CHAPTER ONE: FUNCTIONAL TYPE THEORY

1.1. Compositionality

We have given a compositional semantics for predicate logic, a semantics in which the meaning of any complex expression is a function of the meanings of its parts and the way these meanings are put together.
In predicate logic, complex expressions are formulas, and their parts are formulas, terms (constants or variables), n-place predicates, and the logical constants.

As is well known to anybody who has gone through lists of exercises of the form: 'translate into predicate logic', predicate logic does a much better job on representing the meanings of various types of natural language **sentences**, than it does on representing the meanings of their parts.

Take for example the following sentence:

(1) Some old man and every girl kissed and hugged Ronya.

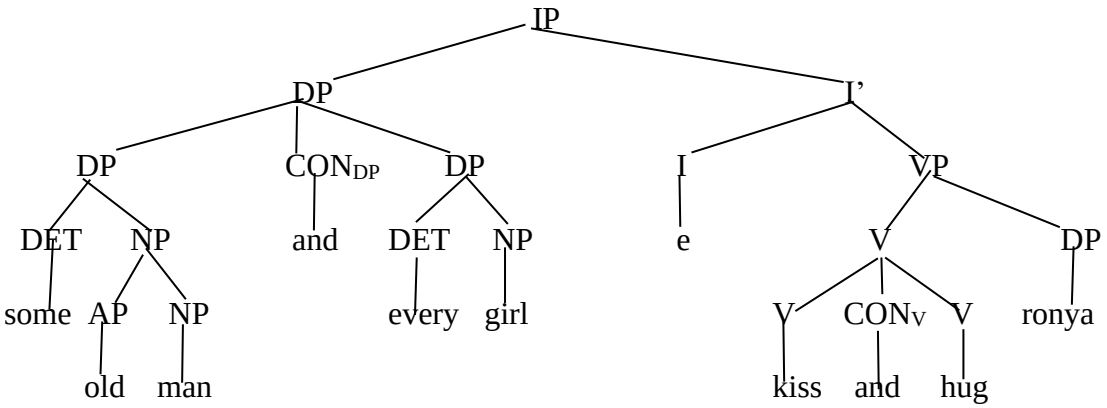
With the one-place predicate constants MAN, OLD, GIRL and the two-place predicate constants KISSED and HUGGED, we can quite adequately represent (1) in predicate logic as (1'):

(1') $\exists x[MAN(x) \wedge OLD(x) \wedge KISSED(x,RONYA) \wedge HUGGED(x,RONYA)] \wedge$
$\forall y[GIRL(y) \rightarrow KISSED(y,RONYA) \wedge HUGGED(y,RONYA)]$

The problem is that without an explicit syntax and a compositional semantics, representing (1) as (1') is a completely creative exercise, and it shouldn't be.

What we really need is a semantics that gives us the right semantic representation for sentence (1) (and hence the right meaning), and that tells us at the same time how this semantic representation is related to the meanings of the parts of (1).
Our representation (1') has various parts (like $\rightarrow$) that don't correspond to any part of the sentence;
some parts of the representation occur twice in different forms, while in the sentence there is only one part (*kissed and hugged Ronya*);
the representation contains sentence conjunction $\wedge$, whereas the sentence contains DP conjunction and verb conjunction;
OLD is separated from MAN in the representation, which is motivated by what the meaning should be, but nothing explains how it got separated, etc.
What we really need is a logical language in which we can represent the meaning of sentence (1) and the meanings of its parts:

We will assume a syntax along the following lines (invent your own syntax if you don't like this. The only important thing here is the constituent structure).
I will have a V and an I node but for simplicity not show V to I raising. Also, I ignore the syntax and semantics of tense here.



SOME + OLD + MAN + AND + EVERY + GIRL + KISSED + AND + HUGGED + RONYA
OLD MAN + EVERY GIRL + KISSED AND HUGGED
SOME OLD MAN + KISSED AND HUGGED RONYA
SOME OLD MAN AND EVERY GIRL
SOME OLD MAN AND EVERY GIRL KISSED AND HUGGED RONYA

The theory we will use to represent these meanings and the operations to put them together to give the correct sentence meanings is functional type theory, which is based on the two basic principles of
function-argument application (**functional application**) and
function definition (**lambda abstraction**).
Such a theory we will now formulate.

1.2. The syntax of functional type logic.

We will start by specifying the language of functional type logic and its semantics. For compactness sake, I will give the whole language, including the lambda-operator and its semantics, but we will ignore that part till later.
The language of functional type logic gives us a rich theory of functions and arguments.

Our theory of functions is **typed**.
This means that for each function in the theory it is specified what its domain and range is, and functions are restricted to their domain and range.
This means that the theory of functions is a theory of **extensional** functions, in the set-theoretic sense of the word:

> a **function** f from set A into set B: $f:A \rightarrow B$ is a set of ordered pairs in $A \times B$ such that: 1. for every $a \in A$: there is a $b \in B$ such that $<a,b> \in f$
> 2. for every $a \in A$, $b,b' \in B$: if $<a,b> \in f$ and $<a,b'> \in f$ then $b=b'$
>
> The **domain** of f, $dom(f) = \{a \in A: \exists b \in B: <a,b> \in f\}$
> The **range** of f, $ran(f) = \{b \in B: \exists a \, e \, A: <a,b> \in f\}$

When we talk about, say, the function + of addition on the natural numbers as being the same function as addition on the real numbers, we are using an intensional notion of function: extensionally $+:N \times N \rightarrow N$ and $+:R \times R \rightarrow R$ are different functions, because their domain and range are different and because they are different sets of ordered pairs. They are the same function, intensionally, in that these functions have the same relevant mathematical properties. Our theory of functions will be a theory of extensional functions, and that means that each function is extensionally specified as a set of ordered pairs with a fixed domain and fixed range.
This means that functions can only apply to arguments which are in their domain.
Partly as a means to avoid the set-theoretic paradoxes, we carry this over into our logical language, which contains expressions denoting these functions. We assume that each functional expression comes with an indication of its **type**, which in essence is an indication of what the domain and range of its interpretation will be. The syntax of our language will only allow a functional expression f of a certain type a to apply to an argument of the type of expressions that denote entities in the domain of the interpretation of f.

