Anaphora and the Logic of Change*

Reinhard Muskens

1 Introduction

There are three major currents in semantic theory these days. First there is what Chierchia [1990] aptly calls "what is alive of classical Montague semantics". Secondly, there is Discourse Representation Theory. Thirdly, there is Situation Semantics. Each of these three branches of formal semantics has its own specialities and its particular focuses of interest. Each can boast of its own successes. Thus Montague semantics models the Fregean building block theory of meaning in a particularly elegant way, gives a unified account of the semantics of noun phrases as generalized quantifiers and a natural but sophisticated treatment of coordination phenomena. Discourse Representation Theory (DRT), on the other hand, treats different kinds of anaphora succesfully, extends the field of operation of semantic theory to the level of texts, handles Geach's so-called 'donkey' sentences in a convincing way and generally deepens our understanding of semantics by its insistence on the dynamic rather than static nature of meaning. Situation Semantics, lastly, emphasizes the partial character of meaning and information and is very much focussed on the contextual dependance of language. The theory gives a nice treatment of the semantics of perception verbs (see Barwise [1981]) and an interesting new approach to the Liar paradox (Barwise & Etchemendy [1987]).

*From: J. van Eijck (ed.), *Logics in AI,* Lecture Notes in Artificial Intelligence **478**, Springer, Berlin, 1991, 412-428. I would like to thank René Ahn, Nicolas Asher, Johan van Benthem, Martin van den Berg, Gennaro Chierchia, Jaap van der Does, Peter van Emde Boas, Paul Dekker, Jan van Eijck, Jeroen Groenendijk, Theo Janssen, Jan Jaspars, Hans Kamp, Fernando Pereira, Barbara Partee, Frank Veltman and Henk Zeevat for their comments, criticisms and discussion. An earlier version of this paper has circulated under the title 'Meaning, Context and the Logic of Change'.

Unfortunately there is no single semantic framework in which all these niceties can be combined and although the three semantic theories are historically connected (all three derive from Richard Montague's pioneering work) and each claims to be a formal, mathematical theory of meaning, it is difficult to compare the three theories due to the diverging technical setups. It is hard to find a position from which all three can be viewed simultaneously and it should be noted that each is lacking in the sense that it cannot explain or copy all successes of the others.

What is needed, clearly, is a synthesis, and indeed some work has been done that goes in the direction of a unified theory of semantics. So, for example, Barwise [1987] compares Montague's [1973] generalized quantifier model of natural language quantification, further developed in Barwise & Cooper [1981], with the approach taken in Barwise & Perry [1983]. Rooth [1987] takes Barwise's paper as a starting point and gives a Montague style fragment of English that embodies a version of the Heim / Kamp theory. Groenendijk and Stokhof [1990] develop a Montagovian version of Discourse Representation Theory as well, calling it 'Dynamic Montague Grammar' (DMG), while Muskens [1989a, 1989b], to give a fourth example, shows that an important feature of Situation Semantics—partiality—is compatible with Montague's type theoretic approach to semantics and that the Situation Semantic analyses of perception verbs and propositional attitudes can be recast in a 'Partial Montague Grammar'.[1]

In this paper I want to make a further contribution towards a synthesis of the existing frameworks of formal semantics. I want to try my hand at another version of the theory of reference developed by Kamp and Heim. This version will be compatible with Montague's framework and compatible to a large extent with my previous unification of that framework with the partiality of Situation Semantics. I shall make extensive use of some of the very interesting ideas that were developed in Groenendijk & Stokhof's DMG and its predecessor Dynamic Predicate Logic (DPL, see Groenendijk & Stokhof [1989]). But while these systems are based on rather unorthodox

[1] The partial theory of types is a simple (four-valued, Extended Strong Kleene) generalization of the usual, total, theory of types that we shall employ below. The setup is relational as in Orey [1959], not functional. The logic is weaker than the total logic but it shares many of the latter's model-theoretic properties. So, for example, it has the property of generalized completeness (validity with respect to Henkin's generalized models can be axiomatized). For technical information see the works mentioned.

logics,[2] I simply use the (many-sorted) theory of types to model the DRT treatment of referentiality. Ordinary type theory is not only much simpler to use than the 'Dynamic Intensional Logic' that Groenendijk & Stokhof employ[3] (or, for that matter, than Montague's IL), it is also much better understood at the metamathematical level. Logics ought not to be multiplied except from necessity.

It turns out that the cumulative effect of this and other simplifications makes the theory admit of generalizations more readily. In a sequel to this paper (Muskens [to appear]) I'll show that, apart from the formalization of Kamp's and Heim's treatment of nominal anaphora given here, the essentially Reichenbachian theory of *tenses* that has been developed within the DRT framework can be formalized in my theory. That Montague's treatment of *intensionality* can be incorporated without any complications will be shown as well.

Our theory will be based on two assumptions and one technical insight. The first assumption is that meaning is compositional. The meanings of words (roughly) are the smallest building blocks of meaning, and meanings may combine into larger and larger structures by the rule that the meaning of a complex expression is given by the meanings of its parts.

The second assumption is that meaning is computational. Texts effect change, in particular, texts effect changes in context. The meaning of a sentence or text can be viewed as a relation between context states, much in the way that the meaning of a computer program can be viewed as a relation between program states.

What is a context state? Evaluation of a text may change the values of many contextual parameters: the temporal point of reference may move, the universe of discourse may grow larger or smaller, possible situations may become relevant or irrelevant to a particular modality, presuppositions may spring into existence, and so on. If we want to keep track of all this, we must set up a 'conversational scoreboard' in the sense of Lewis [1979], a list of all current values of contextual parameters. We may then try to study

[2]For example, in DPL an existential quantifier can bind variables that are outside of its syntactic scope. This directly reflects the fact that in natural language indefinite noun phrases create discourse referents that can be picked up later by anaphoric pronouns not in their scope. While it may be nice to have such a close connection between logic and language, I consider the price that is to be paid in the form of technical complications much too high.

[3]Dynamic Intensional Logic (DIL) is due to Theo Janssen (see Janssen [1983]).

the kinematics of score, the rules that govern score change. On top of this, if we want to be able to interpret a text, we must have a list of all discourse referents active at a particular point in discourse. Texts dynamically create referents that can be referred to at a later instance (see Karttunen [1976], Heim [1982]). For example, if we read the short text in (1) then after reading its first sentence a discourse referent is set up that is picked up by the pronoun she_1 in the second sentence.

