantic algebra would be considered as the product of two copies of the same semantic algebra (however, details of their proposal give rise to complications).

The derivers described in this section together with AddOp are the only derivers we will use. They constitute the basis for the way in which I will introduce derived algebras. A derived algebra will be defined by providing in some way the following information
0) what the old algebra is
1) what the sorts of the new algebra are
2) what the generators of the new algebra are
3) what the operators of the new algebra are.

This information can be used in several ways to build a derived algebra. One might first add the new sorts and then the new operators, or vice versa. One might use the derivers described above, or variants of them. But all methods yield the same algebra, as follows from the uniqueness proof given in the next theorem.

8.13. THEOREM. *Let* $A = \langle (A_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ *and* $B = \langle (B_s)_{s\in S}, (G_\gamma)_{\gamma\in\Gamma}\rangle$ *be similar algebras and let* $h \in Epi(A,B)$. *Let furthermore be given*
1) *a collection of sorts* T *and a mapping* $\sigma: T \to S$,
2) *a collection of new generators* $(H_t)_{t\in T}$, *where* $H_t \subset A_{\sigma(t)}$
3) *a family of polynomials* $P = (p_i)_{i\in I}$ *and a type giving function* $f: (p_i)_{i\in I} \to \bigcup_n T^n \times T$ *such that* $f(p_i) = \langle w,t\rangle$ *only if* $p_i \in POL^A_{\langle\sigma(w),\sigma(t)\rangle}$.

Then there is a unique algebra $D = \langle (D_t)_{t\in T}, \Pi \rangle$ *where*

1) $D_t \subset A_{\sigma(t)}$, *i.e. the carriers of* D *are subsets of the carriers of* A

2) $\Pi = \{p_{i,\langle w,v\rangle,D} \mid p_i \in (P_i)_{i\in I},\ f(p_i) = \langle w,v\rangle$ *and*

$p_{i,\langle w,v\rangle,D}(d_1,\ldots,d_n) = p_A(d_1,\ldots,d_n)\}$

3) D *is generated by the collection* $(H_t)_{t\in T}$, *i.e.* $D = \langle [(H_t)_{t\in T}], \Pi \rangle$

Moreover, there is a unique algebra E *such that for* $\hat{h} = h \restriction D$ *we have* $\hat{h} \in Epi(D,E)$.

PROOF I) *Existence of* D

The derived algebra will be defined in four steps which are indicated in figure 3.

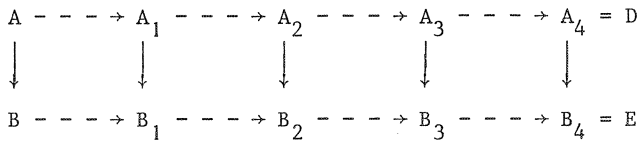


Figure 3. Construction of D

The algebras mentioned in figure 3 are obtained using the derivers defined before.

$A_1 = AddOp_P(A)$, so A_1 is obtained by adding all polynomial symbols mentioned in P.

$A_2 = AddSorts_{T,\sigma}(A_1)$ so A_2 is obtained from A_1 by adding the sorts of T

$A_3 = DelOp_\Delta(A_2)$ where $\Delta = \{\delta \mid f$ is an operator symbol of A_2 and $\delta \notin \Pi\}$

so A_3 has only operators symbols given in P with the type indicated by f.

$A_4 = SubAlg_{(H_t)_{t\in T}}(A_3)$ so A_4 is the algebra built from $(H_t)_{t\in T}$.

From the previous definitions concerning derivers it follows that A_4 is an algebra which satisfies the three conditions mentioned in the theorem.

II. *Uniqueness of* D

Suppose that D1 and D2 are algebras satisfying the requirements for D. Define $D3_t = D1_t \cap D2_t$. Then $D3 = \langle (D3_t)_{t\in T}, \Pi \rangle$ is a subalgebra of D1 because

1) $D3_t \subset D1_t$

2) $(D3_t)_{t\in T}$ is closed under the operations in Π:
Let the type of $\pi \in \Pi$ be $\langle\langle t_1,\ldots,t_n\rangle,t_{n+1}\rangle$ and suppose $d_1 \in D3_{t_1},\ldots,d_n \in D3_{t_n}$. Then $\pi(d_1,\ldots,d_n) \in (D1_{t_{n+1}} \cap D2_{t_{n+1}})$ since π has the same interpretation in D1 and D2, and both D1 and D2 are closed under π. Hence D3 is closed under π.

Moreover, $H_t \subset D3_t$. So D3 is a subalgebra of D1 containing the generators of D1, hence D3 = D1. From this follows that D2 = D1.

III. *Existence and unicity of* E

Algebra E is defined by analogy to the definition of D. So $B_1 = \mathrm{AddOp}_{h(P)}(B)$, B2 = $\mathrm{AddSorts}_{T,\sigma}$(B1), B3 = $\mathrm{DelOp}_{\Delta}(B_2)$ and B4 = $\mathrm{SubAlg}_{(H_t)_{t\in T}}$(B3). Here h(P) is defined as in theorem 7.3 and Δ is defined as above. From the previous theorems about derivers it follows that $h \in \mathrm{Epi}(A_i,B_i)$ for $i \in \{1,2,3\}$ and that $h\restriction A4 \in \mathrm{Epi}(A4,B4)$. It also follows that each B_i is the unique algebra with this property. In particular B4 is the unique algebra which satisfies the requirement for E.

8.14. EXAMPLE. This example consists of the syntax and semantics of a small fragment of English. The meanings of the sentences of the fragment are obtained by translating them into an algebra derived from predicate logic. Its semantics is very primitive: the meaning of a sentence is a truth value. This aspect is not important because the purpose of the example is to illustrate the derivers described in this section.

The generators of the algebraic syntax are as follows:

$$B_T = \{John, Mary\}$$
$$B_{IV} = \{run, walk\}$$
$$B_{CN} = \{child, professor\}.$$

The rules of the syntax are as follows

F_1: T × IV → S defined by $F_1(\alpha,\beta) = \alpha\ \beta s$.

F_2: T × CN → S defined by $F_2(\alpha,\beta) = \alpha\ is\ a\ \beta$.

Examples are:

$F_1(John, run)$ = *John runs*

$F_2(Mary, professor)$ = *Mary is a professor.*

This information determines the following algebraic grammar

$$\langle\langle[\{John, Mary\}_T, \{run, walk\}_{IV}, \{child, professor\}_{CN}], \{F_1, F_2\}\rangle, S\rangle.$$

The fragment is translated into a derived algebra which is determined by the following information.
The elements of B_T are translated respectively into *John,Mary* which are constants of type e. The elements of B_{IV} and B_{CN} are translated into the following constants of type <e,t>: *run,walk,child,professor*. Notice the different type face used for English words and logical constants. The application of operator T_1 (corresponding to rule F_1) is described by the polynomial symbol $x_{2,\langle e,t\rangle}(x_{1,e})$. Consequently, if α' and β' are the translations of α and β respectively, then the translation of the term $F_1(\alpha,\beta)$ is $\beta'(\alpha')$. The operator T_2 (corresponding with rule F_2) is defined by the same polynomial symbol. Examples are:

translation of $F_1(John, run)$ is $run(john)$.

translation of $F_2(Mary, professor)$ is $professor(mary)$.

The description of the derived algebra given above has not the form used in theorem 8.12. But implicitly all that information is given, as appears from the following.

1. The sorts of the derived algebra are the same as those of the syntax: T,IV,CN and S. The mapping σ to the old sorts is $\sigma(T) = e$, $\sigma(IV) = \langle e,t\rangle$, $\sigma(CN) = \langle e,t\rangle$ and $\sigma(S) = t$.
2. The generators of the derived algebra are the translations of the generators of the syntactic algebra.
3. The polynomial operator is $x_{2,\langle e,t\rangle}(x_{1,t})$, and the type-giving function f says $f((x_{2,\langle e,t\rangle}(x_{1,t})) = \{\langle\langle T,IV\rangle,S\rangle,\langle\langle T,CN\rangle,S\rangle\}$.

The process of making a derived algebra as is described in the proof of theorem 8.12 proceeds as follows.
step 1. The polynomial operator $x_{2,\langle e,t\rangle}(x_{1,t})$ is added to the algebra of predicate logic.

step 2. The categories T,CN,IV and S are added. Their carriers consist of the expressions of type e, type <e,t>, type <e,t> and type t respectively. Moreover, the operators are multiplied. For instance, the operator $x_{2,<e,t>}(x_{1,e})$ gets 12 incarnations. Examples of the types of these incarnations are <<s,e>,e>,t>, <<CN,T>S>, <<s,e>,T>S> and <IV,T>t>.
step 3. Everything that is not needed will be removed. The carriers of sorts CN and IV are reduced, which has the effect that they become disjunct. Sort t is removed, and most incarnations of the polynomial are removed, except for <<CN,T>S> and <<IV,T>,S>.

The derived algebra which results from this process is the unique algebra guaranteed in theorem 8.13. In the sequel I will present the information needed to apply theorem 8.11 in the implicit way used here. The process of forming a derived algebra will not be described explicitly.

9. DISCUSSION

The framework defined in this paper is closely related to two proposals in the literature. These proposals are developed in two quite different fields of semantics. The first one is developed by Richard Montague for the treatment of the semantics of natural languages. It is presented in "Universal Grammar" (MONTAGUE 1970b), henceforth UG. The first sentence of this article reads

> *There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians; indeed, I consider it possible to comprehend the syntax and semantics of both kinds of languages within a single natural and mathematical precise theory.*

It is striking to discover that this statement also holds for the languages of computer scientists. Independent of Montague's work, and independent of the philosophical tradition this work was based on, the same ideas were developed in the field of semantics of programming languages by the group called Adj (Goguen, Thatcher, Wagner, Wright). Their motivation had nothing to do with the compositionality principle; they have a practical justification for their framework. The second sentence of ADJ 1979 reads:

> *The belief that the ideas presented here are key, comes from our experience over the last eight years in developing and applying these concepts.*

A more detailed comparision between these proposals and the one described in this chapter will be given below.

The basic difference between Montague's framework and the present one, is that Montague did not have the notion 'many sorted algebra' available. He worked with a one sorted algebra and his syntax consisted of the description of a very special one-sorted algebra: one with much additional structure. I have a much more general algebraic concept of syntax and his one-sorted algebra is a special case. However, the mathematical object Montague defines is the same as the object I define. The two frameworks present different views of the same mathematical object. These different views have some consequences for the details of the framework.

1. In the present framework operators are typed. In Montague's framework operators are typeless, but rules are typed. This has the following consequence. If we apply the UG framework to PTQ, then not the syntactic rules (i.e. S4,S5,...) are the operators in the algebraic sense, but the operations on strings (i.e. $F_1, F_2 \ldots$). The framework requires for each F a single corresponding semantic operation. But this is not for all F's the case (e.g. not for F8: conjunction-operation). This illustrates that the present framework, in which rules and operators coincide, gives an approach which is closer to practice than the UG framework.

2. Both frameworks require that the operators be total. In my framework this means that an operator has to be defined for the whole carrier of the type of its arguments. In Montague's framework it means that the operators have to be defined for all elements on the algebra, even for those elements to which it will never be applied. A similar remark holds for homomorphism. For instance the semantic interpretation has to be defined for expressions which are not expressions of logic such as $\rightarrow \rightarrow p$. In practice no one actually defines homomorphisms for such arguments. In the present framework this practice is sanctioned.

3. In the present framework, there is a natural relation between the disambiguated and the generated language: from an expression in the term algebra one obtains the corresponding expression in the generated language by evaluating the expression. In UG there is a (further unspecified) relation R relating the disambiguated language with the generated language. Such a relation can be used for several purposes: for neat ones such as deleting brackets, but also for filtering, completely reformulating the expression, or building new structures and other obscure operations. That R can be any such relation is not good. As far as I know, no one working in Montague grammar actually uses this extreme power of R. Hence it is attractive to restrict R as we have done.

4. The present framework has some built in restrictions to guarantee that the grammar be effective. The restrictions are obeyed by all existing proposals. The original, unrestricted definitions allow for too uninteresting grammars.

Summarizing, the differences between the present framework and Montague's have as a consequence that the present framework is much closer to practice, and that unwanted, and unused, facilities are no longer available.

Next I will consider the relation of our framework to that of Adj. The basic idea underlying their approach is formulated in ADJ (1977, p.69).

> *In the cases we examine, syntax is an initial algebra, and any other algebra* A *in the class is a possible domain (or semantic algebra); the semantic function is the uniquely determined homomorphism* h_A: S → A, *assigning a meaning* $h_A(s)$ *in* A *to each syntactic structure* s *in* S.

This statement implies that the group Adj works with what we have called 'simple Montague grammars'. They have, however, not explicitly described the framework in which they work. They are interested primarily in practical work concerning the semantics of programming languages. It appears that their work is in accordance with what we have defined as being a (standard) Montague grammar. For instance, a central aspect of the present framework is that polynomial operators are used to define complex operations on meanings. Adj certainly knew about the benefit of polynomials: their papers are full of such operators. But no explicit formulation is given of the role of polynomials in their approach. Since a framework is only implicit, it is possible, that the algebraic theory developed in this chapter is hidden in their works. The most fundamental difference between the two approaches is that they base the semantics on the algebraic grammar for the language, whereas we base it on the corresponding term algebra (i.e. on derivational histories). However, since the framework of Adj is not made explicit, it is difficult to compare their approach with ours. Therefore I restrict myself to the general remarks given above. Below I will discuss some technical differences in the definition of algebra and homomorphism.

In the Adj approach it is required that all similar algebras have the same operator symbols. I prefer to have the possibility of using different operator symbols because that is standard in the field of Montague grammars (the operators from the syntactic algebra are usually denoted S_i, and those of the logical algebra T_i). Furthermore, the Adj definition has as a consequence that renaming the sorts gives rise to a completely new algebra:

an algebra obtained by renaming the sorts is not isomorphic to the original algebra, it is not even similar! For these reasons I prefer the more general definition of many sorted algebra and of similar algebras which are used in this chapter.

In the theory of universal algebras one usually allows for nullary operations, i.e. for operations which do not take an argument and which always yield the same value. In our definition such operations are not allowed. To consider constants as nullary operators is intuitively difficult, and practically inconvenient. For, instance, after having presented their definition, which allows for nullary operations, ADJ (1977,p.71) says that the uniformity is 'mathematically nice', but 'it is often more convenient' to separate them out from the more general operators. Another difficulty is the following. Let an algebraic grammar be given for a certain fragment. Suppose that a new element is added to an existing carrier (a new word is added of an already present category). Then one would judge intuitively that nothing essential is added. If a new rule is added (i.e. a non-nullary operation), then a new type of syntactic constructions is added to the fragment. In such a situation one would say that something essentially is added. If nullary operations would be allowed for, then these two kinds of addition would have the same status, which is not in accordance with practice. However, this difference concerning operators does not give rise to essential differences in the algebraic theory (e.g. because I have adapted suitably the definition of 'polynomial symbol').

In our approach a homomorphism is a mapping with as domain the elements of the carriers. In the Adj approach it is a sorted collection of mappings. For each sort there is a separate mapping. So in case an element occurs in several carriers it is treated as if there are two different elements. The images under the homomorphism can be different for the same element in different sorts. This is not acceptable in our approach. In the process of making a derived algebra we impose a new structure of sorts on the logical algebra, and the interpretation homomorphism has to determine uniquely the interpretation of the elements of the new sorts (which are also elements of the old sorts). If the homomorphism is defined as a sorted collection of functions, the interpretation of the new carriers is arbitrary. Hence theorem 8.12 would not be valid. In order to guarantee an unique interpretation for the derived algebra, the Adj-definition was corrected.

Summarising, the main difference between our approach and that of Adj

is that we base the semantics on the term algebra, whereas Adj does not. Another difference is that we have an explicit framework. Differences in technical details are a consequence of this framework or of requirements from established practice. The present framework might be considered as a synthesis of the idea's of Montague with technical tools of Adj.

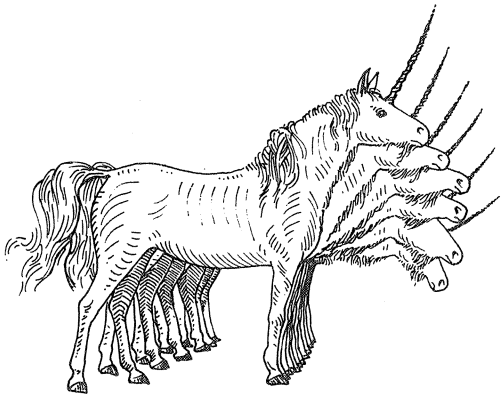
Finally, I will mention some afterthoughts about two points made in this chapter. The choice not to allow for nullary operators is non-standard and has some advantages for the explication. Along this way we came just far enough. But this choice has the disadvantage that existing theory cannot be applied directly. More in particular, I did not succeed in obtaining a handsome definition of a 'free algebra'. Maybe this is a sign that it might be wiser to follow the standard definition and accept the didactic difficulties. The second point concerns the discussion in section 5(p.67) of some remarks of ADJ concerning the context-freeness of the generated language. Joe Goguen (pers.comm) explained that ADJ's remarks should not be understood in literal way. Ind ADJ 1977 the possibility is mentioned that the set of operators is infinite. In that way non-context free languages can be dealt with. The criticism on their approach should therefore not be that ADJ can only deal with context-free languages, but that they can only deal with grammars with context free rules, but not with arbitrary syntactic operations.

CHAPTER III

INTENSIONAL LOGIC

ABSTRACT

In this chapter the language of intensional logic is introduced; this language is a useful tool for representing meanings of e.g. English. The semantic interpretation of intensional logic is defined by a translation into the language Ty2 of two-sorted type theory. Several properties of intensional logic are explained using this translation into Ty2.



1. TWO FACETS

1.1. Introduction

Our aim is to associate in a systematic way the expressions of a language with their meanings. Hence we need a method to represent meanings. The most convenient way to do so, is to use some suitable logical language. Once the interpretation of that language has been defined, it can further be used to represent meanings. The language we will use in this book, is the language of intensional logic, henceforth IL. This language is especially suitable for representing the intended meanings because it 'wears its interpretation upon its sleeves' (Van Benthem, pers.comm.).

In chapter 1 some consequences of the principle of compositionality are discussed. Here I will pay special attention to two of them.

I) The meanings associated with expressions of a natural language or a programming language are intensions, i.e. functions on a domain consisting of a set of 'indices'. The indices formalize several factors which influence the meaning of an expression.

II) The meanings form a many sorted algebra which is similar to the syntactic algebra. Hence we have for each category in the syntactic algebra a corresponding sort in the semantic algebra: the semantic model is 'typed'.

In the light of the close connection between IL and its models, it is not surprising that these two facets of meaning are reflected in IL. This language contains operators connected with indices (e.g. tense operators), as well as operators reflecting the typed structure of the semantic domain (e.g. λ abstraction). This means that IL can be considered as the *amalgamation* of two kinds of languages: type logic and modal tense logic. With this characterization in mind, many properties of IL can be explained. This will be done in the sequel.

1.2. Model - part I

The set of indices plays an important role in the formalization of the notion 'meaning' since meanings are functions with indices as their domain. The definition of the model will not say much about what indices are: they are defined as an arbitrary set. This level of abstraction has the advantage that the meanings of such different languages as English and Algol can be described by them. But one might like to have an intuitive understanding of what indices are, what they are a formal counterpart of, and which degree of reality they have. Several views on these issues are possible, and I will

mention some of them. Thereafter I will give my personal opinion.

1. Our semantic theory gives a model of how the reality is, or might have been. An index represents one of these possibilities. In application to natural language this means that an index represents a possible state of affairs of the reality. In application to programming languages this means that an index represents a possible internal state of the computer.
2. Our semantical theory gives a model of a psychologically acceptable way of dealing with meanings. In this conception an index formalizes a perceptually possible state of affairs (cf. PARTEE 1977b).
3. Languages describe concepts, and users of a language are equipped with a battery of identification procedures for such concepts. An index represents a class of possible outcomes of such procedures (cf. TICHY 1971).
4. Our semantic theory describes how we deal with data. An index represents a maximal, non-contradictory set of data (cf. VELTMAN 1981).
5. 'In order to say what meaning is, we may first ask what a meaning does, and then find something that does that.' (LEWIS 1970). We want meanings to do certain things (e.g. formalize implication relations among sentences), we define meanings in an appropriate way, and indices form a technical tool which is useful to this purpose. Indices are not a formalization of something; they are just a tool.

Conception 1 is intuitively very appealing, and most widespread in the literature. But the interpretations 2,3, and 4 are also intuitively appealing. The reader is invited to choose that conception he likes best. An intuitively conceivable interpretation might help him to understand how and why everything works. But the reader should only stick to his interpretation as long as it is of use to him. For the simple cases indices can probably be considered as an adequate formalization of his intuitions. But once comes the day that his intuition does not help him any more. Then he should switch to conception 5; no interpretation but a technical tool. Such a situation arises, for instance, with the treatment of questions. Do you have an idea of what the meaning of a question should be in the light of conception 1,2,3, or 4? For instance the treatment of indirect questions given in GROENENDIJK & STOKHOF (1981) cannot be explained on the basis of the first four conceptions. They have chosen as meanings those semantic entities which do what they wanted them to do: the indices play just a technical role.

1.3. Model - part II

In the model theory of type-logic the models are constructed from a few basic sets by adding sets of functions between already available sets. Two kinds of models can be distinguished, depending on how many functions are added. In the so called 'standard models', the addition clause says that if A and B are sets in the model, then the set A^B of all functions from B to A also is a set in the model. In the so called 'generalized models' one needs not to take this whole set, but one may take some subset. There is a condition on the construction of models which guarantees that not too few elements are added to the model: every object that can be described in the logic should be incorporated in the model.

The laws of type logic which hold in standard models are not axiomatizable. In order to escape this situation, the generalized models were introduced (HENKIN 1950). By extending the class of possible models, the laws were restricted to an axiomatizable class: the more models the more possible counter examples, and therefore the fewer laws.

What kind of models will be used for the interpretation of intensional logic? I mention four options.

1. the class of all standard models
2. a subclass of the standard models
3. the class of all generalized models
4. a subclass of the generalized models.

Which choice is made, depends on the application one has in mind, and what conception one has about the role of the model (see section 1.2). If one intends to model certain psychological insights, then one might argue that the generalized models with countably many elements are the best choice (cf. PARTEE 1977b, and the discussion in chapter 7). If the model is used for dealing with the semantics of programming languages then a certain subset of the generalized models is required (see chapter 4). In the application of Montague grammar to natural language, one works with option 2. A subclass of the standard models is characterized by means of meaning postulates which give restrictions on the interpretation of the constants of the logic. I will follow this standard approach in the sequel.

1.4. Laws

Most of the proof-theoretic properties of IL can be explained by considering IL as the union of two systems: type logic and modal tense logic. The modal laws of IL are the laws of the modal logic S5. Many of the laws

of type logic are laws of IL, exceptions are variants of the laws which are not valid in modal logic. The laws of type logic (i.e. those which hold in all standard models) are not axiomatizable. Since IL has (on sorted) type logic as a sublanguage, IL is not axiomatizable either. For modal logic there is an axiomatization of the laws which hold in all generalized models. This is expressed by saying that type logic has the property of generalized completeness. By combining these two completeness results, the generalized completeness of IL can be proved (see also section 3).

1.5. Method

I have explained that many aspects of IL can be understood by considering IL as the amalgamation of type logic and modal tense logic. Nevertheless, the formal introduction of IL will not proceed along this line. I will first introduce some other language: Ty2, the language of two sorted type theory. On the basis of Ty2 I will define IL: the algebraic grammar of IL is an algebra derived from the algebraic grammar for Ty2. The reasons for preferring this approach are the following:

1. *Model theoretic*

In Ty2 the indices are treated as elements of a certain type just like all elements. This is not the case for IL. In the interpretation of IL indices occur only as domains of certain functions, but not as range. Therefore the models for IL become structures in which carriers of certain types are deleted, whereas, from the viewpoint of an elegant construction, these carriers should be there. In the models for Ty2 they are there. Remarkable properties of IL can be explained from the fact that these carriers are not incorporated in its models. It appears to be better for understanding, and technically more convenient, to describe first the full model, and to remove next certain sets, instead of to start immediately with the remarkable model.

2. *Homomorphisms*

From the viewpoint of our framework, it is essential to demonstrate that the interpretation of IL is a homomorphism. It seems, however, rather difficult to show that the interpretation homomorphism for type logic and that for modal tense logic can be combined to a single homomorphism for IL. Furthermore, we should, in such an approach, consider first the interpretations of these two languages separately. It is easier to consider only Ty2.

3. *Laws*

Many of the proof rules for Ty2 are easy to formulate and to understand. This is not the case with IL. It is for instance much easier to prove lambda conversion first for Ty2, and derive from this the rule for IL, than to prove the IL rule directly.

4. *Speculation*

We will use IL for expressing meanings of natural language expressions because it is a suitable language for that purpose. No explicit reference to indices is possible in IL, and there is no need to do so for the fragment we will consider. But one may expect that for larger fragments it is unavoidable to have in the logic explicit reference to indices. NEEDHAM (1975) has given philosophical arguments for this opinion, Van BENTHEM (1977) has given technical arguments, and GROENENDIJK & STOKHOF (1981) treat a fragment of natural language where the use of Ty2 turned out to be required. Furthermore we will consider in chapter 4 a kind of semantics for programming languages which requires that states can be mentioned explicitly in the logical language, and we will use Ty2 for that purpose.

2. TWO-SORTED TYPE THEORY

In this section the language Ty2 will be defined: the language of two sorted type theory. The name (due to GALLIN 1975) reflects that the language has two basic types (besides the type of truth values). It is a generalization of one sorted type theory which has (besides the type of truth values) one basic type. The language is defined here by means of an algebraic grammar.

Since in logic it is customary to speak of types, rather than of sorts, I will use this terminology, even in an algebraic context. The collection of types of Ty2 is the smallest set Ty such that

1. $\{e,s,t\} \subset Ty$ (e='entity',s='sense',t='truth value').
2. if $\sigma \in Ty$ and $\tau \in Ty$ then $\langle\sigma,\tau\rangle \in Ty$.

This is the standard notation for types which is used in Montague grammar. It is, however, not the standard notation in type theory. Following CHURCH (1940), the standard notation is $(\sigma)\tau$ instead of $\langle\sigma,\tau\rangle$. I agree with LINK & VARGA (1975) that if we would adopt that notation and some standard conventions from type theory, this would give rise to a simpler notation than

the one defined above. But I prefer not to confuse readers familiar with Montague grammar, and therefore I will use his notation.

For each type $\tau \in Ty$ we have two denumerable sets of symbols:

$$CON_\tau = \{c_{1,\tau}, c_{2,\tau}, \ldots\} \qquad \text{the constants of type } \tau$$

and

$$VAR_\tau = \{v_{1,\tau}, v_{2,\tau}, \ldots\} \qquad \text{the variables of type } \tau.$$

So the constants and variables are indexed by a natural number and a type symbol. The elements of VAR_τ are called variables since they will be used in Ty2 as variables in the logical sense (they should not be confused with variables in the algebraic sense which occur in polynomials over Ty2).

The generators of type τ are the variables and constants of type τ. The carrier of type τ is denoted as ME_τ (meaningful expression of type τ). An element of the algebra is called a (meaningful) expression. The standard convention is to call the meaningful expressions of type t 'formulas', but I will call all meaningful expressions 'formulas'. (this gives the possibility to distinguish them easily from expressions in other languages).

There are denumerable many operators in the algebra of Ty2, because the operators defined below are rather schemes of operators, in which the types involved occur as parameters. For instance the operator for equalities (i.e. $R_=$) corresponds with a whole class of operators: for each type $\tau \in Ty$ there is an operator $R_{=,\tau}$. These operators $R_{=,\tau}$ all have the same effect: $R_{=,\tau}(\alpha,\beta)$ is defined as being the expression $[\alpha=\beta]$. Therefore we can define a whole class with a single scheme. The scheme for $R_=$ should contain τ as a parameter, and other operations should contain two types as parameter. These types are not explicitly mentioned as parameter, since they can easily be derived from the context. The proliferation of operators just sketched is a consequence of the algebraic approach and caused by the fact that, if two expressions belong to different sorts for one operation (say for function application), they belong to different sorts for all operations (so for equality).

The operators of Ty2 are defined as follows

1. *Equality*

$$R_=: ME_\tau \times ME_\tau \to ME_t \quad \text{where } R_=(\alpha,\beta) = [\alpha=\beta].$$

2. *Function Application*

$$R_{(\ )}: ME_{<\sigma,\tau>} \times ME_\sigma \to ME_\tau \quad \text{where } R_{(\ )}(\alpha,\beta) = [\alpha(\beta)].$$

3. *Quantification*

$$R_{\exists v}: ME_t \to ME_t \qquad \text{where } v \in VAR_\tau \text{ and } R_{\exists v}(\phi) = \exists v[\phi].$$

For universal quantification ($R_{\forall v}$) analogously. Recall that in chapter 1 arguments were given for not analyzing $\exists v$ any further.

4. *Abstraction*

$$R_{\lambda v}: ME_\tau \to ME_{<\sigma,\tau>} \qquad \text{where } v \in VAR_\sigma \text{ and } R_{\lambda v}(\alpha) = \lambda v[\alpha].$$

5. *Connectives*

$$R_\wedge: ME_t \times ME_t \to ME_t \qquad \text{where } R_\wedge(\alpha,\beta) = [\alpha \wedge \beta].$$

Analogously for $R_\vee$, $R_\to$, and $R_\leftrightarrow$.

$$R_\neg: ME_t \to ME_t \qquad \text{where } R_\neg(\phi) = [\neg\phi].$$

The (syncategorematic) symbols [and] are used to guarantee unique readability of the formulas. They will be omitted when no confusion is likely. The syncategorematic symbols $\exists v$ (existential quantifier), $\forall v$ (universal quantifier) and λv (the lambda-abstraction) are called binders. A variable is called free when it does not occur within the scope of $\exists v$, $\forall v$, or λv. The notions 'scope' and 'free' can be defined rigorously in the usual way.

This completes the definition of the operators of Ty2. For the semantics of English, two more operators are needed. They introduce ordering symbols between expressions of type s (i.e. between index expressions).

6. *Ordering*

$$R_<: ME_s \times ME_s \to ME_t \qquad R_<(\alpha,\beta) = [\alpha < \beta]$$

$$R_>: ME_s \times ME_s \to ME_t \qquad R_>(\alpha,\beta) = [\alpha > \beta].$$

After having described the sets of sorts, generators and operators of Ty2, I will present the algebraic grammar for Ty2. Let $\underline{R}$ be the collection of operators introduced in clauses 1-5. Then an algebraic grammar for Ty2 is

$$<< \left[(CON_\tau \cup VAR_\tau)_{\tau \in Ty}\right], \underline{R}>, t>.$$

If $\underline{R}$ is replaced by $\underline{R} \cup \{R_<, R_>\}$: then an algebraic grammar for Ty2< is obtained.

3. THE INTERPRETATION OF Ty2

The semantic domain in which Ty2 will be interpreted, consists of a large collection of sets, which are built from a few basic ones. These basic sets are the set A of entities, the set I of indices, and the set {0,1} of truth values. The sets D_τ of possible denotations of type τ are defined by;

1. $D_t = \{0,1\}$, $D_e = A$, $D_s = I$
2. $D_{<\sigma,\tau>} = D_\tau^{D_\sigma}$.

In order to deal with logical variables, we need functions which assign semantical objects to them. The collection AS of variable assignments (based on A and I) is defined by

$$AS = \prod_{\tau \in Ty} (D_\tau^{VAR_\tau}).$$

Let as, as' $\in$ AS. We say that as' $\underset{v}{\sim}$ as (as' is a v-variant of as) if for all $w \in$ VAR such that $w \neq v$ holds that as(w) = as'(w). If as' $\underset{v}{\sim}$ as and as'(v) = d then we write [v→d]as for as'.

Now the necessary preparations are made to say what the elements of the semantic domains are. Let A and I be non-empty sets, and let D_τ be defined as above. The sets M_τ of meanings of type τ (based upon A and I) are:

$$M_\tau = D_\tau^{AS}.$$

By the semantic domain based upon A and I, we understand the collection $(M_\tau)_{\tau \in Ty}$. In such domains we will interpret Ty2. For Ty2< additional structure is required. The set I has to be the cartesian product of two sets W and T, where T is linearly ordered by a relation <. Here W is called the collection of possible worlds, and T the collection of time points. An element i $\in$ W × T is called a *reference point* or *index*.