This is the basic idea of the functional language:
-each expression is of a certain type and will be interpreted into a certain semantic domain corresponding to that type.

-a functional expressions of a certain type can only apply to expressions which denote entities in the domain of the function which is the interpretation of that functional expression and hence are of the type of the input for that function.

-The result of applying a functional expression to an expression of the correct input type is an expression which is of the output type, and hence is interpreted as an entity within the range of the interpretation of that functional expression.

We make this precise by defining the language of functional type logic TL.
We first define all the possible **types** that we will have expressions of in the language:

Let e and t be two symbols.

$TYPE_{TL}$ is the smallest set such that:

1. $e,t \in TYPE_{TL}$
2. if $a,b \in TYPE_{TL}$ then $\langle a,b\rangle \in TYPE_{TL}$

e and t are our basic types. They are the only types in the theory that will not be types of functions.

-e (entity) is the type of expressions that denote individuals.
Thus e is the type of terms in predicate logic.

-t (truth value) is the type of expressions that denote truth values.
Thus t is the type of formulas in predicate logic.

-For any types a and b, <a,b> is the type of expressions that denote functions from entities denoted by expressions of type a into entities denoted by expressions of type b.
In short, <a,b> is the type of expressions that denote functions mapping a-entities onto b-entities.

This definition of types gives us a rich set of types of expressions:

Since e and t are types, <e,t> is a type:
the type of functional expressions denoting functions that map individuals (the denotations of expressions of type e) onto truth values (the denotations of expressions of type t, formulas).
As we will see, this is the type of properties (or sets) of individuals.

Assuming <e,t> to be the type of property-denoting expressions,
then <<e,t>,<e,t>> is the type of expressions mapping properties onto properties:
the corresponding functions take a function from individuals to truth values as input and deliver similarly a function from individuals into truth values as output.
As we will see, this is the just the type for expressions like prenominal adjectives.

Since <e,t> is a type and t is a type, <<e,t>,t> is a type as well.
This is the type of expressions that denote functions that take themselves a function from individuals into truth values as input and deliver a truth value as output.
As we will see, this is the right type for interpreting noun phrases like *every cat*.
We call expressions of type <<e,t>,t> generalized quantifiers.

Since <e,t> is a type and <<e,t>,t> is a type, also <<e,t>,<<e,t>,t>> is a type.
The type of expressions that denote functions which take functions from individuals into truth values as input, and give noun phrase interpretations as output.
This is the type of determiners.

We really have a rich theory of types.
Many types are included in the theory that we may not have any use for.
Like the type <<t,e>,<t,e>>:
the type of expressions that denote functions which map functions from truth values into individuals onto functions from truth values into individuals.
It is unlikely that we can argue plausibly that there are natural language expressions that will be interpreted as functions of this type. To have a general and simple definition of the types and our language, we will nevertheless include these in our logical language (they just won't be used in given interpretations to natural languages expressions).

We will very soon give some ways of thinking about the types described above in a more intuitive or simple way. The above examples were only meant to show how the definition of types works.

Based on this definition, we now define the logical language TL. The language TL has the following logical constants: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$, =, (,), λ.
As we can see, for ease, we take over all the logical constants of predicate logic and we add another symbol, λ, the λ-operator.
I will from now on in the definitions leave out subscript TL, indicating that a set is particular to the language. I take that to be understood.

Constants and variables:
For each type a $\in$ TYPE, we specify a set CON_a of non-logical constants of type a and a set VAR_a of variables of type a.

As before, we can think about the non-logical constants of type a as the lexical items whose semantics we do not further specify.

For every a $\in$ TYPE: $CON_a = \{c^a_1, c^a_2, \ldots\}$
a set of constants of type a (at most countably many)
For every a e TYPE: $VAR_a = \{x^a_1, x^a_2, \ldots\}$
a set of variables of type a (countably many)

We will never write these type-indices explicitly in the formulas, rather we will make typographical conventions about the types of our expressions.
For instance, I will typically use the following conventions:

I write my constants in capital letters: RONYA, PIM, CAT, PURR
x,y,z are variables of type e
P,Q are variables of type <e,t>

But generally I will indicate the typographical conventions where needed.

So we have as many constants as we want for each type (that is up to our choice), and we have countably many variables of each type.

With this given we will specify the syntax of the language. We do that by specifying for each type a $\in$ TYPE the set EXP_a, the set of well formed expressions of type a.

For each $a \in$ TYPE, EXP_a is the smallest set such that:

1. Constants and variables:

$CON_a \cup VAR_a \subseteq EXP_a$

Constants and variables of type a are expressions of type a.

2. Functional application:

If $\alpha \in EXP_{<a,b>}$ and $\beta \in EXP_a$ then $(\alpha(\beta)) \in EXP_b$

3. Functional abstraction:

If $x \in VAR_a$ and $\beta \in EXP_b$ then $\lambda x\beta \in EXP_{<a,b>}$

4. Connectives:

If $\varphi,\psi \in EXP_t$ then $\neg\varphi$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi) \in EXP_t$

5. Quantifiers:

If $x \in VAR_a$ and $\varphi \in EXP_t$ then $\forall x\varphi$, $\exists x\varphi \in EXP_t$

6. Identity:

If $\alpha \in EXP_a$ and $\beta \in EXP_a$ then $(\alpha=\beta) \in EXP_t$

As said before, we will ignore clause 3 of this definition for the moment.
Focussing on clauses 4-6, we see that type logic contains predicate logic as part.
Clause 4, concerning the connectives is just the same clause that we had in predicate logic (because FORM = EXP_t).