(1) A_1 girl walked by. She_1 was pretty

So we must keep track of two lists. One list tells us what values Lewis's components of conversational score have and one tells us what value each discourse referent has at each point in discourse.[4] We may combine these two lists into one and call it a (*context*) *state.*

If we were to design a computer program to keep track of the state of discourse (as in Karttunen [1976]) it would be a natural choice to associate each component of conversational score and each discourse referent with a variable in that program. In fact, we may entertain the metaphor that a natural language text *is* a program, continually effecting the values of a long list of variables. Interpretation of a text continually changes the context state and the context state at each point in discourse in its turn effects interpretation. In much the same way a computer program changes the values of its variables while the values of these variables effect the course the computation takes.

The technical insight I referred to above is that virtually all programming concepts to be found in the usual imperative computer languages are available in classical type theory. We can do any amount of programming in type theory. This suggests that type theory is an adequate tool for studying how languages can program context change. Since there is also some evidence that type theory is a good vehicle for modelling how the meaning of a complex expression depends on the meaning of its parts, we may hope that it is adequate for a combined theory: a compositional theory of the computational aspects of natural language meaning.

The logic of programming is usually studied in a theory called *dynamic logic* and I'll show how to generalize this logic to the full theory of types in

[4] It may seem that we need more than one value for the referent that was set up by a_1 *girl* in (1), but see the discussion on nondeterminism below.

the next section. When this is done I'll show how to apply the resulting generalization to some phenomena that are central to Discourse Representation Theory in section 2.

2 Type Theory and Dynamic Logic

Dynamic Logic (Pratt [1976], for an excellent survey see Harel [1984], for a transparent introduction Goldblatt [1987]) is a logic of computation. In dynamic logic the meaning of a computer program is conceived of as a relation between *machine states*, execution of a program changes the state a machine is in. In an actual computer a machine state could be thought of as consisting of the contents of all registers and memory locations in the device at a particular moment. But in theory we make an abstraction and consider the abstract machines that are associated with programs. We can identify the states of such program machines with functions that assign values to all program variables.

Thus, for example, suppose we have seven variables in our program, of which u, v and w range over natural numbers, X and Y range over sets of natural numbers and *card* and *bool* range over playing cards and the values *yes* and *no* respectively. Then the columns in the following figure can be identified with machine states.

	i_1	i_2	i_3	i_4	i_5	...
u :	0	22	7	22	22	...
v :	0	2	44	2	2	...
w :	5	2	7	22	22	...
X :	$\{n \mid n \geq 9\}$	$\{3, 0\}$	$\{5\}$	$\{3, 0\}$	$\{3, 0\}$	...
Y :	$\{3, 14, 8\}$	$\{n \mid n \geq 2\}$	$\emptyset$	$\{3, 0\}$	$\{n \mid n \geq 2\}$	...
card:	3$\heartsuit$	10$\diamondsuit$	8$\clubsuit$	10$\diamondsuit$	10$\diamondsuit$	...
bool:	*yes*	*yes*	*no*	*yes*	*yes*	...

fig. 1

The meaning of a given program is identified with the set of all pairs of states $\langle i,j \rangle$ such that starting in state i we may end in state j after execution of that program. For example, suppose that our abstract machine is in state i_2 and that the statement to be executed is the assignment $w := u$. Then after execution the machine will be in state i_5. The value that was assigned to u in i_2 is now assigned to the program variable w as well. This means that the pair $\langle i, j\rangle$ is considered to be an element of the meaning of the atomic program $w := u$. More generally, the meaning of $w := u$ is the set of all pairs $\langle i, j\rangle$ such that the value of w at j equals the value of u at i, while the values of all other program variables remain unaltered.

Apart from programs we may also consider *formulae* like the identity expression $u = w$. Formulae express no relation between machine states, but are just true or false at any given state. For example $u = w$ is false at states i_1 and i_2, but true at states i_3, i_4 and i_5. Consequently, the meaning of a formula is identified with a set of machine states.[5]

Let us consider programs and formulae that are more complex than those that consist of just one assignment statement or just one identity expression. The syntax of dynamic logic offers the following constructions: Suppose that γ and δ are programs and that A and B are formulae, then $\perp$, $A \to B$ and $[\gamma]A$ are formulae and $\gamma;\delta$, $\gamma\cup\delta$, $A?$ and γ^* are programs. The formula $\perp$ is defined to be false at every state, $A \to B$ is false at a state if and only if A is true and B is false at that state. In Goldblatt's book we find the following other intended meanings:

$\gamma;\delta$	do γ and then δ
$\gamma\cup\delta$	do either γ or δ non-deterministically
$A?$	test A : continue if A is true, otherwise "fail"
γ^*	repeat γ some finite number (≥ 0) of times
$[\gamma]A$	after every terminating execution of γ, A is true

I'll discuss these constructions one by one now. The first is the *sequencing* of statements $\gamma;\delta$, This sequencing has a lot in common with the consecution of sentences in a text and with the behaviour of the word "and" in English. If we start in state i_2 of fig. 1 and execute the sequential statement $w := u; Y := X$ then execution of the first part will take us to state i_5 as before, after which an execution of the second part will bring us to i_4. Thus the pair $\langle i_2, i_4\rangle$ is an element of the meaning of the program $w := u; Y := X$. In general, if the meaning of program γ is the relation R_γ and the meaning of δ is the relation R_δ then the meaning of γ ; δ is the set of all pairs $\langle i, j\rangle$ such that $\langle i,k\rangle \in R_\gamma$ and $\langle k,j\rangle \in R_\delta$ for some state k. The resulting relation is sometimes called the *product* of R_γ and R_δ. If both relations happen to be functions, that is

[5]Readers familiar with Discourse Representation Theory will note that the distinction between formulae and programs in dynamic logic mirrors the distinction between conditions and DRSs in DRT.

if we are considering *deterministic* programs, this product is nothing but the *composition* of these functions.