As is suggested by the definition of semantic domain, the interpretation homomorphism of Ty2 will assign to an expression of type τ some element of M_τ, i.e. the meaning of $\phi \in ME_\tau$ is some function f: AS → D_τ. In chapter one we have formalized the meaning of an expression of predicate logic as a function f: AS → D_t, and here this approach is generalized to other types. In the case of predicate logic a geometrical interpretation of this process was possible, and this led us towards the cylindric algebras. For the case of Ty2 it is not that easy to find a geometrical

interpretation. In any case, I will not try to give one. But I consider the interpretation of Ty2 given here as a generalization of the interpretation of predicate logic with cylindric algebras. Therefore I will call the interpretation of quantifiers of Ty2 'cylindrifications'.

Analogous to the interpretation of variables, there are functions for the interpretation of constants. The collection F (based upon A and I) of functions interpreting constants is defined by:

$$F = \prod_{\tau \in Ty} D_\tau^{CON_\tau}.$$

By a model for Ty2 we understand a pair <M,F> where

1. M is a semantic domain (based on A and I).
2. $F \in F$, hence F is a function for the interpretation of constants (based on the same sets A and I).

In order to define a homomorphism from Ty2 to some model, the models should obtain the structure of an algebra similar to the syntactic algebra of Ty2. That means that I have to say what the carriers, the generators, and the operators of the models are. The generators and operators will be defined below, along with the definition of the interpretation homomorphism V (V = 'valuation'). The carrier of type τ has already been defined; viz. M_τ. So the value of an element of ME_τ under this interpretation V is a function from AS to D_τ. This function will be defined by saying what its value is for an arbitrary assignment as $\in$ AS. I will write $V_{as}(\alpha)$ instead of $V(\alpha)(as)$ because the former notation is the standard one.

The generators of the semantic algebra are the images of the generators of the syntactic algebra. These are defined by

a) $V_{as}(v_{\tau,n}) = as(v_{\tau,n})$

b) $V_{as}(c_{\tau,n}) = F(c_{\tau,n})$.

As for the last clause, one should remember that we are defining the interpretation with respect to some model, and that models are defined as consisting of a large collection of sets and a function F which interprets the constants.

The interpretation of compound expressions of Ty2 will be defined next. Let R be some operator of the syntactic algebra of Ty2. Then the value $V_{as}(R(\alpha))$ will be defined in terms of $V_{as}(\alpha)$. In this way it is determined how $V(R(\alpha))$ is obtained from $V(\alpha)$. Then it is also determined how the operator T, which produces $V(R(\alpha))$ out of $V(\alpha)$ is defined. For each clause

in the definition of $V_{as}(\alpha)$, I will informally describe which semantic operator T is introduced.

1. *Equality*

$$V_{as}(\alpha=\beta) = \begin{cases} 1 & \text{if } V_{as}(\alpha) = V_{as}(\beta) \\ 0 & \text{otherwise.} \end{cases}$$

So $V(\alpha=\beta)$ is a function from assignments to truth values yielding 1 if $V(\alpha)$ and $V(\beta)$ get the same interpretation for that assignment. Consequently $T_{=}$ is the assignment-wise evaluated equality. To be completely correct, I have to say that there is a class of semantic operators described here and not a single one: for each type τ there is an equality operator $T_{=,\tau}$.

2. *Function application*

$$V_{as}(\alpha(\beta)) = V_{as}(\alpha)(V_{as}(\beta)).$$

So if $V(\alpha) \in M_{<\sigma,\tau>}$, $V(\beta) \in M_{\sigma}$, then $V(\alpha(\beta)) \in M_{\tau}$. And $T_{(\,)}: M_{<\sigma,\tau>} \times M_{\sigma} \to M_{\tau}$, where $T_{(\,)}$ is assignment-wise function application of the assignment-wise determined function to the assignment-wise determined argument.

3. *Quantification*

$$V_{as}(\exists v\phi) = \begin{cases} 1 & \text{if there is an as' } \underset{v}{\sim} \text{ as such that } V_{as'}(\phi) = 1. \\ 0 & \text{otherwise.} \end{cases}$$

The element $V(\exists v\phi) \in M_t$ is obtained from $V(\phi) \in M_t$ by application of $T_{\exists v}$. This operation $T_{\exists v}$ is a cylindrification operation like the ones introduced in chapter 1. $V_{as}(\forall v\phi)$ is defined analogously.

4. *Abstraction*

Let $v \in VAR_{\sigma}$ and $\phi \in ME_{\tau}$. Then $V_{as}(\lambda v\phi)$ is that function f with domain D_{σ} such that whenever d is in that domain, then f(d) is $V_{as'}(\phi)$, where as' = $[v \to d]as$. In the sequel I will symbolize this rather long phrase as

$$V_{as}(\lambda v\phi) = \underline{\lambda} d\ V_{[v \to d]as}(\phi).$$

Here $\underline{\lambda}$ might be considered as an abstraction in the meta language, but its role is nothing more than abbreviating the phrase mentioned above: 'that

function which ...'. The semantic operator $T_{\lambda v}$ corresponding to $R_{\lambda v}$ is a function from M_τ to $M_{<\sigma,\tau>}$ where $T_{\lambda v}$ associates with an element $e \in M_\tau$ some function $f \in M_{<\sigma,\tau>}$. This function f assigns to an as $\in$ AS the function that for argument d has the value e(as'), where as' = [v↦d]as.

5. *Connectives*

$$V_{as}(\phi \wedge \psi) = \begin{cases} 1 & \text{if } V_{as}(\phi) = V_{as}(\psi) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

So T is the assignment-wise evaluated conjunction.
The corresponding operators $T_\vee, T_\rightarrow, T_\neg$ and $T_\leftrightarrow$ are defined analogously to $T_\wedge$: assignment-wise evaluated connectives.

This completes the interpretation of Ty2. For the interpretation of Ty2< an additional clause is required.

6. *Ordering*

$$V_{as}(\alpha<\beta) = \begin{cases} 1 & \text{if the world component of } V_{as}(\alpha) \text{ equals the world component of } V_{as}(\beta), \text{ and the time component of } V_{as}(\alpha) \text{ is before the time component of } V_{as}(\beta) \text{ in the linear ordering of T.} \\ 0 & \text{otherwise.} \end{cases}$$

Analogously for $V_{as}(\alpha>\beta)$.

This definition means that in case the world components of α and β are different, then $V_{as}(\alpha<\beta) = 0$. The relation-symbol < does not correspond with a total ordering, and consequently $\neg(\alpha<\beta)$ is not equivalent with $[\alpha>\beta \vee \alpha=\beta]$.

4. PROPERTIES OF Ty2

In the definition of the language Ty2, we introduced a lot of operators (corresponding with connectives and quantifiers). Abstraction was just one among them. In a certain sense, however, it is the most important and powerful operator. The other operators are unnecessary since they can be defined in terms of λ-operators (and =). Also expressions denoting truth values can be defined in this way. I will present the definitions (originating

from HENKIN 1963), without further explications because I will not use their details in the sequel. They are presented here for illustrating the central role of λ-abstraction in this system.

4.1. EXAMPLE *definitions based on λ-operators*.
Let $x,y \in VAR_t$ and $f \in VAR_{<t,t>}$. Then

$$T = [\lambda x[x] = \lambda x[x]]$$
$$F = [\lambda x[x] = \lambda x[T]]$$
$$\neg = [\lambda x[F=x]]$$
$$\wedge = [\lambda x \lambda y[\lambda f[f(x) = y] = \lambda f[f(T)]]]$$
$$\rightarrow = [\lambda x \lambda y[[x \wedge y] = x]]$$
$$\vee = [\lambda x \lambda y[\neg x \rightarrow y]]$$

Let $\tau \in Ty$ and $z \in VAR_\tau$; then:

$$\forall zA = [\lambda zA = \lambda zT].$$

4.1. END

In certain circumstances, we may simplify formulas of the form $\lambda v[\phi](\alpha)$ by substituting the argument α for the free occurrences of v in φ. This kind of simplification is called λ-reduction or λ-conversion. In the theory of λ-calculi this reduction is known under the name β-reduction (α-reduction is change of the bound variable *v*). I described above the central position of λ-operators. This implies that the prooftheory of Ty2 is essentially the prooftheory of λ-calculus. Therefore it is of theoretical importance to know under what circumstances λ-conversion is allowed. But there is also an important practical motivation. In the next chapters we will encounter frequently formulas with many λ-operators. Then, by λ-conversion, these formulas can be reduced to a manageable size. This practical aspect of reducing formulas is the main motivation for considering λ-conversion here in detail. I start with recalling a theorem which says which kinds of reductions are allowed in all contexts. Thereafter some theorems will be given concerning the reduction of formulas containing λ-operators.

4.2. THEOREM. *Let* $\alpha,\alpha' \in ME_\sigma$ *and* $\beta,\beta' \in ME_\tau$ *such that*
a) β *is part of* α
b) β' *is part of* α'
c) α' *is obtained from* α *by substitution of* β' *for* β.

Suppose that for all as $\in$ AS *holds* $V_{as}(\beta) = V_{as}(\beta')$.
Then for all as $\in$ AS *holds* $V_{as}(\alpha) = V_{as}(\alpha')$.

PROOF. Recall that β is a part of α if in the production process of α some rule is used which has β as one of its arguments. Hence β is used in a construction step $R(..,\beta,..)$, where R is an operator from the algebraic grammar for Ty2. That $V_{as}(\beta) = V_{as}(\beta')$ for all as $\in$ AS, means that β' has the same meaning as β. This means that the present theorem is a reformulation of theorem 6.4 in chapter two (here adapted for the present algebra and the present notion of meaning). Hence the same proof applies.
4.2. END

As a consequence of this theorem, two expressions with the same meaning may be replaced by each other 'salva veritate'. This is a generalization of Leibniz' principle (just as the corresponding theorem in chapter 2) since it applies to formulas of any type to be replaced within formulas of any (other) type. The theorem provides a foundation for all reductions (simplifications) of IL-formulas we will meet in the sequel. A (sub)-formula may be replaced by a formula with the same meaning. This may even be done in case it is not yet known in which larger expression they will occur as subformula. The theorem holds due to the fact that we have an algebraic interpretation of Ty_2.

4.3. DEFINITION. Let $\phi \in ME_\sigma$, $\alpha \in ME_\tau$ and $v \in VAR_\tau$. Then $[\alpha/v]\phi$ denotes the formula obtained from ϕ by *substitution* of α for all free occurrences of v in ϕ. This substitution is defined recursively as follows

$$[\alpha/v]v \overset{=}{d} \alpha,\ [\alpha/v]\,w \overset{=}{d} w \text{ (if } w \neq v),\ [\alpha/v]c \overset{=}{d} c$$

$$[\alpha/v][\psi = \eta] \overset{=}{d} [[\alpha/v]\psi] = [[\alpha/v]\eta]$$

$$[\alpha/v][\psi(\eta)] \overset{=}{d} [[\alpha/v]\psi]([\alpha/v]\eta)$$

$$\begin{cases} [\alpha/v][\exists w\phi] \overset{=}{d} \exists w[[\alpha/v]\phi] & \text{if } w \neq v \\ [\alpha/v]\exists v\phi \overset{=}{d} \exists v\phi \end{cases}$$

analogously for $\forall w\phi$ and for $\lambda w\phi$

$$[\alpha/v][\phi \wedge \psi] \overset{=}{d} [[\alpha/v]\phi] \wedge [[\alpha/v]\psi]$$

analogously for the other connectives.

4.3. END

4.4. THEOREM. *Suppose no free variable in* α *becomes bound by substitution of* α *for* v *in* ϕ. *Then* λ*-conversion is allowed; i.e. for all* $as \in AS$:

$$V_{as}(\lambda v[\phi](\alpha)) = V_{as}([\alpha/v]\phi).$$

PROOF. The clause concerning function application in the definition of Ty2 says:

$$V_{as}(\lambda v[\phi](\alpha)) = V_{as}(\lambda v[\phi])(V_{as}(\alpha)).$$

By definition $V_{as}(\lambda v[\phi])$ is that function which for argument d yields value $V_{[v\to d]as}(\phi)$. So, writing A for $V_{as}(\alpha)$, we have

$$V_{as}(\lambda v[\phi])(V_{as}(\alpha)) = V_{as}(\lambda v[\phi])(A) = V_{[v\to A]as}(\phi).$$

We first will prove, that for all $as \in AS$

$$V_{[v\to A]as}(\phi) = V_{as}([\alpha/v]\phi).$$

From this equality the proof of theorem easily follows. The proof of the equality proceeds with induction to the construction of ϕ.

1. $\phi \equiv c$, where $c \in CON_\tau$

$$V_{[v\to A]as}(c) = F(c) = V_{as}(c) = V_{as}([\alpha/v]c).$$

2. $\phi \equiv w$, where $w \in VAR_\tau$.

2.1. $w \neq v$

$$V_{[v\to A]as}(w) = V_{as}(w) = V_{as}([\alpha/v]w).$$

2.2. $w \equiv v$

$$V_{[v\to A]as}(v) = A = V_{as}(\alpha) = V_{as}[\alpha/v]v.$$

3. $\phi \equiv [\psi = \eta]$

$$V_{[v\to A]as}(\psi=\eta) = 1 \quad \text{iff} \quad V_{[v\to A]as}(\psi) = V_{[v\to A]as}(\eta).$$

By induction hypothesis, this is true iff

$$V_{as}([\alpha/v]\psi) = V_{as}([\alpha/v]\eta),$$

hence iff $V_{as}([\alpha/v][\psi=\eta]) = 1$.

4. $\phi \equiv \lambda w \psi$

4.1. $w \equiv v$

$$V_{[v \to A]as}(\lambda v \psi) = V_{as}(\lambda v \psi) = V_{as}[\alpha/v][\lambda v \psi].$$

4.2. $w \not\equiv v$.

The conditions of the theorem guarantee that w does not occur in α. This fact is used in equality I below. Equality II holds since we may apply the induction hypothesis for assignment $[w \to d]$ as, and equality III follows from the definition of substitution.

$$V_{[v \to A]as}(\lambda w \psi) = V_{[v \to V_{as}(\alpha)]as}(\lambda w \psi) =$$

$$\underline{\lambda} d V_{[w \to d]([v \to V_{as}(\alpha)]as)}(\psi) =_{I} \lambda d V_{[w \to d]([v \to V_{[w \to d]as}(\alpha)]as)}(\psi) =$$

$$\underline{\lambda} d V_{[v \to V_{[w \to d]as}(\alpha)]([w \to d]as)}(\psi) =_{II}$$

$$\underline{\lambda} d V_{[w \to d]as}([\alpha/v]\psi) =$$

$$= V_{as} \lambda w [\alpha/v]\psi =_{III}$$

$$V_{as}[\alpha/v]\lambda w \psi.$$

The proof for $\forall v \psi$ and $\exists v \psi$ proceeds analogously.

5. $\phi \equiv \neg \psi$

$$V_{[v \to A]as}(\neg \psi) = 1 \quad \text{iff} \quad V_{[v \to A]as}(\psi) = 0$$

by induction hypothesis we have

$$V_{[v \to A]as}(\psi) = V_{as}[\alpha/v]\psi.$$

So $V_{[v \to A]as}(\neg \psi) = 1$ iff $V_{as} \neg [\alpha/v]\psi = 1$.

Analogously for the other connectives.

4.4. END

From theorem 4.2 it follows that in case λ-conversion is allowed on a certain formula, it is allowed in whatever context the formula occurs. So, given a compound formula with several λ-operators, one may first reduce the operators with the smallest scope and so further, but one may reduce also first the operator with the widest scope, or one may proceed in any other sequence. Does this have consequences for the final result? In other words, is there a unique λ-reduced form ('a λ-normal form')? The answer is affirmative. The only reason which prevents a correct application of the λ-conversion is the syntactic constraint that a free variable in α should not become bound by substitution in φ. Using α-conversion (renaming of bound variables), this obstruction can be eliminated. It can then be shown that each formula in Ty_2 can be reduced by use of α- and λ-conversion to a λ-reduced form which is unique, up to the naming of bound variables (see the proof for typed λ-calculus in ANDREWS 1971 of PIETRZYKOWSKI 1973, which proof can be applied to Ty_2 as well). This property of reduction system is known under the name 'Church-Rosser property'.

The theorem we proved concerning λ-conversion gives a syntactic description of situations in which λ-conversion is allowed. It is, however, possible that the condition mentioned in the theorem is not satisfied, but that nevertheless λ-conversion leads to an equivalent formula. A semantic description of situations in which λ-conversion is allowed, is given in the theorem below. This semantic description is not useful for simplifying Ty2 formulas, since there are no syntactic properties which correspond with the semantic description in the theorem. In applications for the semantics of natural languages or programming languages we will have additional information (for instance from meaning postulates) which makes it possible to apply this theorem on the basis of syntactic criteria.

4.5. THEOREM (JANSSEN 1980). *Let* $\lambda v[\phi](\alpha) \in ME$, *and suppose that for all* as,as' $\in$ As: $V_{as}(\alpha) = V_{as'}(\alpha)$.
Then for all as $\in$ AS: $V_{as}(\lambda v[\phi](\alpha)) = V_{as}([\alpha/v]\phi)$.

PROOF. Consider the proof of theorem 4.4. The only case where is made use of the fact that no variable in α becomes bound, is in the equality I in case 4. Since the condition for the present theorem requires that the interpretation of α does not depend on the choice of as, we have

$$V_{as}(\alpha) = V_{[w \mapsto d]as}(\alpha).$$

Consequently the proof of 4.4 applies, using this justification for equality I.

4.5. END

Ty2 contains typed λ-calculus as a sub-theory. Since typed λ-calculus is not axiomatizable, Ty2 is not axiomatizable either. That typed λ-calculus is not axiomatizable becomes evident if one realizes that its models are very rich: they contain models for the natural numbers. The formal proof of the non-axiomatizability is based upon standard techniques and seems well known. GALLIN (1975) does not give a reference when remarking that typed λ-calculus is not axiomatizable, and HENKIN (1950) only gives some hints concerning a proof. A sketch of a possible proof is as follows. An effective translation of Peano arithmetic into typed λ-calculus is defined (see below for an example). Then it is proven that every formula ϕ from Peano arithmetic is true in the standard model of natural numbers iff the translation of ϕ is true in all standard models for Ty2. Since arithmetic truth is not axiomatizable, Ty2 cannot be axiomatizable either.

An example of an effective translation of Peano arithmetic into typed λ-calculus is given in CHURCH (1940). For curiosity I mention the translations of some numbers and of the successor operator S. Also the Peano-axioms can be formulated in typed λ-calculus. The formulas translating 0,1,2 and S, contain the variables $x \in \mathrm{VAR}_e$, $f \in \mathrm{VAR}_{<e,e>}$, and $v \in \mathrm{VAR}_{<<<e,e>,e>,e>}$. One easily checks that S(0) = 1 and S(1) = 2.

arithmetics	*translation*
0	$\lambda f \lambda x[x]$
1	$\lambda f \lambda x[f(x)]$
2	$\lambda f \lambda x[f(f(x))]$
S	$\lambda v \lambda f \lambda x[f(v(f)(x))]$.

As I already said in section 1, Ty2 is generalized complete: i.e. the class of formulas valid in all generalized models is axiomatizable. The proof is a simple generalization of the proof for one-sorted type theory (HENKIN 1950). I will define below the generalized models and present the axioms without further proof.

The generalized domains of type $\tau \in \mathrm{Ty}$, denoted GD_τ, are defined by

1. $GD_e = A$, $GD_s = I$, $GD_t = \{0,1\}$
2. $GD_{<\sigma,\tau>} \subset GD_\tau^{GD_\sigma}$ $(\sigma,\tau \in Ty)$.

The generalized meanings of type τ denoted GM_τ, are defined by

$$GM_\tau = GD_\tau^{AS} \quad \text{where AS is the set of variable assignments.}$$

A generalized model is a pair <GM,F>, where

1. $GM = \bigcup_\tau GD_\tau$
2. $F \in \mathcal{F}$ where $\mathcal{F}$ is the collection of interpretations of constants
3. the pair <GM,F> is such that there exists a function V which assigns to each formula a meaning, and which satisfies the clauses a,b,1,...6 from the definition of V for Ty2 in section 3.

Without the last requirement concerning generalized models, there might arise difficulties to define V for some model: the interpretation of λ might fail because the required function may fail to belong to the model. The addition of requirement 3 makes that such a situation cannot arise. On the other hand, the third condition makes that it is not evident that generalized models exist since the given definition is not an inductive definition. It can be shown, however, that out of any consistent set of formulas such a model can be built.

The axioms for Ty2 are as follows (GALLIN 1975, p.60):

A1) $g(\mathrm{T}) \wedge g(\mathrm{F}) = \forall x[g(x)]$ $\quad x \in VAR_t$, $g \in VAR_{<t,t>}$

A2) $x = y \rightarrow f(x) = f(y)$ $\quad x \in VAR_\tau$, $f \in VAR_{<\tau,t>}$

A3) $\forall x[f(x) = g(x)] = [f = g]$ $\quad x \in VAR_\sigma$, $f,g \in VAR_{<\sigma,\tau>}$

AS4) $\lambda x[A(x)](B) = [B/x](A(x))$ where the condition of th.4.4 is satisfied.

Furthermore, there is the following rule of inference:

From $A = A'$ and the formula B one may infer formula B', where B' comes from B by replacing one occurrence of A which is part of B, by the formula A' (cf. theorem 4.2).

5. INTENSIONAL LOGIC

The language of intensional logic is for the most part the same as the language Ty2. The collection of types of IL is a subset of the collection types of Ty2. The set T of types of IL (sorts of IL) is defined as the

smallest set such that

1. $e \in T$ and $t \in T$
2. if $\sigma,\tau \in T$ then $\langle\sigma,\tau\rangle \in T$
3. if $\tau \in T$ then $\langle s,\tau\rangle \in T$.

The language IL is defined by the following algebra.

$$IL = \langle\langle[CON_\tau \cup VAR_\tau]_{\tau\in T}, \underline{R} \cup \{R_\vee, R_\wedge, R_\Box, R_W, R_H\}\rangle, t\rangle$$

where

a. CON_τ and VAR_τ are the same as for Ty2 (as far as the types involved belong to T)

b. $\underline{R}$ consists of all the operations of Ty2 (as far as the types involved belong to T).

The new operators are as follows

1. $R_{\vee,\tau}: ME_{\langle s,\tau\rangle} \to ME_\tau$ defined by $R_{\vee,\tau}(\alpha) = [^\vee\alpha]$
2. $R_{\wedge,\tau}: ME_\tau \to ME_{\langle s,\tau\rangle}$ defined by $R_{\wedge,\tau}(\alpha) = [^\wedge\alpha]$
3. $R_\Box: ME_t \to ME_t$ defined by $R_\Box(\phi) = [\Box\phi]$
4. $R_W: ME_t \to ME_t$ defined by $R_W(\phi) = [W\phi]$
5. $R_H: ME_t \to ME_t$ defined by $R_H(\phi) = [H\phi]$.

The symbol $^\vee$ is pronounced as 'extension' or 'down', $^\wedge$ as 'intension' or 'up', $\Box$ as 'necessarily', and W and H are the future tense operator (W ~ 'will'), and the past tense operator respectively (H ~ 'has'). This use of the symbols H and W follows Montague's PTQ. It should be observed that his notation conflicts with the tradition in tense logic, where P('past') and F('future') are used for this purpose. The operators W and H are used in tense logic for respectively 'it always will be the case' and 'it has always be the case'.

The semantics of IL will be defined indirectly: by means of a translation of IL into Ty2<. I will employ the techniques developed in chapter 2 and design a Montague grammar for IL. This means that a homomorphism Tr will be defined from the term algebra T_{IL} associated with IL, to an algebra Der(Ty2) which is derived from Ty2< (see figure 1). The meaning of an IL expression ϕ is then defined as its image under the composition of this translation Tr and the interpretation homomorphism V for Ty_2 (where V is restricted to the derived algebra). This composition Tr∘V will be denoted

V as well, since from the context it will be clear whether the interpretation of an IL expression or of a Ty2 expression is meant.

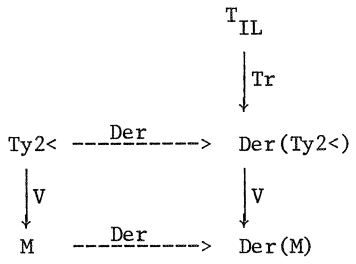


Figure 1. IL as a Montague grammar

The translation Tr will introduce the variables $v_{1,s}$ and $v_{2,s}$, which will play a special role in the interpretation of IL. Following GROENENDIJK & STOKHOF 1981, these variables will be written *a* and *a'* respectively. The translation Tr introduces only variable *a* as free variable of type s. Since the expressions of IL contain no variables of type s, this means that the interpretation of an IL-expression is determined by the interpretation of the IL-variables and the interpretation of *a*. Hence the meaning of an IL-expression can be considered as a function with as domain the assignments to pairs consisting of the value of *a*, and an assignment to IL-variables.

Let the collection G of assignments to IL variables be defined by

$$G = \prod_{\tau} D_{\tau}^{VAR_{\tau}}.$$

The meaning of an IL-formula $\phi \in ME_{\tau}$ is then an element of $D^{I\times G}$, where the component I determines the value assigned to *a*. The interpretation of α with respect to $i \in I$ and $g \in G$ is denoted by $V_{i,g}(\alpha)$.

From chapter 2, I recall a special method for the description of derived algebras. Let the original algebra be $A = \langle (A_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$. Then a derived algebra is uniquely determined by the following information (see chapter 2, theorem 8.13).

1) A collection S' of sorts of the derived algebra and a mapping $\tau: S' \to S$ which associates new sorts with old sorts.
2) A collection $(H_{s'})_{s'\in S'}$ of generators for the new algebra, such that $H_{s'} \subset A_{\tau(s')}$.
3) A collection $P \subset POL^A$ of operators of the new algebra, and a typegiving

function for these operators which has to satisfy certain requirements.

The interpretation of IL will be defined by means of an algebra which is derived from Ty2 in the just mentioned way. The specification of the three components is as follows.

1) The set T of types of IL is a subset of set of types of Ty2<. Hence the set of types of the derived algebra is T. For the mapping τ we take the identity mapping.

2) There are two kinds of generators in the derived algebra: those which correspond with variables of IL, and those which correspond with the constants of IL. As generators corresponding with the IL-variables, the same Ty2-variables are taken. For the constants we do not proceed in this way. Would we have done so, then a constant of IL would always been associated with one and the same semantical object, because the Ty2-constants have this property. For applications this is not desirable. For instance, the constant *walk* will be interpreted (at a given index) as a function from entities to truth values, thus determining the set of entities walking on that index. We desire, however, that for another index this set may be different. Therefore it is not attractive to take constants of Ty2< as generators of the derived algebra. We will use the variable $a \in VAR_s$ for indicating the current index. The generator corresponding to the IL-constant $c_{n,\sigma}$ is the compound formula $c_{n,<s,\sigma>}(a)$. Note that this formula contains the constant with the same index, but of one intension level higher. These considerations explain the following definition

$$H_{\sigma'} = VAR_{\sigma'} \cup \{c_{n,<s,\sigma'>}(a) \mid n \in \mathbb{N}\}.$$

3) Note first that the type giving function for the polynomials does not need to be specified because all types of IL are types of Ty2. There are two kinds of operators. Some operators of IL are also operators of Ty2< as well. The polynomial symbols for these operators (see chapter 2, remark after theorem 4.6) are incorporated in P. The polynomial symbols corresponding with the other operators of IL are as follows

$$R_{\vee,\tau}: X_{1,\tau}(a)$$
$$R_{\wedge,\tau}: \lambda a[X_{1,\tau}]$$
$$R_{\Box}: \forall a[X_{1,t}]$$

R_W: $\exists a'>a[\lambda a[X_{1,t}](a')]$

R_H: $\exists a'<a[\lambda a[X_{1,t}](a')]$.

In the polynomials for $R_\square$ and $R_\wedge$ a binder for a is introduced. In most cases this does not give rise to vacuous quantification or abstraction since the variable X will often be replaced by an expression containing a free variable a introduced by the translation of some constant. The polynomial for R_W might be read as $\exists a'>a[[a'/a]X_{1,t}]$ and for R_H analogously (but these expressions are not polynomial symbols).

The information given above completely determines a unique derived algebra. Theorems of chapter 2 guarantee that in this indirect way the interpretation of IL is defined as the composition of the translation of IL into Der(Ty2<) with the interpretation of this derived algebra.

6. PROPERTIES OF IL

Below, and in the next section, some theorems concerning IL will be presented. The proofs will rely on the way in which the meaning of IL is defined: as the composition of the translation homomorphism Tr and the meaning homomorphism V for Ty2<. Hence the interpretation $V_{i,g}(\phi)$ of an IL-expression ϕ equals the interpretation $V_{as}(Tr(\phi))$ of its translation in Ty2<, where as is an assignment to Ty2-variables such that $as(a) = i$ and $as(v) = g(v)$ for all IL-variables v. Hence we may prove $V_{i,g}(\phi) = V_{i,g}(\psi)$ by proving $V_{as}(Tr(\phi)) = V_{as}(Tr(\psi))$ for such a Ty-assignment as. If η is an expression of Ty2, then the notation $V_{i,g}(\eta)$ will be used for $V_{as}(\eta)$, where as is an arbitrary assignment to Ty2 variables such that as(a) = i and as(v) = g(v) for all IL-variables v. If ϕ is of type t, we will often write $i,g \models \phi$ instead of $V_{i,g}(\phi) = 1$. When i or g are arbitrary, they will be omitted. Hence $\models \phi$ means that for all i and g it is the case that $i,g \models \phi$.

In section 4 we notified the theoretical importance of λ-conversion for Ty2: all quantifiers and connectives can be defined by means of lambda operators. For Ty2 the same holds, so it is of theoretical importance to know under which circumstances λ-conversion is allowed. But there also is an important practical motivation. We will frequently use λ-conversion for simplifying formulas. For these reasons I will consider the IL-variants of the theorems concerning the substitution of equivalents and concerning λ-conversion.

6.1. THEOREM. *Let* $\alpha,\alpha' \in ME_\sigma$ *and* $\beta,\beta' \in ME_\tau$ *such that*
a) β *is part of* α
b) β' *is part of* α'
c) α' *is obtained from* α *by substitution of* β' *for* β.
Suppose that for all $i \in I$ *and* $g \in G$

$$V_{i,g}(\beta) = V_{i,g}(\beta').$$

Then for all $i \in I$ *and* $g \in G$

$$V_{i,g}(\alpha) = V_{i,g}(\alpha').$$

PROOF. This theorem could be proven in the same way as the corresponding theorem for Ty2: by reference to theorem 6.4 from chapter 2. I prefer, however, to prove the theorem by means of translation into Ty2 because this shows some arguments which will be used (implicitly or explicitly) in the other proofs.

From (1) we may conclude that (2) holds, from which (3) immediately follows:
(1) for all $i \in I$, $g \in G$: $V_{i,g}(\beta) = V_{i,g}(\beta')$
(2) for all $i \in I$, $g \in G$: $V_{i,g}(Tr(\beta)) = V_{i,g}(Tr(\beta'))$
(3) for all $as \in As$: $V_{as}(Tr(\beta)) = V_{as}(Tr(\beta'))$.
Recall that β is a part of α if there is in the production of α an application $R(..,\beta,..)$ of an operator R with β as one of its argument. Since Tr is a homomorphism defined on such production processes, it follows that $Tr(\alpha) = Tr(..,R(..,\beta,..)..) = ...R'(..,Tr(\beta),..)...$ (here is R' the polynomial operator over Ty2 which corresponds with the IL-operator R). This says that the translation of a part of α is a part of the translation of α. Consequently we may apply to (3) theorem 4.2 and conclude that (4) holds.
(4) For all $as \in AS$: $V_{as}(Tr(\alpha)) = V_{as}(Tr(\alpha'))$.
From this follows
(5) For all $i \in I$, $g \in G$: $V_{i,g}(\alpha) = V_{i,g}(\alpha')$.
6.1. END

An important class of expressions are the expressions which contain neither constants, nor the operators $^\vee$, H or W. Such expressions are called modally closed; a formal definition is as follows.