Clause 5 for quantifiers looks just the same as the corresponding clause in predicate logic, but note that in predicate logic we could only quantify over individual variables (that is, variable of type e), while in type logic, we allow quantification over variables of any type: in type logic we have higher order quantification.

Similarly, while in predicate logic we can form identity statements between terms (where TERM = EXP_e), in type logic we can form identity statements between any two expressions of any type, as long as they have the same type.
The resulting identity statement is a formula, an expression of type t.

We will show in a later chapter that there is a lot of redundancy in this definition: we only need functional application, functional abstraction and identity to define everything else.

Let us now focus on clause 2, the clause for functional application.

To see how this works let us specify some constants.
Let us assume that we have the following constants:

RONYA $\in$ CON_e
e is the type of individuals.

MAN, GIRL $\in$ $CON_{<e,t>}$
<e,t> is the type of one-place predicates of individuals.

KISSED, HUGGED $\in$ $CON_{<e,<e,t>>}$
<e,<e,t>> is the type of two-place relations between individuals.

SOME, EVERY $\in$ $CON_{<<e,t>,<<e,t>,t>>}$
<<e,t>,<<e,t>,t>> is the type of determiners.

OLD $\in$ $CON_{<<e,t>,<e,t>>}$
<<e,t>,<e,t>> is the type of one-place predicate modifiers: these are functions that take a property and yield a property.

AND_1 $\in$ $CON_{<<<e,t>,t>,<<<e,t>,t>,<<e,t>,t>>>}$
<<e,t>,t> is the type of noun phrase meanings.
AND_1 takes a noun phrase meaning, and another noun phrase meaning and produces a noun phrase meaning.

AND_2 $\in$ $CON_{<<e,<e,t>>,< <e,<e,t>>,<e,<e,t>>>>}$
Here AND is conjunction of two-place relations, it takes two two place relations, and produces a two place relation.

Let us now do some functional applications:

HUGGED $\in$ $CON_{<e,<e,t>>}$, a two-place predicate, hence (by clause 1):
HUGGED $\in$ $EXP_{<e,<e,t>>}$
Similarly AND_2 $\in$ $EXP_{<<e,<e,t>>,<<e,<e,t>>,<e,<e,t>>>>}$

We see that: AND_2 $\in$ $EXP_{<<e,<e,t>>,<<e,<e,t>>,<e,<e,t>>>>}$

$<$ ------, ---------->
a b

HUGGED $\in$ $EXP_{<e,<e,t>>}$
----
a

Hence (AND_2(HUGGED)) $\in$ $EXP_{<<e,<e,t>>,<e,<e,t>>>}$
----
b

Next: $(AND_2(HUGGED)) \in EXP_{<<e,<e,t>>,<e,<e,t>>>}$

<-------, ------>

a b

$KISSED \in EXP_{<e,<e,t>>}$

----

a

Hence $((AND_2(HUGGED))(KISSED)) \in EXP_{<e,<e,t>>}$

----

b

We see that $((AND_2(HUGGED))(KISSED))$ is itself a two-place predicate. This makes it very suitable to serve as a representation for *kissed and hugged* (of course, we still would need to give AND_2 the correct interpretation).

$((AND_2(HUGGED))(KISSED)) \in EXP_{<e,<e,t>>}$

<-, --->

a b

$RONYA \in EXP_e$

-

a

Hence $(((AND_2(HUGGED))(KISSED))(RONYA)) \in EXP_{<e,t>}$

----

b

$(((AND_2(HUGGED))(KISSED))(RONYA))$ is a one-place predicate, again, that is what we would expect *kissed and hugged Ronya* to be.

$OLD \in EXP_{<<e,t>,<e,t>>}$
$MAN \in EXP_{<e,t>}$
Hence $(OLD(MAN)) \in EXP_{<e,t>}$

$SOME \in EXP_{<<e,t>,<<e,t>,t>>}$
$(OLD(MAN)) \in EXP_{<e,t>}$
Hence $(SOME((OLD(MAN)))) \in EXP_{<<e,t>,t>}$

$EVERY \in EXP_{<<e,t>,<<e,t>,t>>>}$
$GIRL \in EXP_{<e,t>}$
Hence $(EVERY(GIRL)) \in EXP_{<<e,t>,t>}$

$AND_1 \in EXP_{<<<e,t>,t>,<<<e,t>,t>,<<e,t>,t>>>}$
$(EVERY(GIRL)) \in EXP_{<<e,t>,t>}$
Hence $(AND_1((EVERY(GIRL)))) \in EXP_{<<<e,t>,t>,<<e,t>,t>>}$

$(AND_1((EVERY(GIRL)))) \in EXP_{<<<e,t>,t>,<<e,t>,t>>}$
$(SOME((OLD(MAN)))) \in EXP_{<<e,t>,t>}$
Hence $((AND_1((EVERY(GIRL))))((SOME((OLD(MAN)))))) \in EXP_{<<e,t>,t>}$

Assuming that <<e,t>,t> is the correct type for noun phrase interpretations,
we have see that all of
(SOME((OLD(MAN)))) (*some old man*),
(EVERY(GIRL)) (*every girl*) and
((AND$_1$((EVERY(GIRL))))((SOME((OLD(MAN)))))) (*some old man and every girl*)
are of that type <<e,t>,t>.

