Group	: Computational Linguistics
Topic	: General

Title	: **Types**
Author	: Jan Odijk

Doc.Nr.	: 0353
Date	: September 18, 1989
Status	: informal
Supersedes	: -
Distribution	: Project
Clearance	: Project
Keywords	: Type system, sorts, semantic component

Institute for Perception Research

Contents

1 Introduction

This document deals with types in Rosetta. A proposal is made for a type-system that is to be used in Rosetta. In section 2 some formal aspects of the type-system are discussed. Everything proposed there is fairly standard, I think, except for one part: the system proposed allows for a simple way to deal with cross-classifications. Section 3 deals with the substantive part of the types: Which types exist, and how do we determine types? A strategy to solve this problem is proposed, and some examples are given. Section 4 discusses some aspects of robustness and implementation.

The status of this paper is exploratory and the proposal tentative. The system as proposed here fits in nicely with (a restricted version of) the proposal by Joep Rous concerning the semantic component (doc. R0401).

2 Types

A *type* is an object associated to set of meanings. The set of meanings associated to a type is called the *domain* of the type.

Both basic meanings (e.g. meanings of single words and idioms) and non basic meanings (e.g. the meaning of an NP or a sentence) are members of a *type domain* .

I'll notate type domains as D_{type}. The meaning of some expression α is represented as $\|\alpha\|$.

Examples

D_{human} = { ‖man‖, ‖vrouw‖, ‖minister‖, ... }
$D_{abstract}$ = { ‖kunst‖, ‖tijd‖,.. }
D_{mass} = { ‖vlees‖, ‖goud‖, ... }
$D_{question}$ = { ‖of hij ziek is‖, ‖Is dat correct?‖, ‖wat heeft hij gedaan?‖,...}
D_{plural} = { ‖de mannen‖, ‖enkele vrouwen‖, ‖sommige auto's‖, ...}

The types from Montague grammar can be represented in this way as well. The type t has domain D_t = { 0,1 }; the type e equals the set of all entities. The type c suggested for the meaning of *wheather-it* in doc. R0308 would be a type consisting of exactly one member.

Sometimes a distinction is made between *types* and *sorts*. Types are restricted to the types e, t and complex types built out of these elements, whereas *sorts* further subdivide types (in particular the type e). This distinction is not made here. Some of the types that will be proposed correspond clearly to *sorts*, e.g. *human, animate*

etc. Others, however, will probably correspond to complex types formulated in terms of *e*'s and *t*'s, e.g. *question*. In the system proposed here both have the same status, though they probably will differ in their place on a hierarchy to be defined below.

The domains of types of basic meanings are large, but finite sets. The domains of types of nonbasic meanings are infinite sets. For these reasons sets of meanings cannot be used directly in a real system. For basic meanings it would be impractical because the sets are very large, for nonbasic meanings it is in principle impossible, since the sets are infinite. Instead *types* are used to achieve the desired effects.

2.1 Cross classification

Types are used in *(enclosed) type descriptions* to indicate what type domains a meaning is a member of. Enclosed type descriptions are defined by means of the following syntax:

<enclosed type description> →'<' <type description> '>'

<type description> →<signed type> {<type description>}

<signed type> →['∼'] <type>

Examples

< animate >

< ∼animate >

‖de mannen‖ : < human plural masculine>

‖handdoek‖: < concrete horizontal>

‖zwemt de man‖ : < question activity >

where *horizontal* represents the class of meanings that can function as the first argument of the meaning of the verb *liggen*, but not of *zitten* or *staan*, and where *activity* is an *aktionsart*.

Every meaning has an enclosed type description. If a certain meaning M has an enclosed type-description of the form $< \tau_1..\tau_n >$ (where τ_i is a signed type), then this is interpreted in such a way that M is a member of $D_{\tau_1} \cap .. \cap D_{\tau_n}$. The interpretation of $\sim\tau$ (τ a type) is: a certain meaning can have the specification $\sim\tau$ if it is a member of the complement of D_τ. The set relative to which the complement is taken (the universe) is here the set of all meanings. It might perhaps be better to take some other set as the universe. For discussion, see below.