6.2. DEFINITION. An expression of IL is called *modally closed* if it is an element of the subalgebra

$$\langle [(VAR_\tau)_{\tau \in Ty}], \underline{R} \cup \{R_\wedge, R_\Box\} \rangle$$

where $\underline{R}$ consists of the operators of Ty2.
6.2. END

The theorem for λ-conversion which corresponds with theorem 4.4 reads as follows

6.3. THEOREM. *Let* $\lambda v[\phi](\alpha) \in ME_\tau$, *and suppose that no free variable in* α *becomes bound by substitution of* α *for* v *in* ϕ. *Suppose that one of the following two conditions holds:*
1. *no occurrence of* v *in* ϕ *lies within the scope of* $^\wedge$,H,W, *or* $\Box$
2. α *is modally closed.*
Then for all $i \in I$ *and* $g \in G$

$$i,g \models \lambda v[\phi](\alpha) = [\alpha/v]\phi.$$

PROOF. *Part 1.*
Suppose condition 1 is satisfied.
The translation $Tr(\alpha)$ of α contains the same variables as α, except for the possible introduction of variables of type s. The translation $Tr(\phi)$ of ϕ contains the same binders as ϕ since only $^\wedge$,H,W, and $\Box$ introduce new binders (see the definition of Tr). Since ϕ itself does not contain binders for variables of type s, we conclude that:

No free variable in $Tr(\phi)$ becomes bound by substitution of $Tr(\alpha)$ for v in $Tr(\phi)$.

Theorem 6.1 allows us to conclude from this that for all $as \in AS$.

$$V_{as}(\lambda v[Tr(\phi)](Tr(\alpha))) = V_{as}([Tr(\alpha)/v]Tr(\phi)).$$

Note that $Tr(\phi)$ has the same occurrences of v as ϕ, hence one easily proves with induction that

$$[Tr(\alpha)/v]Tr(\phi) = Tr([\alpha/v]\phi).$$

Consequently $V_{as}(Tr(\lambda v[\phi](\alpha))) = V_{as}(Tr([\alpha/v]\phi))$.

So $Tr \circ V(\lambda v[\phi](\alpha)) = Tr \circ V([\alpha/v]\phi)$.
From this it follows that for all $g \in G$ and $i \in I$

$$g,i \models \lambda v[\phi](\alpha) = [\alpha/v]\phi.$$

Part 2.
Suppose that condition 2 is satisfied.
The translation of ϕ may introduce binders for variables of type s, but it does not introduce binders for variables of other types (see the definition of Tr). The expression α does not contain free variables of type s, and the translation in this case does not introduce such variables since the only kind of expressions which give rise to new free variables are constants, and the operators $^{\vee}$,H, and W. So we may conclude that:

> No free variable in $Tr(\alpha)$ becomes bound by substitution of $Tr(\alpha)$ for v in $Tr(\phi)$.

From this we can prove the theorem in the same way as we did for the first condition.
6.3. END

In theorem 4.5 a semantic description was given of situations in which λ-conversion is allowed. The IL variant of this theorem reads as follows.

6.4. THEOREM (IL). *Let* $\lambda v[\phi](\alpha) \in ME$ *and suppose for all* $i,j \in I$ *and* $g,h \in G$: $V_{i,g}(\alpha) = V_{j,h}(\alpha)$. *Then for all* $i \in I$ *and* $g \in G$:
$V_{i,g}(\lambda v[\phi](\alpha)) = V_{i,g}([\alpha/v]\phi)$.

PROOF. By translation into Ty2 and application of theorem 4.5.
6.4. END

In the light of the role of λ-conversion, it is interesting to know whether λ-conversion in IL has the Church-Rosser property, i.e. whether there is an unique lambda-reduced normal form for IL. In much practical experience with intensional logic I learned that it does not matter in which order a formula containing several λ-operators is simplified: first applying λ-reduction to the most embedded operators, or first the most outside ones, the final result was the same. It was a big surprise that FRIEDMAN & WARREN (1980b) found an IL-expression where different reduction sequences yield different final results. Their example is

$$\lambda x[\lambda y[\hat{}\, y = u(x)\,](x)\,](c)$$

where x and y are variables of some type τ, c a constant of type τ, and u a variable of type $<\tau,<s,\tau>>$. For each of the λ operators the conditions for the theorem are satisfied. Reducing first λx yields

$$\lambda y[\hat{}\, y = u(c)\,](c)$$

which cannot be reduced further since the conditions for λ-conversion are not satisfied. Reducing first λy yields

$$\lambda x[\hat{}\, x = u(x)\,](c)$$

which cannot be reduced either. We end up with two different, although logical equivalent, formulas; i.e. there is no λ-reduced normal form for IL.

The example depends on the particular form for λ-contraction: for all occurrences of the variable the substitution takes place in one and the same step. FRIEDMAN & WARREN (1980
is equivalent to

$$[\lambda x[\hat{}\, x]\,](c) = u(c)\,.$$

This formula is in some sense further reduced. They conjecture that for a certain reformulation of λ-conversion the Church-Rosser property could be provable.

It is interesting to compare the above discussion with the situation in Ty2, where there is a unique λ-reduced form. The Ty2-translation of the Friedman-Warren formula is ($c' \in CON_{<s,\tau>}$!)

$$\lambda x[\lambda y[\lambda a[y] = u(x)\,](x)\,]c'(a)\,.$$

This reduces to

$$\lambda y[\lambda a[y] = u(c'(a))\,]c'(a)\,.$$

After renaming the bound variable a to i, this reduces further to

$$\lambda y[\lambda i[c'(a)] = u(c'(a))].$$

Note that this last reduction is possible here (and not in IL) because of the explicit abstraction λi, instead of the implicit abstraction in $^{\wedge}y$.

A lot of laws of IL are variants of well known laws for predicate logic and type logic. An exception to this description is formed by alws involving constants. The constants of IL are interpreted in a remarkable way: their interpretation is state dependent. Invalid is, for instance, the existential generalization $\Box A(a) \rightarrow \exists x \Box A(x)$, whereas $\forall y(\Box A(y) \rightarrow \exists x \Box A(x))$ is valid. Invalid is $\forall x[x = c \rightarrow \Box[x = c]]$, whereas $\forall x \forall y[x = y \rightarrow \Box[x = y]]$ is valid. Other examples of invalid formulas are $\exists y \Box[y = c]$ and $\forall x[\Diamond[A(x)] \rightarrow \Diamond A(a)]$, where $\Diamond$ abbreviates $\neg \Box \neg$.

Since IL contains type theory as a sublanguage, there is no axiomatization of IL (see also section 4). But IL is generalized complete as is proved by GALLIN (1975). The proof is obtained by combining the proof of generalized completeness of type theory (HENKIN 1950), and the completeness proof for modal logic (see HUGHES & CRESSWELL 1968). The following axioms for IL are due to GALLIN (1975); the formulation is adapted

A1	$[g(T) \wedge g(F)] = \forall x[g(x)]$	$x \in VAR_t$, $g \in VAR_{<t,t>}$
A2	$x = y \rightarrow f(x) = f(y)$	$x \in VAR_\sigma$, $f \in VAR_{<\sigma,t>}$
A3	$\forall x[f(x) = g(x)] = [f = g]$	$x \in VAR_\sigma$, $f,g \in VAR_{<\sigma,\tau>}$
A4	$\lambda x[\alpha](\beta) = [\beta/x]\alpha$	if the conditions of theorem 5.2. are satisfied
A5	$\Box[^{\vee}f = {}^{\vee}g] = [f = g]$	$f,g \in VAR_{<s,\tau>}$
A6	$^{\vee\wedge}\alpha = \alpha$	$\alpha \in ME_\sigma$.

The rule of inference is:

> From $A = A'$ and the formula B one may infer to formula B', where B' comes from B by replacing one occurrence of A, that is part of B, by A'

Notice that the translation of axiom A5 into Ty2, would lead to a formula of the form of A3, and that the translation of A6 into Ty2 would be of the form of A4. Axiom A6 will be considered in detail in the next section.

I will not consider details of this axiomatization for the following three reasons.

1) This axiomatization was designed for constituting a basis for the completeness proof, and not for proving theorems in practice. To prove the most simple theorems on the basis of the above axioms would be rather difficult. All proofs that will be given in the sequel are semantic proofs and not syntactic proofs: i.e. the proofs will be based upon the interpretation of formulas and not on axioms.
2) We will work with models which obey certain postulates. These postulates express many important semantic details, and most of the proofs we are interested in, are based upon these special properties and not on the general properties described by the axioms.
3) We do not work with generalized models,but with standard models. So the axiomatization is not complete in this respect.

7. EXTENSION AND INTENSION

In this section special attention is paid to the interaction of the extension operator and the intension operator. In this way some insight is obtained in these operators and their sometimes remarkable properties. The 'Bigboss' example, which will be given below, is important since the *Bigboss* will figure as (counter)example on several occasions.

7.1. THEOREM. *For all* i,g: $V_{i,g}{}^{\vee\wedge}[\alpha] = V_{i,g}\alpha$.

PROOF. $Tr({}^{\vee\wedge}\alpha) = Tr({}^{\wedge}\alpha)(a) = \lambda a[Tr(\alpha)](a) = [a/a]Tr(\alpha) = Tr(\alpha)$.

Note that λ-conversion is allowed because the condition of theorem 4.5 is satisfied.
7.1. END

It was widely believed that the extension operator should be the right inverse of the intension operator as well. This believe is expressed in PARTEE (1975,p.250) and in GEBAUER (1978,p.47). It is true, however, only in certain cases. In order to clarify the situation, consider the following description of the effect of ${}^{\wedge\vee}$. Let I and D be denumerable, so $D_\tau = \{d_1,d_2,\ldots\}$ and $I = \{i_1,i_2,\ldots\}$. Let $\alpha \in ME_\tau$, hence the meaning of α is a function with domain I and range D_τ. We may represent α as a denumerable sequence of elements from D_τ. An example is given in figure 2.

arguments	i_1 i_2 i_3 i_4 ..
$V_{i_1,g}(\alpha)$: values for the respective arguments	$\underline{d_1}$ d_2 d_1 d_3 ..
$V_{i_2,g}(\alpha)$: values for the respective arguments	d_2 $\underline{d_2}$ d_3 d_1 ..
$V_{i_3,g}(\alpha)$: values for the respective arguments	d_1 d_1 $\underline{d_2}$ d_1 ..

Figure 2. The interpreation of $\alpha \in ME_\tau$

The interpretation for index i of $^{\wedge\vee}\alpha$ is some function with domain I and range D_τ. Which function it is does not depend on the choice of i, because $Tr(^{\wedge\vee}\alpha)$ which equals $\lambda a[Tr(\alpha)(a)]$, contains no free variables of type s. The function $V_{i,g}(^{\wedge\vee}\alpha)$ yields for argument i_n as value the value of $V_{i_n,g}(\alpha)$ for argument i_n. So in the above example for argument i_1 it yields as value d_1, for i_2 it yields d_2, and for i_3 it yields d_2 (the underlined elements). One observes that $^{\wedge\vee}\alpha$ is the diagonalization of α. So $^{\wedge\vee}\alpha = \alpha$ will hold for all i,g if for all $i,j \in I$: $V_{i,g}(\alpha) = V_{j,g}(\alpha)$. A syntactic description of a class of formulas for which the equality holds, is given in the following theorem.

7.2. THEOREM. *Suppose* α *is modally closed. Then for all* i,g: $V_{i,g}[^{\wedge\vee}\alpha] = V_{i,g}(\alpha)$.

PROOF. The functions $Tr(^{\wedge\vee}\alpha)$ and $Tr(\alpha)$ denote functions with domain I, and their values for an arbitrary argument i are the same:

$$Tr(^{\wedge\vee}\alpha)(i) = \lambda a[Tr(\alpha)(a)](i) = [i/a]Tr(\alpha)([i/a]a) = Tr(\alpha)(i).$$

Notice that $[i/a]Tr(\alpha) = Tr(\alpha)$ since α is modally closed. So for all as $\in$ AS: $V_{as}\ Tr(^{\wedge\vee}\alpha) = V_{as}(Tr(\alpha))$, which proves the theorem.
7.2. END

The insights obtained from the 'diagonalization' point of view, can be used to obtain a counterexample for the case that α is not modally closed. It suffices to find an expression α of type τ which has at index i_1 as its denotation a constant function from I to D_τ, say with constantly value d_1, and at index i_2 as its denotation a constant function yielding some other value, say d_2. This situation is represented in figure 3.

arguments	i_1 i_2 i_3 i_4
$V_{i_1,g}(\alpha)$: values for the respective arguments	$\underline{d}_1$ d_1 d_1 d_1
$V_{i_2,g}(\alpha)$: values for the respective arguments	d_2 $\underline{d}_2$ d_2 d_2

Figure 3. A counterexample for $^{\wedge\vee}\alpha$.

Now $V_{i_1,g}[^{\wedge\vee}\alpha](i_1) = d_1 \neq d_2 = V_{i_2,g}[^{\wedge\vee}\alpha](i_1)$.

One way to obtain this effect is by means of a constant. I give an example of a somewhat artificial nature (due to JANSSEN 1980). Let the valuation of the constant *Bigboss* $\in CON_{<s,e>}$ for index i be the function constantly yielding the object $d \in D_e$ to which the predicate 'is the most powerful man on earth' applies on that index. A possible variant of this example would be the constant *Miss-world* $\in CON_{<s,e>}$, to which the predicate applies 'is elected as most beautiful woman in the world'.

Assume that for the constant *Bigboss* holds that for all $j \in I$ both

$$V_{i_1,g}[\textit{Bigboss}](j) = V_{i_1,g}[\textit{Reagan}]$$

and

$$V_{i_2,g}[\textit{Bigboss}](j) = V_{i_2,g}[\textit{Bresnjev}].$$

Then

$$V_{i_1,g}[^{\wedge\vee}\textit{Bigboss}](i_2) = \underline{\lambda}i[V_{i,g}[\textit{Bigboss}](i)](i_2) =$$

$$= V_{i_2,g}[\textit{Bigboss}](i_2) = V_{i_2,g}[\textit{Bresnjev}].$$

So

$$V_{i_1,g}[^{\wedge\vee}\textit{Bigboss}](i_2) \neq V_{i_1,g}[\textit{Bigboss}](i_2).$$

This effect does not depend on the special interpretations for constants. Another way to obtain the desired effect is by taking for α the expression x where x is a variable of type <s,<s,e>>. Let $g(x)$ be defined such that for all $j \in I$: $g(x)(j) = V_{j,g}[\textit{Bigboss}]$. Then $^{\wedge\vee\vee}x \neq {}^{\vee}x$:
because $V_{i_1,g}(^{\wedge\vee\vee}x)(i_2) = \underline{\lambda}i[g(x)(i)(i)](i_2) = V_{i_2,g}[\textit{Bigboss}](i_2) =$
$= V_{i_2,g}[\textit{Bresnev}]$
whereas $V_{i_1,g}(^{\vee}x)(i_2) = g(x)(i_1)(i_2) = V_{i_1,g}[\textit{Bigboss}](i_2) = V_{i_1,g}[\textit{Reagan}]$.

The next example concerns the situation that IL is extended with the *if-then-else* construct. Let β be of type t and ϕ and ψ of type τ, and define

$$V_{i_1,g}[\underline{if}\ \beta\ \underline{then}\ \phi\ \underline{else}\ \psi] = \begin{cases} V_{i_1,g}(\phi) & \text{if } V_{i_1,g}(\beta) = 1 \\ V_{i,g}(\psi) & \text{otherwise.} \end{cases}$$

Let x and y be variables of type <s,e> and assume that $g(x) \neq g(y)$, but that for some i holds that $g(x)(i) = g(y)(i)$.
Then it is not true that for all i,g

$$i,g \models {}^{\wedge\vee}[\underline{if}\ {}^{\vee}x = {}^{\vee}y\ \underline{then}\ x\ \underline{else}\ y] = [\underline{if}\ {}^{\vee}x = {}^{\vee}y\ \underline{then}\ x\ \underline{else}\ y].$$

This kind of expression is rather likely to occur in the description of semantics of programming languages.

The last example is due to GROENENDIJK & STOKHOF (1981). They consider the semantics of *whether*-complements. An example is

John knows whether Mary walks.

The verb *know* is analysed as a relation between an individual and a proposition. Which proposition is John asserted to know? If it is the case that Mary walks, then John is asserted to know that Mary walks. And if Mary does not walk, then he is asserted to know that Mary does not walk. So the proposition John knows appears to be

if walk(mary) then $^{\wedge}$*walk(mary) else* $^{\wedge}[\neg$*walk(mary)*$]$.

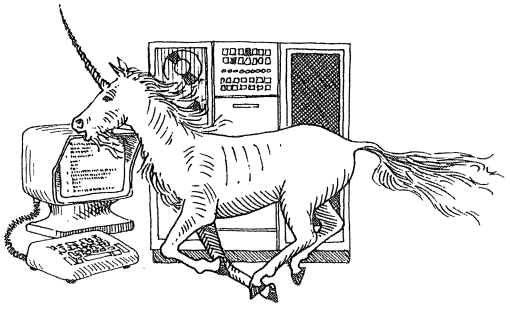
This example provides for a rather natural example of a formula ϕ for which ${}^{\wedge\vee}\phi$ does not reduce.

CHAPTER IV

MONTAGUE GRAMMAR AND PROGRAMMING LANGUAGES

ABSTRACT

The present chapter starts with an introduction to the semantics of programming languages. The semantics of the assignment statement is considered in detail, and the traditional approaches which use predicate transformers are shown to give rise to problems. A solution is presented according to the algebraic framework defined in the first chapters of this book; it uses an extension of intensional logic.



1. ASSIGNMENT STATEMENTS

1.1. Introduction

Programs are pieces of text, written in some programming language. These languages are designed for the special purpose of instructing computers. They also are used in communication among human beings for telling them how to instruct computers or for communicating algorithms which are not intended for computer execution. So for programming languages we are in the same situation as for natural languages. We have a syntax and we have intended meanings, and we wish to relate these two aspects in a systematic way. Since we are in the same situation, we may apply the same framework. In this chapter we will do so for a certain fragment of the programming language ALGOL 68.

There exists nowadays several thousands of mutually incompatible programming languages. They are formal languages with a complete formal definition of the syntax of the language. Such a definition specifies exactly when a string of symbols over the alphabet of the language is a program and when not. The definition of a programming language also specifies how a program should be executed on a computer, or, formulated more generally, what the program is intended to do. In fact, however, several programming languages are not adequately documented in this respect. Each programming language has its own set of strange idiosyncracies, design errors, perfectly good ideas and clumsy conventions. However, there are a few standard types of instructions present in most of the languages. The present chapter deals mainly with the semantics of one of those instructions: the assignment statement which assigns a value to a name.

It appears that assignment statements exhibit the same phenomena as intensional operators in natural languages. A certain position in the context of an assignment statement is transparent (certain substitutions for names are allowed), whereas another position is opaque (such substitutions are not allowed). The traditional ways of treating the semantics of programming languages do not provide tools for dealing with intensional phenomena. A correct treatment of simple cases of the assignment statement can be given, but for the more complex cases the traditional approaches fail. I will demonstrate that the treatment of intensional operators in natural language, as given in the previous chapters, may also be applied to programming languages, and that in this way a formalized semantics of

assignment statements can be given which deals correctly with the more complex cases as well. Hence we will use the same logic: intensional logic (see chapter 3). The idea to use this logic goes back to JANSSEN & Van EMDE BOAS (1977a,b). We will however, not only use the same logic, but also the same compositional, algebraic framework. In chapter 1 the background of this framework was discussed, and in chapter 2 it was defined formally and compared with the algebraic approach of Adj. For a bibliography of universal algebraic and logical approaches in computer science see ANDREKA & NEMETI 1969. The first sections of the present chapter are a revision of JANSSEN & Van EMDE BOAS (1981).

1.2. Simple assignments

One may think of a computer as a large collection of *cells* each containing a *value* (usually a number). For some of these cells names are available in the programming language. Such names are called *identifiers* or, equivalently, *variables*. The term 'identifier' is mainly used in contexts dealing with syntax, 'variable' in contexts dealing with semantics. The connection of a variable with a cell is fixed at the start of the execution of a program and remains further unchanged. So in this respect a variable does not vary. However, the cell associated with a variable stores a value, and this value may be changed several times during the execution of a program. So in this indirect way a variable can vary. The *assignment statement* is an instruction to change the value stored in a cell.

An example of an assignment statement is: $x := 7$, read as 'x becomes 7'. Execution of this assignment has the effect that the value 7 is placed in the cell associated with x. Let us assume that initially the cells associated with x, y and w contain the values 1, 2 and 4 respectively (figure 1a). The execution of $x := 7$ results in the situation shown in figure 1b. Execution of $y := x$ has the effect that the value stored in the cell associated with x is copied in the cell associated with y (figure 1c). The assignment $w := w + 1$ applied in turn to this situation, has the effect that the value associated with w is increased by one (figure 1d).

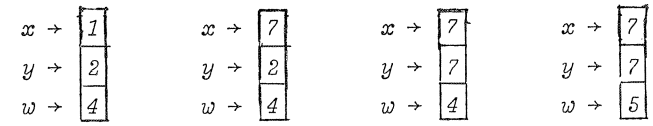


Figure 1a Initial Situation — Figure 1b After $x := 7$ — Figure 1c After $y := 7$ — Figure 1d After $w := w + 1$

Now the necessary preparations are made for demonstrating the relation with natural language phenomena. Suppose that we are in a situation where the identifiers x and y are both associated with value 7. Consider now the assignment

(1) $x := y + 1$.

The effect of (1) is that the value associated with x becomes 8. Now replace identifier y in (1) by x:

(2) $x := x + 1$.

Again, the effect is that the value associated with x becomes 8. So an identifier on the right hand side of ':=' may be replaced by another which is associated with an equal value, without changing the effect of the assignment. One may even replace the identifier by (a notation for) its value:

(3) $x := 7 + 1$.

Replacing an identifier on the left hand side of ':=' has more drastic consequences. Replacing x by y in (1) yields:

(4) $y := y + 1$.

The value of y is increased by one, whereas the value associated with x remains unchanged. Assignment (1), on the other hand, had the effect of increasing the value of x by one; likewise both (2) and (3). So on the left hand side the replacement of one identifier by another having the same value is not allowed. While (2) and (3) are in a certain sense equivalent with (1), assignment (4) certainly is not. Identifiers (variables) behave differently on the two sides of ':='.

It is striking to see the analogy with natural language. I mention an example due to QUINE (1960). Suppose that, perhaps as result of a recent appointment, it holds that

(5) *the dean = the chairman of the hospital board.*

Consider now the following sentence:

(6) *The commissioner is looking for the chairman of the hospital board.*

The meaning of (6) would not be essentially changed if we replaced *the commissioner* by another identification of the same person; a thus changed sentence would be true in the same situations as the original sentence. But consider now (7).

(7) *The commissioner is looking for the dean.*

Changing (6) into (7) does make a difference: it is conceivable that the commissioner affirms (6) and simultaneously denies (7) because of the fact that he has not been informed that (5) recently has become a truth. Sentence (7) is true in other situations than sentence (5). Hence they have a different meaning. In the terminology for substitution phenomena, the subject position of *is looking for* is called (*referentially*) *transparent*, and its object position (*referentially*) *opaque* or *intensional position*. Because of the close analogy, we will use the same terminology for programming languages, and call the right hand side of the assignment 'transparent', and its left hand side 'opaque' or 'intensional'.

The observation concerning substitutions in assignments statements, as considered above, is not original. It is, for instance, described in TENNENT 1976 and STOY 1977 (where the term 'transparent' is used) and in PRATT 1976 (who used both 'transparent' and 'opaque'). The semantic treatments of these phenomena which have been proposed, are, however, far from ideal, and in fact not suitable for assignments which are less simple than the ones above. The authors just mentioned, like many others, avoid these difficulties by considering a language without the more complex constructions.

1.3. Other assignments

Above we only considered assignments involving cells which contain an integer as value. In this section I will describe two other situations: cells containing an identifier as value (pointers) and rows of cells (arrays).

Some programming languages also allow for handling cells which contain a variable (identifier) as value (e.g. the languages Pascal and Algol-68). Names of such cells are called *pointer identifiers* or equivalently *pointer variables*, shortly *pointers*. The situation that pointer p has the identifier x as its value, is shown in figure 2a. In this situation, p is indirectly related to the value of x, i.e. *7*. The assignment $p := w$ has the effect that the value stored in p's cell becomes w (figure 2b). Thus p is indirectly related to the value of w: the integer *5*. When next the assignment $w := 6$ is executed, the integer value indirectly associated with p becomes *6* (figure 2c). So an assignmet can have consequences for pointers which are not mentioned in the assignment statement itself: the value of the variable associated with the pointer may change.

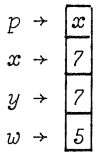


Figure 2a
Initial Situation

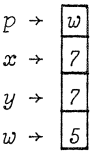


Figure 2b
After $p := w$

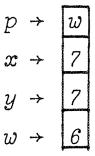


Figure 2c
After $w := 6$

In a real computer, a cell does not contain an integer or a variable, but rather a code for an integer or an code for a variable. For most real computers it is not possible to derive from the contents of a cell, whether it should be interpreted as an integer code or a variable code. In order to prevent the unintended use of an integer code for a variable code, or vice versa, some programming languages (e.g. Pascal) require for each identifier a specification of the kind of values to be stored in the corresponding cells. The syntax of such a programming language then prevents unintended use of an integer code for an identifier code (etc.) by permitting only programs in which each identifier is used for a single kind of value. Other languages leave it to the discretion of the programmer whether to use an identifier for only one kind of value (e.g. Snobol-4). Our examples are from a language of the former type: ALGOL 68.

The programming language ALGOL 68 also allows for higher order pointers, such as pointers to pointers to variables for integer values. They are related to cells which contain as value (the code of) a pointer of the kind described above. These higher order pointers will be treated analogously to the pointers to integer identifiers.

Several programming languages have names for rows of cells (arrays of cells). Names of such rows are called array identifiers, or equivalently array variables. An individual cell can be indicated by attaching a subscript to the array identifier. The element of an array a associated with subscript i is indicated by $a[i]$. The cells of an array contain values of a certain kind: the cells of an integer array contain integers (see figure 3a), and the cells of an array of pointers contain pointers. The execution of the assignment $a[2] := 2$ has the effect that in the cell indicated by $a[2]$ the value 2 is stored (see figure 3b). The subscript may be a complex integer expression. The effect of the assignment $a[a[1]] := 2$ is that the value in $a[1]$ is determined, it is checked whether the value obtained (i.e. 1) is an acceptable index for the array and the assignment $a[1] := 2$

is performed (figure 3c). In the sequel I, will assume that all integers are acceptable indices for subscripts for an array, i.e. that all arrays are of infinite length (of course an unrealistic assumption; but I am interested in formalizing other aspects of arrays). Other kinds of assignment which involve arrays are in the fragment (e.g. the assignment of the whole array in a single action), but I will deal primarlily with assignments of the form just discussed.

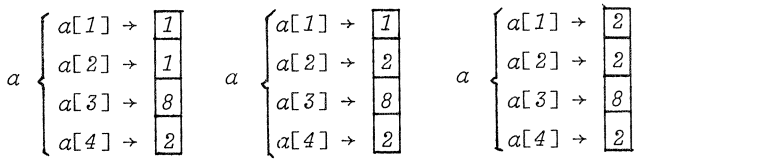


Figure 3a — Initial Situation

Figure 3b — After $a[2] := 2$

Figure 3c — After $a[a[1]] := 2$

2. SEMANTICS OF PROGRAMS

2.1. Why?

Let us consider, as an example, a program which computes solutions of the quadratic equation $ax^2 + bx + c = 0$. The program is based upon the well-known formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad . \tag{8}$$

The program reads as follows:

```
1.  begin real a, b, c, disc, d, x1, x2;
2.  read ((a,b,c));
3.  disc := b*b - 4*a*c;
4.     d := sqrt (disc);
5.    x1 := -b + d; x1 := x2/(2*a);
6.    x2 := -b - d; x2 := x2/(2*a);
7.  print ((a,b,c,x1,x2,newline))
8.  end.
```

The first line of the program says that the identifiers mentioned there, will only be used as names of locations containing real numbers as values (e.g. *3.14159*). The second and seventh line illustrate that the computer

may obtain data from outside (input) and communicate results to the outside world (output). The program also shows that the mathematical formula looks much more compact than the program, but that this compactness is made possible by the use of some conventions which have to be made explicit for the computer. For example, in the program we must write *4*a*c* for *4* times *a* times *c*, while in the formula *4ac* suffices. In the formula we use two dimensional features, which are eliminated in the program (*sqrt*(..) instead of √...). This linear character is necessitated by the fact that programs have to be communicated by way of a sequential channel; for example, the wire connecting the computer with a card reader. The symbol *real* indicates that the identifiers mentioned may only be associated with real values, and the symbols *begin* and *end* indicate the begin and the end of the program.

There exists a considerable confusion among programmers, theoreticians, and designers as to what we should understand by the semantics of a programming language. There are, however, some properties of programs for which there is a measure of agreement on the need for a treatment within the field of semantics. These properties are:

correctness: A program should perform the task it is intended to perform. For example the program given above is incorrect: it does not account for $a = 0$ or $disc < 0$.

equivalence: Two different programs may yield the same results in all circumstances. For example, in the program under discussion we may interchange the order of the computation of *x1* and *x2*, but we cannot compute *d* before we compute *disc*.

termination: If we start the execution of a program, will it ever stop? It might be the case that the computer keeps on trying to find the square root of *-1*, and thus for certain values of *a*, *b* and *c* never halts.

Each of the above properties tells us something about the possible computations the program will perform when provided with input data. We want to predict what may happen in case ...; more specifically, we want to prove that our predictions about the capabilities of the program are correct. How can we achieve this goal? Clearly it is impossible to try out all possible computations of the program, instead one is tempted to run the program on a 'representative' set of input data. This activity is known as program debugging. This way one may discover errors, but one can never *prove* the program to be correct. Still, in practice, most programs used nowadays have been verified only in this way. One might alternatively try to understand

the program simply by reading its text. Again this is not of great help, since mistakes made by the programmer can be remade by the reader. The only way out is the invention of a mathematical theory for proving correctness, equivalence, termination etc.. We need a formalized semantics on which such a theory can be based.

2.2. How?

What does a formal semantics for a program look like? The most common approach is a so-called *operational semantics*. One defines the meaning of a program by first describing some abstract machine (a mathematical model of an idealized computer) and next specifying how the program is to be executed on the abstract machine. Needless to say the problem is transferred in this way from the real world to some idealistic world. The possibly infinitely many computations of the program remain as complex as before. On the other hand, it is by use of an operational semantics that the meaning of most of the existing programming languages is specified. Examples are the programming languages PL/1 in LUCAS & WALK 1971, and, underneath its special description method, ALGOL 68 in Van WIJNGAARDEN 1975.

For about 15 years so-called *denotational semantics* have been provided for programming languages (see e.g. TENNENT 1976, STOY 1977, De BAKKER 1980) of a program is given as a mathematical object in a model; usually some function which describes the input-output behaviour of the program. By abstracting from the intermediate stages of the computation, the model has far less resemblance to a real computer than the abstract machines used in operational semantics. The programs are not considered so much to be transforming values into values, but rather as transforming the entire initial state of a computer into some final state. In this approach, states are highly complex descriptions of all information present in the computer.

Mostly, we are not interested in all aspects of a computer state, but only in a small part (for instance the values of the input and output variables). This leads to a third approach to semantics, which uses so-called *predicate transformers* (FLOYD 1967, HOARE 1969, DIJKSTRA 1974, 1976 and DE BAKKER 1980). A (state) predicate is a proposition about states. So a predicate specifies a set of states: all states for which the proposition holds true. We need to correlate propositions about the state before the execution of the program (*preconditions*) with propositions about the state afterwards (*postconditions*). This is the approach to semantics that we will

follow in the sequel. Usually one distinguishes approaches which associate preconditions and postconditions, but do not consider termination of the execution of the program, and approaches which consider termination as well. The former approaches are said to deal with *partial correctness*, and the latter with *total correctness*. Since all programs we will discuss are terminating programs, the distinction is for our fragment not relevant and will not be mentioned any further.