Finally:
((AND$_1$((EVERY(GIRL))))((SOME((OLD(MAN)))))) $\in$ EXP$_{<<e,t>,t>}$
(((AND$_2$(HUGGED))(KISSED))(RONYA)) $\in$ EXP$_{<e,t>}$
Hence
(((AND$_1$((EVERY(GIRL))))((SOME((OLD(MAN))))))
((((AND$_2$(HUGGED))(KISSED))(RONYA)))) $\in$ EXP$_t$

We can make this more perspicuous by introducing a notation convention:

> **Notation convention**: **Infix notation**:
> Let R be an expression of type <a,<a,c>>, α and β expression of type a. Then ((R(α))(β)) is an expression of type c.
> We define: (β R α) := ((R(α))(β)

:= means 'is by definition'. Thus (β R α) is notation that we add to type theory: it is not defined as part of the syntax of the language, but is just a way in which we make our expressions more readable.
With infix notation, we can write the above expression with infix notation applied to both AND$_1$ and to AND$_2$ as:

((EVERY(GIRL)) AND$_1$ (SOME((OLD(MAN)))))
((KISSED AND$_2$ HUGGED)(RONYA)))

This is a wellformed expression of type logic of type t, a sentence, and it can be used as the representation of sentence (1).
Of course, we haven't yet interpreted the expressions occurring in this expression, but we can note that, unlike the predicate logical expression, in the above expression we can find a wellformed sub-expression corresponding to each of the parts of sentence (1).

(1) Some old man and every girl kissed and hugged Ronya.

1.3. The semantics of functional type logic.

Let us now move to the semantics of functional type logic.
As before, we start with specifying the models for our type logical language TL.
I will follow the practice here to include in the definition of the models only those aspects that vary from model to model.

A model for functional type logic is a pair:
M = <D,F>, where:
1. D, the **domain of individuals**, is a non-empty set.
2. F is the **interpretation function** for the non-logical constants.

This looks just like predicate logic. However, since we possibly have many more types of constants, it is in the specification of the interpretation function F that differences will come in.
In the model we only specify domain D, the domain of individuals. this is obviously not the only domain in which expressions are interpreted. We don't put the other domains in the definition for the model, because, once D is specified, all other domains are predictable and are given in a domain definition:

Let M = <D,F> be a model for TL.
For any type a $\in$ $TYPE_{TL}$, we define D_a, the domain of type a:
1. $D_e = D$
2. $D_t = \{0,1\}$
3. $D_{<a,b>} = (D_a \rightarrow D_b)$

So, the domain of type e, De is the domain of individuals given in the model M.
The domain of type t, Dt, is the set of truth values {0,1}.

For any sets A,B, $(A \rightarrow B) = \{f: f \text{ is a function from A into B}\}$

Thus $(A \rightarrow B)$ is the set of all functions with domain A and range included in B.
Hence $D_{<a,b>}$ is the set of all functions with domain D_a
- the set of entities in the domain of a – and with their range included in D_b –
the set of entities in the domain of b.
In other words, $D_{<a,b>}$ is the set of all functions from a-entities into b-entities.

Thus, $D_{<e,t>} = (D_e \rightarrow D_t)$
$= (D \rightarrow \{0,1\})$
The set of all functions from individuals into truth values.

$D_{<e,<e,t>>} = (D_e \rightarrow (D_e \rightarrow D_t))$
$= (D \rightarrow (D \rightarrow \{0,1\}))$
The set of all functions that map each individual into a function from individuals into truth values.

$D_{<<e,t>,t>} = ((D_e \rightarrow D_t) \rightarrow D_t)$
$= ((D \rightarrow \{0,1\}) \rightarrow \{0,1\})$

The set of all functions that map functions from individuals into truth values onto truth values.

$D_{<<e,t>,<<e,t>,t>>} = ((D_e \rightarrow D_t) \rightarrow ((D_e \rightarrow D_t) \rightarrow D_t)) =$
$((D \rightarrow \{0,1\}) \rightarrow ((D \rightarrow \{0,1\}) \rightarrow \{0,1\}))$
The set of all functions that map functions from individuals into truth values onto functions that map functions from individuals into truth values onto truth values.

$D_{<<e,t>,<e,t>>} = ((D_e \rightarrow D_t) \rightarrow (D_e \rightarrow D_t)) =$
$((D \rightarrow \{0,1\}) \rightarrow (D \rightarrow \{0,1\})$
The set of all functions that map each function from individuals into truth values onto a function from individuals onto truth values.

$D_{<t,t>} = (D_t \rightarrow D_t) =$
$(\{0,1\} \rightarrow \{0,1\})$
The set of all functions from truth values into truth values
(= the set of all one place truth functions).

$D_{<t,<t,t>>} = (D_t \rightarrow (D_t \rightarrow D_t)) =$
$(\{0,1\} \rightarrow (\{0,1\} \rightarrow \{0,1\}))$
The set of all functions from truth values into one-place truth functions
(=, as we will see, the set of all two-place truth functions).

The domain definition specifies a semantic domain for every possible type in TYPE. With that, we can specify the interpretation function F for the non-logical constants of the model:

A **model** for functional type logic TL is a pair M = <D,F>,
where D is a non-empty set and:
for every $a \in TYPE_{TL}$: $F:CON_a \rightarrow D_a$

For every type a, F maps any constant $c \in CON_a$ onto an entity $F(c) \in D_a$.

So F interprets a non-logical constant of type a as an entity in the domain of a.

Let M = <D,F> be a model for TL.
An assignment function (on M) is any function g such that:
for every $a \in TYPE_{TL}$ for every $x \in VAR_a$: $g(x) \in D_a$

Thus an assignment function maps every variable of every type into an entity of the domain corresponding to that type.
Hence, g maps variable $x \in VAR_e$ onto an individual g(x) in D, it maps predicate variable $P \in VAR_{<e,t>}$ onto some function g(P) in $(D \rightarrow \{0,1\})$, etc.