But we do not restrict ourselves to the consideration of deterministic programs (programs expressing functions), as the second construct, the *choice,* makes clear. Suppose we are in state i_5, then execution of $w := v$ will bring us to i_2, but execution of $Y := X$ will bring us to i_4. Thus execution of $w := v \cup Y := X$ may either land us in i_2 or in i_4. It follows that both $\langle i_5, i_2 \rangle$ and $\langle i_5, i_4 \rangle$ are elements of the meaning of $w := v \cup Y := X$ In general, the meaning of $\gamma \cup \delta$ is the union of the relations that are the meanings of γ and δ respectively.

From a programming point of view it might at first blush not seem very realistic to include a nondeterministic construction in the syntax: the computers that you and I have at our desks certainly operate in a deterministic way. But the allowance of nondeterminism greatly facilitates the study of the semantics of programming languages and computer language semanticists view deterministic programs as an interesting special case to which the results of their more general studies can be applied. In natural language nondeterminism seems to be the rule rather than the exception. Consider the following short text.

(2) A_1 man entered. He_1 ordered a beer.

Suppose we have a program that is designed to read and interpret texts like these (a program like the one in Karttunen [1976]). The program does not operate on (symbolic representations of) natural numbers, sets of natural numbers and cards, but on (symbolic representations of) things in the world, relations among these things, and so on. After reading the first sentence, the program must have stored some man who entered in some internal store, say in v_1 ; this man can then be picked up later as the referent of he_1 in the second sentence. Now, which man should be stored in v_1? This appears to be a great problem if we think of the program as embodying a deterministic automaton. Suppose that in fact both Bill and John entered, but that only John ordered a beer (while Bill ordered a martini). Then if the program stores Bill in v_1 the text will be interpreted as being false, while if John is stored, it will (correctly) come out true. But the program cannot know this in advance, that is, after processing the first sentence it has no information that allows it to discriminate between the two men. So, which man should be stored, the 'indeterminate' man? This solution would seem to land us right into the middle of Mediaeval philosophy and into the knot of problems from which modern post-Fregean logic has freed us.

But if we allow our program to operate nondeterministically, the problem vanishes. We can then let the meaning of the first sentence consist of those pairs of machine states $\langle i, j\rangle$ such that i is like j except that the value of v_1 in j is some man who entered. In j_1 this may be Bill, in j_2 it may be John and it may be some other man who entered in some other state (in fact, speaking loosely, we might say now that we have stored an 'indeterminate' man in v_1). Some of the men stored may not have ordered a beer, but states in which the value of v_1 did not order a beer will be ruled out by (2)'s next sentence.

How does (2)'s second sentence manage to rule out such states? This question brings us to the third syntactic construct of dynamic logic in the list above, the *test*. The meaning of a program $A?$ (where A is a formula) is the set of all pairs $\langle i, i\rangle$ such that i is an element of the meaning of A. To see how this can be used to rule out certain possible continuations of the computation, consider the program $(w := v \cup Y := X); u = w?$ and start its execution in state i_5. After executing the choice $w := v \cup Y := X$ we land in states i_2 and i_4 as before, but now execution of the test $u = w?$ ensures that i_2 is ruled out. The pair $\langle i_5, i_4\rangle$ is an element of the meaning of the construct as a whole, but the pair $\langle i_5, i_2\rangle$ is not, and all possible continuations starting in i_2 have now become irrelevant. In a similar way we may think of the second sentence in (2) as performing a test, ruling out certain possible continuations of the interpretation process.

Thus the first three syntactic constructs in our list have a close correspondence to phenomena in natural language. *Sequencing* of programs is strongly reminiscent of the sequencing of sentences in a text and of natural language conjunction generally. The nondeterminism that is introduced by *choice* is closely connected to the indefinite character of indefinites. And *tests* rule out certain possibilities in much the same way as natural language expressions may do.

But for the last two constructs in the list I see no direct application to natural language semantics. I have merely included them for the sake of completeness and I should like to confine myself to stating their semantics without discussion: The meaning of an *iteration* γ^* is the reflexive transitive closure of the meaning of γ and the meaning of a formula $[\gamma]A$ is the set of states i such that for all j such that $\langle i, j\rangle$ is in the meaning of γ, j is in the meaning of A.

Now suppose we want to consider natural language phenomena in the light of the dynamic logic sketched above and that we want to do this in the general (Montagovian) setting of Logical Semantics. A first problem to solve then is of a logical character. On the one hand Montague semantics is based on the theory of types, on the other we want to have the main concepts of dynamic logic at our disposal. How can we work in type theory and use dynamic logic too? The solution is simple and takes the form of a translation of dynamic logic into type theory.

We'll work with the two-sorted type theory TY_2 of Gallin [1975]. Essentially this is just Church's [1940] type theory, be it that there are three basic types, where Church uses only two. The basic types are e, s and t, standing for *entities, states* and *truth values* respectively. As I stated above the syntactic constructs of dynamic logic can be divided into two categories: formulae and programs. Formulae are true or false at a given state and thus should translate as terms of type st (sets of states), while programs are state changers and get type $s(st)$ (relations between states). Define the translation function † from the constructs of dynamic logic to those of type theory inductively by the following clauses (i, j, k and l are variables of type s, X is a variable of type st):

$$\begin{aligned}
(\bot)^\dagger &= \lambda i \,\bot \\
(A \rightarrow B)^\dagger &= \lambda i(A^\dagger i \rightarrow B^\dagger i) \\
(\gamma;\delta)^\dagger &= \lambda ij\exists k(\gamma^\dagger ik \wedge \delta^\dagger kj) \\
(\gamma \cup \delta)^\dagger &= \lambda ij(\gamma^\dagger ij \vee \delta^\dagger ij) \\
(A?)^\dagger &= \lambda ij(A^\dagger i \wedge j = i) \\
(\gamma^*)^\dagger &= \lambda ij\forall X((Xi \wedge \forall kl((Xk \wedge \gamma^\dagger kl) \rightarrow Xl)) \rightarrow Xj) \\
([\gamma]A)^\dagger &= \lambda i\forall j(\gamma^\dagger ij \rightarrow A^\dagger j)
\end{aligned}$$

The clauses here closely follow the discussion of dynamic logic given above. We see that the translation of $\bot$ is the empty set of states, that $A \rightarrow B$ translates as the set of states at which either the translation of A is false or the translation of B is true, that the meaning of $\gamma;\delta$ is given as the product of the meanings of γ and δ, that the meaning of $\gamma \cup \delta$ is the union of the meanings of its components and that the meaning of a test $A?$ is given as the set of all pairs $\langle i, i\rangle$ such that A is true at i. The translations of γ^* and of $[\gamma]A$ are again listed for the sake of completeness only. The first gives the reflexive transitive closure of the meaning of γ by means of a second order quantification;[6] the second treats $[\gamma]$ essentially as a modal operator with an accessibility relation given by γ.