In this way cross-classifications can be made in a systematic way. These cross-classifications are absolutely necessary to be able to make a realistic system. Once the system is fixed, it might be possible to do away with the cross classifications by partitioning the set of meanings in sufficiently small subsets, though this probably will not serve any purpose.

2.2 Hierarchy

We define a relation $\leq$ on types (actually: signed types, see below), as follows:

a $\leq$ b iff $D_a \subseteq D_b$

Analogously, the relations $<$, $>$ and $\geq$ can be defined.

Since the actual system cannot deal with real sets, a partial ordering of types w.r.t the operator $\leq$ must be specified in a real system. The $\leq$ relation must be computed on the basis of such a specification.

Example The following might be parts of such a partial ordering:

human < animate < concrete
drink < fluid < concrete
concrete < e
weapon < instrument < concrete

2.3 Type requirements

Every function specifies for each of its arguments to what type it must belong. This is specified by means of a *type requirement* for each argument.

A type requirement is defined by the following syntax:[1]

<req-descr> →<req-descr> '|' <req-descr>
<req-descr> →<signed type> { <signed type> }
<signed type> →['∼'] <type>

Examples

animate
animate | t
∼animate | t
human feminine | weapon

It is possible to make enclosed requirement descriptions from req-descr's in the following manner:

<enclosed req-descr> →'< [<req-descr>['-' <req-descr> ['-' <req-descr>]]] '>'

[1] The syntax given is arbitrary to a certain extent, and it can be extended where necessary, e.g. one might perhaps want to allow requirement descriptions of the form *horizontal (weapon — cloths)*, etc.

These can be used to specify the type requirements for each argument (0,1,2,3) of an argument-taking meaning.

Compatibility of a type description τ with a requirement description ρ is defined as follows:

alternatives if $\rho{=}\rho_1|\rho_2$ (ρ_1 and ρ_2 requirement descriptions), and τ_1 is a type description, then τ_1 is compatible with ρ iff τ_1 is compatible with ρ_1 or with ρ_2

concatenation if $\tau = \tau_1..\tau_n$, $\rho = \rho_1..\rho_m$ (τ_i and ρ_i signed types for all i), then τ is compatible with ρ iff $\forall_{\rho_i}\exists_{\tau_j}\tau_j \leq \rho_i$

Example Given the partial ordering sketched above, the following holds:

concatenation < plural human masculine > is compatible with *animate*, since *animate* $\geq$ *human*

alternation < human > is compatible with *abstract* | *concrete* , since *human* $\leq$ *concrete*

It appears to be more convenient to define '$\sim$' as creating a complement w.r.t. the *root type* than w.r.t all types. The *root type* for some type τ is defined as follows:

Roottype(τ) = τ_1 iff $\tau_1 \geq \tau$ and there is no type τ_2 such that $\tau_2 > \tau_1$

In words, the roottype of some type τ is the type occurring as the root of the hierarchy tree that τ is a member of. This will give the result that e.g. the type description $\sim$animate is equivalent to nonanimate entities (< $\sim$animate entity >), so that e.g. meanings of type *t* are not included, which appears to be the intention in most cases..

Other interpretations of '$\sim$' can be imagined as well. If no consistent, intuitively appealing definition can be found, then it might perhaps be better to do away with it, because it will then certainly create errors, or we might replace it by a difference operator.

It might perhaps be convenient to define a special type that is compatible with any type and one (perhaps the same) that any type is compatible with (e.g. *anytype*).

2.4 How to use types in the semantics?

The idea is that any argument-taking meaning has an enclosed type requirement as part of its semantic specification and that any meaning has a type description. Combining a argument-taking meaning with its arguments entails:

1 checking whether the type description of each argument is compatible with the relevant type requirement as specified in the argument taking meaning.

2 (if the conditions under **1** are satisfied) computing a new type for the result of the application of this rule.

Thus, the operation working on these types for an n-ary syntactic rule is an n-ary function from an n-tuple of feature bundles to a set of feature bundles.

If there is a type mismatch, no new type will be computed for the result of the application of the relevant rule. If no case passes the type compatibility test, no resulting type will occur, and the associated semantic derivation tree is (semantically) ill-formed.