We can now give the semantics for our functional type logical language, by specifying, as before $[\![\alpha]\!]_{M,g}$, the interpretation of expression α in model M, relative to assignment function g:

The semantic interpretation: $[\![\alpha]\!]_{M,g}$
For any a,b ∈ $TYPE_{TL}$:

1. Constants and variables.

If $c \in CON_a$, then $[\![c]\!]_{M,g} = F(c)$
If $x \in VAR_a$, then $[\![x]\!]_{M,g} = g(x)$

2. Functional application.

If $\alpha \in EXP_{<a,b>}$ and $\beta \in EXP_a$ then:
$[\![(\alpha(\beta))]\!]_{M,g} = [\![\alpha]\!]_{M,g}([\![\beta]\!]_{M,g})$

3. Functional abstraction.

If $x \in VAR_a$ and $\beta \in EXP_b$ then:
$[\![\lambda x\beta]\!]_{M,g} = h$,
where h is the unique function in $(D_a \rightarrow D_b)$ such that:
for every $d \in D_a$: $h(d) = [\![\beta]\!]_{M,g^d_x}$

4. Connectives.

If $\varphi,\psi \in EXP_t$, then:
$[\![\neg\varphi]\!]_{M,g} = \neg([\![\varphi]\!]_{M,g})$
$[\![(\varphi \wedge \psi)]\!]_{M,g} = \wedge(<[\![\varphi]\!]_{M,g},[\![\psi]\!]_{M,g}>)$
$[\![(\varphi \vee \psi)]\!]_{M,g} = \vee(<[\![\varphi]\!]_{M,g},[\![\psi]\!]_{M,g}>)$
$[\![(\varphi \rightarrow \psi]\!]_{M,g} = \rightarrow(<[\![\varphi]\!]_{M,g},[\![\psi]\!]_{M,g}>)$

Here on the right hand side we find the following truth functions:
$\neg:\{0,1\} \rightarrow \{0,1\}$; $\wedge,\vee,\rightarrow$ are functions from {0,1} x {0,1} into {0,1}, given by:

$\neg$ = {<0,1>,<1,0>}
$\wedge$ = {<<1,1>,1>,<<1,0>,0>,<<0,1>,0>,<<0,0>,0>}
$\vee$ = {<<1,1>,1>,<<1,0>,1>,<<0,1>,1>,<<0,0>,0>}
$\rightarrow$ = {<<1,1>,1>,<<1,0>,0>,<<0,1>,1>,<<0,0>,1>}

5. Quantifiers.

If $x \in VAR_a$ and $\varphi \in EXP_t$ then:
$[\![\forall x\varphi]\!]_{M,g} = 1$ iff for every $d \in D_a$: $[\![\varphi]\!]_{M,g^d_x} = 1$; 0 otherwise
$[\![\exists x\varphi]\!]_{M,g} = 1$ iff for some $d \in D_a$: $[\![\varphi]\!]_{M,g^d_x} = 1$; 0 otherwise

6. Identity.

If $\alpha, \beta \in EXP_a$ then:
$[\![(\alpha=\beta)]\!]_{M,g} = 1$ iff $[\![\alpha]\!]_{M,g} = [\![\beta]\!]_{M,g}$

A sentence is a formula without free variables.
Let X be a set of sentences and φ a sentence.

$[\![\varphi]\!]_M = 1$, φ is true in M, iff for every g: $[\![\varphi]\!]_{M,g} = 1$
$[\![\varphi]\!]_M = 0$, φ is false in M, iff for every g: $[\![\varphi]\!]_{M,g} = 0$

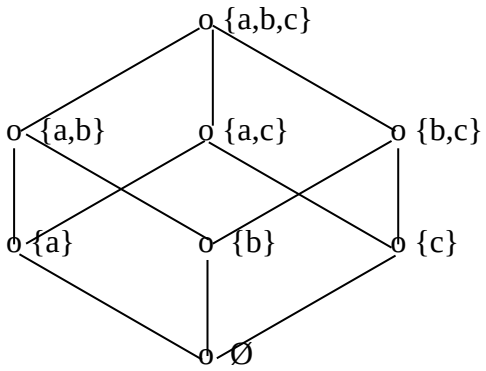
$X \Rightarrow \varphi$, X entails φ, iff for every model M:
if for every $\delta \in X$: $[\![\delta]\!]_M = 1$ then $[\![\varphi]\!]_M = 1$

1.4. Structures and Isomorphisms

Take the set {a, b, c}. The powerset of {a,b,c} is given by:
pow({a, b, c}) = {Ø, {a}, {b}, {c}, {a,b}, {a,c}, {a,b,c}}

This set is closed under the operations of union, ∪, intersection, ∩ and complementation —, meaning:
-For every X, Y ∈ **pow**({a,b,c}): X ∪ Y ∈ **pow**({a,b,c}),
X ∩ Y ∈ **pow**({a,b,c}),
X — Y ∈ **pow**({a,b,c})
We write: ¬X for {a,b,c} — X

-We order the elements of **pow**({a,b,c}) by the subset relation ⊆ as in the following diagram:

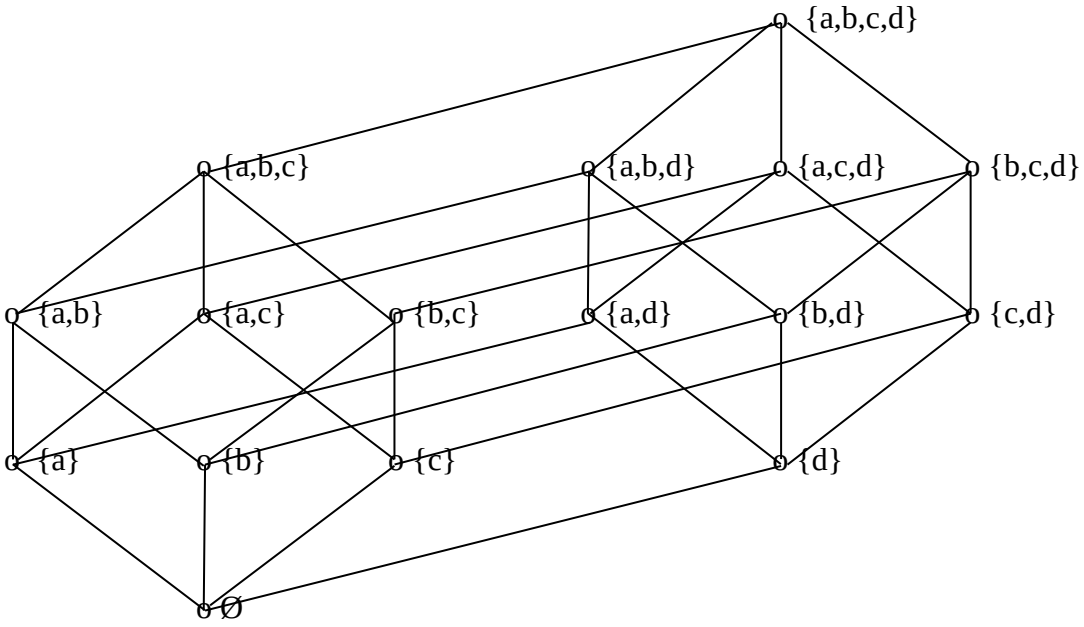


Ø ⊆ {a} ⊆ {a,b} ⊆ {a,b,c}
{a,b} ∩ {a,c} = {a}
{a} ∪ {c} = {a,c}
¬{a} = {a,b,c}—{a} = {b,c}

X ∩ ¬X = Ø
X ∪ ¬X = {a,b,c}
¬¬X = X
¬(X ∪ Y) = ¬X ∩ ¬Y
¬(X ∩ Y) = ¬X ∪ ¬Y

The structure <**pow**({a,b,c}), ⊆, ∪, ∩, —, Ø, {a,b,c}> is called a (powerset) Boolean algebra.
We will discuss Boolean algebras more formally later in this class when we discuss Boolean semantics for plurality.

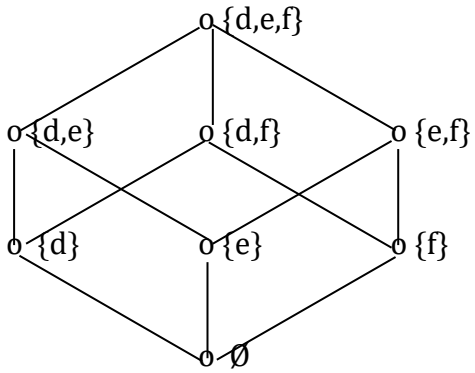
Another example: the 16 element boolean algebra: **pow**({a,b,c,d}):



We call the top element 1 and the bottom element 0, and define: $\neg X = 1 - X$
The fact that it is a Boolean algebra means that it satisfies the same conditions as the other structure, among others:

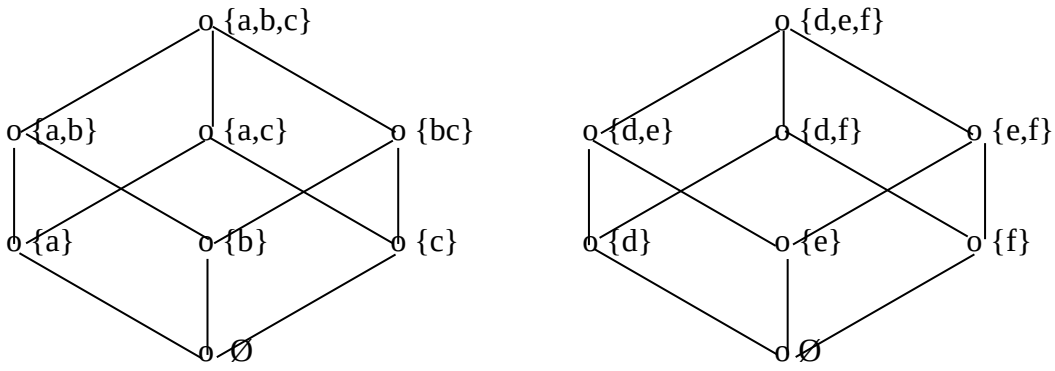
$X \cup \neg X = 1$
$X \cap \neg X = 0$
$\neg\neg X = X$
$\neg(X \cup Y) = \neg X \cap \neg Y$

Now look at the structure: <**pow**({d,e,f}), ⊆, ∪, ∩, –, Ø, {d,e,f}>:



It is not strictly speaking the same structure as <**pow**({a,b,c}), ⊆, ∪, ∩, –, Ø, {a,b,c}>, but only because the objects are different:
the relations between the objects are *exactly the same*.