As an example we consider the program from Section 2.1. An initial state may be described by specifying that on the input channel three numbers a, b and c are present such that $a \neq 0$, and $b^2 - 4ac \geq 0$. The execution of the program will lead such a state to a state where $x1$ and $x2$ contain the solutions to the equation $ax^2 + bx + c = 0$. Conversely, we observe that, if one wants the program to stop in a state where $x1$ and $x2$ represent the solutions of the equation $ax^2 + bx + c = 0$, it suffices to require that the coefficients a, b and c are present on the input channel (in this order!) before the execution of the program, and that moreover $a \neq 0$ and $b^2 - 4ac \geq 0$. In the semantics we will restrict our attention to the real computation, and therefore consider a reduced version of the program from which the input and output instructions and the specifications of the identifiers such as *real* are removed. Let us call this reduced program 'prog'. In presenting the relation between predicates and programs, we follow a notational convention due to HOARE 1969. Let π be a program, and ϕ and ψ predicates expressing properties of states. Then $\{\phi\}\pi\{\psi\}$ means that if we execute π starting in a state where ϕ holds true, and the execution of the program terminates, then predicate ψ holds in the resulting state. Our observations concerning the program are now expressed by:

$$(9) \qquad \{\alpha \neq 0 \wedge (b^2-4ac) \geq 0\} \text{ prog } \{\alpha(x1)^2 + b(x1) + c = 0 \wedge a(x2)^2 + b(x2) + c = 0 \wedge \forall z[az^2 + bz + c = 0 \rightarrow z = x1 \vee z = x2]\}.$$

There are two variants of predicate transformer semantics. The aim of the first variant, the *forward approach* or (*Floyd-approach*) can be described as follows. For any program π, find, according to the structure of π, a predicate transformer which for any state predicate ϕ yields a state predicate ψ, such that if ϕ holds before the execution of π, then ψ gives all information about the final state which can be concluded from ϕ and π. Such a predicate ψ is called a *strongest postcondition* with respect to ϕ and π.

Mathematically a strongest postcondition sp (with respect to ϕ and π) is defined by

(I) $\{\phi\}\ \pi\ \{sp\}$ and

(II) If $\{\phi\}\ \pi\ \{\eta\}$ then from sp we can conclude η.

Suppose that we have two predicates sp_1 and sp_2, both satisfying (I) and (II). Then they are equivalent. From (I) follows that $\{\phi\}\ \pi\ \{sp_1\}$ and $\{\phi\}\ \pi\ \{sp_2\}$. Then from (II) follows that sp_1 implies sp_2 and vice versa. Since all strongest postcondition with respect to ϕ and π, are equivalent, we may speak about *the* strongest postcondition with respect to ϕ and π. For this the notation $sp(\pi,\phi)$ is used.

Instead of this approach, one frequently follows an approach which reverses the process: the *backward-approach* or *Hoare-approach*. For a program π and a predicate ψ one wants to find the weakest predicate which still ensures that, after execution of π, predicate ψ holds. Such a predicate is called a *weakest precondition*. Mathematically a weakest precondition wp (with respect to π and ψ) is defined by

I $\{wp\}\ \pi\ \{\psi\}$

II If $\{\eta\}\ \pi\ \{\psi\}$ then from η we can conclude wp.

Analogously to the proof for postconditions, it can be shown that all weakest preconditions are equivalent. Therefore we may speak about *the* weakest precondition with respect to π and ϕ. For this the notation $wp(\pi,\phi)$ is used (see DIJKSTRA 1974, 1976 for more on this approach).

Above, I used the phrase 'based upon the structure of π'. This was required since it would be useless to have a semantics which attaches to each program and predicate a strongest postcondition in an ad-hoc way, in particular because there are infinitely many programs. One has to use the fact that programs are formed in a structured way according to the syntax of the programming language, and according to our framework, we aim at obtaining these predicate transformers by means of a method which employs this structure.

3. PREDICATE TRANSFORMERS

3.1. Floyd's forward predicate transformer

Below, Floyd's description is given of the strongest postcondition for the assignment statement. But before doing so, I give some suggestive

heuristics. Suppose that $x = 0$ holds before the execution of $x := 1$. Then afterwards $x = 1$ should hold instead of $x = 0$. As a first guess at a generalization one might suppose that always after execution of $v := \delta$ it holds that $v = \delta$. But this is not generally correct, as can be seen from inspection of the assignment $x := x + 1$. One must not confuse the old value of a variable with the new one. To capture this old value versus new-value distinction, the information about the old value is remembered using a variable (in the logical sense!) bound by some existential quantifier and using the operation of substitution. So after $v := \delta$ one should have that v equals 'δ with the old value of v substituted (where necessary) for v in δ'. This paraphrase is expressed by the expression $v = [z/v]\,\delta$, where z stands for the old value of v and $[z/v]$ is the substitution operator. Thus we have obtained information about the final situation from the assignment statement itself. Furthermore we can obtain information from the information we have about the situation before the execution of the assignment. Suppose that ϕ holds true before the execution of the assignment. From the discussion in Section 2 we know that the execution of $v := \delta$ changes only the value of v. All information in ϕ which is independent of v remains true. So after the execution of the assignment $[z/v]\phi$ holds true. If we combine these two sources of information into one formula, we obtain Floyd's *forward predicate transformation rule* for the assignment statement (FLOYD 1967).

(10) $\quad \{\phi\}\ v:=\delta\ \{\exists z[[z/v]\phi \wedge v = [z/v]\delta]\}.$

Here ϕ denotes an assertion on the state of the computer, i.e., the values of the relevant variables in the program before execution of the assignment, and the more complex assertion $\exists z[[z/v]\phi \wedge v = [z/v]\delta]$ describes the situation afterwards.

The examples below illustrate how the assignment rule works in practice.

1) assignment: $x := 1$; precondition: $x = 0$
   obtained postcondition:
   $\exists z[[z/x](x=0) \wedge x = [z/x]1]$, i.e. $\exists z[z=0 \wedge x=1]$, which is equivalent to $x = 1$.
2) assignment: $x := x + 1$; precondition: $x > 0$
   obtained postcondition:
   $\exists z[[z/x](x>0) \wedge x = [z/x](x+1)]$, i.e. $\exists z[z>0 \wedge x=z+1]$, which is equivalent to $x > 1$.

3) Assignment: $a[1] := a[1] + 1$; precondition: $a[1] = a[2]$.
Obtained postcondition:
$\exists z[[z/a[1]](a[1] = a[2]) \wedge a[1] = [z/a[1]](a[1]+1)]$, i.e.
$\exists z[z = a[2] \wedge a[1] = z+1]$, which is equivalent to $a[1] = a[2] + 1$.

3.2. Hoare's backward predicate transformer

Below Hoare's description will be given of the weakest precondition for the assignment statement. First I will give some heuristics. Suppose we want $x = 4$ to hold after the execution of $x := y + 1$. Then it has to be the case that before the execution of the assignment, $y + 1 = 4$ holds. More generally, every statement about x holding after the assignment has to be true about $y + 1$ before its execution. This observation is described in the following rule for the backward predicate transformer (HOARE 1969)

(11) $\quad \{[\delta/v]\phi\}\ v := \delta\ \{\phi\}.$

Some examples illustrate how the rule works in practice.

1) Assignment: $x := 1$; postcondition: $x = 1$.
Obtained precondition:
$[1/x](x=1)$, i.e. $1 = 1$, or *true*.
This result says that for *all* initial states $x = 1$ holds after the execution of the assignment. If the postcondition had been $x = 2$, the obtained precondition would have been $1 = 2$ or *false*, thus formalizing that for *no* initial state does $x = 2$ hold after execution of $x := 1$.
2) Assignment: $x := x + 1$; postcondition $x > 1$.
Obtained precondition:
$[x+1/x](x>1)$, i.e. $x + 1 > 1$ which is equivalent to $x > 0$.
3) Assignment: $a[1] := a[1] + 1$, postcondition $a[1] = a[2] + 1$.
Obtained precondition:
$[a[1] + 1/a[1]](a[1] = (a[2]+1))$, i.e. $a[1] + 1 = a[2] + 1$, which is equivalent with $a[1] = a[2]$.

3.3. Problems with Floyd's rule

Since 1974 it has been noticed by several authors that the assignment rules of Floyd and Hoare lead to incorrect results when applied to cases where the identifier is not directly associated with a cell storing an integer value. Examples are given in Van EMDE BOAS (1974), (thesis 13),

De BAKKER (1976), GRIES (1977), JANSSEN & Van EMDE BOAS (1977a,b). The examples concern assignments involving an identifier of an integer array, or a pointer to an integer identifier. In this section I will consider only examples concerning Floyd's rule.

An example concerning assignment to a subscripted array identifier is

(12) $a[a[1]] := 2$.

Suppose that the assertion which holds before the execution of the assignment is

(13) $a[1] = 1 \wedge a[2] = 1$.

Then Floyd's rule implies that after the execution of the assignment holds

(14) $\exists z[[z/a[a[1]]](a[1]=1 \wedge a[2]=1) \wedge a[a[1]] = [z/a[a[1]]]2]$

i.e.

(15) $\exists z[a[1] = 1 \wedge a[2] = 1 \wedge a[a[1]] = 2]$

which is equivalent to

(16) $a[1] = 1 \wedge a[2] = 1 \wedge a[a[1]] = 2$.

This formula is a contradiction, whereas the assignment is a correctly terminating action. Compare this result with the situations in figure 3, where this assignment is performed in a situation satisfying the given precondition. Then it is clear that the postcondition should be

(17) $a[1] = 2 \wedge a[2] = 1$.

It turns out that problems also arise in the case of pointers (JANSSEN & Van EMDE BOAS 1977a). An example is the following program consisting of three consecutive assignment statements. The identifier p is a pointer and x an integer variable.

(18) $x := 5$; $p := x$; $x := 6$.

Suppose that we have no information about the state before the execution of this program. This can be expressed by saying that the predicate *true* holds in the initial state. By application of Floyd's rule, we find that after the first assignment $x = 5$ holds (analogously to the first example above). Note that the state presented in figure 2a (Section 1) satisfies this predicate. For the state after the second assignment Floyd's rule yields:

(19) $\exists z[[z/p](x=5) \wedge p = [z/p]x]$

i.e.

(20) $\exists z[x=5 \wedge p=x]$

which is equivalent to

(21) $x = 5 \wedge p = x$.

It is indeed the case that after the second assignment the integer value related with p equals 5 (of figure 2b). According to Floyd's rule, after the third assignment the following is true:

(22) $\exists z[[z/x](x=5 \wedge p=x) \wedge x = [z/x]6]$

i.e.

(23) $\exists z[z = 5 \wedge p = z \wedge x = 6]$.

This formula says that the integer value related with p equals 5. But as the reader may remember from the discussion in Section 2, the integer value related with p is changed as well (figure 2c).

3.4. Predicate transformers as meanings

Floyd's assignment rule is one rule from a collection of proof rules: for each construction of the programming language there is a rule which describes a relation between precondition and post condition. The meaning of a construction is defined in a completely different way. A computer-like model is defined, and the meaning of a statement (e.g. the assignment statement) is described as a certain state-transition function (a function from computer states to computer states). The proof rule corresponding to the construction can be used to prove properties of programs containing this construction. A prime example of this approach is De BAKKER (1980). It is, however, not precisely the approach that I will follow in this chapter.

In the discussion in section 2.2 I have mentioned arguments why predicate transformers are attractive from a semantic viewpoint, and why state-transition function are less attractive. I will give predicate transformers a central position in my treatment: the meaning of a program, and in particular of an assignment statement, will be *defined* by means of a predicate transformer.

In theory I could define the meaning of an assignment by any predicate transformer I would like. But then there is a great danger of loosing contact with the behaviour of computer programs in practice. Therefore I will

give a justification of my choice of the predicate transformers. This will be done by defining a state-transition function that resembles the usual state-transition semantics. Then it will be proven that the defined predicate transformers are correct and yield strongest postconditions (or weakest preconditions). In the light of this connection with practice, it is not surprising that there is a resemblence between Floyd's (Hoare's) predicate transformer and the one I will define. But the formal position of the predicate transformers is essentially different in this approach. Actually, I shall argue that Floyd's (Hoare's) predicate transformer cannot be used for our purposes. The problems with the standard formulation of the transformers are mentioned below; they are solvable by some modifications which will be discussed in the next section. The discussion will be restricted to the Floyd-approach; for the Hoare approach similar remarks apply.

In the Floyd-approach the predicate-transformation rule for the assignment is an axiom in a system of proof rules. It can be considered as an instruction how to change a given predicate into its strongest postcondition. In our approach an assignment statement has to be considered semantically as a predicate transformer. Hence it has to correspond with a single expression which is interpreted in the model as a predicate transformer. This requires that Floyd's rule has to be reformulated into such an expression. This can be done by means of a suitable λ-abstraction. The predicate transformer corresponding with assignment $x := \delta$ will look like (24).

(24) $\lambda\phi\exists z[[z/x]\phi \wedge x = [z/x]\delta]$.

This expression is not quite correct because of an inconsistency in the types of ϕ. The subexpression $[z/x]\phi$ is part of a conjunction. Therefore both $[z/x]\phi$ and ϕ have to denote a truth-value. But in the abstraction $\lambda\phi$ the ϕ is not intended as an abstraction over truth-values (there are only two of them), but as an abstraction over predicates (there are a lot of them). This means that the types of ϕ in (24) are not consistent, so it cannot be the predicate-transformer which we will use.

A second problem is the occurrence of the substitution operator in Floyd's rule (and in (24)). It is an operator which operates on strings of symbols. The operator does not belong to the language of logic and there is no semantic interpretation for it. Hence expressions containing the operator have no interpretation. To say it in the terminology of our framework: expressions like (24) are not a polynomial operator over the logic used. Remember that no logical language has the substitution operator

as one of its operators. Substitution belongs to the meta-language, and is used there to indicate how an expression of the logic has to be changed in order to obtain a certain other expression. Since proof rules and axioms are, by their nature, rules concerning syntactic objects, there is no objection against a substitution operator occurring in a proof rule. But we wish to use predicate transformers to determine meanings. If we would use substitution operators in predicate-transformers, then our transformers would be instructions for formula manipulation, and we would not do semantics. The same observation is made by Tennent with respect to another rule. He stated in a discussion (NEUHOLD 1978, p.69):

Substitution is purely syntactic, function modification semantic.

The third problem can be illustrated by considering the assignment $x := y + 1$. The identifier x is used in the execution of the program in an essentially different way than the identifier y. The y is used to indicate a certain value. The x is used as the name of a cell, and not to indicate a value. This different use corresponds with the semantic difference : in section 1.2 we observed that the left-hand side of the assignment statement is referentially opaque, whereas the right-hand side is transparent. Floyd's rule does not reflect these differences. The rule makes no clear distinction between a name and the value associated with that name. In my opinion this is the main source of the problems with Floyd's rule. Remember that all problems we considered above, arose precisely in those situations where there are several ways available for referring to a certain value in the computer: one may use an identifier or a pointer to that identifier; one may use an array identifier subscripted with an integer, or subscripted with an compound expression referring to the same value.

In the field of semantics of natural languages an approach which identified name and object-referred-to was employed in the beginnings of this century. Ryle epitomizes this feature of these theories in his name for them: 'Fido'-Fido theories! The word 'Fido' means Fido, the dog, which is its meaning (see STEINBERG & JAKOBOVITS 1971, p.7). The approach was abandoned, because it turned out to be too simple for treating the less elementary cases. In view of the analogy of the behaviour of names in natural languages and in programming languages we observed in section 1, it is not too surprising that Floyd's rule is not completely successful either.

4. SEMANTICAL CONSIDERATIONS

4.1. The model

In section 5 the syntax and semantics of a small fragment of a programming language will be presented; in section 7 a larger fragment will be dealt with. The treatment will fit the framework developed in the first chapter. So we will translate the programming language into some logical language, which is interpreted in some model. In the present section the semantical aspects (model, logic) will be discussed which are relevant for the treatment of the first fragment. In sections 6 and 7 this discussion will be continued.

In section 2.1 we observed that the assignment statement creates an intensional context. Therefore it is tempting to try to apply in the field of programming languages the notions developed for intensional phenomena in natural languages. The basic step for such an application is the transfer of the notion 'possible world' to the context of programming languages. It turns out that possible worlds can be interpreted as internal states of the computer. Since this is a rather concrete interpretation, I expect that the ontological objections which are sometimes raised against the use of possible world semantics for natural languages (e.g. POTTS 1976), do not apply here. The idea to use a possible world semantics and some kind of modal logic can be found with several authors. An influencing article in this direction was PRATT 1976; for a survey, see Van EMDE BOAS 1978 or PRATT 1980.

An important set in the model is the set of possible worlds, which in the present context will be called set of states. This set will be introduced in the same way as possible worlds were introduced in the treatment of natural languages. It is just some non-empty set (denoted by ST). They are not further analysed; so we do not build explicitly in our semantic domains some abstract model of the computer. But this does not mean that every model for intensional logic is an acceptable candidate for the interpretation of programming languages. Below I will formulate some restrictions on these models, which determine a certain subclass, and these restrictions have, of course consequences for the set ST as well. In this indirect way certain properties of the computer are incorporated in the model. The formulation of the restrictions only concern the simple assignment statement, and they will be generalized in section 7.

An integer identifier is associated with some cell in the computer, and for each state we may ask which value is contained in this cell. The semantic property of an integer identifier we are interested in, is the function which relates a state with the value contained (in that state) in the cell corresponding to that identifier. So we wish to associate with an identifier a function from states to values, see chapter 1 for a discussion (the same idea can be found in ADJ 1977 or 1979). In order to obtain this effect, integer identifiers are translated into constants of type <s,e> (e.g. the identifiers x,y and w are translated into the constants x,y and w of type <s,e>). But something more can be said about their interpretation. The standard interpretation of constants of intensional logic allows that for a given constant we obtain for different states different functions from states to values as interpretation. But we assume that on the computers on which the programs are executed, the relation between an identifier and the corresponding cell is never changed, so that for all states the function associated with an identifier is the same. The interpretations of x,y and w have to be state independent (in chapter 5, section 2 a related situation will arise for natural language; one uses there for such constants the name 'rigid designators'). This requirement implies that not all models for intensional logic are acceptable as candidates for formalizing the meaning of programming languages. We are only interested in those models in which the following postulate holds.

4.1. Rigidness Postulate

Let $c \in \mathrm{CON}_{<s,e>}$ and $v \in \mathrm{VAR}_{<s,e>}$. Then the following formula holds:

$$\exists v \Box [c=v].$$

4.1. END

The above argumentation in favour of the rigidness postulate is not completely compelling. For a fragment containing only simple assignment statements one might alternatively translate integer identifiers into constants of type e which are interpreted non-rigidly. In such an approach the constant relation between an identifier and a cell would not have been formalized. This aspect will, however, become essentail if the fragment is extended with pointers. Although there are no essentially non-rigid constants in the fragment under consideration, it is also possible to consider

such constructs e.g. the integer identifier *xory* which denotes the same as the integer identifier x or the integer identifier y, depending on which of both currently has the greatest integer value. The rigidness postulate guarantees that the interpretation of constants is state independent. Therefore we may replace the usual notation for their interpretation, being $F(c)(s)$, by some notation not mentioning the current state. I will use $V(c)$ as the notation for the interpretation of a constant with respect to an arbitrary state.

Two states which agree in the values of all identifiers should not be distinguishable, since on a real computer such states (should) behave alike. Two states only count as different if they are different with respect to the value of at least one identifier. This is expressed in the following postulate.

4.2. Distinctness Postulate

Let $s,t \in ST$. If for all $c \in CON$, $V(c)(s) = V(c)(t)$, then $s = t$.

4.2. END

The execution of an assignment modifies the state of the computer in a specific way: the value of a single identifier is changed, while the values of all other identifiers are kept intact. This property is expressed by the update postulate, which requires that the model to be rich enough to allow for such a change. The term 'update' should not be interpreted as stating that we change the model in some way; the model is required to have a structure allowing for such a transition of states.

4.3. Update Postulate

For all $s \in ST$, $c \in CON_{<s,e>}$, $n \in \mathbb{N}$ there is a $t \in ST$ such that

$$V(c)(t) = n$$
$$V(c')(t) = V(c')(s) \quad \text{if} \quad c' \neq c.$$

4.3. END

The update postulate requires the existence of a certain new state, and the distinctness postulate guarantees the uniqueness of this new state.

I formulated the update postulate for constants of type <s,e> only, but in section 7 it will be generalized to constants of many other types as well. If the update postulate holds for a constant c and a value d, then the (unique) state required by the postulate is denoted <c←d>s.

Note that the postulates differ from the meaning postulates given for natural languages in the sense that they are formulated in the meta-language and not in intensional logic itself. This allowed us to use quantification over states and over constants in the formulation of the postulates.

One might wish to construct a model which satisfies these three postulates. It turns out that the easiest way is to give the states an internal structure. The rigidness postulate and the distinctness postulate say that we may take for elements of ST sets of functions from (translations of) identifiers to integers. The update postulate says that ST has to be a sufficiently large set. Let ID be the set of integer identifiers. Then we might take $ST = \mathbb{N}^{ID}$. Another possibility (suggested by J. Zucker) is $ST = \{s \in \mathbb{N}^{ID} \mid s(x) \neq 0 \text{ for only finitely many } x\}$. Sets of states with a completely different structure are, in principle, possible as well.

In the introduction I have said that the set of states (set of possible worlds) is just some set. This means that states are, in our approach, a primitive notion and that no internal structure is required for them. But the models just described correspond closely with the models know from the literature (e.g. the one defined by De BAKKER (1980, p.21)); for the larger fragment we will consider this correspondence is less obvious (see section 7). The difference between these two approaches is that here we started with requiring certain properties, whereas usually one starts defining a model. A consequence is that we are only allowed to use the properties we explicitly required, and that we are not allowed to use the accidental properties of a particular model. This is an advantage when a model has to be explicit about a certain aspect, whereas a theory is required to be neutral in this respect. An example could be the way of initialization of identifiers as discussed in De BAKKER (1980, p.218). He says about a certain kind of examples that it: '[..] indicates an overspecification in our semantics [..], it also leads to an incomplete proof theory'. He avoids the problem by eliminating them from his fragment. By means of the present approach such an overspecification could probably avoided.

4.2. The logic

We will use a possible-world semantics for dealing with phenomena of opaque and transparant contexts. Therefore it is tempting to use as logical language the same language as we used in the previous chapters: intensional logic. Since we deal with a programming language, some of the semantic phenomena will differ considerably from the ones we considered before. Intensional logic will be extended with some new operators which allow us to cope with these new phenomena.

The programs deal with numbers, and this induces some changes. The constants of type e ($v_{1,e}, v_{2,e}, \ldots$) will be written in the form *0,1,2,3* .. and interpreted as the corresponding numbers. The logic is extended with operators on numbers: +, x, -, ≤, ≥, =. The symbols *true* and *false* abbreviate *1 = 1* and *1 ≠ 1* respectively. The programming language has an *if-then-else-fi* construction (the *fi* plays the role of a closing bracket; it eliminates syntactic ambiguities). A related construction is introduced in the logic. Its syntax and semantics are as follows:

4.4. DEFINITION. For all $\tau \in Ty$, $\alpha \in ME_t$, $\beta \in ME_\tau$ and $\gamma \in ME_\tau$ we have

$$\textit{if}\ \alpha\ \textit{then}\ \beta\ \textit{else}\ \gamma\ \textit{fi} \in ME_\tau.$$

The interpretation is defined by:

$$V_{s,g}(\textit{if}\ \alpha\ \textit{then}\ \beta\ \textit{else}\ \gamma\ \textit{fi}) = \begin{cases} V_{s,g}(\beta) & \text{if } V_{s,g}(\alpha) = 1 \\ V_{s,g}(\gamma) & \text{otherwise.} \end{cases}$$

4.4. END

The update postulate and the distinctness postulate guarantee for $n \in \mathbb{N}$ and $c \in CON_e$ existence and uniqueness of a state $\langle c \leftarrow n \rangle s$. It is useful to have in the logic an operator which corresponds with the semantic operator $\langle c \leftarrow n \rangle$. These operators, which I will call *state switchers*, are modal operators (since they change the state, (i.e. world) with respect to which its argument is interpreted). The syntax and semantics of state switchers is defined as follows.

4.5. DEFINITION. For all $\sigma,\tau \in CAT$, $\phi \in ME_\sigma$, $c \in CON_{\langle s,\tau \rangle}$, $\alpha \in ME_\tau$ we have

$\{\alpha/^{\vee}c\}\phi \in ME_{\sigma}$.

The interpretation is defined by:

$$V_{s,g}(\{\alpha/^{\vee}c\}\phi) = \begin{cases} V_{<c\leftarrow V_{s,g}(\alpha)>s,g}(\phi) & \text{if } <c\leftarrow V_{s,g}(\alpha)>s \\ & \text{is defined,} \\ V_{s,g}(\phi) & \text{otherwise.} \end{cases}$$

Note that in the present stage of exposition, the 'defined' case only applies for $c \in CON_{<s,e>}$.
4.5. END

One might wonder why the state-switcher contains an extension operator, for only the constant c and the expression α are relevant for determing which state-switcher is intended. The reason is that state-switchers have many properties in common with the well-known substitution operators. The state-switcher determined by c and α behaves almost the same as the substitution operator $[\alpha/^{\vee}c]$. This will be proven in section 4.3.

The meaning of a program will be defined as a predicate transformer. Since we will represent meanings in intensional logic, we have to find a representation of predicate transformers in intensional logic. Let us first consider state-predicates. These are properties of states. For some states the predicate holds, for others it does not hold, so a state predicate is a function f: $S \rightarrow \{0,1\}$. Since the interpretation of intensional logic is state-dependent, such a state predicate can be represented by means of an expression of type t.

A predicate transformer should, in the present approach, not be an operation on expressions, but a semantic function which relates state-predicates with state-predicates. So it should be a function f: $(S\rightarrow\{0,1\}) \rightarrow (S\rightarrow\{0,1\})$. This means that it is a function which yields a truth-value, and which takes two arguments: a state-predicate, and a state. Changing the order of the arguments does not change the function essentially. We may consider a state-predicate as a function which takes a state and a state-predicate, and yields a truth-value. Hence we may say that a predicate transformer is a function f: $S \rightarrow ((S\rightarrow\{0,1\}) \rightarrow \{0,1\})$. This view is, in a certain sense, equivalent to the one we started with. A formula of type <<s,t>,s> has as its meaning such a function, hence formulas of type

<<s,t>,t> are suitable as representations of predicate transformers. Therefore programs and assignments can be translated into expressions of this type.

One might have expected that programs and assignments are translated into expressions of type <<s,t>,<s,t>>. This was the type of the translations of programs and assignments in JANSSEN & Van EMDE BOAS (1977a,b). The first argument for using the type <<s,t>,t> of theoretical nature. An expression of type <<s,t>,<s,t>> has as its meaning a function $f: S \to ((S \to \{0,1\}) \to (S \to \{0,1\}))$, and this is not a predicate transformer (although it is closely connected, and could be used for that purpose). The second argument is of practical nature: the type of the present translation gives rise to less occurrences of the $^\wedge$ and $^\vee$ signs.

A consequence of the representations which we use for (state-)predicates and predicate transformers is the following. Suppose that program π is translated into predicate transformer π', and that this program is executed in a state which satisfies predicate ϕ. Then in the resulting state the predicate denoted by $\pi'(^\wedge\phi)$ holds; it is intended as the strongest condition with respect to program π and predicate ϕ (i.e. $sp(\pi,\phi)$).

4.3. Theorems

The substitution theorem says that the state-switcher behaves almost the same as the ordinary substitution operator. The iteration theorem describes a property of the iteration of state-switchers.

4.6. SUBSTITUTION THEOREM. *The following equalities hold with respect to all variable assignments and states.*

1. $\{a/^\vee c\}c' = c'$ *for all* $c' \in$ CON.
2. $\{\alpha/^\vee c\}v = v$ *for all* $v \in$ VAR.
3. $\{\alpha/^\vee c\}(\phi\wedge\psi) = \{\alpha/^\vee c\}\phi \wedge \{\alpha/^\vee c\}\psi$
   analogously for $\vee,\to,\leftrightarrow,\neg$, *if-then-else-fi constructs.*
4. $\{\alpha/^\vee c\}(\exists x\phi) = \exists x\{\alpha/^\vee c\}\phi$ *if x does not occur free in* α
   analogously for $\forall x\phi$, $\lambda x\phi$.
5. $\{\alpha/^\vee c\}(\beta(\gamma)) = [\{\alpha/^\vee c\}\beta](\{\alpha/^\vee c\}\gamma)$.
6. $\{\alpha/^\vee c\}^\wedge\beta = {}^\wedge\beta$ *analogously for* $\Box\beta$.
7. $\{\alpha/^\vee c\}^\vee c = \alpha$.

Consequence

The state switcher $\{\alpha/^{\vee}c\}$ behaves as the substitution operator $[\alpha/^{\vee}c]$, except if applied to $^{\wedge}\beta, \Box\beta$ or $^{\vee}\beta$ (where $\beta \not\equiv c$). The formulas $\{\alpha/^{\vee}c\}^{\wedge}\beta$ and $\{\alpha/^{\vee}c\}\Box\beta$ reduce to $^{\wedge}\beta$ and $\Box\beta$ respectively, whereas $\{\alpha/^{\vee}c\}^{\vee}\beta$ cannot be reduced any further.

PROOF. Let t be the state $<c\leftarrow V_{s,g}(\alpha)>s$, so $V_{s,g}\{\alpha/^{\vee}c\}\phi = V_{t,g}\phi$.

1. $V_{s,g}(\{\alpha/^{\vee}c\}c') = V_{t,g}(c') = V_{s,g}(c')$.
   The equalities hold because of the Rigidness Postulate.
2. $V_{s,g}\{\alpha/^{\vee}c\}(v) = V_{t,g}(v) = g(v) = V_{s,g}(v)$.
3. $V_{s,g}\{\alpha/^{\vee}c\}(\phi\wedge\psi) = 1 \iff V_{t,g}(\phi\wedge\psi) = 1 \iff V_{t,g}(\phi) = 1$ and $V_{t,g}(\psi) = 1 \iff V_{s,g}(\{\alpha/^{\vee}c\}\phi) = 1$ and $V_{s,g}(\{\alpha/^{\vee}c\}\psi) = 1 \iff V_{s,g}(\{\alpha/^{\vee}c\}\phi \wedge \{\alpha/^{\vee}c\}\psi) = 1$.
   Analogously for the other connectives.
4. $V_{s,g}(\{\alpha/^{\vee}c\}\exists x\phi) = 1 \iff V_{t,g}(\exists x\phi) = 1 \iff$ there is a $g' \sim_x g$ such that $V_{t,g'}(\phi) = 1 \iff$ {x not free in α!} $\iff$ there is a $g' \sim_x g$ such that $V_{s,g}(\{\alpha/^{\vee}c\}\phi) = 1 \iff V_{s,g}(\exists x\{\alpha/^{\vee}c\}\phi) = 1$.
5. $V_{s,g}\{\alpha/^{\vee}c\}(\beta(\gamma)) = V_{t,g}(\beta(\gamma)) = V_{t,g}(\beta)(V_{t,g}(\gamma)) = V_{s,g}(\{\alpha/^{\vee}c\}\beta)(V_{s,g}\{\alpha/^{\vee}c\}\gamma) = V_{s,g}(\{\alpha/^{\vee}c\}\beta(\{\alpha/^{\vee}c\}\gamma))$.
6. $V_{s,g}(\{\alpha/^{\vee}c\}^{\wedge}\beta) = V_{t,g}(^{\wedge}\beta) = \underline{\lambda}t'\ V_{t',g}(\beta) = V_{s,g}(^{\wedge}\beta)$.
7. $V_{s,g}(\{\alpha/^{\vee}c\}^{\vee}c) = V_{t,g}(^{\vee}c) = V(c)(<c\leftarrow V_{s,g}(\alpha)>s) = V_{s,g}(\alpha)$.

4.7. ITERATION THEOREM.

$\{\alpha_1/^{\vee}c\}(\{\alpha_2/^{\vee}c\}\phi) = \{\{\alpha_1/^{\vee}c\}\alpha_2/^{\vee}c\}(\phi)$.