In certain cases certain information from arguments or argument taking meanings must be retained. A concrete example might be VP-modifying adverbs. For such adverbs it might be necessary that we compute a type for the VP and store it in the featurebundle associated to the rule forming a VPPROP (or CLAUSE) in order to be able to later compare the type requirement of the adverb with the type of the VP. This would be artificial, but it would be a direct consequence of the treatment of such adverbs of which we knew in advance that it was semantically probably not fully correct.

A second case where this might be relevant concerns such words as *krijgen* 'gedaan krijgen', *laten* with complements such as *hem in de kast*, *hem in de tuin* etc., where it is perhaps necessary to record the type of the PP (path, location, etc.) occurring in this complement. I am not sure about this yet, but it should be investigated. At the moment, these verbs have been given (somewhat ad-hoc) type requirements combining a truth value with the type of the PP (*tlocation*, *tpath*, etc.).

Also the treatment of certain kinds of adjuncts as proposed in doc. R0308 might make it necessary to retain certain information of the head.

We should also investigate whether there are other cases of this kind.

3 How to determine types?

A more difficult problem than the formal part of the type system is the substantive part: How to determine which types should be assumed? This is a very difficult problem, because it is possible to take subsets from the sets of meanings in an infinite number of ways. The problem is: how do we select a choice of subsets that has linguistic significance, and that is useful for disambiguation purposes?

My suggestion would be to tackle this problem in the following manner:

1 First, tentatively make type-requirements for all arguments of meanings of many (ideally: all) argument-taking words. Specify the hierarchy required for this system, and define the types postulated precisely (add tests etc.)

2 Try to fill all meanings of nouns (more generally: potential heads of argument phrases) etc. for their types, using the hierarchy of types postulated. The hierarchy of types will consist of a set of trees: Pick for each tree (and each

subtree, etc.) the lowest leaf that is appropriate. Form a type description by putting all these types in a list.

As an experiment, I started to do the first part of this strategy for some of the verbs in the current dictionaries that I met anyway doing other dictionary activities. This led to a first list of types and an associated hierarchy. This list is supplied in appendix A. The actual list in the dictionary has been extended slightly already.

While filling these type requirements I found out that it was very important to have a clear idea of what types the rules of the current Rosetta3 system would yield. Since I did not have very clear ideas about this, this will probably have yielded inconsistent fillings. To avoid this in the future, I made a first proposal as to what kind of types are yielded by the final rules of subgrammars. They are listed in appendix B.

4 Implementation and Robustness

4.1 Robustness and Metaphorical Uses

Of course, the Rosetta system must be robust enough to deal with cases that are not foreseen. Among these there might be simply semantically illformed sentences, but also all kinds of metaphorical uses.

One way to guarantee this robustness, would be to give the type check the status of an ordering imposing mechanism: the type check never filters, but it orders trees according to plausibility. In this way the system will always be able to cope with metaphorical uses, etc.

It is my opinion that this should *not* be done, and that the type check should be a real filter. When considering the cases in Van Dale, one finds that cases of metaphors etc. are limited to very special cases. If metaphors are to be dealt with in a principled way, then a theory of metaphors should be built in. Since there probably is no time (nor anyone who would like to do it) to develop such a theory, I think that cases of metaphor should be treated by general robustness measures. This is especially the case since Van Dale already distinguishes several 'metaphorical' uses as separate meanings.

Also such sentences as *Dieren kunnen niet spreken* etc. should be dealt with by means of robustness measures.

One way to implement robustness (suggested to me by Harm Smit), is to go up higher in the hierarchy step by step for each requirement description that is violated, if no semantic derivation tree would be well-formed otherwise, until a semantic derivation tree is well-formed. Another would be to specify such changes in a list explicitly (This could in fact be a first, very primitive, theory of metaphors). E.g. If *human*

is required and *animate* is found, then *animate* may be lowered to *human* to achieve the effect of *personalization*, one of a very limited number of possible metaphors or related devices. Of course, also both methods can be used.