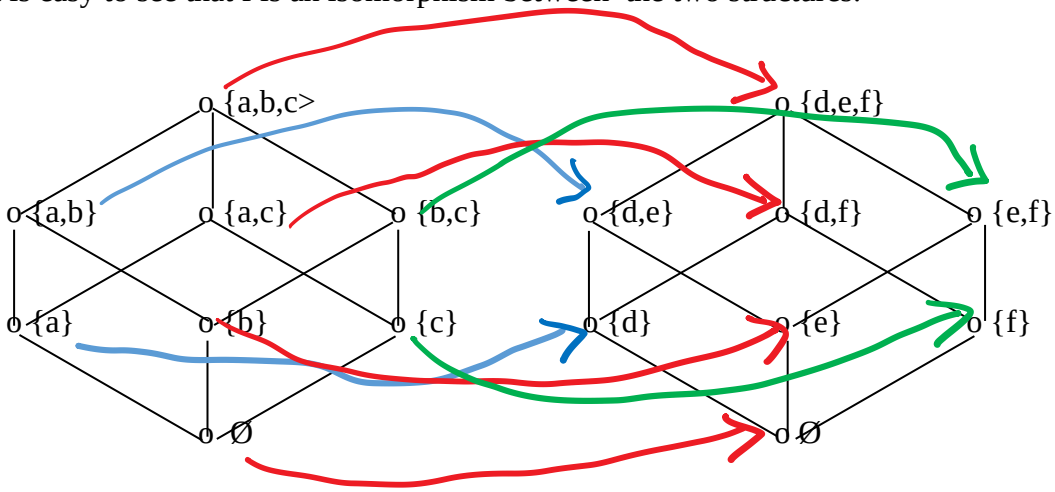
The mathematical expression for this is that the structures
<**pow**({a,b,c}), ⊆, ∪, ∩, –, Ø, {a,b,c}> and <**pow**({d,e,f}), ⊆, ∪, ∩, –, Ø, {d,e,f}>
are *isomorphic*, and this means that
there is a one-one function (an isomorphism) relating
the elements of <**pow**({a,b,c}), ⊆, ∪, ∩, –, Ø, {a,b,c}> one-one to
the elements of <**pow**({d,e,f}), ⊆, ∪, ∩, –, Ø, {d,e,f}>
a function that completely preserves the relations and operations ⊆ and ∩ and ∪ and – in both directions.



Define g: {a,b,c} → {d,e,f} by: g(a) = d
g(b) = e
g(c) = f

And define f: **pow**({a,b,c}) → **pow**({d,e,f}) by:
for every X ∈ **pow**({a,b,c}): f(X) = {g(x): x ∈ X}

Then it is easy to see that f is an isomorphism between the two structures:



Examples:
- {a} ⊆ {a,b} and f({a}) ⊆ f({a,b}),
because f({a}) = {d} and f({a,b}) = {d,e} and {d} ⊆ {d,e}

-{a,b} ∩ {b,c} = {b}
f({a,b}) ∩ f({b,c}) = {d,e} ∩ {e,f} = {e} = f({b})

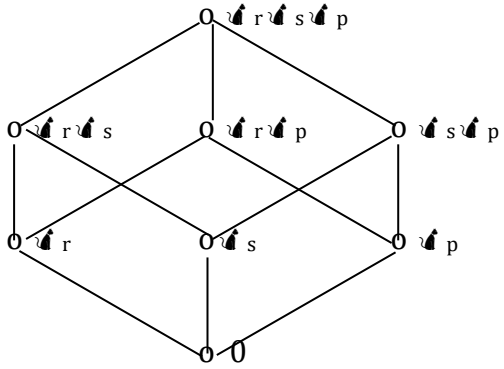
We say that the structures <**pow**({a,b,c}), ⊆, ∪, ∩, –, Ø, {a,b,c}> and <**pow**({d,e,f}), ⊆, ∪, ∩, –, Ø, {d,e,f}> are *identical up to isomorphism*,
and the crucial point is that, with Mathematics, we do not distinguish between structures that are identical up to isomorphism,
for all relevant purposes, they are *the same structure*.

Suppose now we have a set of group objects:

{0, ʎ r, ʎ s, ʎ p, ʎ r ʎ s, ʎ r ʎ p, ʎ s ʎ p , ʎ r ʎ s ʎ p }

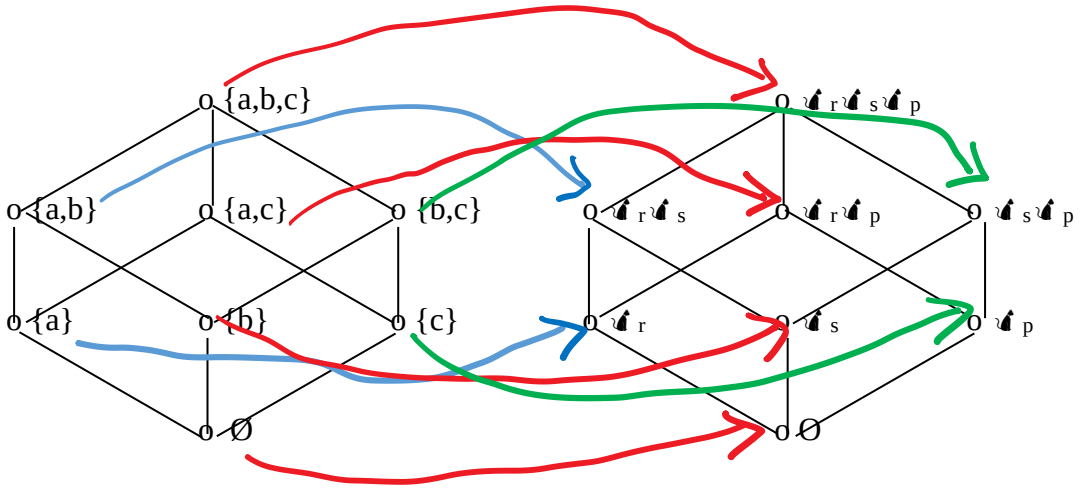
ronya, shunra, pim,
ronya and shunra, ronya and pim, shunra and pim,
ronya and shunra and pim

and we define three operations ⊔, ⊓, ¬ and a relation ⊑
such that the relations are given as in the picture below:



and we set ʎ r ⊔ ʎ s = ʎ r ʎ s, etc.
ʎ r ʎ p ⊓ ʎ s ʎ p = ʎ p, etc.
¬ ʎ r = ʎ s ʎ p

Then it is easy to see that the following function f is an isomorphism between the two structures:



So the objects in the two structures may be completely different types of objects, and the operations may be defined as is appropriate for those objects.
Yet, the objects can be one to one mapped so as that the relations and operations are preserved in both directions.
This means that the structures are isomorphic, and hence identical up to isomorphism, and we don't distinguish between them.