This translation embeds the propositional part of dynamic logic into type theory, the part that contains no variables (or quantification) and hence no assignment statements. But we do want to study how assignments are being made, for it seems that language has a capacity to update the components of conversational score in a way reminiscent of the updating of variables in a program. So let us return to our discussion of states, variables and assignment statements now.

The reader may have noted a contradiction between our treatment of states as primitive objects and our earlier declaration that states are functions from program variables to the possible values of these variables. We could try to remove this contradiction by taking states to be objects of some complex type $\alpha\beta$, where α is the type of variables and β is the type of their values. But this plan fails, for in general there is no *single* type of variables and no *single* type of the values of variables. Programming languages can handle variables ranging over many different data types and human languages seem to be capable of storing many different sorts of things as items of conversational score. It seems that we have a problem here. Was it caused by an all too strict adherence to a typed system?

There is an ingenious little trick due to Theo Janssen [1983] that helps us out: Janssen simply observed that we may go on treating states as primitive if we treat program variables as functions from states to values. That is, we may shift our attention from the columns in figure 1 to the rows, and instead of viewing (say) i_2 as the function that assigns the number 22 to u, the number 2 to v, the set $\{n \mid n \geq 2\}$ to Y, the card 10$\diamondsuit$ to *card* and so on, we may view (say) w as the function assigning the number 5 to i_1, the number 2 to i_2, the number 7 to i_3 etcetera. This procedure is clearly equivalent to the older one and it saves us from the type clash we encountered above.

This means that we can regard states as inhabitants of our typed domains and the same holds for the things that are denoted by program variables. States all live in the same basic domain D_s, while the denotations of program variables may live in different domains. For example, if n is the type of natural numbers then the denotation of u in figure 1 lives in D_{sn}, but the

[6]The treatment of iteration improves upon the results in Janssen [1983]. A treatment of recursion in the typed models of classical higher order logic is given in Muskens [in preparation].

denotation of X lives in $D_{s(nt)}$. A program variable that has values of type α is a function of type $s\alpha$ itself.

Treating states as primitive and treating program variables as functions from states to values thus allows us to have many different types of things that can be stored as the value of a variable at a certain state. But now that we have assured ourselves of this possibility we shall refrain from using it. For reasons of exposition we shall allow only type e objects to be values of program variables and program variables consequently will have type se. In a sequel to this paper (Muskens [to appear]), however, we'll make a more extensive use of our possibilities and there the theory will be generalized so that we can have any finite number of types of program variables.

We should, by the way, remove a possible source of confusion. We are treating the denotations of program variables as *objects* in our ontology. Objects can be referred to in two ways, by means of constants or by means of variables, and there is no reason to make an exception for objects of type se. In view of this, the term program *variable* is perhaps less than felicitous and I want to change terminology now. Referring to the object I shall from now on use the term *store,* a constant denoting a store is a *store name* and a (logical) variable ranging over stores a *store variable.*[7] I take it that the syntactic objects that are usually called program variables are in fact store *names,* not store *variables.* Stores are functions, and of course the values of a function may vary in the sense that a function may assign different values to different arguments.

What effect does the execution of an assignment statement $v := u$ have on a state? It changes the value of the store named by v to the value of the store named by u, but it leaves all other stores as they are. Consequently, if we write $i[v]j$ for "states i and j agree in all stores, except possibly in the store (named by) v", the following should be our translation of the assignment statement into type logic.

$$(v := u)^\dagger = \lambda ij(i[v]j \wedge vj = ui)$$

The intuitive meaning of the formula $i[v]j \wedge vj = u$ is that i and j agree in all stores, except possibly in store v and that the value of store v in j is identical to the value of store u in i.

In order to make this really work two conditions must be fulfilled. The first of these is that the expression $i[v]j$ really means what we want it to mean.

[7] This is the official position. Once the basic confusion is removed there seems to be no harm in some happy sinning against strict usage.

This we can ensure by letting $i[v]j$ be an abbreviation of $\forall u_{se}((STu \wedge u \neq v) \to uj = ui)$, where ST is a non-logical constant of type $(se)t$ with the intuitive interpretation "is a store". The second condition that is to be fulfilled if we want our treatment of assignments to be correct, is that for each i there really *is* a j in the model such that $i[v]j \wedge vj = ui$. Until now there is nothing that guarantees this. For example, some of our typed models may have only one state in their domain D_s. In models that do not have enough states an attempt to update a store may fail; we want to rule out such models. In fact, we want to make sure that we can *always* update a store selectively with each appropriate value we may like to. This we can do by means of the following axiom.

AX1 $\forall i \forall v_{se} \forall x_e (STv \to \exists j(i[v]j \wedge vj = x))$

This makes sure that an assignment is always possible by postulating that the required state always exists. The axiom scheme is closely connected with Goldblatt's [1987, pp. 102] requirement of 'Having Enough States' and with Janssen's 'Update Postulate'. We'll refer to it as the Update Axiom. It follows from the axiom that not all type se functions are stores (except in the marginal case that D_e contains only one element), since, for example, a constant function that assigns the same value to *all* states cannot be updated to another value. The Update Axiom imposes the condition that contents of stores may be varied at will.

Of course store names should refer to stores and that is just what the following axiom scheme requires.

AX2 STv for each store name v

The combined effect of these axioms and the definition of $i[v]j$ now guarantees that assignment statements always get the interpretation that is desired.