PROOF. Note that also here the state switcher behaves as a substitution operator: first a substitution of α_2 for all occurrences of $^{\vee}c$, and next a substitution of α_1 for the new occurrences of $^{\vee}c$, is equivalent with an immedaite substitution of $[\alpha_1/^{\vee}c]\alpha_2$ for all occurrences of $^{\vee}c$. The proof of the theorem is as follows.

First consider $<c\leftarrow d_1>(<c\leftarrow d_2>)s$, where d_1 and d_2 are possible values of c. This denotes a state in which all identifiers have the same value as in s, except for c which has value d_1. So it is the same state as $<c\leftarrow d_1>s$

(due to the distinctness postulate). This equivalence is used in the proof below.

$$V_{s,g}\{\alpha_1/^{\vee}c\}(\{\alpha_2/^{\vee}c\}\phi) = V_{<c\leftarrow V_{s,g}(\alpha_1)>s,g}\{\alpha_2/^{\vee}c\}(\phi) =$$

$$V_{<c\leftarrow V_{<c\leftarrow V_{s,g}(\alpha_1)>s,g}(\alpha_2)>(<c\leftarrow V_{s,g}(\alpha_1)>)s,g}(\phi) =$$

$$V_{<c\leftarrow V_{<c\leftarrow V_{s,g}(\alpha_1)>s,g}(\alpha_2)>s,g}(\phi) =$$

$$V_{<c\leftarrow V_{s,g}(\{\alpha_1/^{\vee}c\}\alpha_2)>s,g}(\phi) = V_{s,g}\{\{\alpha_1/^{\vee}c\}\alpha_2/^{\vee}c\}(\phi)$$

4.7. END

5. FIRST FRAGMENT

5.1. The rules

In this section the syntax and semantics will be presented of a small fragment of a programming language. The fragment contains only programs which consist of a sequence of simple assignment statements; many programming languages have a fragment like the one presented here. The treatment will be in accordance with the framework developed in the first chapters of this book. This means that for each basic expression (generator of the syntactic algebra) there has to be a translation into the logic, and that for each syntactic rule there has to be a corresponding semantic rule which says how the translations of the parts of a syntactic construction have to be combined in order to obtain the meaning of the compound construction.

The syntax of the fragment has the following five categories:

1. INT The set of representations of integers. Basic expressions in this category are: *1,2,3,...,12,...,666,...* .
2. ID The set of integer identifiers. Basic expressions are x,y and z.
3. ASS The set of assignments.
4. PROG The set of programs.
5. BOOL The set of boolean expressions.

The basic expressions of the category INT translate into corresponding constants of type e; the translation of *1* is *1* etc. The identifiers x,y and w translate into corresponding constants of type <s,e>: the translation of x is x.

The syntactic rules are presented in the same way as in previous chapters. In the clause called 'rule', the categories involved are mentioned;

first the categories of the input expressions, then the category of the resulting expression. The F-clause describes the operation which is performed on the input expressions; here α always stands for the first input expression, β for the second, and γ for the third. The T-clause describes how the translation of the resulting expression is built up from the translations of the input expressions. Here α' denotes the translation of the first input expression, β' of the second, and γ' of the third.

Rule S_{1a}: INT × INT → BOOL
F_{1a}: $\alpha = \beta$
T_{1a}: $\alpha' = \beta'$.

Example S_{1a}: Out of the integer expressions *1* and *2*, we may build the boolean expression *1* < *2*, with as translation *1* < *2*.

Rules $S_{1b}..S_{1e}$: Analogously for the relations >, ≤, ≥, =.

Rule S_{2a}: INT × INT → INT
F_{2a}: $\alpha + \beta$
T_{2a}: $\alpha' + \beta'$

Example : *(1+2)' = 1 + 2*

Rules S_{2b}, S_{2c}: Analogously for the operations × and ÷

Rule S_3 : ID → INT
F_3 : α
T_3 : $^{\vee}\alpha'$

Example : The integer identifier *x* can be used to denote an integer.

Rule S_4 : ID × INT → ASS
F_4 : $\alpha := \beta$
T_4 : $\lambda P[\exists z[\{z/^{\vee}\alpha'\}^{\vee}P \wedge {}^{\vee}\alpha' = \{z/^{\vee}\alpha'\}\beta']]$ $(z \in VAR_e)$

Example : See below. Notice the similarity and differences between this predicate transformer and Floyd's original rule. Some extension operators have been added, and the substitution operator is replaced by an operator with a semantical interpretation.

Rule S_5 : ASS → PROG
F_5 : α
T_5 : α'

Example : Every assignment statement can be used as a (reduced) program.

Rule S_6 : PROG × PROG → PROG
F_6 : α;β
T_6 : $\lambda P[\alpha'(^{\wedge}\beta'(P))]$.

Rule S_7 : BOOL × PROG × PROG → PROG
F_7 : *if* α *then* β *else* γ *fi*
T_7 : $\lambda P[\beta'^{\wedge}(\alpha' \wedge {}^{\vee}P) \vee \gamma'^{\wedge}(\neg\alpha' \wedge {}^{\vee}P)]$

5.2. Examples

5.1. EXAMPLE: $x := y$.

The derivational history of this assignment is presented in figure 4. Also the successive steps of the translation process are presented in the tree. At each stage the number of the rule used and the category of the produced expression are mentioned between braces.

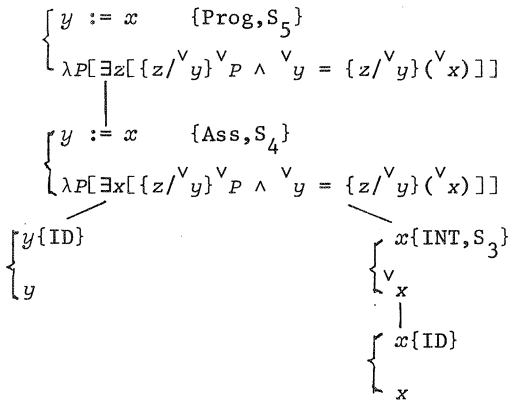


Figure 4. $y := x$

The obtained translation of the program can be reduced, using the substitution theorem, to (25)

(25) $\lambda P[\exists z[\{z/{}^{\vee}y\}{}^{\vee}P \wedge {}^{\vee}y = {}^{\vee}x]]$.

Now suppose that before the execution of the assignment x equals *7* and y equals *2* (cf. Section 1, Figure 2c). So the initial state satisfies predicate (26):

(26) ${}^{\vee}x = 7 \wedge {}^{\vee}y = 2$.

Then after the execution of the assignment the following holds:

(27) $\lambda P[\exists z[\{z/^{\vee}y\}{}^{\vee}P \wedge {}^{\vee}y={}^{\vee}x]](\hat{}[{}^{\vee}x=7 \wedge {}^{\vee}y=2])$.

This reduces to (28), and further to (29) and (30).

(28) $\exists z[\{z/^{\vee}y\}({}^{\vee}x=7 \wedge {}^{\vee}y=2) \wedge {}^{\vee}y={}^{\vee}x]$

(29) $\exists z[{}^{\vee}x=7 \wedge z=2 \wedge {}^{\vee}y={}^{\vee}x]$

(30) ${}^{\vee}x=7 \wedge {}^{\vee}y={}^{\vee}x$.

5.2. EXAMPLE: $y := x$; $y := y + 1$.

The translation of the second assignment statement is obtained in the same way as the translation of $y := x$ in example 5.1. Its translation is (31), which reduces to (32).

(31) $\lambda P\exists z[\{z/^{\vee}y\}{}^{\vee}P \wedge {}^{\vee}y = \{z/^{\vee}y\}({}^{\vee}y+1)]$

(32) $\lambda P\exists z[\{z/^{\vee}y\}{}^{\vee}P \wedge {}^{\vee}y = z+1]$.

The translation of the whole program is therefore

$\lambda Q[y:=y+1]'(\hat{}[[y:=x]'(Q)]) =$

$\lambda Q\lambda P[\exists z[\{z/^{\vee}y\}{}^{\vee}P \wedge {}^{\vee}y = z+1]](\hat{}\exists z[\{z/^{\vee}y\}{}^{\vee}Q \wedge {}^{\vee}y={}^{\vee}x]) =$

$\lambda Q\exists z[\{z/^{\vee}y\}(\exists w[\{w/^{\vee}y\}{}^{\vee}Q \wedge {}^{\vee}y={}^{\vee}x]) \wedge {}^{\vee}y = z+1] =$

$\lambda Q\exists z\exists w[\{\{z/^{\vee}y\}w/^{\vee}y\}{}^{\vee}Q \wedge z={}^{\vee}x \wedge {}^{\vee}y = z+1] =$

$\lambda Q\exists z\exists w[\{w/^{\vee}y\}{}^{\vee}Q \wedge z={}^{\vee}x \wedge {}^{\vee}y = z+1]$.

Suppose now that before the execution of the program $x > 0$ holds. Then afterwards (33) holds, which reduces in turn to (34) and further to (35).

(33) $\exists z\exists w[\{w/^{\vee}y\}({}^{\vee}x>0) \wedge z={}^{\vee}x \wedge {}^{\vee}y=z+1]$

(34) $\exists z[{}^{\vee}x>0 \wedge z={}^{\vee}x \wedge {}^{\vee}y=z+1]$

(35) ${}^{\vee}x>0 \wedge {}^{\vee}y={}^{\vee}x+1$.

In the treatment of this program we first determined the translation of the program, and then considered some specific precondition. If we knew the precondition beforehand, and were only interested in obtaining the postcondition (and not in obtaining the translation of the whole program), we could first calculate the postcondition after the first assignment. This postcondition could then be taken as precondition for the second assignment.

5.3. EXAMPLE: *if* $y < 0$ *then* $y := x$ *else* $y := y+1$ *fi*.

The predicate transformer corresponding with this program is

(36) $\lambda Q[\lambda P[\exists z\{z/^{V}y\}^{V}P \wedge {}^{V}y={}^{V}x]\wedge({}^{V}y<0 \wedge {}^{V}Q) \vee \lambda P[\exists z\{z/^{V}y\}^{V}P \wedge$

$\wedge {}^{V}y=z+1]\wedge(\neg[{}^{V}y<0] \wedge {}^{V}Q)]$.

This reduces to

(37) $\lambda Q[\exists z[z<0 \wedge \{z/^{V}y\}^{V}Q \wedge {}^{V}y={}^{V}x] \vee \exists z[\neg[z<0]\wedge\{z/^{V}y\}^{V}Q \wedge {}^{V}y=z+1]$.

Suppose that we have no information about the state before the execution of the assignment. This is expressed by the precondition $1=1$. Then afterwards (38) holds, which reduces to (39).

(38) $\exists z[z<0 \wedge {}^{V}y={}^{V}x] \vee \exists z[\neg[z<0] \wedge {}^{V}y=z+1]$

(39) ${}^{V}y={}^{V}x \vee {}^{V}y \geq 1$.

5.3. END

6. POINTERS AND ARRAYS

6.1. Pointers

An application of Floyd's rule to assignments containing pointers may give rise to problems, see the example in section 3. In sections 4-6 we have developed a compositional, algebraic approach for simple assignments. This algebraic approach can be generalized in a straightforward way to the case of pointers. I will consider at this moment only pointers to integer identifiers; a more general and formal treatment will be given in section 7.

Pointers to integer identifiers are expressions which have as value in a given state some integer identifier. In another state they may have another identifier as value. Therefore we associate with a pointer some function from states to interpretations of integer identifiers. In analogy to the treatment of integer identifiers, this is done by translating the pointer into a rigid constant; so pointer p translates into constants $p \in \mathrm{CON}_{<s,<s,e>>}$. The execution of the assignment $p := y$ has as an effect that the current state is changed in such a way that in the new state all identifiers have the same value as before, except for p which now has value y. This effect can be described by means of a state switcher like the ones we introduced in relation with simple assignments. Below I will

introduce some postulates which guarantee that the state switchers $\{z/{}^{\vee}p\}$ can be interpreted in the way we intend, and satisfies equalities analogous to the substitution theorem (4.6). But first an example. We assume that the predicate transformer corresponding with the assignment $p := \delta$ reads:

(40) $\lambda P[\exists z[\{z/{}^{\vee}p\}{}^{\vee}P \wedge {}^{\vee}p = \{z/{}^{\vee}P\}\delta]] \qquad (z \in \mathrm{VAR}_{<s,e>})$.

6.1. EXAMPLE. $x := 5$; $p := x$; $x := 6$.
Let us assume that this program is executed in an arbitrary state, so the precondition is *true*. We are interested in the postcondition after the last assignment. That postcondition is obtained by calculating the postcondition of each assignment in turn, and taking that postcondition as input for the predicate transformer of the next assignment. The postcondition of the first assignment for precondition *true* reduces as follows.

$$\lambda P[\exists z[\{z/{}^{\vee}x\}{}^{\vee}P \wedge {}^{\vee}x=5]]({}^{\wedge}\mathit{true}) = \exists z[{}^{\vee}x=5] = {}^{\vee}x=5.$$

The postcondition of the second assignment (using the predicate transformer described above) reduces as follows

$$\lambda P[\exists z[\{z/{}^{\vee}p\}{}^{\vee}P \vee {}^{\vee}p=x]({}^{\wedge}[{}^{\vee}x=5]) = \exists z[\{z/{}^{\vee}p\}({}^{\vee}x=5) \wedge$$
$$\wedge {}^{\vee}p=x] = [{}^{\vee}x=5 \wedge {}^{\vee}p=x].$$

Finally, the postcondition of the last assignment reduces as follows:

$$\lambda P\exists z[\{z/{}^{\vee}x\}{}^{\vee}P \wedge {}^{\vee}x=6]({}^{\wedge}[{}^{\vee}x=5 \wedge {}^{\vee}p=x]) = \exists z[\{z/{}^{\vee}x\}({}^{\vee}x=5 \wedge {}^{\vee}p=x) \wedge {}^{\vee}x=6] =$$
$$[\exists z[z=5 \wedge {}^{\vee}p=x] \wedge {}^{\vee}x=6] = [{}^{\vee}p=x \wedge {}^{\vee}x=6].$$

From this formule follows ${}^{\vee\vee}p=6$, so the postcondition has as consequence that the integer value related with p is 6. This is as it should be (see figure 2).

If we compare the treatment of this program with the treatment using Floyd's rule see (18)-(23), then we observe that this success is due to a careful distinction between the representation of the interpretation of identifier x, namely x, and the representation of the *value* of that identifier, namely ${}^{\vee}x$. This has as its effect that in the calculation of the last postcondition the x in the identity $p=x$ is not replaced by z as would be the case if Floyd's rule were used.
6.1. END

For the constants which translate pointers, we have postulates analogous to the ones we have for constants translating integer identifiers (rigidness

postulate, distinctness postulate, update postulate). Something more, however, has to be said about the possible values of pointer constants. Consider $p \in CON_{<s,<s,e>>}$. This constant is interpreted as a function from states to objects of type <s,e>. Not all such objects are possible values of pointers. In a given state the extension of p has to be the interpretation of some integer identifier and we have already formulated some requirements concerning such interpretations (update postulate etc.). For instance, the interpretation of an integer identifier cannot be a constant function yielding for all states the same value. Consequently the extension of p cannot be such an object. Thus we arrive at the following postulate concerning the constants of type <s,<s,e>> (for higher order pointers analogous requirements will be given).

6.2. Properness postulate

For all $c \in CON_{<s,<s,e>>}$, $s \in ST$

$$V(c)(s) \in \{V(c') \mid c' \in CON_{<s,e>}\}.$$

6.2. END

6.2. Arrays

In section 3 it was shown that a straightforward application of Floyd's rule to assignments containing subscripted array identifiers may yield incorrect results. Here a compositional treatment of the semantics of such assignments will be developed (the formal treatment will be given in 7). In order to have a comparision for the treatment, I will first sketch a treatment due to De BAKKER (1976, 1980).

De BAKKER presents an extension of Floyd's proof rule for the case of assignment statements. His treatment is based on the definition of a new kind of substitution operator $[\alpha/\beta]$. In most cases this operator behaves as the ordinary substitution operator, but not in the case that both α and β are of the form array-identifier-with-subscript. Then this substitution may result in a compound expression containing an *if-then-else* construction. The relevant clause of the definition of the operator is as follows.

(41) $\begin{cases} [t/a[s_1]](b[s_2]) = b[[t/a[s_1]](s_2)] \\ [t/a[s_1]](a[s_2]) = \underline{if}\ [t/a[s_1]]s_2 = s_1\ \underline{then}\ t\ \underline{else}\ a[[t/a[s_1]]s_2]. \end{cases}$

Using this operator, De Bakker gives a variant of Floyd's rule for assignment statements:

(42) $\{\phi\}\ a[s] := t\ \{\exists y \exists z[[y/a[z]](\phi) \wedge z = [y/a[z]](s) \wedge a[z] = [y/a[z]](t)]\}.$

6.3. EXAMPLE.

Assignment: $a[a[1]] := 2$. Precondition: $a[1] = 1 \wedge a[2] = 1$.

Postcondition:

$\exists y \exists z[[y/a[z]](a[1] = 1 \wedge a[2] = 1) \wedge z = [y/a[z]](a[1]) \wedge a[z] = [y/a[z]](2)]$.

By the definition of substitution this reduces to

$\exists y \exists z[\underline{if}\ 1=z\ \underline{then}\ y\ \underline{else}\ a[1]\ \underline{fi} = 1 \wedge \underline{if}\ 2=z\ \underline{then}\ y\ \underline{else}\ a[2]\ \underline{fi} = 1 \wedge z = \underline{if}\ 1=z\ \underline{then}\ y\ \underline{else}\ a[1]\ \underline{fi} \wedge a[z]=2]$.

From the second and the third boolean expression in the conjunction, we see that we must take $z=1$, and the whole expression reduces to:

$\exists y[y=1 \wedge a[2]=1 \wedge 1=y \wedge a[1]=2]$.

This is in turn equivalent to $a[1]=2 \wedge a[2]=1$, from which it follows that $a[a[1]]=2$.

This proof rule works correctly! It is not easy to understand why the rule works, but De Bakker has proven its correctness.

6.3. END

From our methodological point of view this solution has the same disadvantages as Floyd's original proposal, the main one being that the substitution operator defined in (41) has no semantic interpretation. In order to obtain a solution within the limits of our framework, let us consider the 'parts' of the assignment $a[s] := t$. The usual syntax says that there are two parts: the left hand side, (i.e. $a[s]$), and the right hand side (i.e. t). The left hand side has as its parts an array identifier (i.e. a) and an integer expression (i.e. s). This analysis has as a consequence that, in our algebraic approach, we have to associate with the array identifier a some semantic object. In the papers by De Bakker this is not done, nor is this done by several other authors in the field. One usually employs a model which is an abstract computer model with cells, and it is not possible to associate some cell with a. In our model, on the other hand, it is not difficult to associate some semantic object with a. For each state an array identifier determines a function from integers (subscripts) to

integers (the value contained in the cell with that subscript). In analogy to the treatment of integer identifiers, this relation between an identifier and the associated function, is given by translating the array identifier into a rigid constant of type <s,<e,e,>>.

Using the fact that it makes sense to speak about the value of (the translation of) an array identifier a, we can easily describe the effect of the execution of an assignment $a[\beta] := \gamma$. By this assignment a state is reached in which the value associated with a differs for one argument from its old value. If the old value of a is denoted by z, then the new value of a is, roughly, described by: $\lambda n[\underline{if}\ n{=}\beta\ \underline{then}\ \gamma\ \underline{else}\ z(n)\ \underline{fi}]$. I said 'roughly' since it is not yet expressed, for β and γ, to take here the old value of a. These considerations give rise to the following predicate transformer associated with $a[\beta] := \gamma$:

(43) $\lambda P[\exists z\{z/{}^{\vee}a\}{}^{\vee}P \wedge {}^{\vee}a = \{z/{}^{\vee}a\}(\lambda n[\underline{if}\ n = \beta'\ \underline{then}\ \gamma\ \underline{else}\ {}^{\vee}a[n]\ \underline{fi}])]$.

Notice the direct analogy of this predicate transformer with the predicate transformer for the simple assignment. The correctness of (43) is, I believe, much clearer than of the one given by De Bakker. This perspicuity is due to the fact that we treat the array identifiers as having a meaning. In a model based upon the use of 'cells', such an approach does not come naturally. The main point of the present approach (arrays as functions) is the basis for the treatment of arrays in GRIES 1977. It turned out that already in HOARE & WIRTH 1973 arrays are considered as denoting functions (however not in the context of the problems under discussion).

6.4. EXAMPLE. Consider the assignment $a[a[1]] := 2$, executed in a state in which $a[1] = 1$ and $a[2] = 1$. We wish to find the strongest postcondition in this situation. This is found by application of the predicate transformer (associated with the assignment) to the precondition expressing the mentioned property of the state. In the logical formulas given below I should write $a(1)$ etc., since we interpret a as a function. But in order to keep in mind what we are modelling, I prefer the notation $a[1]$

$$[a[a[1]] := 2]'({}^{\wedge}[[{}^{\vee}a[1]=1 \wedge {}^{\vee}a[2]=1]) =$$
$$= \lambda P[\exists z\{z/{}^{\vee}a\}{}^{\vee}P \wedge {}^{\vee}a = \{z/{}^{\vee}a\}(\lambda n\ \underline{if}\ n={}^{\vee}a[1]\ \underline{then}\ 2\ \underline{else}\ {}^{\vee}a[n]\underline{fi})](\wedge[{}^{\vee}a[1]=1 \wedge {}^{\vee}a[2]=1])$$
$$= \exists z[z[1]=1 \wedge z[2]=1 \wedge {}^{\vee}a = \lambda n\ \underline{if}\ n=z[1]\ \underline{then}\ 2\ \underline{else}\ z[n]\underline{fi}]$$
$$= \exists z[z[1]=1 \wedge z[2]=1 \wedge {}^{\vee}a = \lambda n\ \underline{if}\ n=1\ \underline{then}\ 2\ \underline{else}\ z[n]\underline{fi}].$$

From this postcondition the value of $a[a[1]]$ can be calculated:

$$^{V}[a[^{V}a[1]] = {}^{V}a[\lambda n \ \underline{if}\ n=1\ \underline{then}\ 2\ \underline{else}\ z[n]\ \underline{fi}\ [1]] = {}^{V}a[2] = z[2] = 1.$$

6.4. END

Now that we know which predicate transformer should be used, let us look at how it was obtained. We could have tried to find some translation for the left hand side of the assignment (i.e. for $a[n]$), out of which the predicate transformer could be formed. It turned out to be preferable to use the insights obtained from considerations based on the principle of compositionality. We observed that $a[s] := t$ is a notation for changing the function associated with a. This suggests to consider such an assignment as a three-place syntactic operation which takes as inputs the array identifier, the subscript expression, and the expression at the right hand side of the := sign. In such an approach it is easy to obtain the desired predicate transformer, and therefore this approach will be followed. This shows that semantic considerations may influence the design of the syntactic rules (however, a binary approach to the assignment is not forbidden).

In JANSSEN & Van EMDE BOAS (1977a) assignments to multi-dimensional arrays are treated. Since the proposal given there, is not strictly in accordance with the principle of compositionality, it is not mentioned here. One could incorporate assignments to n-dimensional arrays by introducing a separate rule for each choice of n; then an n-dimensional array is considered as a function of n arguments.

7. SECOND FRAGMENT

7.1. The rules

In this section I will present the syntax and semantics of a fragment of the programming language ALGOL-68 (Van WIJNGAARDEN 1975). The fragment contains integer identifiers, pointers to integer identifiers, pointers to such pointers, etc., so there is in principle an infinite hierarchy of pointers. The fragment also contains arrays of integers, arrays of integer identifiers, arrays of pointers to integer identifiers, etc., so in principle an infinite hierarchy of arrays. In order to deal with such infinite sets, the syntax will contain rule schemata. These schemata are like the hyperrules used in the official ALGOL-68 report (VAN WIJNGAARDEN 1975). Following the framework from ch.1, the semantics of the fragment will be

described by means of a translation into intensional logic. As explained in the previous section, this logic has to be interpreted in a restricted class of models. The models have to satisfy certain postulates; these will be presented in section 7.3. In section 7.4 a model will be constructed that satisfies these postulates.

The names of the categories used in section 5 have to be changed in order to follow the ALGOL-68 terminology. The category of integers will be called 'int id' (i.e. integer identifier) and the category of integer identifiers ID will be called 'ref int id' (i.e. reference to integer identifier). As explained above there will be an infinite set of categories. In the description of a category name we may use the meta notion mode. The possible substitutions for this metanotion are described by the following meta rules;

mode → int
mode → ref mode
mode → row of mode.

These modes correspond with types of intensional logic; this correspondence is formalized by the mapping τ which is defined as follows.

τ(int) = e
τ(bool) = t
τ(ref mode) = <s,τ(mode)>
τ(row of mode) = <e,τ(mode)>.

For each 'mode' there is a category 'mode id' which contains denumerable many expressions: the identifiers of that mode. Examples are:

Category	Typical identifiers
int id	$1,2,3,\ldots,666,\ldots$
ref int id	$x,y,w,x_1,x_2,\ldots$
ref ref int id	$p,q,p_1,p_2,\ldots$
row of int id	$a,a_1,a_2,\ldots$

The rule schemata of the fragment are presented in the same way as the rules presented in section 5. The main difference is that in section 5 we had actual rules, whereas we here have schemata which become actual rules by means of a substitution for mode. Important is that throughout one scheme the same substitution for mode has to be used.

Rule $B_1..B_5$: int exp × int exp → bool exp
$FB_1..FB_5$: $\alpha \ast \beta$ where $\ast$ stands for $<,>,\leq,\geq,$ or $=$.
$TB_1..TB_5$: $\alpha' \ast \beta'$ idem for $\ast$.

Rule $I_1..I_3$: int exp × int exp → int exp
$FI_1..FI_3$: $\alpha \oplus \beta$ where $\oplus$ stands for $+,\times,\div$ respectively
$TI_1..TI_1$: $\alpha' \oplus \beta'$ idem for $\oplus$.

Rule E_1 : mode id → mode unit
FE_1 : α
TE_1 : α'.

Rule E_2 : mode unit → mode exp
FE_2 : α
TE_2 : α'.

Rule E_3 : ref mode exp → mode exp
FE_3 : α
TE_3 : $^{\vee}\alpha'$.

Rule E_4 : bool exp × mode unit × mode unit → mode unit
FE_4 : *if* α *then* β *else* γ *fi*
TE_4 : *if* α' *then* β' *else* γ' *fi*
comment : The rule is defined for units and not for exp's in order to avoid the problems of 'balancing' (see e.g. Van WIJNGAARDEN 1975).

Rule E_5 : ref row of mode unit × int exp → ref mode unit
FE_5 : $\alpha'[\beta']$
TE_5 : $^{\wedge}[^{\vee}\alpha'[\beta']]$.

Rule A_1 : ref mode id × mode exp → ass
FA_1 : $\alpha := \beta$
TA_1 : $\lambda P \exists z[\{z/^{\vee}\alpha'\}^{\wedge}P \wedge {}^{\vee}\alpha' = \{z/^{\vee}\alpha'\}\beta']$ where $z \in VAR_{\tau}(\text{mode})$.

Rule A_2 : ref row of mode id × int exp × mode exp → ass
TA_2 : $\alpha[\beta] := \gamma$
FA_2 : $\lambda P[\exists z\{z/^{\vee}\alpha'\}^{\vee}P \wedge {}^{\vee}\alpha' = \{z/^{\vee}\alpha'\}[\lambda n$ *if* $n=\beta'$ *then* γ' *else* $z[n]$ *fi*$]]$.

Rule P_1 : ass → simple prog
FP_1 : α
TP_1 : α'.

Rule P_2 : simple prog → prog
FP_2 : α
TP_2 : α'.

Rule P_3 : prog × simple prog → prog
FP_3 : $\alpha;\beta$
TP_3 : $\lambda P(\alpha'(\hat{}[\beta'(P)]))$.

Rule P_4 : bool exp × prog × prog → prog
FP_4 : *if* α *then* β *else* γ *fi*
TP_4 : $\lambda P[\beta'(\hat{}[\alpha' \wedge {}^{\vee}P]) \vee \gamma'(\hat{}[\neg\alpha' \wedge {}^{\vee}P])]$.

7.1. EXAMPLE. In section 5 I have given several examples of assignment statements. Therefore now as example a somewhat more complex program

$p := a[1];\ a[1] := 2$ precondition $a[1]=1 \wedge a[2]=2$.

The postcondition after the first assignment is:

$$[p := a[1]]'(\hat{}[{}^{\vee}a[1]=1 \wedge {}^{\vee}a[2]=2]) =$$

$$\lambda P\exists z[\{z/{}^{\vee}p\}{}^{\vee}P \wedge {}^{\vee}p = \{z/{}^{\vee}p\}\hat{}[{}^{\vee}a[1]]](\hat{}[{}^{\vee}a[1]=1 \wedge {}^{\vee}a[2]=2]) =$$

$$[{}^{\vee}a[1]=1 \wedge {}^{\vee}a[2]=2 \wedge {}^{\vee}p = \hat{}[{}^{\vee}a[1]]].$$

Then the postcondition after the second assignment is:

$$\exists z\{z/{}^{\vee}a\}({}^{\vee}\hat{}[{}^{\vee}a[1]=1 \wedge {}^{\vee}a[2]=2 \wedge {}^{\vee}p = \hat{}[{}^{\vee}a[1]]]) \wedge$$

$${}^{\vee}a = \{z/{}^{\vee}a\}[\lambda n \text{ if } n = 1 \text{ then } 2 \text{ else } a[n] \text{ fi}] =$$

$$= \exists z[z[1]=1 \wedge z[2]=2 \wedge {}^{\vee}p = \hat{}[{}^{\vee}a[1]] \wedge$$

$${}^{\vee}a = \lambda n \text{ if } n = 1 \text{ then } 2 \text{ else } z[n] \text{ fi}].$$

From this we conclude that ${}^{\vee\vee}p = {}^{\vee}a[1] = 2$.

7.1. END

7.2. The postulates

In order to formulate the postulates, I will first define the set AT of achievable types. This set consists of the types which are achievable by translating expressions of categories which have a name obtained from

the name-scheme 'mode exp'.

7.2. DEFINITION. The set AT $\subset$ Ty is defined by the following clauses:

1. $e \in AT$
2. If $\tau \in AT$ then $\langle s,\tau\rangle \in AT$ and $\langle e,\tau\rangle \in AT$.

7.2. END

The rigidness postulate says that all constants are rigid designators.

7.3. Rigidness Postulate

For all $\tau \in AT$ and $c \in CON_{\langle s,\tau\rangle}$: $\exists v \Box [c=v]$.

7.3. END

The distinctness postulate says that two states are different only if they give rise to a different extension of some constant.

7.4. Distinctness Postulate

Let $s,t \in ST$. If for all $\tau \in AT$ and $c \in CON_{\langle s,\tau\rangle}$ we have $V(c)(s) = V(c)(t)$, then $s = t$.

7.4. END

The properness postulate says, roughly, that the extension of a constant has to be a value that can be achieved by executing instructions from the programming language. First we define these sets AV_τ of achievable values of type τ.

7.5. DEFINITION. The sets AV_τ ($\tau \in AT$) of *achievable values* of type τ are defined as the smallest sets satisfying the following clauses.

I $AV_e = \mathbb{N}$

II $\{V(c) \mid c \in CON_{\langle s,\tau\rangle}\} \subset AV_{\langle s,\tau\rangle}$

III if $\rho \in AV_{\langle s,\langle e,\tau\rangle\rangle}$ and $n \in \mathbb{N}$ then $\lambda s[[\rho(s)](n)] \in AV_{\langle s,\tau\rangle}$

IV $AV_{\langle e,\tau\rangle} = AV_\tau^{\mathbb{N}}$.

7.6. Properness Postulate

For all $s \in ST, \tau \in AT$, $c \in CON_{\langle s,\tau\rangle}$ we have $V(c)(s) \in AV_\tau$.

7.6. END

The update postulate says that the model should have such a richness that the value of one identifier can be changed into arbitrary achievable value, without changing the values of other identifiers.

7.8. Update Postulate

For all $s \in ST, \tau \in AT$, $c \in CON_{<s,\tau>}$, $d \in AV_\tau$, there is a state $t \in ST$ such that

1. $V(c)(t) = d$
2. $V(c')(t) = V(c')(s)$ for all constants $c' \neq c$.