4.2 Implementation

The implementation of the system proposed can be done in many ways. The $\leq$-relation might be computed on the basis of a hierarchy specification and represented in an incidence matrix, or in a tree.

The actual rules doing the type-check can be formulated in the rules that will be used in the semantic component (see the document by Joep Rous).

As far as efficiency is concerned, the following should be noted. Let us consider a typical case of a verb taking 3 arguments, which is m-fold ambiguous. Let us assume that the arguments themselves are also ambigous, resp. *i,j,k*-fold ambigous.

In this situation (if no other relevant properties hold) the current implementation of M-Generator would require in the worst case $i.j.k.m$ compatibility tests. If $i = j = k = m = 10$ this would imply 10^4 comparisons.

However, if the implementation is done differently, in the way sketched very informally below, the number of comparisons can be reduced to $(i + j + k).m$ comparisons (i.e. 300 if $i = j = k = m = 10$).

The implementation should proceed as follows. First all substituents (arguments) are evaluated, for all their meanings. Next the pre-action associates corresponding variables with the relevant semantic features. Now a set of such semantic features for each variable exists.

The derivation proceeds by evaluating each meaning of the verb separately. For each of these meanings a type check is done for each argument with all alternative semantic features. This will (in the worst case) require k comparisons for the third argument, j comparisons for the second and i for the first argument. If some test yields incompatibility (in the worst case this is never so), the relevant variable and its associated substituent are thrown out. This yields in the worst case $(i + j + k)$ comparisons.

This process is repeated for each meaning of the verb, i.e. m times, so that in total $(i + j + k).m$ comparisons need be made.

This implementation does presuppose that there are no interdependencies between arguments w.r.t. their type restrictions. I think that that is indeed never the case, but if such cases are found, this should be reported, for it probably makes this implementation impossible.

It should be investigated whether it is possible to make an implementation as sketched here and whether it can be incorporated into the rest of M-Generator, and also whether it is necessary.

A Types as occurring in the dictionaries now

The types mentioned occur in type requirements listed in the dutch dictionaries as comment in the notation proposed above. (search for the string {<, or use the edit command file [odijkje.mrules]typedt.edt to get them into a file). The first subsection contains a list of all types encountered, the second subsection contains the hierarchy of types, the last subsection lists the meaning keys with meaning description and their postulated typing.

A.1 List of types

The following additional conventions have been adopted:

***** an asterisk indicates that the typing should be reconsidered

” Quotes are used to mention keys directly if no characterization of the relevant class could be found.

non the prefix ∼ has not been used yet. The prefix *non* or *in* has been used instead

loc,t stands for the type *tlocation*, see the discussion above w.r.t *krijgen*.

The list contains type names, plus their number of occurrences (no number means that they occurred once). The list is up to date until July 14, 1989. After this date new type requirements have been specified, which are not listed in this document.

Index

A.2 Hierarchy of postulated types

entity	concrete	animate	human	
			feminine	
			masculine	
		possessable		
			bel,telefoon,zoemer	
			builtobject	
			celestialbody	
			cloths	
			fluid	drink
			food	
			fuel	
			geounit	
			medicine	
			musicalinstrument	
			ship	
			suitcase	
			transportmeans	
	abstract	'koers'		
		infocontainer		
		'deling'		
		exam		
		game		
		line		
		music		
		punishment		
		taste		
c				
t				
loc,t				
question				
location				
path	sourcepath			
plural				
spacemeasure				
timeref				