There is one more axiom scheme that we shall need, an axiom scheme that is completely reasonable from a programming point of view: although different stores may have the same value at a given state, we don't want two different store names to refer to the same store. An assignment $v := u$ should not result in an update of w simply because v and w happen to be names for the same store and from $i[v]j$ we want to be able to conclude that $ui = uj$ if u and v are different store names. This we enforce simply by demanding

AX3 $u \neq v$ for each two syntactically different store names u and v

This ends our discussion of the assignment statement and it ends our discussion of the more general part of the theory. All programming concepts that are needed in the rest of the paper have been introduced now. Essentially we have shown how to treat the class of so-called **while** programs in Montague Grammar.[8] Since every computable function can be implemented with the help of a **while** program this means that we can do any amount of programming in classical type theory.

3 Nominal Anaphora

In this section I'll define a little Montague fragment of English, treating anaphora in the way of Kamp [1981] and Heim [1982]. The result can be viewed as a direct generalization of Groenendijk & Stokhof's system of 'Dynamic Predicate Logic' (Groenendijk & Stokhof [1989]) to the theory of types.[9] The fragment will be based on a system of categories that is defined in the following manner.

i. S and E are categories;

ii. If A and B are categories, then $A/^nB$ is a category ($n \geq 1$).

Here S is the category of sentences (and texts). The category E does not itself correspond to any class of English expressions, but it is used to build up complex categories that do correspond to such classes. The notation $/^n$ stands for a sequence of n slashes. I'll employ some familiar abbreviations for category notations, writing

VP (verb phrase)	for	S/E,
N (common noun phrase)	for	$S//E$
NP (noun phrase)	for	S/VP ,
TV (transitive verb phrase)	for	VP/NP, and
DET (determiner)	for	NP/N.

The analogy that we have noted between programs and texts motivates us to treat sentences, and indeed texts, as relations between states, objects of

[8] The statement **while** A **do** α can be defined as $(A?;\alpha)^*;\neg A?$.

[9] In fact the present system is closer to DPL than Groenendijk & Stokhof's own generalization, DMG, is. Roughly, what Groenendijk & Stokhof do on the metalevel of DPL I do on the object level of type theory.

type $s(st\ )$, just like programs. The category E we associate with type e. More generally, we define a correspondence between types and Montague's categories as follows.

i. $TYP(S) = s(st)$; $TYP(E\ ) = e$;

ii. $TYP(A/^n B) = (TYP(B), TYP(A))$.

The idea is that an expression of category A is associated with an object of type $TYP(A)$ and that an expression that seeks an expression of category B in order to combine with it into an expression of category A is associated with a function from $TYP(B)$ objects to $TYP(A)$ objects, or, equivalently, with a $(TYP(B), TYP(A))$ object.

To improve readability let's abbreviate our notation for types somewhat and let's write $[\alpha_1 \dots \alpha_n]$ instead of $(\alpha_1(\alpha_2(\dots \alpha_n(s(st))\dots)$. Under this convention, the rule above assigns the types listed in the second column of the table below to the categories listed in its first column.

Category	*Type*	*Some basic expressions*
VP	$[e]$	*walk, talk*
N	$[e]$	*farmer, donkey, man, woman, bastard*
NP	$[[e]]$	$Pedro_n$, $John_n$, it_n, he_n, she_n $(n \geq 1)$
TV	$[[[e]]e]$	*own, beat, love*
DET	$[[e][e]]$	a_n, $every_n$, the_n, no_n $(n \geq 1)$
$(N\ /\ N)\ /\ VP$	$[[e][e]e]$	*who*
$(S\ /\ S\ )\ /\ S$	$[[][]]$	*and, or,.* (the stop)
$(S\ /\ S\ )\ //\ S$	$[[][]]$	*if*

Some basic expressions belonging to these categories I have listed in the third column. From these the complex expressions of our fragment are built. An expression of category $A/^n B$ will combine with an expression of category B and the result will be an expression of category A. For example, the word a_n of category DET (defined as NP/N) combines with the word *farmer* of category N to the phrase a_n *farmer,* which belongs to the category NP. The exact nature of the way expressions are combined need hardly concern us here. Mostly, combination is just concatenation, but some syntactic fine-tuning is needed in order to take care of things like word order and agreement.

Determiners, proper names and pronouns are indexed, as the reader will have noticed. As usual, coindexing is meant to indicate the relation between a dependent (for example an anaphoric pronoun) and its antecedent. So in the short text

(3) *A_1 farmer owns a_2 donkey. The_1 bastard beats it_2*

the coindexing indicates that *the bastard* depends on *a farmer* and that *it* depends on *a donkey*. In this paper we study only the semantic aspects of the dependent / antecedent relation, but our considerations should be supplemented with a syntactic theory of the same relation, such as the Binding Theory (see e.g. Reinhart [1979], Bach & Partee [1981]). The Binding Theory rules out certain coindexings that are logically possible but syntactically impossible. Our version of Dynamic Montague Grammar is designed to answer the question how in a syntactically acceptable text a dependent manages to pick up a referent that was introduced by its antecedent; so we may restrict ourselves to the study of texts that are coindexed in a syntactically correct way.

In order to provide our little fragment of English with an interpretation we shall translate it into type theory. Expressions of a category A will be translated into terms of type $TYP(A)$. The translation of an expression, or, to be precise, the set of terms that are equivalent (given the axioms) with the translation of an expression, we identify with its meaning. Thus we can make predictions about the semantic behaviour of expressions on the basis of the logical behaviour of their translations. The function that assigns translations to expressions is defined as usual, rule-to-rule, inductively, by specifying (a) the translations of basic expressions and (b) how the translation of a complex expression depends on the translations of its parts.

To start with (b), our rule for combining the translation of a category $A/^nB$ expression with the translation of an expression of category B is always functional application. That is, if σ is a translation of the expression Σ of category $A/^nB$ and if ξ translates the expression Ξ of category B, then the translation of the result of combining Σ and Ξ is the term $\sigma\xi$.