7.8. END

The update postulate only requires 'updating' to an achievable value. This means that the interpretation of $\{\alpha/^{\vee}c\}$ can be defined as follows.

7.9. DEFINITION.

$$V_{s,g}\{\alpha/^{\vee}c\}\phi = \begin{cases} V_{<c \leftarrow V_{s,g}\alpha>s,g}\,\phi & \text{if } V_{s,g}(\alpha) \text{ is achievable} \\ V_{s,g}\phi & \text{otherwise.} \end{cases}$$

7.9. END

7.3. A model

The postulates concerning the model can be distinguished in two groups. Some of the postulates require a certain richness of the model (the distinctness postulate and the update postulate), other postulates limit this richness (rigidness postulate and properness postulate). I will show that it is possible to steer a course between this Scylla and Charibdis by constructing a model which satisfies all these postulates.

The model will be built from the set of natural numbers and a set of states. This set of states should have a certain richness since the model has to fulfill the update postulate (every constant can take every achievable value). In order to obtain this effect one would like to take as set of states the cartesian product of the sets AV_τ of achievable values of type τ. This method cannot be used since the achievable values themselves are defined using the set of states (clause III of their definition). Therefore we will first introduce a collection of expressions which will turn out to be in a one-one correspondence with the achievable values. The

set of states will be defined as the cartesian product of the sets of these expressions.

7.11. DEFINITION. The sets $AE_\tau (\tau \in AT)$ of achievable value expressions of type τ are defined as the smallest sets satisfying the following clauses:

(1) $\forall\, i \in CON_e$ if $\underline{i} \in AE_e$

(2) $\forall\, c \in CON_{<s,\tau>}$ if $c \in AE_{<s,\tau>}$

(3) $\forall\, i \in AE_e$ and for $\forall\, \rho \in AE_{<s,<e,\tau>>}$: $\underline{\rho[i]} \in AE_{<s,\tau>}$

(4) If for all $n \in \mathbb{N}$: $\phi_n \in AE_\tau$ then $(\phi_n)_{n\in\mathbb{N}} \in AE_{<e,\tau>}$.

7.10. END

Clause (4) introduced infinite sequences of symbols. They arise since we did not formalize the finiteness of arrays. The above definition has as a consequence that corresponding to each achievable value given by the definition of AV, there is an expression in AE.

A model for IL satisfying the postulates is now constructed as follows. We use the sets AE_τ of achievable value denotations and define the set of states by

$$S = \prod_{\tau\in AT} \prod_{CON_{<s,\tau>}} AE_\tau .$$

For $c \in CON_{<s,\tau>}$ we denote the projection on the c-th coordinate of a state s by $\Pi_c(s)$.

Having chosen the set S, the sets D_τ are determined for each type τ. To complete the description of the model we must explain how $V(c)$ is defined for constants. This function is defined simultaneously with a mapping $G: \bigcup_{\tau\in AT} AE_\tau \to \bigcup_{\tau\in AT} AV_\tau$.

(1) $V(i) = G(\underline{i}) = i$ for $\underline{i} \in AE_e$

i.e. a number denotations are mapped onto the integers denoted by them.

(2) $V(c) = G(\underline{c}) = \underline{\lambda}s[G(\Pi_c(s))]$ for $c \in CON_{<s,\tau>}$

(3) $G(\underline{\rho[i]}) = \underline{\lambda}s[G(\rho)(s)[G(\underline{i})]]$ for $\rho \in AE_{<s,<e,\tau>>}$

(4) $G((\phi_n)_{n\in\mathbb{N}}) = \underline{\lambda}n[G(\phi_n)]$ for $(\phi_n)_{n\in\mathbb{N}} \in AE_{<e,\tau>}$.

Clearly the map $G: \bigcup_{\tau\in AT} AE_\tau \to \bigcup_{\tau\in AT} AV_\tau$ in this way becomes a bijection. So all elements in the model which are of an achievable type, are achievable values. Moreover the model satisfies all postulates, due to the definition of the set S.

8. CORRECTNESS AND COMPLETENESS

8.1. State transition semantics

In the previous section the meaning a program is defined, and one might expect that the story ends there. But the kind of meanings (predicate transformers) are far removed from the behaviour of a computer while executing a program. One might ask whether we did not loose the connection with a notion of meaning that is more connected with the behaviour of computers. In order to answer this question another kind of semantics will be considered; one in which the meanings of assignments and programs are defined as mappings from states to states, rather than as predicate transformers. I will call it a state-transition semantics; it is related with the standard denotational semantics.

In order to express such a state transition semantics, we need a language in which states can be represented. In the present context the best choice seems to be Ty2: two sorted type theory (see chapter 3, or GALLIN 1975, for a definition). For our purposes this language is extended with state switchers:

8.1. DEFINITION. If $\tau \in AT$, $c \in CON_{<s,\tau>}$, $\beta \in ME_{\tau}$ and $s \in ME_{s}$, then $<c \leftarrow \beta> s \in ME_{s}$. The interpretation of such an expression is defined by

$V_g(<c \leftarrow \beta> s) = V_{g'}(s)$, where $g' \underset{s}{\sim} g$ and $g'(s)$ is the unique state t, such that $V(c)(t) = V(\beta)$ and $V(c')(t) = V(c')(s)$ if $c' \neq c$, if such a state exists (the update postulate guarantees uniqueness); otherwise $g'(s) = s$.

8.1. END

The state-transition semantics of the fragment is defined by means of providing for a translation into Ty2. The translation function will be denoted as ". For the identifiers the translation into Ty2 is the same as the translation into IL, so $\chi' = \chi''$ for all identifiers χ.

For most of the translation rules into Ty2 the formulation can easily be obtained from the translation rules into IL using the standard formulation of IL in Ty2 (see chapter 3). Therefore I will present here only those rules which are essentially different: the rules concerning assignments and programs.

Rule P_1 : Ass → Simple Prog
FP_1 : α
OP_1 : α".

Rule P_2 : Simple Prog → Prog
FP_2 : α
OP_2 : α".

Rule P_3 : Prog × Simple Prog → Prog
FP_3 : $\alpha;\beta$
OP_3 : $\lambda s[\alpha"(\beta"(s))]$.

Rule P_4 : Bool Exp × Prog × Prog → Prog
EP_4 : *if* α *then* β *else* γ *fi*
OP_4 : $\lambda s[$*if* $\alpha"(s)$ *then* $\beta"$ *else* $\gamma"$ *fi*$]$.

Rule A_1 : Ref mode Id × mode Exp → Ass
FA_1 : $\alpha := \beta$
OA_1 : $\lambda s[<\alpha" \leftarrow \beta">(s)]$.

Rule A_2 : Ref Row of mode Id × Int Exp × mode Exp → Ass
TA_2 : $\alpha[\beta] := \gamma$
OA_2 : $\lambda s[<\alpha" \leftarrow \lambda n$ *if* $n = \beta"$ *then* $\gamma"$ *else* $\alpha"[n]$ *fi*$>(s)]$.

8.2. Strongest postconditions

Our aim is to prove that the predicate transformers we have defined in the previous section, are correct with respect to the operational semantics", and that these predicate transformers give as much information about the final state as possible. The relevant notions are defined as follows.

8.2. DEFINITION. A forward predicate transformer π' is called *correct* with respect to program π if for all state predicates ϕ and all states s.

if $s \models \phi$ then $\pi"(s) \models \pi'(\hat{}\phi)$.

8.3. DEFINITION. A forward predicate transformer π' is called *maximal* with respect to program π if for all pairs of state predicates ϕ,ψ holds:

if for all states s: $s \models \phi$ implies $\pi"(s) \models \psi$,
then $\models \pi'(\hat{}\phi) \rightarrow \psi$.

8.4. THEOREM. *Let π be a program, and π_1 and π_2 be forward predicate transformers which are correct and maximal with respect to π. Then for all ϕ:*

$$\models \pi_1(\hat{}\phi) \leftrightarrow \pi_2(\hat{}\phi).$$

PROOF. Since π_2 is correct we have:

$$\text{if } s \models \phi \text{ then } \pi''(s) \models \pi_2(\hat{}\phi).$$

Since π_1 is maximal, from the above implication follows:

$$\models \pi_1(\hat{}\phi) \rightarrow \pi_2(\hat{}\phi).$$

Analogously we prove

$$\models \pi_2(\hat{}\phi) \rightarrow \pi_1(\hat{}\phi).$$

8.4. END

A consequence of this theorem is that all predicate transformers which are correct and maximal with respect to a certain program yield equivalent postconditions. This justifies the following definition.

8.5. DEFINITION. Let π be a program and ϕ an expression of type t. Now $sp(\pi,\phi)$ is a new expression of type t, called the *strongest postcondition* with respect to π and ϕ. The interpretation of $sp(\pi,\phi)$ is equal to the interpretation of $\pi'(\hat{}\phi)$, where π' is a forward predicate transformer which is correct and maximal with respect to π.

8.5. END

A notion which turns out to be useful for proving properties of predicate transformers is

8.6. DEFINITION. A predicate transformer π' is called *recoverable* with respect to program π if for all states t and state-predicates ϕ

$$\text{if } t \models \pi'(\hat{}\phi) \text{ then there is a state s such that } s \models \phi \text{ and } \pi''(s) = t.$$

8.7. THEOREM. *If π' is recoverable, then π' is maximal.*

PROOF. Suppose that π' is recoverable and assume that $s \models \phi$ implies that $\pi''(s) \models \psi$, but that not $\models \pi'(\hat{}\phi) \rightarrow \psi$. Then there is a state t such that $t \models \pi(\hat{}\phi)$ and $t \models \neg\psi$. Since π' is recoverable there is a state s such that $s \models \phi$ and $\pi''(s) = t$. By assumption we also have $\pi''(s) \models \psi$. Contradiction.

8.8. THEOREM. *The translation function ' defined in section 7 yields strongest postconditions.*

PROOF. By induction to the structure of the possible programs. We only consider the case $\chi := \delta$ because for other cases the proof is straightforward.

Part 1: Correctness. Let $s \models \phi$ and $t = \pi''(s)$. Thus $t = \langle \chi \leftarrow \delta'' \rangle s$. We have to prove that

(44) $t \models \exists z[\{z/{}^{V}\chi'\}\phi \wedge {}^{V}\chi' = \{z/{}^{V}\chi'\}\delta']$.

Let h be such that $h(z) = V_s({}^{V}\chi)$. Then for every formula ψ:

(45) $V_{t,h}(\{z/{}^{V}\chi'\}\psi) = V_{\langle \chi' \leftarrow h(z) \rangle t,h}(\psi) = V_{s,h}(\psi)$.

Therefore

(46) $t,h \models \{z/{}^{V}\chi'\}\phi$.

Moreover

(47) $V_{t,h}(\{z/{}^{V}\chi'\}\delta') = V_{\langle \chi \leftarrow h(z) \rangle t,h}\ \delta' = V_{s,h}\ \delta' = V_{t,h}\ \chi'$.

This means that ' is correct.

Part 2: Recoverability. Let

(48) $t \models \exists z[\{z/{}^{V}\chi\}\phi \wedge {}^{V}\chi = \{z/{}^{V}\chi'\}\delta']$.

Thus there is a g such that (49) and (50) hold

(49) $t,g \models \{z/{}^{V}\chi'\}\phi$

(50) $V_t({}^{V}\chi') = V_{t,g}(\{z/{}^{V}\chi'\}\delta')$.

We define $s = \langle \chi \leftarrow g(z) \rangle t$, then we immediately conclude that $s \models \phi$. We prove now that the value of ${}^{V}\chi'$ is the same in $\pi''(s)$ and in t. Since this is the only identifier in which they might differ we conclude that the states are the same (the update postulate guarantees uniqueness!)

(51) $V_{\pi''(s)}\ ({}^{V}\chi') = V_{\langle \chi \leftarrow \delta'' \rangle s}\ ({}^{V}\chi') = V_s(\delta') = V_{\langle \chi \leftarrow g(z) \rangle t}(\delta') =$
$= V_{t,g}\{z/{}^{V}\chi'\}\delta' = V_t({}^{V}\chi')$.

Notice that this proof also holds in case that δ is an λ-expression, or in case g(z) is not achievable. This means that π' is recoverable, hence π' is maximal.

8.8. END

8.3. Completeness

The notions 'completeness' and 'soundness' of a collection proof rules play an important role in the literature concerning the semantics of programming languages. Such collections are intended to be used for proving properties of programs. Our main aim was not to prove properties, but to define meanings. However, in the discussions about our approach the possibility to prove properties of programs played an important role. In the examples several proofs concerning effects of programs were given, and one of the arguments for using predicate transformers was their usefulness in proofs. Therefore it is interesting to consider our approach in the light of the notions 'soundness' and 'completeness'. First I will informally discuss these notions in their relation to the traditional approach (for a survey see APT 1981), there after I will try to transfer them to our approach.

In the traditional approaches one describes the relation between the assertions (state predicates) ϕ and ψ and the program π by means of the correctness formula $\{\phi\}\pi\{\psi\}$. This formula should be read as stating that if ϕ holds before the execution of program π, then ψ holds afterwards (for a discussion see section 2). Formula ϕ is called a *precondition*, and ψ a *postcondition*. collection C of proof rules for such formulas consists of axioms, and of proof rules which allow to derive new formulas from already derived ones. For the basic constructions of the programming language certain formulas are given as axioms (e.g. Floyd's axiom for the assignment statement). An important proof rule is (52); the so called *rule of consequence*. It allows us to replace a precondition by a stronger statement, and a postcondition by a weaker statement.

(52) If $p \rightarrow p_1$, $\{p_1\}S\{q_1\}$, $q_1 \rightarrow q$ are derived, then $\{p\}S\{q\}$ follows.

The notion $\vdash_C$ (derivable in C) is then defined as usual. Hence (53) means that the formula $\{\phi\}\pi\{\psi\}$ can be derived from the axioms by using only rules from C.

(53) $\vdash_C \{\phi\}\pi\{\psi\}$.

Besides the syntactic notion $\vdash_C$, the semantic notion $\models_M$ of satisfaction in a model M is used. A model M is defined, in which assertions ϕ and ψ can be interpreted and in which the execution of π is modelled. Then (54) says that it is true in M that if ϕ holds before the execution of π, then

ψ holds afterwards.

(54) $\models_M \{\phi\}\pi\{\psi\}$.

The notions soundness and completeness of collection C of proof rules relate the syntactic notion $\vdash_C$ with the semantic notion $\models_M$. The collection C is called *sound* if for all ϕ,ψ and π

(55) $\vdash_C \{\phi\}\pi\{\psi\}$ implies $\models_M \{\phi\}\pi\{\psi\}$.

The collection C is called *complete* if for all ϕ,ψ and π

(56) $\models_M \{\phi\}\pi\{\psi\}$ implies $\vdash_C \{\phi\}\pi\{\psi\}$.

Most identifiers in computer programs have to be associated with numbers, and the assertions in correctness formulas may say something about the numerical values of these identifiers. We may consider a trivial program α (e.g. $x := x$) a trivial assertion β (e.g. *true*), and an arbitrary assertion γ from number theory. Then (57) holds.

(57) $\models_M \{\beta\}\alpha\{\gamma\}$ if and only if $\models_M \gamma$.

Suppose now that we had a complete collection C of proof rules for correctness formulas. Then combination of (56) with (57) would learn us that (58) holds

(58) $\vdash_C \{\beta\}\alpha\{\gamma\}$ if and only if $\models_M \gamma$.

Thus a complete proof system for correctness formulas would give us a complete proof system for arithmetic. Since arithmetic is not completely axiomatizable, there cannot be such a complete system C for correctness formulas. Concerning this situation De BAKKER (1980, p.61) says the following:

> '[..] *we want to concentrate on the programming aspects of our language, and* [..] *pay little attention to questions about assertions which do not interact with* [*assignment*] *statements (so that even if an axiomatization of validity were to exist, we might not be interested in using it).*

For this reason De Bakker takes all valid assertions as axioms of C, i.e. if $\models_M \phi$, then by definition $\vdash_C \phi$. This notion of completeness, viz. where certain assertions are taken as axioms, is called *complete in the sense of Cook*. For a formal definition see COOK (1978), or APT (1981). This notion is defined only for logical languages which are *expressive*: languages in which all strongest postconditions can be expressed (for the class of programs under consideration). From the results in 8.2 follows that our extension of IL is expressive.

In order to define the notions 'soundness' and 'completeness' for our approach, we have to find notions that can be compared with $\vdash_C$ and with $\models_M$. First I will consider the syntactic notion $\vdash_C$. In our approach the logical deductions are performed on the level of intensional logic. So if we would introduce a system S of proof rules, it would be proof rules of intensional logic. Hence we have to find an expression of IL which corresponds with (52).

We have characterized (in intensional logic) the meaning of a program π by means of a predicate transformer π', and we have proven that this transformer yields strongest postconditions, i.e. $sp(\pi,\phi) = \pi'(\hat{}\phi)$. Consider now (59)

(59) $\pi'(\hat{}\phi) \rightarrow \psi$.

Formula (59) expresses that if ϕ holds before the execution of π, then ψ holds afterwards. So (59) corresponds with the correctness formula $\{\phi\}\pi\{\psi\}$. An alternative approach would of course be to use the corresponding backward predicate transformer. The discussion below will be restricted to forward predicate transformers; for backward predicate transformers related remarks could be made. Suppose now that we have a system S of proof rules of intensional logic. The notion $\vdash_S$ can be defined as usual. Then (60) says about S the same as (53) says about C. Therefore I will consider (60) as the counterpart of (53).

(60) $\vdash_S \pi'(\hat{}\phi) \rightarrow \psi$.

In section 7 we have defined a class of models. Let $\models$ denote the interpretation in these models. In the light of the above discussion (61) can be considered as the counterpart in our system of (54).

(61) $\models \pi'(\hat{}\phi) \rightarrow \psi$.

A system of proof rules for IL is called *sound* if for all ϕ,ψ and π (62) holds.

(62) $\vdash_S \pi'(\hat{}\phi) \rightarrow \psi$ implies $\models \pi'(\hat{}\phi) \rightarrow \psi$.

A system S of proof rules is called *complete* if for all ϕ,ψ and π (63) holds

(63) $\models \pi'(\hat{}\phi) \rightarrow \psi$ implies $\vdash_S \pi'(\hat{}\phi) \rightarrow \psi$.

We might consider again trivial program α, trivial condition β, and an arbitrary IL formula δ. Then (64) holds

(64) $\models \alpha'(\hat{}\beta) \rightarrow \delta$ if and only if $\models \delta$.

Suppose now that proof system S contains modus ponens. Then (65) holds

(65) $\vdash_S \alpha'(\hat{}\beta) \rightarrow \delta$ if and only if $\vdash_S \delta$.

Suppose moreover that S is complete. Then from (64) and (65) it follows that (66) holds

(66) $\models \delta$ if and only if $\vdash_S \delta$.

Thus a complete system of proof rules would give us a complete axiomatization of IL. Such an axiomatization does not exist (see chapter 3). Hence S cannot be complete either. In this situation we might follow De Bakker, and make the notion of completeness independent of the incompleteness of the logic we use. So we might take all formulas of our extension of IL as axioms. But then S is complete (in the sense of Cook) in a trivial way since all correctness formulas are formulas of our extension of IL.

This completeness result is not very exciting, and one might try to find another notion of completeness. A restriction of the axioms to only arithmetical assertions seems me to be unnatural for the fragment under discussion because our programs do not only deal with natural numbers, but also with pointers of different kinds. From a logical viewpoint it is attractive to try to prove for our extension a kind of generalized completeness (see chapter 3). This would require that Gallin's axiom system for IL (see chapter 3) is extended with rules concerning state-switchers. Thus we might show that a system S is generalized complete, i.e. that it is complete with respect to the formulas which are true in all generalized models. The models defined in section 7 constitute a subclass of the set of generalized models. I do not know any reason to expect that the formulas valid in all models of this subclass are the same as those valid in all generalized models (because our subclass does not contain an important class: the standard models). Hence generalized completeness would be an interesting result that proves a certain degree of completeness, but it would not correspond with the traditional completeness results in computer science. I doubt whether computer scientists would be happy with such a completeness result.

Another concept between 'incomplete' and trivially 'complete', is suggested by Peter van Emde Boas. The formula $\pi'(\hat{}\phi) \rightarrow \psi$ was intended to be the analogue of the Hoare formula $\{\phi\}\pi\{\psi\}$. The language in which we express ϕ and ψ contains state switchers, but in most cases a programmer will be interested in cases were ϕ and ψ are state-switcher free. However, our

analogue of the Hoare formula, viz. $\pi'(\hat{}\phi) \rightarrow \psi$, will always contain a state-switcher introduced by the predicate transformer π'. Now one might hope for a result which says that this state-switcher can always be eliminated. In the examples we described this was indeed the case. There are however situations where no reduction rule is applicable (if values of pointers are involved, where these values are unknown). This makes it unlikely that it will always be possible to eliminate the state-switcher from a formula obtained by application of a predicate transformer to a state-switcher free formula (i.e. such an expressibility result is not te be expected). It would however, be interesting to know whether the reduction formulas are sufficient to eliminate the state-switchers from those translations of Hoare formulas where elimination is possible. This gives the following intermediate concept of 'completeness'.

If ϕ and ψ are state-switcher free and $\models \pi'(\hat{}\phi) = \psi$ then $\vdash_S \pi'(\hat{}\phi) = \psi$.

9. THE BACKWARD APPROACH

9.1. Problems with Hoare's rule

Besides the approach discussed up till now, there is the approach based on backward predicate transformers. In section 3 we have already met Hoare's rule for the assignment statement

(67) $\{[\delta/v]\ \psi\}\ v := \delta\ \{\psi\}$.

Hoare's rule may yield incorrect results when applied to assignment containing pointers or arrays, just as was the case with Floyd's rule. I mention three examples.

De BAKKER (1976) presents for Hoare's rule the following example

(68) $\{[1/a[a[2]]](a[a[2]]=1)\}\ a[a[2]] := 1\ \{a[a[2]]=1\}$.

The precondition in (68) reduces to $1=1$. That would imply that, for any initial state, the execution of $a[a[2]] := 1$ has the effect that afterwards $a[a[2]] = 1$ holds. This is incorrect (consider e.g. an initial state satisfying the equality $a[2]=2 \wedge a[1]=2$).

GRIESS (1977) presents the following example

(69) $\{1=a[j]\}\ a[i] := 1\ \{a[i] = a[j]\}$.

Whereas in example (68) the obtained precondition was too weak, in the present example the obtained precondition is too restrictive. The postcondition holds also in case the initial state satisfies $i=j$.

An example of the failure of Hoare's rule for the treatment of pointers is (JANSSEN & VAN EMDE BOAS 1977b):

(70) $\{x=x+1\}\ p := x;\ \{p=x+1\}\ x := x+1\ \{p=x\}$.

It is impossible to satisfy the precondition mentioned in (70), whereas for any initial state the postcondition will be satisfied.

Besides the objection that (67) gives incorrect results in certain cases, the same more fundamental problems arise as were mentioned in section 3 for Floyd's rule (e.g. the use of textual substitution).

9.2. Backward predicate transformers

Using a state switcher a formulation can be given for the backward predicate transformers which satisfies our algebraic framework. The transformer corresponding to $v := \delta$ is

$$\lambda P[\{\delta'/{}^{V}v\}{}^{V}P].$$

The transformer corresponding with $a[\beta] := \gamma$ is

$$\lambda P[\{\lambda n\ \underline{if}\ n = \beta'\ \underline{then}\ \gamma'\ \underline{else}\ {}^{V}a[n]\ \underline{fi}/{}^{V}a\}{}^{V}P].$$

9.1. EXAMPLE. Assignment $a[i] := 1$; Postcondition ${}^{V}a[i] = {}^{V}a[j]$
Precondition: $\{\lambda n\ \underline{if}\ n = i\ \underline{then}\ 1\ \underline{else}\ {}^{V}a[n]\ \underline{fi}/{}^{V}a\}({}^{V}a[i]={}^{V}a[j])$ reducing to:
$1 = (\underline{if}\ j = i\ \underline{then}\ 1\ \underline{else}\ a[j]\ \underline{fi})[j]$ and
further to: $j = i \vee a[j] = 1$ (compare this with (69)).

9.2. EXAMPLE. Assignment $a[a[2]] := 1$; postcondition ${}^{V}a[{}^{V}a[2]] = 1$.
Precondition: $\{\lambda n\ \underline{if}\ n = {}^{V}a[2]\ \underline{then}\ 1\ \underline{else}\ a[n]\ \underline{fi}/{}^{V}a\}({}^{V}a[{}^{V}a[2]]=1)$.
We have to apply the state switcher to both occurrences of ${}^{V}a$ in the postcondition. If we apply it to ${}^{V}a[2]$ then we obtain

$$\underline{if}\ 2 = {}^{V}a[2]\ \underline{then}\ 1\ \underline{else}\ {}^{V}a[2]\ \underline{fi}.$$

This leads us to consider the following two cases.

I. $2 = {}^{V}a[2]$.
Then the precondition reduces to
$(\{\lambda n\ \underline{if}\ n = a[2]\ \underline{then}\ 1\ \underline{else}\ a[n]\ \underline{fi}/{}^{V}a\}{}^{V}a)[1] = 1$ which reduces to
$a[1] = 1$.

II. $2 \neq {}^{V}a[2]$.
Then the precondition reduces to
$(\{\lambda n\ \underline{if}\ n = a[2]\ \underline{then}\ 1\ \underline{else}\ a[n]\ \underline{fi}/{}^{V}a\}{}^{V}a)[2] = 1$ which reduces to
$1 = 1$.

So the precondition is $({}^{\vee}a[2] = 2 \wedge {}^{\vee}a[1] = 1) \vee ({}^{\vee}a[2] \neq 2)$. (Compare this result with (68)).
9.2. END

9.3. Weakest preconditions

We aim at obtaining backward predicate transformers which yield a result that is correct with respect to the operational semantics ", and which require assumptions as weak as possible about the initial state (i.e. dual to the requirements concerning the forward predicate transformers). The relevant notions are defined as follows.

9.3. DEFINITION. A backward predicate transformer $'\pi$ is called *correct* with respect to a program π if for all state predicates ϕ and all states s

$$\text{if } s \models {}'\pi({}^{\wedge}\phi) \quad \text{then} \quad \pi''(s) \models \phi.$$

9.4. DEFINITION. A backward predicate transformer $'\pi$ is called *minimal* with respect to a program π if for all pairs of state predicates η and ϕ, the following holds:

$$\text{if for all states s: } s \models \eta \text{ implies } \pi''(s) \models \phi,$$
$$\text{then } \models \eta \rightarrow {}'\pi({}^{\wedge}\phi).$$

9.5. THEOREM. *Let π be a program, and π_1 and π_2 be backward predicate transformers which are correct and minimal with respect to π. Then for all ϕ:*

$$\models \pi_1({}^{\wedge}\phi) \leftrightarrow \pi_2({}^{\wedge}\phi).$$

PROOF. Since π_1 is correct, we have:

$$\text{if } s \models \pi_1({}^{\wedge}\phi) \quad \text{then} \quad \pi''(s) \models \phi.$$

Since π_2 is minimal, from this implication follows

$$\models \pi_1({}^{\wedge}\phi) \rightarrow \pi_2({}^{\wedge}\phi).$$

Analogously we prove $\models \pi_2({}^{\wedge}\phi) \rightarrow \pi_1({}^{\wedge}\phi)$.
9.5. END

A consequence of this theorem is that all backward predicate transformers which are correct and minimal with respect to a certain program, yield equivalent preconditions. This justifies the following definition.

9.6. DEFINITION. Let π be a program and ϕ a state predicate. Then $wp(\pi,\phi)$ is a new expression of type t, called the weakest precondition with respect to π and ϕ. The interpretation of $wp(\pi,\phi)$ is equal to the interpretation of $'\pi(^{\wedge}\phi)$, where $'\pi$ is a backward predicate transformer which is correct and minimal with respect to π. If a state predicate is equivalent with $wp(\pi,\phi)$ it is called a weakest precondition with respect to π and ϕ.
9.6. END

In 9.2 backward predicate transformers are defined for the assignment statements. We wish to prove that they yield a weakest precondition. This will not be proven in a direct way because it turns out that backward and forward predicate transformers are closely related. The one can be defined from the other, and correctness and maximality of the forward predicate transformers implicate correctness and minimality of the backward predicate transformers. These results will be proven in the next subsections.

9.4. Strongest and weakest

Strongest postconditions and weakest preconditions can syntactically be defined in terms of each other. This connection is proved in the following theorem.

9.7. THEOREM. *Let* $Q \in \mathrm{VAR}_{<s,t>}$ *and let*

(I) *be* $\exists Q[{}^{\vee}Q \wedge \Box[sp(\pi,{}^{\vee}Q) \rightarrow \phi]]$

and

(II) *be* $\forall Q[\Box[\phi \rightarrow wp(\pi,{}^{\vee}Q)] \rightarrow {}^{\vee}Q]$.

Then it holds that
formula (I) *is equivalent to* $wp(\pi,\phi)$, *and formula* (II) *is equivalent to* $sp(\pi,\phi)$.

PROOF.
part A
I show that (I) is correct (A_1) and minimal (A_2) with respect to ϕ,π and ". From this follows that (I) is equivalent to $wp(\pi,\phi)$.

part A_1
Suppose that s satisfies (I), so

$$s \models \exists Q[{}^{\vee}Q \wedge \Box[sp(\pi,{}^{\vee}Q) \rightarrow \phi]].$$

Then for some g (71) and (72) holds

(71) $s,g \models {}^{\vee}Q$

(72) $s,g \models \Box [sp(\pi,{}^{\vee}Q) \rightarrow \phi]$.

By definition of *sp*, from (71) follows

$\pi''(s),g \models sp(\pi,{}^{\vee}Q)$.

By definition of $\Box$ from (72) follows

$\pi''(s),g \models sp(\pi,{}^{\vee}Q) \rightarrow \phi$.

Therefore

$\pi''(s),g \models \phi$, or equivalently $\pi''(s) \models \phi$.

This means that (I) is correct.

part A_2

Suppose that for all s holds

(73) $s \models \eta$ implies $\pi''(s) \models \phi$.

By definition of *sp* from (73) follows

$\models sp(\pi,\eta) \rightarrow \phi$.

So for all s

$s \models \Box [sp(\pi,\eta) \rightarrow \phi]$.

Let g be an assignment such that $g,s \models Q = {}^{\wedge}\eta$. Then

$s,g \models \Box [sp(\pi,{}^{\vee}Q) \rightarrow \phi]$.

So (for the choice $Q = {}^{\wedge}\eta$)

$s \models \eta \rightarrow \exists Q[{}^{\vee}Q \wedge \Box [sp(\pi,{}^{\vee}Q) \rightarrow \phi]]$.

This means that (II) is minimal.

part B

I show that (II) is correct (B_1) and maximal (B_2) with respect to ϕ,π, and". From this it follows that (II) is equivalent with $sp(\pi,\phi)$.

part B_1

Let

$s \models \phi$.

Suppose that for variable assignment g holds

(74) $g \models \Box [\phi \rightarrow wp(\pi,{}^{\vee}Q)]$.

Then from (74) follows

$$s,g \models wp(\pi, {}^{\vee}Q).$$

So

(75) $\pi''(s),g \models {}^{\vee}Q.$

From (74) and (75) it follows that for all g holds

$$\pi''(s),g \models \Box[\phi \rightarrow wp(\pi, {}^{\vee}Q)] \rightarrow {}^{\vee}Q.$$

So

$$\pi''(s) \models \forall Q[\Box[\phi \rightarrow wp(\pi, {}^{\vee}Q] \rightarrow {}^{\vee}Q].$$

This means that (II) is correct.

part B_2

Suppose that for all s holds

(76) $s \models \phi$ implies $\pi''(s) \models \eta.$

Then, by definition of *wp* it follows that

(77) $\models \phi \rightarrow wp(\pi,\eta).$

Suppose that t satisfies (II), so

(78) $t \models \forall Q[\Box[\phi \rightarrow wp(\pi, {}^{\vee}Q)] \rightarrow {}^{\vee}Q].$

Let g be an assignment such that $g,t \models Q = {}^{\wedge}\eta$. Then:

(79) $t,g \models \Box[\phi \rightarrow wp(\pi, {}^{\vee}Q)] \rightarrow {}^{\vee}Q.$

Then from (77) and (79) follows

(80) $t,g \models {}^{\vee}Q.$

So

(81) $\models \forall Q[[\Box[\phi \rightarrow wp(\pi, {}^{\vee}Q)] \rightarrow {}^{\vee}Q] \rightarrow {}^{\vee}Q].$

This means that II is maximal.