A.3 Meaning keys and their type requirements

mkey	#	meaning descr	typing
m_aV_0002_beveel_aan	s1	"aanraden"	<human-entity-human>
m_aV_0001_pers_af	s1	"dwingen te geven"	<human-human>
m_aV_0002_pers_af	s2	"geheel en al persen"	<human-cloths>
m_aV_0001_betref	s1	"aangaan"	<entity-human>
m_aV_0002_betref	s2	"handelen over"	<entity-entity>
m_aV_0001_ga	s1	"zich verplaatsen,begeven"	<concrete-path>
m_aV_0002_ga	s1	"vertrekken, weggaan"	<concrete-timeref>
m_aV_0013_ga	s2	"begrepen zijn in,passen"	<concrete-location>
m_aV_0004_ga	s3	"<+ onbep. wijs>beginnen te"	<c|entity-t>
m_aV_0005_ga	s5	"rinkelen"	<?bel,telefoon,zoemer>
m_aV_0014_ga	s1	"beheren"	<human-entity>
m_aV_0015_ga	s2	"tot onderwerp hebben"	<*infocontainer-entity>
m_aV_0001_hebben	s1	"bezitten"	<human-possessable>
m_aV_0005_hebben	s5	"in de genoemde toestand verkeren"	<animate-t|loc,t>
m_aV_0014_hebben	s1	"nut ondervinden van"	<animate-entity|t>
m_aV_0001_help	s1	"bijstaan"	<animate-entity|t-animate>
m_aV_0002_help	s2	"verzorgen"	<human-human>
m_aV_0006_help	s6	"baten"	<nonanimate|t>
m_aV_0001_neem_in	s1	"mbt. geneesmiddelen"	<human-medicine>
m_aV_0002_neem_in	s2	"mbt. een plaatsruimte"	<entity-spacemeasure>
m_aV_0003_neem_in	s3	"veroveren"	<human-geounit|builtobject>
m_aV_0006_neem_in	s6	"aan boord nemen"	<ship|human-fuel|food|fluid>
m_aV_0007_neem_in	s7	"inkorten"	<human-cloths>
m_aV_0101_stort_in	s1	"neerstorten"	<nonanimate>
m_aV_0102_stort_in	s2	"een inzinking krijgen"	<human>
m_aV_0001_isoleer	s1	"afzonderen"	<human-human>
m_aV_0002_isoleer	s2	"uit een geheel halen"	<human-nonhuman>
m_aV_0001_kleef	s1	"vast blijven zitten"	<concrete-concrete>
m_aV_0002_kleef	s2	"<fig.> onverbrekelijk verbonden zijn met"	<abstract-abstract>
m_aV_0002_koester	s2	"vertroetelen"	<human-animate>
m_aV_0003_koester	s3	"uit waardering beschermen"	<human-entity>
m_aV_0013_laat	s13	"bij zijn dood nalaten"	<human-concrete-human>
m_aV_0014_laat	s14	"afstaan"	<human-entity-human>

mkey	#	meaning descr	typing
m_aV_0001_luister	s1	"horen om iets te vernemen"	<human-entity|question>
m_aV_0003_luister	s3	"aandacht schenken aan"	<human-human>
m_aV_0004_luister	s4	"gehoorzamen aan"	<human-human>
m_aV_2101_moet	s1	"mogen, believen"	<human-human>
m_aV_0101_naai	s1	"vasthechten (van boeken)"	<human-inanimate>
m_aV_0103_naai	s3	"benadelen"	<human-human>
m_aV_0001_noteer	s1	"aantekenen"	<human-t>
m_aV_0002_noteer	s2	"bepalen, opgeven (van koersen)"	<human-'koers'>
m_aV_0001_omhels	s1	"omarmen"	<human-human>
m_aV_0002_omhels	s2	"<fig.>aannemen"	<human-abstract>
m_aV_0003_omhels	s3	"omvatten"	<nonanimate-entity>
m_aV_0003_onthaal	s3	"<fig.>vergasten op"	<human-human-abstract>
m_aV_0001_ontmoet	s1	"onvoorzien tegenkomen"	<human-human>
m_aV_0002_ontmoet	s2	"volgens afspraak treffen"	<human-human>
m_aV_0003_ontmoet	s3	"ondervinden"	<human-abstract>
m_aV_0004_ontmoet	s4	"<wisk.> (van lijnen)"	<line-line>
m_aV_0101_open	s1	"openmaken"	<animate-concrete>
m_aV_0103_open	s3	"openstellen"	<animate-abstract>
m_aV_0002_ga_op	s2	"mbt. de zon"	<celestialbody>
m_aV_0004_ga_op	s4	"examen afleggen"	<human-exam>
m_aV_0005_ga_op	s5	"opgegeten/opgedronken worden"	<food|drink>
m_aV_0006_ga_op	s6	"juist zijn"	<abstract>
m_aV_0007_ga_op	s7	"in beslag genomen worden"	<human-abstract>
m_aV_0008_ga_op	s8	"mbt. delingen"	<deling>
m_aV_0009_ga_op	s9	"in elkaar overgaan"	<entity-entity>
m_aV_0101_hang_op	s1	"in de hoogte hangen"	<human-inanimate>
m_aV_0102_hang_op	s2	"ter dood brengen"	<human-human>
m_aV_0101_houd_op	s1	"omhooghouden"	<human-concrete>
m_aV_0102_houd_op	s2	"verdedigen"	<human-abstract>