Translations of basic expressions, to continue with (a), can be specified by simply listing them and this I'll do now. A detailed explanation will be given shortly.[10]

[10]Not all basic expressions given in the table above can be found in this list but for each item in the table an example is listed. So, e.g., the translation of *own* will be analogous

$$
\begin{array}{lll}
a_n & \rightsquigarrow & \lambda P_1 P_2 \lambda ij \exists kh(i[v_n]k \wedge P_1(v_n k)kh \wedge P_2(v_n k)hj) \\
no_n & \rightsquigarrow & \lambda P_1 P_2 \lambda ij(i = j \wedge \neg\exists khl(i[v_n]k \wedge P_1(v_n k)kh \wedge P_2(v_n k)hl)) \\
every_n & \rightsquigarrow & \lambda P_1 P_2 \lambda ij(i = j \wedge \forall kl((i[v_n]k \wedge P_1(v_n k)kl) \rightarrow \exists h\, P_2(v_n k)lh)) \\
the_n & \rightsquigarrow & \lambda P_1 P_2 \lambda ij \exists k(P_1(v_n k)ik \wedge P_2(v_n k)kj) \\
Pedro_n & \rightsquigarrow & \lambda P \lambda ij(v_n i = pedro \wedge P(v_n i)ij) \\
he_n & \rightsquigarrow & \lambda P \lambda ij(P(v_n i)ij) \\
if & \rightsquigarrow & \lambda pq \lambda ij(i = j \wedge \forall h(pih \rightarrow \exists k\, qhk)) \\
and & \rightsquigarrow & \lambda pq \lambda ij \exists h(pih \wedge qhj) \\
. & \rightsquigarrow & \lambda pq \lambda ij \exists h(pih \wedge qhj) \\
or & \rightsquigarrow & \lambda pq \lambda ij(i = j \wedge \exists h(pih \vee qih)) \\
who & \rightsquigarrow & \lambda P_1 P_2 \lambda x \lambda ij \exists h(P_2 xih \wedge P_1 xhj) \\
farmer & \rightsquigarrow & \lambda x \lambda ij(farmer\ x \wedge i = j) \\
walk & \rightsquigarrow & \lambda x \lambda ij(walk\ x \wedge i = j) \\
love & \rightsquigarrow & \lambda Q \lambda y(Q \lambda x \lambda ij(love\ xy \wedge i = j))
\end{array}
$$

In these translations we let h, i, j, k and l be type s variables; x and y are type e variables; (subscripted) P is a variable of type $TYP(VP)$; Q a variable of type $TYP(NP)$; p and q are variables of type $s(st)$; *pedro* is a constant of type e ; *farmer* and *walk* are type et constants; *love* is a constant of type $e(et)$ and each v_n is a store name.

To grasp how things work one is advised to make a few translations and by way of example I'll work out some translations in detail, making comments as I go along. I'll start with text (3).

(3) *A_1 farmer owns a_2 donkey. The_1 bastard beats it_2*

The combination *a_1 farmer* is translated by the translation of *a_1* applied to the translation of *farmer.* Some lambda-conversions reduce this to

(4) $\lambda P \lambda ij \exists kh(i[v_1]k \wedge farmer(v_1 k) \wedge k = h \wedge P(v_1 k)hj)$

and by predicate logic this is equivalent to

(5) $\lambda P \lambda ij \exists k(i[v_1]k \wedge farmer(v_1 k) \wedge P(v_1 k)kj)$.

In a completely analogous way we find that *a_2 donkey* translates as

(6) $\lambda P \lambda ij \exists k(i[v_2]k \wedge donkey(v_2 k) \wedge P(v_2 k)kj)$.

to that of *love,* the translation of it_n will be analogous (and in fact identical) to that of he_n.

And from this we derive that *own a_2 donkey* has a translation equivalent to

(7) $\lambda y \lambda ij(i[v_2]j \wedge donkey(v_2 j) \wedge own(v_2 j)y)$,

so that for *a_1 farmer owns a_2 donkey* we find

(8) $\lambda ij \exists k(i[v_1]k \wedge farmer(v_1 k) \wedge k[v_2]j \wedge donkey(v_2 j) \wedge own(v_2 j)(v_1 k))$.

Thus far everything was lambda-conversion and ordinary logic; but now we come to a reduction that is specific to our system. First, using the definition of $k[v_2]j$ (and AX3), note that the term above is equivalent to

(9) $\lambda ij \exists k(i[v_1]k \wedge farmer(v_1 j) \wedge k[v_2]j \wedge donkey(v_2 j) \wedge own(v_2 j)(v_1 j))$.

Now let us write $i[v_1, v_2]j$ for $\exists k(i[v_1]k \wedge k[v_2]j)$. Then our term reduces to

(10) $\lambda ij(i[v_1, v_2]j \wedge farmer(v_1 j \wedge donkey(v_2 j) \wedge own(v_2 j)(v_1 j))$.

A moment's reflection and an application of the Update Axiom learns us that $i[v_1, v_2]j$ means 'states i and j agree in all stores except possibly in v_1 and v_2'. Since this new notation will prove useful on many occasions we may generalize it somewhat. Let $u_1, \ldots u_n$ be store names, then by induction $i[u_1, \ldots u_n]j$ is defined to abbreviate $\exists k(i[u_1]k \wedge k[u_2, \ldots u_n]j)$. Again, by the Update Axiom the formula $i[u_1, \ldots u_n]j$ means: 'states i and j agree in all stores except possibly in $u_1, \ldots u_n$'.

The upshot of the translation process thus far is that we have associated a certain relation between context states with the sentence *a_1 farmer owns a_2 donkey.* The relation in question holds between states i and j if these states differ in maximally two of their stores, v_1 and v_2, and if the values of these stores in j are a farmer and a donkey that he owns respectively. In fact the sentence *a_1 farmer owns a_2 donkey* now has aspects that we find in assignment statements in a programming language: it assigns a farmer to v_1 and a donkey to v_2 and imposes the further constraint that the farmer owns the donkey. Of course the assignment is nondeterministic: there may be more than one farmer and one donkey in the model that satisfy the description, or there may be none.

Let's continue our translation. By a procedure that is now entirely familiar we find that *the_1 bastard beats it_2* translates as

(11) $\lambda ij(bastard(v_1 i) \wedge beat(v_2 i)(v_1 i) \wedge i = j)$.

This means that the sentence functions as a test: it denotes the set of all pairs $\langle i, i\rangle$ such that the value of store v_1 at i is a bastard that beats the value of store v_2.