9.7. END

That $wp(\phi,\pi)$ and $sp(\pi,\phi)$ are closely connected is also observed by RAULEFS (1978). He gives a semantic connection. Theorem 9.7 goes further, because an explicit syntactic relation is given.

9.5. Correctness proof

The theorem 9.7 has as a consequence that weakest preconditions and strongest postconditions can be defined in terms of each other. Now it is unlikely that the formulas with quantification over intensions of predicates are the kind of expressions one would like to handle in practice. The importance of the theorem is that given some expression equivalent with $sp(\pi,\phi)$, it allows us to prove that some expression (found on intuitive considerations) is equivalent with $wp(\pi,\phi)$. From the correctness and maximality of the predicate transformers defined in section 5 and 7, it follows that the backward predicate transformers defined in this section are correct and minimal.

9.8. THEOREM. *The following two statements are equivalent*

(I) $sp(\chi := \delta,\ \phi) = \exists z[\{z/^{\vee}\chi\}\phi \wedge {}^{\vee}\chi' = \{z/^{\vee}\chi'\}\delta']$

(II) $wp(\chi := \delta,\ \phi) = \{\delta'/^{\vee}\chi\}\phi$.

PROOF.

part 1: (I) ⇒ (II).

Assume that (I) holds. Then from theorem 9.7 follows:

(82) $\models wp(\chi := \delta,\phi) = \exists Q[{}^{\vee}Q \wedge \Box[\exists z[\{z/^{\vee}\chi'\}{}^{\vee}Q \wedge {}^{\vee}\chi' = \{z/^{\vee}\chi'\}\delta'] \rightarrow \phi]]$.

So we have to prove that for arbitrary assertion ϕ and state s holds that

(83) $s \models \{\delta'/^{\vee}\chi'\}\phi$

if and only if

(84) $s \models \exists Q[{}^{\vee}Q \wedge \Box\, \exists z[\{z/^{\vee}\chi'\}{}^{\vee}Q \wedge {}^{\vee}\chi' = \{z/^{\vee}\chi'\}\delta'] \rightarrow \phi]]$

part 1a: (83) ⇒ (84)

Assume that (83) holds. Let g be a variable assignment such that

(85) $g \models Q = {}^{\wedge}\{\delta'/^{\vee}\chi'\}\psi$.

Then (due to (83)) we have

(86) $s,g \models {}^{\vee}Q$.

In order to prove (84) we have next to prove the necessary validity of the formula mentioned after the □ for this choice of Q. So we have to prove that for arbitrary state t (87) implies (88).

(87) $t \models \exists z[\{z/^{\vee}\chi'\}\{\delta'/^{\vee}\chi'\}\phi \wedge {}^{\vee}\chi' = \{z/^{\vee}\chi'\}\delta']$

(88) $t \models \phi$.

Let h be a variable assignment for which (87) holds. Then using the iteration theorem, we find

(89) $t,h \models \{\{z/{}^{V}\chi'\}\delta'/{}^{V}\chi'\}\phi \wedge {}^{V}\chi' = \{z/{}^{V}\chi'\}\delta'$.

The second conjunct gives us information about the value of ${}^{V}\chi'$ in this state. The state switcher says that we have to interpret ϕ with respect to the state where ${}^{V}\chi'$ precisely has that value. So the state switcher does not change the state! This means that

(90) $t \models \phi$.

So (87) implies (88), and therefore (84) holds.

part 1b: (84) $\Rightarrow$ (83)
Assume (84) holds. Then there is a variable assignment g such that (91) and (92) hold

(91) $s,g \models {}^{V}Q$

(92) $s,g \models \Box[\exists z[\{z/{}^{V}\chi'\}{}^{V}Q \wedge {}^{V}\chi' = \{z/{}^{V}\chi'\}\delta'] \rightarrow \phi]$.

In (92) it is said that a certain formula is necessarily valid. Application of this to state $<\chi'\leftarrow\delta'>s$ gives

(93) $<\chi'\leftarrow\delta'>s,g \models \exists z[\{z/{}^{V}\chi'\}{}^{V}Q \wedge {}^{V}\chi' = \{z/{}^{V}\chi'\}\delta'] \rightarrow \phi$.

Let $g' \underset{z}{\sim} g$ be such that $g'(z) = V_s({}^{V}\chi')$ so $<\chi'\leftarrow z><\chi'\leftarrow\delta'>s = s$. Since (91) holds, we have

(94) $<\chi'\leftarrow z><\chi'\leftarrow\delta'>s,g' \models {}^{V}Q$.

Consequently

(95) $<\chi'\leftarrow\delta'>s,g' \models \{z/{}^{V}\chi'\}{}^{V}Q$.

Moreover

(96) $<\chi'\leftarrow\delta'>s,g' \models {}^{V}\chi' = \{z/{}^{V}\chi'\}\delta'$

because $V_{<\chi'\leftarrow\delta'>s}({}^{V}\chi') = V_s(\delta') = V_{<\chi'\leftarrow z><\chi'\leftarrow\delta'>s}(\delta') = V_{<\chi'\leftarrow\delta'>s}\{z/{}^{V}\chi'\}\delta'$.
From (94) and (95) follows that the antecedent of the implication in (93) holds. Therefore the consequent of the implication holds

(97) $<\chi'\leftarrow\delta'>s,g' \models \phi$

so

(98) $s,g' \models \{\delta'/\chi'\}\phi$.

This means that (83) holds, so (84) ⇒ (83). And this completes the proof of (I) ⇒ (II).

part 2: (II) ⇒ (I)

The proof of (II) ⇒ (I) uses a related kind of arguments. Therefore this proof will be presented in a more concise way. Assume that (II) holds. Then we have to prove that for arbitrary s and ϕ:

(99) $s \models \forall Q[\Box[\phi \rightarrow \{\delta'/{}^{\vee}\chi'\}{}^{\vee}Q] \rightarrow {}^{\vee}Q]$

if and only if

(100) $s \models \exists z[\{z/{}^{\vee}\chi'\}\phi \wedge {}^{\vee}\chi' = \{z/{}^{\vee}\chi'\}\delta']$.

part 2a

Assume (99). Take for Q in (99) the assertion in (100). We now prove that the antecedent of (99) holds, then (100) is an immediate consequent. So suppose $t \models \phi$. We have to prove that

(101) $t \models \exists z[\{\delta'/{}^{\vee}\chi'\}\{z/{}^{\vee}\chi'\}\phi \wedge \{\delta'/{}^{\vee}\chi'\}[{}^{\vee}\chi' = \{z/{}^{\vee}\chi'\}\delta']]$

or equivalently

(102) $t \models \exists z[\{z'/{}^{\vee}\chi'\}\phi \wedge \delta' = \{z/{}^{\vee}\chi'\}\delta']$.

This is true for $g(z) = V_t({}^{\vee}\chi')$, so the antecedent of (99) holds, and from this follows that (100) holds.

part 2b

Assume (100). Let g be arbitrary and assume

(103) $s,g \models \phi \rightarrow \{\delta'/{}^{\vee}\chi'\}{}^{\vee}Q$.

This is the antecedent of (99). We now prove that the consequent holds, so that

(104) $s,g \models {}^{\vee}Q$.

From (100) follows that for some $g' \underset{z}{\sim} g$

(105) $\langle\chi'\leftarrow z\rangle s,g' \models \phi$.

Using (103), from (105) follows

(106) $\langle\chi'\leftarrow z\rangle s,g' \models \{\delta'/{}^{\vee}\chi'\}{}^{\vee}Q$.

Consequently

(107) $s,g' \models \{\{z/{}^{\vee}\chi'\}\delta'/{}^{\vee}\chi'\}{}^{\vee}Q$.

From (100) also follows

(108) $s,g' \models {}^{\vee}\chi' = \{z/{}^{\vee}\chi'\}\delta'$.

From (108 and (107) we may conclude

(109) $s,g' \models {}^{\vee}Q$.

This proves (104), so (99) follows from (100).
9.8. END

9.9. THEOREM. *The following two statements are equivalent*

(I) $\models sp(\alpha[\beta] := \delta,\phi) = \exists z[\{z/{}^{\vee}\alpha'\}\phi \wedge {}^{\vee}\alpha' = \{z/{}^{\vee}\alpha'\}\lambda n$ *if* $n = \beta'$ *then* δ' *else* ${}^{\vee}\alpha\ [n]$ *fi*$]$

(II) $\models wp(\alpha[\beta] := \delta,\phi) = \{\lambda n$ *if* $n = \beta'$ *then* δ' *else* ${}^{\vee}\alpha'[n]$ *fi*$/{}^{\vee}\alpha\}\phi$.

PROOF. The expressions at the right hand side of the equality signs are a special case of the corresponding expressions in the previous theorem. So theorem 9.9 follows from theorem 9.8.
9.9. END

From theorems 9.9 and 9.10 it follows that the predicate transformations for the assignment as defined in section 9.2, yield weakest preconditions.

10. MUTUAL RELEVANCE

In this section I will mention some aspects of the relevance of the study of semantics of programming languages to the study of semantics of natural languages, and vice versa. Most of the remarks have a speculative character.

The present chapter constitutes a concrete example of the relevance of the theory of semantics of natural languages to the study of programming languages. Montague's framework was developed for natural languages, but it is used here for programming languages. The notions 'opaque' and 'transparant', well known in the field of semantics of natural languages, turn out to be useful for the study of semantics of programming languages, see section 1. And the logic developed for the semantics of natural languages turned out to be useful for programming languages as well.

In the semantics of natural languages the principle of compositionality is not only the basis of the framework, but also, as will be shown later, a valuable heuristic tool. It helped us to understand already existing

solutions. It gives rise to suggestions how to deal with certain problems, and it is useful in finding weak points in proposals from the literature. I expect that the principle can play the same role in the semantics of programming languages. The treatment of arrays in this chapter (see section 6) is an example of the influence of the principle. Below I will give some further suggestions concerning possible relevance of the principle.

Consider the treatment of 'labels' and 'goto-statements' by A. de Bruyn (chapter 7 in De BAKKER 1980). The treatment is rather complex, and not much motivation for it is given. I expect, however, that these phenomena are susceptible to the technique explained in chapter 1: if the meaning of some statement seems to depend on certain factors, then incorporate these factors into the notion of meaning. In this way the notion of 'continuation' (used by de Bruyn) might be more easily explained, and thus the proposal more easily understood.

In De BAKKER 1980, the proof rules for certain constructions make use of devices which are, from a compositional point of view, not attractive. These constructions are assignments to subscripted array identifiers, procedures with parameters, and declarations of identifiers at the beginning of blocks. In the proof rules for these constructions mainly syntactic substitution is used. From a compositional point of view it is not surprising that the semantic treatment of these phenomena is not completely satisfactory. For assignments to array elements an alternative was proposed in section 6, and for blocks a suggestion was made in section 4. A compositional approach to the semantics of procedures with parameters would describe the meaning of a procedure-call as being built from the meaning of the procedure and the meaning of its argument. If this argument is a reference parameter (call by variable), then the argument position is opaque. This suggests that the meaning of such a procedure should be a function which takes as argument an intension.

In the semantics of natural language ideas from the semantics of programming languages can be used. The basic expression in a programming language is the assignment statement. For the computer the assignment statement is a command to perform a certain action. I have demonstrated how the semantics of such commands is dealt with by means of predicate transformers. Inspired by this approach, we might do the same for commands in natural language. Some examples (taken from Van EMDE BOAS & JANSSEN 1978) are given below. Consider the imperative

(110) *John, drink tea.*

Its translation as a predicate transformer would become something like

(111) $\lambda P[^{\wedge}(\mathcal{B}(^{\vee}P) \wedge drink\text{-}tea(^{\wedge}john)]$.

This expression describes the change of the state of the world if the command is obeyed. The operator $\mathcal{B}$ is a kind of state-switcher, it indicates the moment of utterance of the command. A similar approach can be used to describe the semantics of actions. One might describe the smenatics of performative sentences like

(112) *We crown Charles emperor*

by means of an predicate transformer.

Often a sequence of sentences is used to perform an action rather than to make a some assertions: sentences can be used to give information to the hearer. Consider the text

(113) *Mary seeks John. John is a unicorn.*

These sentences might be translated into the predicate transformers (114) and (115).

(114) $\lambda P[^{\vee}P \wedge seek_{*}(mary,john)]$

(115) $\lambda P[^{\vee}P \wedge unicorn_{*}(john)]$.

Suppose that the information the hearer has in the beginning is denoted by ϕ. Then by the first sentence this information is changed into

(116) $\phi \wedge seek_{*}(mary,john)$

and by the second sentence into

(117) $\phi \wedge seek_{*}(mary,john) \wedge unicorn_{*}(john)$.

From the final expression the hearer may conclude that Mary seeks a unicorn.

Also on a more theoretical level the semantics of programming languages can be useful for the study of semantics of natural languages. In the study of natural languages the need for partial functions often arises. In the semantics one wants to use partially defined predicates in order to deal with sortal incorrectness and presuppositions, and in the syntax one wishes to have rules that are not applicable to every expression of the category for which the rule is defined. In the field of programming languages phenomena arise for which one might wish to use partial functions. In this field techniques are used which make it possible to use nevertheless total

functions. The basic idea is to introduce in the semantic domain an extra element. Since this approach is, from an algebraic point of view, very attractive, I would like to use this technique in the field of natural languages as well (see chapter 7).

APPENDIX

SAFE AND POLYNOMIAL

In this appendix the theorem will be presented which was announced in chapter 2, at the end of section 7. The theorem states that in an infinitely generated free algebra all safe operations are polynomially definable (free algebras are algebras which are isomorphic to a term algebra). Remind that $f: A_{s_1} \times \ldots \times A_{s_n} \to A_{s_{n+1}}$ is safe in Σ-algebra $\langle A,F\rangle$ if for every Σ-algebra $\langle D,G\rangle$ and every $h \in Epi(\langle A,F\rangle,\langle D,G\rangle)$ there is a unique $\hat{f}: D_{s_1} \times \ldots \times D_{s_n} \to D_{s_{n+1}}$ such that $h \in Epi(\langle A,F \cup \{f\}\rangle,\langle D,G \cup \{\hat{f}\}\rangle)$. The proof originates from F. Wiedijk (pers. comm.).

THEOREM. *Let* $A = \langle (A_s)_{s\in S}, (F_\gamma)_{\gamma\in\Gamma}\rangle$ *be a free algebra, that has a generating set* $(B_s)_{s\in S}$ *where each* B_s *is infinite. Let* $f: A_{s_1} \times \ldots \times A_{s_n} \to A_{s_{n+1}}$ *be a safe operator. Then* f *is a polynomially definable over* A.

HEURISTICS. First I will give some heuristic considerations, there after the theorem will be proved by proving two Lemmas.

Let us assume for the moment that the theorem holds and let us try to reconstruct from f the polynomial p that defines f. Let $\langle b_1,\ldots,b_n\rangle$ be a possible argument for f, where $b_1,\ldots,b_n$ are generators of A. There is a term $t \in T_{\Sigma,A}$ such that $t_A = f(b_1,\ldots,b_n)$. Since A is free, this term is unique. Hence t is obtained from the polynomial p we are looking for, by means of substituting, for the respective variables in p constants corresponding to $b_1,\ldots,b_n$. Term t (probably) contains constants for $b_1,\ldots,b_n$, but it is not yet clear for any given occurrence of such a constant in t, whether it occurs in p as parameter, or is due to substitution for a variable In order to decide in these matters, we consider the value of f for generators $\langle c_1,\ldots,c_n\rangle$ which are different from $\langle b_1,\ldots,b_n\rangle$ and from the constants in t. Suppose that for term u we have $u_A = f(c_1,\ldots,c_n)$. Then u can also be obtained from p by substituting constants. We already know that all constants in p also occur in t. Since $c_1,\ldots,c_n$ do not occur in t, all their occurrences in u are due to of their substitution for variables. So if we replace in u all occurrences of (constants corresponding with) $c_1,\ldots,c_n$ by variables, we have found the polynomial p we were looking for. This idea is followed in the next lemma. We perform these steps and prove that the polynomial so obtained has the desired properties.

LEMMA 1. *There is an infinite sequence* $(z_k)_{k=1,2,\ldots}$ *of disjoint* n*-tuples of generators of* A, *and a polynomial* p *such that for each* z_k, $f(z_k) = p(z_k)$.

PROOF. We define by induction a sequence $(z_k)_{k\in\{0,1,\ldots\}}$. Let $B_{0,s_1} = B_{s_1},\ldots,B_{0,s_n} = B_{s_n}$, and take $z_0 \in B_{0,s_1} \times\ldots\times B_{0,s_n}$ arbitrarily. This n-tuple is used for the first attempt to reconstruct the polynomial p which corresponds with f. Below I will define an infinite sequence of attempts to reconstruct p, and there after it will be proved that from the second attempt on always the same polynomial will be found; this is the polynomial p we were looking for, as will be proven in lemma 2.

Assume that $z_0,\ldots,z_k$ and $p_0,\ldots,p_k$ are already defined. Then we obtain z_{k+1} and p_{k+1} as follows.

Let $C_{k,s}$ be the set of generators of sort s which corresponds with constants in p_k, and let $\{z_{k,s}\}$ be the set of components of z_k of sort s. Define $B_{k,s} = B_{k,s}/(C_{k,s} \cup \{z_{k,s}\})$, and let $z_{k+1} \in B_{k+1,s_1} \times\ldots\times B_{k+1,s_n}$. Since A is generated by $(B_s)_{s\in S}$, $f(z_{k+1})$ can be represented by a term t^n with parameters from $(B_s)_{s\in S}$, including z_{k+1}. This term t can be expressed as a polynomial expression in z_{k+1}, say $p_{k+1}(z_{k+1})$, where, moreover, no component of z_{k+1} occurs as parameter in p_{k+1}. Since for all k and s the sets $C_{k,s}$ and $\{z_{k,s}\}$ are finite, and $B_{k,s}$ was infinite, it follows that $B_{k+1,s}$ is infinite. Hence this construction can be repeated for all k.

Next it will be proven that the polynomials $p_1,p_2,\ldots$, are identical, thus proving the theorem for the sequence $z_1,z_2,\ldots$, (note that p_0 and z_0 are not included). The basic idea of the proof is that we introduce for each k a homomorphism which maps z_k on z_0, and then apply the assumptions of the theorem.

Consider the mapping $\hat{h}$ defined by

$$\begin{cases} \hat{h}(b) = b & \text{if } b \in (B_s/\{z_{k,s}\}) \text{ for some } s \\ \hat{h}(z_k^{(i)}) = z_0^{(i)}, & \text{where } z_k^{(i)} \text{ is the i-th component of } z_k. \end{cases}$$

Since A is free, the mapping $\hat{h}$ determines uniquely a homomorphism $h: \langle A,F\rangle \to \langle[(B_s/\{z_{k,s}\})_{s\in S}],F\rangle$. Moreover, h is an epimorphism since all generators of the 'range'-algebra occur in the range of h. The polynomials p_k were chosen to contain no constants corresponding to components of z_k, therefore $h(p_k(z_k)) = p_k(h(z_k))$ holds for all k.

Since operator f is safe, there is a unique $\hat{f}$ such that

$$h \in \mathrm{Epi}(\langle A, F \cup \{f\}\rangle, \langle [B_s, \{z_{k,s}\})_{s\in S}], F \cup \{\hat{f}\}\rangle).$$

Now the following equalities hold:

I $\quad h(f(z_k)) = \hat{f}(h(z_k)) = \hat{f}(z_0) = \hat{f}(h(z_0)) = h(f(z_0)) = h(p_0(z_0)) = p_0(h(z_0)) = p_0(z_0)$.

II $\quad h(f(z_k)) = h(p_k(z_k)) = p_k(h(z_k)) = p_k(z_0)$.

From I and II follows $p_0(z_0) = p_k(z_0)$. Analogously we can prove that $p_1(z_1) = p_k(z_1)$.

Since A is free, there is a unique term t such that $p_k(z_0) = t = p_0(z_0)$. So if we replace the variables in p_k and p_0 by constants corresponding to the components of z_0, we obtain the same expression. From this, and the fact that no components of z_0 occur as constants in p_0, it follows that the constants in p_k consists of:

a1) all the constants in p_0.

a2) possibly some constants corresponding to components of z_0.

Analogously it follows that the constants in p_k consist of

b1) all the constants in p_1

b2) possibly some constants corresponding to components of z_k.

We have chosen z_1 in such a way that no constant in p_0 corresponds to a component of z_1, and no component of z_0 equals a component of z_1. So if p_k contained constants for components of z_1, this would conflict with a1) and a2). Therefore we have to conclude that the constants in p_k are the same as the constants in p_1, and none of these, moreover, corresponds to components of z_1. So for all $k \geq 1$ we have $p_k \equiv p_1$. Call this polynomial p. Then $f(z_k) = p(z_k)$ for all $k \geq 1$.

LEMMA 2. *Let* p *be the polynomial guaranteed by lemma 1. Then for all* $a \in A_{s_1} \times \ldots \times A_{s_n}$, $f(a) = p(a)$.

PROOF. Let $a = \langle a^{(1)}, \ldots, a^{(n)}\rangle$, and assume that $a^{(i)} = t^{(i)}(b_1^{(i)}, \ldots, b_n^{(i)})$, where $t^{(i)}$ is a polynomial without constants, and the $b_j^{(i)}$'s are generators of A. Assume moreover that $f(a) = t$. Let z_k be the infinite sequence of disjoint n-tuples of generators given by lemma 1. Since there are only finitely many constants in t_1 and finitely many $b_j^{(i)}$'s, there is an m such that the components of z_m are all different from the constants in t and the $b_j^{(i)}$'s.

Define

$$\hat{h} \text{ by } \begin{cases} \hat{h}(b) = b & \text{if } b \in B_s/\{z_{m,s}\} \text{ for some } s \\ \hat{h}(z_m^{(i)}) = a^{(i)} & \text{where } z_m^{(i)} \text{ is the i-th component of } z_m. \end{cases}$$

This mapping $\hat{h}$ defines an epimorphism $h \in \mathrm{Epi}(\langle A,F\rangle, \langle[(B_s \setminus \{z_{m,s}\})]_{s\in S}, F\rangle)$. Since f is safe, there is a unique operation $\hat{f}$ such that

$$h \in \mathrm{Epi}(\langle A, F \cup \{f\}\rangle, \langle[(B_s \setminus \{z_{m,s}\}_s)], F \cup \{\hat{f}\}\rangle).$$

Now the following equalities hold

$$f(a^{(1)},\ldots,a^{(n)}) = t_A \overset{1}{=} h(t_A) = hf(a^{(1)},\ldots,a^{(n)}) =$$

$$h(f(t^1(\vec{b}^{(1)})),\ldots,f(t^n(\vec{b}^{(n)}))) \overset{2}{=} \hat{f}(h(t^{(1)}(\vec{b}^{(1)})),\ldots,h(t^{(n)}(\vec{b}^{(n)}))) =$$

$$= \hat{f}(h(z_m^{(1)}),\ldots,h(z_m^{(n)})) = h(f(z_m^{(1)},\ldots,z_m^{(n)})) =$$

$$= h(p(z_m^{(1)},\ldots,z_m^{(n)})) = p(h(z_m^{(1)}),\ldots,h(z_m^{(2)}))) = p(a^{(1)},\ldots,a^{(n)}).$$

Equalities 1 and 2 hold since z_m has no components which occur in t or b.
END LEMMAS.

From lemma 1 and lemma 2 the theorem follows

END THEOREM.

INDEX OF NAMES

Adj 17,18,30,37,41-43,66-69,80,83,90,92-94,129,145
Andreka 17
Andrews 111
Apt 172,173
de Bakker 135,140,141,147,158,159,173,175,186
Bartsch 9
van Benthem 33,42,64,80,83,96,100
Birkhoff 43
Blok 33
de Bruyn 186
Burstall 83
Carnap 4
Chany 32
Church 71,100,112
Cook 173,175
Cresswell 3,5,122
Davidson 2
Dijkstra 135,137
Dik 36
Dowty 17
Dummett 5-8,11
Ehrig 83
van Emde Boas 17,37,129,139,140,144,150,161,175,186
Faltz 28
Floyd 135,137,138
Fodor 3,15
Frege 1-11,38
Friedman 120,121
Gabriel 7,8,10
Gallin 100,112,113,122,168
Gazdar 66
Geach 10,38
Gebauer 123
Gerhardt 71
Goguen 83,90,94
Graetzer 33,42,45,46
Gries 140,160
Groenendijk 97,100,115,126
Henkin 32,98,107,112,122
Hermes 7,9,10
van Heyenoort 8
Higgins 43
Hoare 135,136,139,160
Hopcroft 53,55,65
Hughes 122
Huntington 7,10
Jakobovits 143
Janssen 17,36,37,111,125,129,140,150,161,186
Jourdain 10
Karttunen 86
Katz 3,15
Keenan 28
Keisler 33

Kreisel 32
Kreowski 83
Krivine 32
Leibniz 71,108
Lewis 29,34,97
Link 44,100
Lipson 43
Lucas 135
Markusz 17
Marcus 34
Mazurkiewics 4,17
Milner 4,17,37
Monk 32
Montague 3,17,21,27,41,42,44,46,74,76,81,90-92,94,114
Needham 100
Nemeti 129
Neuhold 4,143
Padawitz 83
Partee 3,35,36,97,98
Peremans 79
Peters 17,86
Peano 8
Pietrzykowski 111
Popper 2,5
Potts 144
Pratt 30,131,144
Pullum 66
Quine 71,130
Rabin 52
Rasiova 33
Raulefs 181
Rogers 51
Russell 11
Ryle 143
Sain 17
Schuette 35
Schwartz 15
Scott 4,67
Smith 46
Steinberg 143
Stokhof 97,100,115,126
Stoothoff 10,38
Stoy 4,131,135
Strachey 4,67
Szots 17
Tarski 4,32,35
Tennent 4,131,135,143
Thatcher 56,90
Thiel 8
Thomason 2
Tichy 97
Ullman 53,55,65
Varga 44,100
Veltman 97
Wagner 90
Wall 17
Walk 135

Warren 120,121
Wiedijk 189
Van Wijngaarden 135,161
Wirth 160
Wright 90
Zucker 43,67,147

REFERENCES

Adj, 1977,
=(J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright),
'Initial algebra semantics and continuous algebras',
Journal of the Association for Computing Machinery 24, 68-95.

Adj, 1978,
=(J.A. Goguen, J.W. Thatcher, E.G. Wagner),
'An initial algebra approach to the specification, correctness and implementation of abstract data types',
in R. Yeh (ed.), 'Current trends in programming methodology',
Prentice Hall, 1978, pp. 80-149.

Adj, 1979,
=(J.W. Thatcher, E.G. Wagner, J.B. Wright),
'Notes on algebraic fundamentals for theoretical computer science',
in De Bakker & Van Leeuwen 1979, pp. 83-163.

Andreka, H., & I. Nemeti, 1979,
'Applications of universal algebra , model theory and categories in computer science (survey and bibliography),
Computational Linguistics and Computer Languages 13,
251-282. 'Additions' in Computational Linguistics and Computer Languages 14, 7-20.

Andreka, H., & I. Sain, 1981,
'Connections between algebraic logic and initial algebraic semantics of CF languages',
in Domolki & Gergely 1981, pp. 25-83.

Andrews, P.B., 1971,
'Resolution in type theory',
Journal of Symbolic Logic 36, 414-432.

Angelelli, I. (ed.), 1967,
'Gottlob Frege. Kleine Schriften',
Georg Olms, Hildesheim.

Apt, K.R., 1981,
'Ten years of Hoare's logic. Part 1',
A.C.M. Transactions on Programming Languages and Systems 3,
431-483.

Bakker, J.W. de, 1976,
'Correctness proofs for assignment statements',
Report IW 55, Mathematical Centre, Amsterdam.

Bakker, J.W. de & J. van Leeuwen, 1979,
'Foundations of computer science III. part 2: languages, logic, semantics',
Tract 100, Mathematical Centre, Amsterdam.

Bakker, J.W. de, 1980,
'Mathematical theory of program correctness',
Series in Computer Science, Prentice Hall, London.

Bartsch, R., 1978,
'Semantiek en Montague grammatica',
Algemeen Nederlands Tijdschrift voor Wijsbegeerte 70,
117-136.

Benthem, J.F.A.K. van, 1977,
'Tense logic and standard logic',
Logique et Analyse 80, 395-437.

Benthem, J.F.A.K van, 1979a,
'In alle redelijkheid (Openbare les, 29 mei 1979)',
Bulletin centrale interfaculteit Groningen 4,
Universiteit Groningen.
Benthem, J.F.A.K. van, 1979b,
'Universal algebra and model theory. Two excursions
on the border',
Report ZW-7908, dept. of math., Groningen University.
Birkhoff, G. & J.D. Lipson, 1970,
'Heterogeneous algebras',
Journal of Combinatorial Theory 8, 115-133.
Blok, W., 1980,
'The lattice of modal logics, an algebraic investigation',
Journal of Symbolic Logic 45, 221-236.
Chang, C.C. & H.J. Keisler, 1973,
'Model theory',
Studies in logic and the foundations of mathematics 73,
North Holland, Amsterdam.
Church, A., 1940,
'A formulation of the simple theory of types',
Journal of Symbolic Logic 5, 56-68.
Church, A., 1956,
'Introduction to mathematical logic. Vol I.',
Princeton Univ. Press, Princeton (N.J.).
Cook, S.A., 1978,
'Soundness and completeness of an axiom system for
program verification',
SIAM Journal on Computation 7, 70-90.
Cresswell, M.J., 1973,
'Logics and languages',
Methuen, London.
Davidson, D., 1967,
'Truth and meaning',
Synthese 17, 304-323.
Davidson, D. & G. Harman (eds), 1972,
'Semantics of natural language',
Synthese library 40, Reidel, Dordrecht.
Dijkstra, E.W., 1974,
'A simple axiomatic base for programming language
constructs',
Indagationes Mathematicae 36, 1-15.
Dijkstra, E.W., 1976,
'A discipline of programming',
Prentice Hall, Englewood Cliffs (N.J.).
Domolki, B. & T. Gergely (eds), 1981,
'Mathematical logic in programming (Proc. Salgotorjan 1978)',
Colloquia mathematica societatis Janos Bolyai 26,
North Holland, Amsterdam.
Dowty, D.R., R.E. Wall & S. Peters, 1981,
'Introduction to Montague semantics',
Synthese language library 11, Reidel, Dordrecht.
Dummett, M., 1973,
'Frege. Philosophy of language',
Duckworth, London.

Ehrig, H., H.J. Kreowski & P. Padawitz, 1978,
'Stepwise specification and implementation of abstract data types',
in G. Ausiello & C. Boehm (eds), 'Automata, languages and programming (proc. 5 th. coll., Udine)', Lecture notes in computer science 62, Springer, Berlin.

Emde Boas, P. van, 1974,
'Abstract resource-bound classes',
unpublished dissertation, Univ. of Amsterdam.

Emde Boas, P. van & T.M.V. Janssen, 1978,
Montague grammar and programming languages',
in J. Groenendijk & M. Stokhof (eds), 'Proc. of the second Amsterdam coll. on Montague grammar and related topics', Amsterdam papers in formal grammar II, Centrale Interfaculteit, Univ. of Amsterdam, 1978, pp. 101-124.

Emde Boas, P. van, 1978,
'The connection between modal logic and algorithmic logic',
in J. Winkowski (ed.), Mathematical foundations of computer science (7th. symposium Zakopane)', Lecture notes in computer science 64, Springer, Berlin, 1978, pp. 1-18.

Emde Boas, P. van & T.M.V. Janssen, 1979,
'The impact of Frege's principle of compositionality for the semantics of programming and natural languages',
in '"Begriffsschrift". Jenaer Frege-Conferenz 1979', Friedrich-Schiller Universitaet, Jena, 1979, pp. 110-129.
Also: report 79-07, Dep. of Math, Univ. of Amsterdam, 1979.

Floyd, R.W., 1967,
'Assigning meanings to programs',
in J.T. Schwartz (ed.), 'Mathematical aspects of computer science', Proc. Symp. in Applied Mathematics 19, American Mathematical Society, Providence (R.I.), 1967, pp. 19-32.