mkey	#	meaning descr	typing
m_aV_0001_pak	s1	" tevoorschijn halen"	<animate-concrete-sourcepath>
m_aV_0002_pak	s2	" vastnemen"	<animate-entity
m_aV_0003_pak	s3	" betrappen"	<human-human>
m_aV_0004_pak	s4	"inpakken"	<human-suitcase>
m_aV_0005_pak	s5	"gebruik maken van"	<human-transportmeans>
m_aV_0006_pak	s6	"mbt. drank"	<human-drink>
m_aV_0008_pak	s8	"benadelen"	<human-human-entity>
m_aV_0010_pak	s10	"seksueel gebruiken"	<human-human> ?<masculine-feminine>
m_aV_0013_pak	s13	"<sport, voetbal> mishandelen"	<human-human>
m_aV_0101_proef	s1	"een smaak gewaarworden"	<human-food|taste>
m_aV_0102_proef	s2	"bemerken, bespeuren"	<human-abstract>
m_aV_0002_redeneer	s2	"gevolgtrekkingen afleiden"	<human>
m_aV_0003_redeneer	s3	"argumenteren, redetwisten"	<human plural>
m_aV_0001_reinig	s1	"ontdoen van vuil"	<human-nonhuman>
m_aV_0002_reinig	s2	"<relig.>"	<human-human>
m_aV_0102_rem	s2	"<fig.>, vertragen, stoppen"	<t-abstract>
m_aV_0001_roof	s1	"met geweld wegnemen"	<human-nonhuman>
m_aV_0003_roof	s3	"gewelddadig wegvoeren"	<human-human>
m_aV_0001_slik	s1	"innemen"	<human-medicine>
m_aV_0002_slik	s2	"accepteren"	<human-abstract|t>
m_aV_0001_speel	s1	"zich (met een spel) vermaken"	<animate-game>
m_aV_0003_speel	s3	"bespelen"	<human-musicalinstrument>
m_aV_0004_speel	s4	"uitvoeren"	<human-music>
m_aV_0006_speel	s6	"van invloed, belang zijn"	<abstract>
m_aV_0001_spot	s1	"zich met scherts uiten"	<human>
m_aV_0003_spot	s3	"zich niet storen aan"	<human-abstract>
m_aV_0009_sta	s9	"geist worden"	<punishment-*abstract>

mkey	#	meaning descr	typing
m_aV_0001_stroom	s1	"met kracht vloeien"	<fluid-path>
m_aV_0002_stroom	s2	"<fig.>in groten getale komen/gaan"	<nonhuman nonfluid entity-path>
m_aV_0003_stroom	s3	"zich in groten getale voortbewegen"	<human-path>
m_aV_0001_struikelen	s1	"het evenwicht verliezen en evt. vallen"	<human-concrete>
m_aV_0003_struikelen	s3	"<fig.>ten val komen"	<human-abstract>
m_aV_0005_struikelen	s5	"<fig.> (in grote hoeveelheden) aantreffen"	<human-plural entity>
m_aV_0003_stuit	s3	"aantreffen"	<human-concrete>
m_aV_0004_stuit	s4	"<fig.>geconfronteerd worden"	<human|abstract-abstract>
m_aV_0101_stuit	s1	"tegenhouden"	<human-concrete>
m_aV_0102_stuit	s2	"<fig.> tegenhouden"	<human-abstract>

B Types yielded by certain semantic rules

The following list is a first, tentative (and incomplete) proposal for a specification of the types that should be yielded by functions associated to IL-rules.

Index