We can now combine the two sentences. Sentence concatenation is symbolized with the full stop, which is assigned category $(S/S)/S$; its meaning is $\lambda pq\lambda ij\exists h(pih \wedge qhj)$: sequencing. Applying this first to (10) and then applying the result to (11) gives the translation of (3).

(12) $\lambda ij(i[v_1, v_2]j \wedge \mathit{farmer}(v_1j) \wedge \mathit{donkey}(v_2j) \wedge \mathit{own}(v_2j)(v_1j) \wedge \mathit{bastard}(v_1j) \wedge \mathit{beat}(v_2j)(v_1j))$.

We see that the relation expressed by (10) is now restricted properly by the test in (11). Moreover, we see that the discourse referents that were created by the antecedents *a₁ farmer* and *a₂ donkey* in the first sentence of (3) are now picked up by the dependents *the₁ bastard* and *it₂*.

The relation in (12) gives the meaning of text (3), but to get at the truth conditions one further step is needed. We say that a text is *true* in a context state i (in some model) if there is some context state j such that $\langle i, j\rangle$ is in the denotation of the meaning of the text. If R is the meaning of some text then we call its domain $\lambda i\exists jRij$ the set of all states in which the text is true, its *content.* The step from meaning to truth parallels a similar step taken in DRT: a discourse representation structure is true if it has a verifying embedding.

Clearly the content of (3) is

(13) $\lambda i\exists j([v_1, v_2]j \wedge \mathit{farmer}(v_1j) \wedge \mathit{donkey}(v_2j) \wedge \mathit{own}(v_2j)(v_1j)) \wedge \mathit{bastard}(v_1j) \wedge \mathit{beat}(v_2j)(v_1j))$.

But this can be simplified considerably, for it is equivalent to (14). Quantifying over a state has the effect of binding unselectively the contents of all stores in that state.

(14) $\lambda i\exists xy(\mathit{farmer}\ x \wedge \mathit{donkey}\ y \wedge \mathit{own}\ yx \wedge \mathit{bastard}\ x \wedge \mathit{beat}\ yx)$.

To show the equivalence, we may abbreviate the conjunction $\mathit{farmer}\ x \wedge \mathit{donkey}\ y \wedge \mathit{own}\ yx \wedge \mathit{bastard}\ x \wedge \mathit{beat}\ yx$ as φ for the moment. Suppose (13) holds for some i. Then there are objects, namely the values of v_1j and v_2j, that satisfy φ. It follows that (14) is true in i. Conversely, suppose that (14) is true for some i. Then there are d_1 and d_2 that satisfy φ. By the Update

Axiom there is a j, differing from i at most in stores v_1 and v_2, such that $v_1 j = d_1$ and $v_2 j = d_2$. Hence $\exists j(i[v_1, v_2]j \wedge [v_1 j/x, v_2 j/y]\varphi)$ holds, so that (13) is true in i.

The principle underlying the equivalence of (13) and (14) is important enough to state it in full generality. I call it the Unselective Binding Lemma.

UNSELECTIVE BINDING LEMMA. Let $u_1, \ldots, u_n$ be store names, let $x_1, \ldots, x_n$ be distinct variables, let φ be a formula that does not contain j and let

$$[u_1 j/x_1, \ldots, u_n j/x_n]\varphi$$

stand for the simultaneous substitution of $u_1 j$ for x_1 and ... and $u_n j$ for x_n in φ, then:

(i) $\exists j(i[u_1 \ldots u_n]j \wedge [u_1 j/x_1, \ldots, u_n j/x_n]\varphi)$ is equivalent with $\exists x_1 \ldots x_n \varphi$

(ii) $\forall j(i[u_1 \ldots u_n]j \rightarrow [u_1 j/x_1, \ldots, u_n j/x_n]\varphi)$ is equivalent with $\forall x_1 \ldots x_n \varphi$

I omit the proof of this lemma since it is an obvious generalization of the proof of the equivalence of (13) and (14) given above ((ii) follows from (i) of course).

We see that (3) is true in a context state if and only if it is true in all other context states, the content of (3) either denotes the empty set or the set of all states, depending on whether there is a farmer who owns a donkey in the model and whether the bastard beats it. But this does not hold for all texts; let's consider sentence (15) for instance.

(15) *He₁ beats a₂ donkey*

The pronoun *he*$_1$ cannot be interpreted as dependent on some antecedent provided by the text in this case. And so it must be interpreted deictically, its referent must be provided by the context. Now let us look at the meaning and the content of (15), given in (16) and (17) respectively.

(16) $\lambda i(i[v_2]j \wedge \mathit{donkey}(v_2 j) \wedge \mathit{beat}(v_2 j)(v_1 i))$

(17) $\lambda i \exists x(\mathit{donkey}\ x \wedge \mathit{beat}\ x(v_1 i))$

We see that (15) is true only in contexts that provide a referent for the deictic pronoun he_1. The reader may wish to verify that texts containing a proper name or a definite noun phrase that lacks an antecedent are treated likewise.

If a text contains an indefinite right at the start, the discourse referent created by that indefinite will live through the entire text and can be picked up by a dependent at any point. But some discourse referents have only a limited life span. In order to see how our system can account for this, let's work out the translation of the following celebrated example.

(18) *Every$_1$ farmer who owns a$_2$ donkey beats it$_2$*

First we apply the translation of *who,* $\lambda P_1 P_2 \lambda x \lambda ij \exists h(P_2 xih \wedge P_1 xhj)$, which gives a generalized form of conjunction, to the VP *own a$_2$ donkey.* The result, after conversions, is

(19) $\lambda P \lambda x \lambda ij \exists h(Pxih \wedge h[v_2]j \wedge \mathit{donkey}(v_2 j) \wedge \mathit{own}(v_2 j)\mathrm{x})$.

Applying this to the translation of *farmer* results in

(20) $\lambda x \lambda ij(\mathit{farmer}\ x \wedge i[v_2]j \wedge \mathit{donkey}(v_2 j) \wedge \mathit{own}(v_2 j)\mathrm{x})$,

the translation of *farmer who owns a$_2$ donkey.* Next we combine this result with the translation of the determiner *every$_1$*. This gives the following term:

(21) $\lambda P \lambda ij(i = j \wedge \forall l((i[v_1, v_2]l \wedge \mathit{farmer}(v_1 l) \wedge \mathit{donkey}(v_2 l) \wedge \mathit{own}(v_2 l)(v_1 l)) \rightarrow \exists h\, P(v_1 l)lh))$.