Frege, G., 1884,
'Die Grundlagen der Arithmetik. Eine logisch-mathematische Untersuchung ueber den Begriff der Zahl',
W. Koebner, Breslau.
Reprint published by: Georg Olms, Hildesheim, 1961.

Frege, G., 1892,
'Ueber Sinn und Bedeutung',
Zeitschrift fuer Philosophie und philosophische Kritik 100, 25-50.
Reprinted in Angelelli, 1976, pp. 143-162.
Translated as 'On sense and reference' in P.T. Geach & M. Black (eds), 'Translations from the philosophical writings of Gottlob Frege', Basil Blackwell, Oxford, 1952, pp. 56-85.

Frege, G., 1923,
'Logische Untersuchungen. Dritter Teil: Gedankenfuege',
in 'Beitraege zur Philosophie des Deutschen Idealismus. Band III', pp. 36-51.
Reprinted in Angelelli 1976, pp. 378-394.
Translated as 'Compound thoughts' in P.T.Geach & R.H. Stoothoff (transl.), 'Logical investigations. Gottlob Frege', Basil Blackwell, Oxford, 1977, pp. 55-78.

Friedman, J. & D. Warren, 1980,
'Lambda-normal forms in an intensional logic for English'
Studia Logica 39, 311-324.
Gabriel, 1976,
=(G. Gabriel, H. Hermes, F. Kambartel, C. Thiel,
A. Veraart (eds)),
'Gottlob Frege. Wissenschaftliche Briefwechsel',
Felix Meiner, Hamburg.
Gallin, D., 1975,
'Intensional and higher-order modal logic',
Mathematics studies 17, North Holland, Amsterdam.
Gazdar, G., 1982,
'Phrase structure grammar',
in P. Jakobson & G.K. Pullum (eds), 'The nature of syntactic
representation', Synthese language library 15, Reidel, Dordrecht,
1982, pp. 131-186.
Gebauer, H., 1978,
'Montague Grammatik. Eine Einfuerung mit
Anwendungen auf das Deutsche',
Germanistische Arbeitshefte 24, Niemeyer, Tuebingen, 1978.
Gerhardt, C.I. (ed.), 1890,
'Die philosphischen Schriften von Gottfried Wilhelm Leibniz.
Siebenter Band',
Weidmannsche Buchhandlung, Berlin.
Goguen, J.A., & R.M. Burstall, 1978,
Some fundamental properties of algebraic theories:
a tool for semantics of computation.'
Report 53, Department of Artificial Intelligence,
University of Edinburgh.
Graetzer G., 1968,
'Universal algebra',
The univ. series in higher mathematics, van Nostrand, Princeton.
Second edition published by: Springer, New York, 1979.
Gries, D., 1977,
'Assignment to subscripted variables',
Rep. TR 77-305, Dept. of Computer Science, Cornell Univ.,
Ithaca (N,Y.).
Groenendijk J.A.G., T.M.V. Janssen & M.B.J. Stokhof (eds), 1981,
'Formal methods in the study of language. Proceedings of the
third Amsterdam colloquium',
MC-Tracts 135 & 136, Mathematical Centre, Amsterdam, 1981.
Groenendijk, J. & M. Stokhof, 1981,
'Semantics of wh-complements',
in Groenendijk, Janssen, Stokhof, 1981, pp.153 -181.
Groenendijk, J., T.M.V. Janssen, & M. Stokhof (eds), 1984,
'Truth, interpretation, and information. Selected papers from the
third Amsterdam colloquium',
Grass 2, Foris, Dordrecht.
Henkin, L., 1950,
'Completeness in the theory of types',
Journal of Symbolic Logic 15, 81-91.
Henkin, L., 1963,
'A theory of propositional types',
Fundamenta Mathematicae 52, 323-344.

Henkin, L., J.D. Monk & A. Tarski, 1971,
'Cylindric algebras. Part I',
Studies in logic and foundations of mathematics 64,
Norh Holland, Amsterdam.
Hermes, 1969,
=(H. Hermes, F. Kambartel, F. Kaulbach (eds)),
'Gottlob Frege. Nachgelassene Schriften',
Felix Meiner, Hamburg.
Heyenoort, J. van, 1977,
'Sense in Frege',
Journal of Philosophical Logic 6, 93-102.
Higgins, P.J., 1963,
'Algebras with a scheme of operators',
Mathematische Nachrichten 27, 115-132.
Hoare, C.A.R., 1969,
'An axiomatic base for computer programming',
Communications of the Association for Computing Machinery
12, 576-580.
Hoare C.A.R. & N. Wirth, 1973,
'An axiomatic definition of the programming language PASCAL',
Acta Informatica 2, 335-355.
Hopcroft, J.E. & J.D. Ullman, 1979,
'Introduction to automata theory, languages and computation',
Addison-Wesley, Reading (Mass.).
Hughes, G.E. & M.J. Cresswell, 1968,
'An introduction to modal logic',
University Paperbacks 431, Methuen & Co, London.
Janssen, T.M.V. & P. van Emde Boas, 1977a,
'On the proper treatment of referencing, dereferencing
and assignment',
in A. Salomaa & M. Steinby (eds), 'Automata, languages,
and programming (Proc. 4th. coll. Turku)', Lecture notes in
computer science 52, Springer, Berlin, 1977, pp. 282-300.
Janssen, T.M.V. & P. van Emde Boas, 1977b,
'The expressive power of intensional logic in
the semantics of programming languages',
in J. Gruska (ed.), 'Mathematical foundations of
computer science 1977 (Proc. 6th. symp. Tatranska Lomnica)',
Lecture notes in computer science 53, Springer,
Berlin, 1977, pp. 303-311.
Janssen, T.M.V., 1980a,
'Logical investigations on PTQ arising from programming
requirements',
Synthese 44, 361-390.
Janssen, T.M.V., 1981,
'Compositional semantics and relative clause formation
in Montague grammar',
in Groenendijk, Janssen, Stokhof 1981, pp. 237-276.
Janssen, T.M.V., 1981b,
'Montague grammar and functional grammar',
in T. Hoekstra & H v.d. Hulst and M. Moortgat (eds),
'Perspectives on functional grammar', Foris publications,
Dordrecht, 273-297.
also: GLOT 3, 1980, 273-297.

Janssen, T.M.V., & P. van Emde Boas, 1981a,
'On intensionality in programming languages',
in : F. Heny (ed.), 'Ambiguities in intensional contexts',
Synthese library, Reidel, Dordrecht, 1981, pp. 253-269.
Karttunen, L. & Peters, S., 1979,
'Conventional Implicature',
in D.A. Dinneen & C.K. Oh (eds), 'Presuppositions',
Syntax and Semantics 11, Academic Press, New York.
Katz, J.J. & J.A. Fodor, 1963,
'The structure of a semantic theory',
Language 39, pp. 170-210.
Reprinted in J.A. Fodor & J.J. Katz (eds), 'The structure
of language', Prentice Hall, Englewood Cliffs (N.J.), 1964,
pp. 479-518.
Katz, J.J., 1966,
'The philosophy of language',
Harper and Row, London.
Keenan, E.L. & L.M. Faltz, 1978,
'Logical types for natural language',
UCLA occasional papers in linguistics 3.
Kreisel, G. & J.L. Krivine, 1976,
'Elements of mathematical logic. Model Theory',
Studies in logic and the foundations of mathematics 2,
North Holland, Amsterdam.
Kripke, S., 1976,
'Is there a problem about substitutional quantification?',
in G. Evans & J.H. McDowell (eds), 'Truth and meaning.
essays in semantics', Clarendon Press, Oxford, 1976,
pp. 325-419.
Lewis, D., 1970,
'General semantics',
Synthese 22, 18-67.
Reprinted in Davidson & Harman 1972, pp. 169-248.
Reprinted in Partee 1976, pp. 1-50.
Link, G. & M. Varga Y Kibed, 1975,
'Review of R.H. Thomason (ed.), "Formal philosophy. Selected
papers of Richard Montague"',
Erkentniss 9, 252-286.
Lucas, P. & K. Walk, 1971,
'On the formal description of PL/I',
in M.I. Halpern & C.J. Shaw (eds), 'Annual review in
automatic programming, 6', Pergamon Press, Oxford, 1971,
pp. 105-182.
Marcus, R.B., 1962,
'Interpreting quantification',
Inquiry 5, 252-259.
Markusz, Z. & M. Szots, 1981,
'On semantics of programming languages defined by
universal algebraic tools',
in Domolki & Gergely 1981, pp. 491-507.
Mazurkiewicz, A., 1975,
'Parallel recursive program schemes',
in J. Becvar (ed.), 'Mathematical foundations of
computer science (4 th. coll., Marianske Lazne)', Lecture
notes in computer science 32, Springer, Berlin, 1975,
pp. 75-87.

Milner, R., 1975,
'Processes: a mathematical model of computing agents',
in H.E. Rose & J.C. Shepherdson (eds), 'Logic colloquium '73 (Bristol)', Studies in logic and the foundations of mathematics 80, North Holland, Amsterdam, pp. 157-173.

Monk, J.D., 1976,
'Mathematical logic',
Graduate texts in mathematics 37, Springer, Berlin.

Montague, R. 1970a,
'English as a formal language',
in Visentini et al., 'Linguaggi nella societa et nella technica', Edizioni di communita, 1970, (distributed by the Olivetti Corporation, Milan).
Reprinted in : Thomason 1974, pp. 188-221.

Montague, R., 1970b,
'Universal grammar',
Theoria 36, 373-398.
Reprinted in : Thomason 1974, pp. 222-246.

Montague, R., 1973,
'The proper treatment of quantification in ordinary English',
in K.J.J. Hintikka, J.M.E. Moravcsik & P. Suppes (eds), 'Approaches to natural language', Synthese Library 49, Reidel, Dordrecht, 1973, pp. 221-242.
Reprinted in Thomason 1974, pp. 247-270.

Needham, P., 1975,
'Temporal perspective. A logical analysis of temporal reference in English',
Dept. of Philosophy, University of Uppsala.

Neuhold, E.J. (ed.), 1978,
'Formal description of programming concepts (IFIP conf. St. Andrews)',
North Holland, Amsterdam.

Partee, B.H., 1973,
'Some transformational extensions of Montague grammar,'
Journal of Philosophical Logic 2, 509-534.
Reprinted in Partee 1976, pp. 51-76.

Partee, B., 1975,
'Montague grammar and transformational grammar',
Linguistic Inquiry 6, 203-300.

Partee, B.H. (ed.), 1976,
'Montague grammar',
Academic Press, New York.

Partee, B.H., 1977b,
'Possible world semantics and linguistic theory',
The Monist 60, 303-326.

Popper, K, 1976,
'Unended quest. An intellectual autobiography',
Fontana.

Pietrzykowski, T., 1973,
'A complete mechanization of second order theory',
Journal of the Association for Computing Machinery 20, pp. 333-365.

Potts, T., 1976,
'Montague's semiotics. A syllabus of errors',
Theoretical Linguistics 3, 191-208.

Pratt, V.R., 1976,
'Semantical considerations on Floyd-Hoare logic',
in 'Proc. 17th. Symp. on Foundations of Computer Science (Houston)', IEEE Computer Society, Long Beach (Cal.), 1976, pp. 109-121.

Pratt, V.R., 1979,
'Dynamic logic',
in De Bakker & Van Leeuwen 1979, pp. 53-82.

Pratt, V.R., 1980,
'Applications of modal logic to programming',
Studia Logica 39, 257-274.

Pullum, G.K. & G. Gazdar, 1982,
'Natural languages and context-free languages',
Linguistics & Philosophy 4, 471-504.

Quine, W.V.O., 1960,
'Word and object',
The MIT Press, Cambridge (Mass.).

Rabin, M., 1960,
'Computable algebra: general theory and the theory of computable fields',
Transactions of the American Mathematical Society, 95, 341-360.

Rasiowa H., 1974,
'An algebraic approach to non-classical logics',
Studies in logic and foundations of mathematics 78,
North Holland, Amsterdam.

Raulefs, P., 1978,
'The connection between axiomatic and denotational semantics of programming languages',
in K. Alber (ed.), 'Programmiersprachen, 5 Fachtagung der G.I. Univ. Karlsruhe.

Rogers jr., H., 1967,
'Theory of recursive functions and effective computability',
Mc Graw Hill, 1967, New York.

Schuette, K., 1977,
'Proof theory',
Grundlehren der mathematische Wissenschaften 225, Springer, Berlin.

Schwartz, J.T., 1972,
'Semantic definition methods and the evolution of programming languages',
in R. Rustin (ed.), 'Formal semantics of programming languages', Courant Computer Science Symposium 2,
Prentice Hall, Englewood Cliffs (N.J.), 1972, pp. 1-24.

Scott, D. & C. Strachey 1971,
'Towards a mathematical semantics for computer languages',
in J. Fox (ed.), 'Computers and automata (proc. symp. Brooklyn)', Polytechnic Press, Brooklyn (N.Y.), 1971, pp. 16-46.

Steinberg, D.D. & L.A. Jakobovits, 1971,
'Semantics. An interdisciplinary reader in philosophy, linguistics and psychology',
Cambridge Univ. Press.

Stoy, J.E., 1977,
'Denotational semantics: the Scott-Strachey approach to programming language theory',
The MIT Press, Cambridge (Mass.).

Tennent, R.D., 1976,
'The denotational semantics of programming languages',
Communications of the Association for Computing
Machinery 19, 437-453.
Thiel, C., 1965,
'Sinn und Bedeutung in der Logik Gottlob Freges',
Anton Hain, Meisenbach am Glan.
Translated as: 'Sense and reference in Frege's logic',
Reidel, Dordrecht, 1968.
Thomason, R.H. (ed.), 1974,
'Formal philosophy. Selected papers of Richard Montague',
Yale University Press, New Haven.
Tichy, P., 1971,
'An approach to intensional analysis',
Nous 5, 273-297.
Veltman F., 1981,
'Data semantics',
in Groenendijk, Janssen & Stokhof 1981, pp. 541-565.
Revised reprint in Groenendijk, Janssen & Stokhof, 1984,
pp. 43-63.
Wijngaarden, A. van, et al., 1975,
'Revised report on the algorithmic language ALGOL 68',
Acta Informatica 5, 1-236.

MATHEMATICAL CENTRE TRACTS

1 T. van der Walt. *Fixed and almost fixed points.* 1963.

2 A.R. Bloemena. *Sampling from a graph.* 1964.

3 G. de Leve. *Generalized Markovian decision processes, part I: model and method.* 1964.

4 G. de Leve. *Generalized Markovian decision processes, part II: probabilistic background.* 1964.

5 G. de Leve, H.C. Tijms, P.J. Weeda. *Generalized Markovian decision processes, applications.* 1970.

6 M.A. Maurice. *Compact ordered spaces.* 1964.

7 W.R. van Zwet. *Convex transformations of random variables.* 1964.

8 J.A. Zonneveld. *Automatic numerical integration.* 1964.

9 P.C. Baayen. *Universal morphisms.* 1964.

10 E.M. de Jager. *Applications of distributions in mathematical physics.* 1964.

11 A.B. Paalman-de Miranda. *Topological semigroups.* 1964.

12 J.A.Th.M. van Berckel, H. Brandt Corstius, R.J. Mokken, A. van Wijngaarden. *Formal properties of newspaper Dutch.* 1965.

13 H.A. Lauwerier. *Asymptotic expansions.* 1966, out of print; replaced by MCT 54.

14 H.A. Lauwerier. *Calculus of variations in mathematical physics.* 1966.

15 R. Doornbos. *Slippage tests.* 1966.

16 J.W. de Bakker. *Formal definition of programming languages with an application to the definition of ALGOL 60.* 1967.

17 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 1.* 1968.

18 R.P. van de Riet. *Formula manipulation in ALGOL 60, part 2.* 1968.

19 J. van der Slot. *Some properties related to compactness.* 1968.

20 P.J. van der Houwen. *Finite difference methods for solving partial differential equations.* 1968.

21 E. Wattel. *The compactness operator in set theory and topology.* 1968.

22 T.J. Dekker. *ALGOL 60 procedures in numerical algebra, part 1.* 1968.

23 T.J. Dekker, W. Hoffmann. *ALGOL 60 procedures in numerical algebra, part 2.* 1968.

24 J.W. de Bakker. *Recursive procedures.* 1971.

25 E.R. Paërl. *Representations of the Lorentz group and projective geometry.* 1969.

26 European Meeting 1968. *Selected statistical papers, part I.* 1968.

27 European Meeting 1968. *Selected statistical papers, part II.* 1968.

28 J. Oosterhoff. *Combination of one-sided statistical tests.* 1969.

29 J. Verhoeff. *Error detecting decimal codes.* 1969.

30 H. Brandt Corstius. *Exercises in computational linguistics.* 1970.

31 W. Molenaar. *Approximations to the Poisson, binomial and hypergeometric distribution functions.* 1970.

32 L. de Haan. *On regular variation and its application to the weak convergence of sample extremes.* 1970.

33 F.W. Steutel. *Preservation of infinite divisibility under mixing and related topics.* 1970.

34 I. Juhász, A. Verbeek, N.S. Kroonenberg. *Cardinal functions in topology.* 1971.

35 M.H. van Emden. *An analysis of complexity.* 1971.

36 J. Grasman. *On the birth of boundary layers.* 1971.

37 J.W. de Bakker, G.A. Blaauw, A.J.W. Duijvestijn, E.W. Dijkstra, P.J. van der Houwen, G.A.M. Kamsteeg-Kemper, F.E.J. Kruseman Aretz, W.L. van der Poel, J.P. Schaap-Kruseman, M.V. Wilkes, G. Zoutendijk. *MC-25 Informatica Symposium.* 1971.

38 W.A. Verloren van Themaat. *Automatic analysis of Dutch compound words.* 1972.

39 H. Bavinck. *Jacobi series and approximation.* 1972.

40 H.C. Tijms. *Analysis of (s,S) inventory models.* 1972.

41 A. Verbeek. *Superextensions of topological spaces.* 1972.

42 W. Vervaat. *Success epochs in Bernoulli trials (with applications in number theory).* 1972.

43 F.H. Ruymgaart. *Asymptotic theory of rank tests for independence.* 1973.

44 H. Bart. *Meromorphic operator valued functions.* 1973.

45 A.A. Balkema. *Monotone transformations and limit laws.* 1973.

46 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 1: the language.* 1973.

47 R.P. van de Riet. *ABC ALGOL, a portable language for formula manipulation systems, part 2: the compiler.* 1973.

48 F.E.J. Kruseman Aretz, P.J.W. ten Hagen, H.L. Oudshoorn. *An ALGOL 60 compiler in ALGOL 60, text of the MC-compiler for the EL-X8.* 1973.

49 H. Kok. *Connected orderable spaces.* 1974.

50 A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker (eds.). *Revised report on the algorithmic language ALGOL 68.* 1976.

51 A. Hordijk. *Dynamic programming and Markov potential theory.* 1974.

52 P.C. Baayen (ed.). *Topological structures.* 1974.

53 M.J. Faber. *Metrizability in generalized ordered spaces.* 1974.

54 H.A. Lauwerier. *Asymptotic analysis, part 1.* 1974.

55 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 1: theory of designs, finite geometry and coding theory.* 1974.

56 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 2: graph theory, foundations, partitions and combinatorial geometry.* 1974.

57 M. Hall, Jr., J.H. van Lint (eds.). *Combinatorics, part 3: combinatorial group theory.* 1974.

58 W. Albers. *Asymptotic expansions and the deficiency concept in statistics.* 1975.

59 J.L. Mijnheer. *Sample path properties of stable processes.* 1975.

60 F. Göbel. *Queueing models involving buffers.* 1975.

63 J.W. de Bakker (ed.). *Foundations of computer science.* 1975.

64 W.J. de Schipper. *Symmetric closed categories.* 1975.

65 J. de Vries. *Topological transformation groups, 1: a categorical approach.* 1975.

66 H.G.J. Pijls. *Logically convex algebras in spectral theory and eigenfunction expansions.* 1976.

68 P.P.N. de Groen. *Singularly perturbed differential operators of second order.* 1976.

69 J.K. Lenstra. *Sequencing by enumerative methods.* 1977.

70 W.P. de Roever, Jr. *Recursive program schemes: semantics and proof theory.* 1976.

71 J.A.E.E. van Nunen. *Contracting Markov decision processes.* 1976.

72 J.K.M. Jansen. *Simple periodic and non-periodic Lamé functions and their applications in the theory of conical waveguides.* 1977.

73 D.M.R. Leivant. *Absoluteness of intuitionistic logic.* 1979.

74 H.J.J. te Riele. *A theoretical and computational study of generalized aliquot sequences.* 1976.

75 A.E. Brouwer. *Treelike spaces and related connected topological spaces.* 1977.

76 M. Rem. *Associons and the closure statement.* 1976.

77 W.C.M. Kallenberg. *Asymptotic optimality of likelihood ratio tests in exponential families.* 1978.

78 E. de Jonge, A.C.M. van Rooij. *Introduction to Riesz spaces.* 1977.

79 M.C.A. van Zuijlen. *Emperical distributions and rank statistics.* 1977.

80 P.W. Hemker. *A numerical study of stiff two-point boundary problems.* 1977.

81 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 1.* 1976.

82 K.R. Apt, J.W. de Bakker (eds.). *Foundations of computer science II, part 2.* 1976.

83 L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system.* 1979.

84 H.L.L. Busard. *The translation of the elements of Euclid from the Arabic into Latin by Hermann of Carinthia (?), books vii-xii.* 1977.

85 J. van Mill. *Supercompactness and Wallman spaces.* 1977.

86 S.G. van der Meulen, M. Veldhorst. *Torrix I, a programming system for operations on vectors and matrices over arbitrary fields and of variable size.* 1978.

88 A. Schrijver. *Matroids and linking systems.* 1977.

89 J.W. de Roever. *Complex Fourier transformation and analytic functionals with unbounded carriers.* 1978.

90 L.P.J. Groenewegen. *Characterization of optimal strategies in dynamic games.* 1981.

91 J.M. Geysel. *Transcendence in fields of positive characteristic.* 1979.

92 P.J. Weeda. *Finite generalized Markov programming.* 1979.

93 H.C. Tijms, J. Wessels (eds.). *Markov decision theory.* 1977.

94 A. Bijlsma. *Simultaneous approximations in transcendental number theory.* 1978.

95 K.M. van Hee. *Bayesian control of Markov chains.* 1978.

96 P.M.B. Vitányi. *Lindenmayer systems: structure, languages, and growth functions.* 1980.

97 A. Federgruen. *Markovian control problems; functional equations and algorithms.* 1984.

98 R. Geel. *Singular perturbations of hyperbolic type.* 1978.

99 J.K. Lenstra, A.H.G. Rinnooy Kan, P. van Emde Boas (eds.). *Interfaces between computer science and operations research.* 1978.

100 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 1.* 1979.

101 P.C. Baayen, D. van Dulst, J. Oosterhoff (eds.). *Proceedings bicentennial congress of the Wiskundig Genootschap, part 2.* 1979.

102 D. van Dulst. *Reflexive and superreflexive Banach spaces.* 1978.

103 K. van Harn. *Classifying infinitely divisible distributions by functional equations.* 1978.

104 J.M. van Wouwe. *Go-spaces and generalizations of metrizability.* 1979.

105 R. Helmers. *Edgeworth expansions for linear combinations of order statistics.* 1982.

106 A. Schrijver (ed.). *Packing and covering in combinatorics.* 1979.

107 C. den Heijer. *The numerical solution of nonlinear operator equations by imbedding methods.* 1979.

108 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 1.* 1979.

109 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science III, part 2.* 1979.

110 J.C. van Vliet. *ALGOL 68 transput, part I: historical review and discussion of the implementation model.* 1979.

111 J.C. van Vliet. *ALGOL 68 transput, part II: an implementation model.* 1979.

112 H.C.P. Berbee. *Random walks with stationary increments and renewal theory.* 1979.

113 T.A.B. Snijders. *Asymptotic optimality theory for testing problems with restricted alternatives.* 1979.

114 A.J.E.M. Janssen. *Application of the Wigner distribution to harmonic analysis of generalized stochastic processes.* 1979.

115 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 1.* 1979.

116 P.C. Baayen, J. van Mill (eds.). *Topological structures II, part 2.* 1979.

117 P.J.M. Kallenberg. *Branching processes with continuous state space.* 1979.

118 P. Groeneboom. *Large deviations and asymptotic efficiencies.* 1980.

119 F.J. Peters. *Sparse matrices and substructures, with a novel implementation of finite element algorithms.* 1980.

120 W.P.M. de Ruyter. *On the asymptotic analysis of large-scale ocean circulation.* 1980.

121 W.H. Haemers. *Eigenvalue techniques in design and graph theory.* 1980.

122 J.C.P. Bus. *Numerical solution of systems of nonlinear equations.* 1980.

123 I. Yuhász. *Cardinal functions in topology - ten years later.* 1980.

124 R.D. Gill. *Censoring and stochastic integrals.* 1980.

125 R. Eising. *2-D systems, an algebraic approach.* 1980.

126 G. van der Hoek. *Reduction methods in nonlinear programming.* 1980.

127 J.W. Klop. *Combinatory reduction systems.* 1980.

128 A.J.J. Talman. *Variable dimension fixed point algorithms and triangulations.* 1980.

129 G. van der Laan. *Simplicial fixed point algorithms.* 1980.

130 P.J.W. ten Hagen, T. Hagen, P. Klint, H. Noot, H.J. Sint, A.H. Veen. *ILP: intermediate language for pictures.* 1980.

131 R.J.R. Back. *Correctness preserving program refinements: proof theory and applications.* 1980.

132 H.M. Mulder. *The interval function of a graph.* 1980.

133 C.A.J. Klaassen. *Statistical performance of location estimators.* 1981.

134 J.C. van Vliet, H. Wupper (eds.). *Proceedings international conference on ALGOL 68.* 1981.

135 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part I.* 1981.

136 J.A.G. Groenendijk, T.M.V. Janssen, M.J.B. Stokhof (eds.). *Formal methods in the study of language, part II.* 1981.

137 J. Telgen. *Redundancy and linear programs.* 1981.

138 H.A. Lauwerier. *Mathematical models of epidemics.* 1981.

139 J. van der Wal. *Stochastic dynamic programming, successive approximations and nearly optimal strategies for Markov decision processes and Markov games.* 1981.

140 J.H. van Geldrop. *A mathematical theory of pure exchange economies without the no-critical-point hypothesis.* 1981.

141 G.E. Welters. *Abel-Jacobi isogenies for certain types of Fano threefolds.* 1981.

142 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 1.* 1981.

143 J.M. Schumacher. *Dynamic feedback in finite- and infinite-dimensional linear systems.* 1981.

144 P. Eijgenraam. *The solution of initial value problems using interval arithmetic; formulation and analysis of an algorithm.* 1981.

145 A.J. Brentjes. *Multi-dimensional continued fraction algorithms.* 1981.

146 C.V.M. van der Mee. *Semigroup and factorization methods in transport theory.* 1981.

147 H.H. Tigelaar. *Identification and informative sample size.* 1982.

148 L.C.M. Kallenberg. *Linear programming and finite Markovian control problems.* 1983.

149 C.B. Huijsmans, M.A. Kaashoek, W.A.J. Luxemburg, W.K. Vietsch (eds.). *From A to Z, proceedings of a symposium in honour of A.C. Zaanen.* 1982.

150 M. Veldhorst. *An analysis of sparse matrix storage schemes.* 1982.

151 R.J.M.M. Does. *Higher order asymptotics for simple linear rank statistics.* 1982.

152 G.F. van der Hoeven. *Projections of lawless sequences.* 1982.

153 J.P.C. Blanc. *Application of the theory of boundary value problems in the analysis of a queueing model with paired services.* 1982.

154 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part I.* 1982.

155 H.W. Lenstra, Jr., R. Tijdeman (eds.). *Computational methods in number theory, part II.* 1982.

156 P.M.G. Apers. *Query processing and data allocation in distributed database systems.* 1983.

157 H.A.W.M. Kneppers. *The covariant classification of two-dimensional smooth commutative formal groups over an algebraically closed field of positive characteristic.* 1983.

158 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 1.* 1983.

159 J.W. de Bakker, J. van Leeuwen (eds.). *Foundations of computer science IV, distributed systems, part 2.* 1983.

160 A. Rezus. *Abstract AUTOMATH.* 1983.

161 G.F. Helminck. *Eisenstein series on the metaplectic group, an algebraic approach.* 1983.

162 J.J. Dik. *Tests for preference.* 1983.

163 H. Schippers. *Multiple grid methods for equations of the second kind with applications in fluid mechanics.* 1983.

164 F.A. van der Duyn Schouten. *Markov decision processes with continuous time parameter.* 1983.

165 P.C.T. van der Hoeven. *On point processes.* 1983.

166 H.B.M. Jonkers. *Abstraction, specification and implementation techniques, with an application to garbage collection.* 1983.

167 W.H.M. Zijm. *Nonnegative matrices in dynamic programming.* 1983.

168 J.H. Evertse. *Upper bounds for the numbers of solutions of diophantine equations.* 1983.

169 H.R. Bennett, D.J. Lutzer (eds.). *Topology and order structures, part 2.* 1983.

CWI TRACTS

1 D.H.J. Epema. *Surfaces with canonical hyperplane sections.* 1984.

2 J.J. Dijkstra. *Fake topological Hilbert spaces and characterizations of dimension in terms of negligibility.* 1984.

3 A.J. van der Schaft. *System theoretic descriptions of physical systems.* 1984.

4 J. Koene. *Minimal cost flow in processing networks, a primal approach.* 1984.

5 B. Hoogenboom. *Intertwining functions on compact Lie groups.* 1984.

6 A.P.W. Böhm. *Dataflow computation.* 1984.

7 A. Blokhuis. *Few-distance sets.* 1984.

8 M.H. van Hoorn. *Algorithms and approximations for queueing systems.* 1984.

9 C.P.J. Koymans. *Models of the lambda calculus.* 1984.

10 C.G. van der Laan, N.M. Temme. *Calculation of special functions: the gamma function, the exponential integrals and error-like functions.* 1984.

11 N.M. van Dijk. *Controlled Markov processes; time-discretization.* 1984.

12 W.H. Hundsdorfer. *The numerical solution of nonlinear stiff initial value problems: an analysis of one step methods.* 1985.

13 D. Grune. *On the design of ALEPH.* 1985.

14 J.G.F. Thiemann. *Analytic spaces and dynamic programming: a measure theoretic approach.* 1985.

15 F.J. van der Linden. *Euclidean rings with two infinite primes.* 1985.

16 R.J.P. Groothuizen. *Mixed elliptic-hyperbolic partial differential operators: a case-study in Fourier integral operators.* 1985.

17 H.M.M. ten Eikelder. *Symmetries for dynamical and Hamiltonian systems.* 1985.

18 A.D.M. Kester. *Some large deviation results in statistics.* 1985.

19 T.M.V. Janssen. *Foundations and applications of Montague grammar, part 1: Philosophy, framework, computer science.* 1986.

20 B.F. Schriever. *Order dependence.* 1986.

21 D.P. van der Vecht. *Inequalities for stopped Brownian motion.* 1986.

22 J.C.S.P. van der Woude. *Topological dynamix.* 1986.

23 A.F. Monna. *Methods, concepts and ideas in mathematics: aspects of an evolution.* 1986.

24 J.C.M. Baeten. *Filters and ultrafilters over definable subsets of admissible ordinals.* 1986.

25 A.W.J. Kolen. *Tree network and planar rectilinear location theory.* 1986.

26 A.H. Veen. *The misconstrued semicolon: Reconciling imperative languages and dataflow machines.* 1986.

27 A.J.M. van Engelen. *Homogeneous zero-dimensional absolute Borel sets.* 1986.

28 T.M.V. Janssen. *Foundations and applications of Montague grammar, part 2: Applications to natural language.* 1986.

29 H.L. Trentelman. *Almost invariant subspaces and high gain feedback.* 1986.

30 A.G. de Kok. *Production-inventory control models: approximations and algorithms.* 1987.

31 E.E.M. van Berkum. *Optimal paired comparison designs for factorial experiments.* 1987.

32 J.H.J. Einmahl. *Multivariate empirical processes.* 1987.

33 O.J. Vrieze. *Stochastic games with finite state and action spaces.* 1987.