Finally a combination with the VP *beat it$_2$* yields:

(22) $\lambda ij(i = j \wedge \forall l((i[v_1, v_2]l \wedge \mathit{farmer}(v_1 l) \wedge \mathit{donkey}(v_2 l) \wedge \mathit{own}(v_2 l)(v_1 l)) \rightarrow \mathit{beat}(v_2 l)(v_1 l))$,

which by the Unselective Binding Lemma is equivalent to

(23) $\lambda ij(i = j \wedge \forall xy((\mathit{farmer}\ x \wedge \mathit{donkey}\ y \wedge \mathit{own}\ yx) \rightarrow \mathit{beat}\ yx))$.

The translation of a universal sentence thus acts as a *test* ; it cannot change the value of any store but can only serve to rule out certain continuations of the interpretation process. The discourse referents that were introduced by the determiners *every$_1$* and a_2 had a limited life span. Their role was essential in obtaining the correct translation of the sentence, but once this translation was obtained they died and could no longer be accessed. There are more operators that behave in the way of $every_n$ in this respect: in the fragment under consideration the determiner no_n , and the words *if* and *or* have a very similar behaviour.

References

[1] Bach, E. and Partee, B.H.: 1981, Anaphora and Semantic Structure, *CLS* **16**, 1-28.

[2] Barwise, J.: 1981, Scenes and Other Situations, *The Journal of Philosophy*, **78**, 369-397.

[3] Barwise, J.: 1987, Noun Phrases, Generalized Quantifiers and Anaphora, in P. Gärdenfors (ed.), *Generalized Quantifiers,* Reidel, Dordrecht, 1-29.

[4] Barwise, J. and Cooper, R.: 1981, Generalized Quantifiers and Natural Language, *Linguistics and Philosophy* **4**, 159-219.

[5] Barwise, J. and Perry J.: 1983, *Situations and Attitudes*, MIT Press, Cambridge, Massachusetts.

[6] Barwise, J and Etchemendy, 1987, *The Liar: An Essay on Truth and Circularity,* Oxford University Press.

[7] Bäuerle, R., Egli, U., and Von Stechow, A. (eds.): 1979, *Semantics from Different Points of View,* Springer, Berlin.

[8] Chierchia, G.: 1990, Intensionality and Context Change, Towards a Dynamic Theory of Propositions and Properties, manuscript, Cornell University.

[9] Church, A.: 1940, A Formulation of the Simple Theory of Types, *The Journal of Symbolic Logic* **5**, 56-68.

[10] Gabbay, D. and Günthner, F. (eds.): 1983, *Handbook of Philosophical Logic*, Reidel, Dordrecht.

[11] Gallin, D.: 1975, *Intensional and Higher-Order Modal Logic*, North-Holland, Amsterdam.

[12] Goldblatt, R.: 1987, *Logics of Time and Computation,* CSLI Lecture Notes, Stanford.

[13] Groenendijk, J. and Stokhof, M.: 1989, Dynamic Predicate Logic, ITLI, Amsterdam. To appear in *Linguistics and Philosophy.*

[14] Groenendijk, J. and Stokhof, M.: 1990, Dynamic Montague Grammar, in L. Kalman and L. Polos (eds.), *Papers from the Second Symposium on Logic and Language,* Akadmiai Kiado, Budapest, 3-48.

[15] Harel, D.: 1984, Dynamic Logic, in Gabbay & Günthner [1983], 497-604.

[16] Heim, I.: 1982, *The Semantics of Definite and Indefinite Noun Phrases,* Dissertation, University of Massachusetts, Amherst. Published in 1989 by Garland, New York.

[17] Henkin, L.: 1963, A Theory of Propositional Types, *Fundamenta Mathematicae* **52**, 323-344.

[18] Janssen, T.: 1983, *Foundations and Applications of Montague Grammar,* Dissertation, University of Amsterdam. Published in 1986 by CWI, Amsterdam.

[19] Kamp, H.: 1981, A Theory of Truth and Semantic Representation, in J. Groenendijk, Th. Janssen, and M. Stokhof (eds.), *Formal Methods in the Study of Language, Part I,* Mathematisch Centrum, Amsterdam, 277-322.

[20] Karttunen, L.: 1976, Discourse Referents, in J. McCawley (ed.), *Notes from the Linguistic Underground, Syntax and Semantics 7,* Academic Press, New York.

[21] Lewis, D.: 1979, Score Keeping in a Language Game, in Bäuerle, Egli & Von Stechow [1979], 172-187.

[22] Montague, R.: 1973, The Proper Treatment of Quantification in Ordinary English, reprinted in Montague [1974], 247-270.

[23] Montague, R.: 1974, *Formal Philosophy*, Yale University Press, New Haven.

[24] Muskens, R.A.: 1989a, Going Partial in Montague Grammar, in R. Bartsch, J.F.A.K. van Benthem and P. van Emde Boas (eds.), *Semantics and Contextual Expression,* Foris, Dordrecht, 175-220.

[25] Muskens, R.A.: 1989b, *Meaning and Partiality,* Dissertation, University of Amsterdam.

[26] Muskens, R.A.: to appear, Tense and the Logic of Change.

[27] Muskens, R.A.: in preparation, Logical Semantics for Programming Languages.

[28] Orey, S.: 1959, Model Theory for the Higher Order Predicate Calculus, *Transactions of the American Mathematical Society* **92**, 72-84.

[29] Pratt, V.R.: 1976, Semantical Considerations on Floyd-Hoare Logic, *Proc. 17th IEEE Symp. on Foundations of Computer Science,* 109-121.

[30] Reinhart, T.: 1979, Syntactic Domains for Semantic Rules, in F. Günthner and S. Schmidt (eds.), *Formal semantics and Pragmatics for Natural Languages,* Reidel, Dordrecht.

[31] Rooth, M.: 1987, *Noun Phrase Interpretation in Montague Grammar, File Change Semantics, and Situation Semantics,* in P. Gärdenfors (ed.), *Generalized Quantifiers,* Reidel, Dordrecht, 237-268.