This notion underlies the functional type theory that we use in semantics.

1.5.1. Chacteristic functions.

You are used to predicate denotations being sets of individuals:

$[\![CAT]\!]_{M,g} = F_{M,PL}(CAT) \subseteq D_{M,}$ the set of cats.

In functional type theory: $[\![CAT]\!]_{M,g} = F_{M,TL}(CAT)$: $D_M \rightarrow \{0,1\}$.

So in predicate logic we interpet CAT as the set of cats in the domain,
in functional type theory, we interpret CAT as the function that maps every object in the domain onto a truth value 0 or 1,
and that maps individuals in the domain onto 1 iff they are cats.

Suppose we have a domain of three individuals {r,p,s}, ronya, pim and sander
Then there are, as before 8 subsets, but also 8 functions from{r,p,s} into {0,1}.
And we can define on the set of all functions f and g from D_M into {0,1},
($D_M \rightarrow \{0,1\}$), the following relation:

$f \sqsubseteq g$ iff for all $d \in D_M$: if f(d)=1 then g(d)=1

In terms of this we define also $f \sqcup g$, $f \sqcap g$ and $\neg f$.

Here we realize that in Boolean algebras $\sqcup$ and $\sqcap$ and $\neg$ pattern along the truth tables for disjunction $\vee$, conjunction $\wedge$, and negation $\neg$:
So:

$$\begin{pmatrix} r \rightarrow 1 \\ p \rightarrow 0 \\ s \rightarrow 0 \end{pmatrix} \sqcup \begin{pmatrix} r \rightarrow 0 \\ p \rightarrow 1 \\ s \rightarrow 0 \end{pmatrix} = \begin{pmatrix} r \rightarrow 1 \\ p \rightarrow 1 \\ s \rightarrow 0 \end{pmatrix} \qquad \begin{array}{l} \vee(1,0) = 1 \\ \vee(0,1) = 1 \\ \vee(0,0) = 0 \end{array}$$

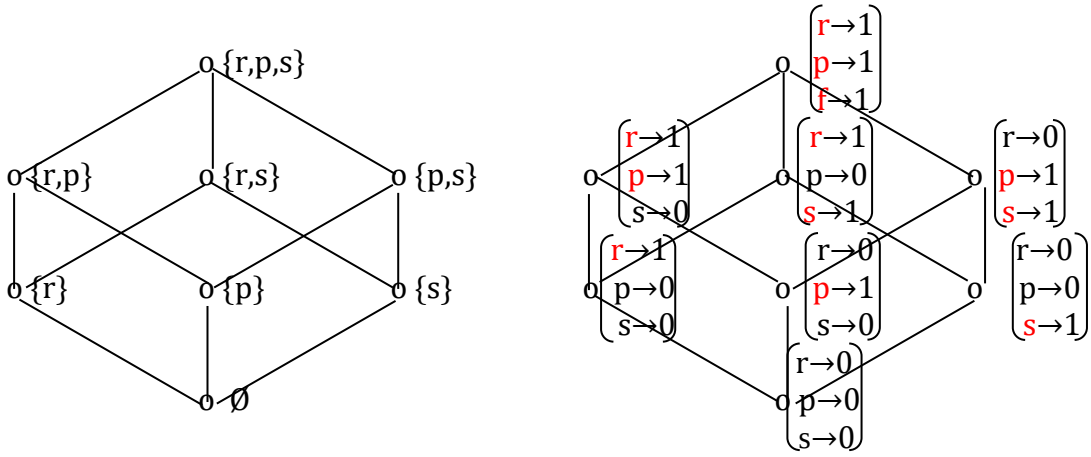
and:

$$\begin{pmatrix} r \rightarrow 1 \\ p \rightarrow 1 \\ s \rightarrow 0 \end{pmatrix} \sqcap \begin{pmatrix} r \rightarrow 0 \\ p \rightarrow 1 \\ s \rightarrow 1 \end{pmatrix} = \begin{pmatrix} r \rightarrow 0 \\ p \rightarrow 1 \\ s \rightarrow 0 \end{pmatrix} \qquad \begin{array}{l} \wedge(1,0) = 0 \\ \wedge(1,1) = 1 \\ \wedge(0,1) = 0 \end{array}$$

and:

$$\neg \begin{pmatrix} r \rightarrow 1 \\ p \rightarrow 0 \\ s \rightarrow 0 \end{pmatrix} = \begin{pmatrix} r \rightarrow 0 \\ p \rightarrow 1 \\ s \rightarrow 1 \end{pmatrix} \qquad \begin{array}{l} \neg 1 = 0 \\ \neg 0 = 1 \\ \neg 0 = 1 \end{array}$$

So the picture we get is the following:



And we see that the structures we get are isomorphic.
The functions on the right are the characteristic functions of the sets on the left:

For A ⊆ D:
char$_D$(A) is the function that maps every element in A onto 1 and every element in D_M − A onto 0.

char$_D$({r,p}) maps r and p onto 1 and s onto 0.

If $[\![CAT]\!]_{M,g,PL}$ = {r,p}, and we switch in type logic from sets to characteristic functions, it follows that $[\![CAT]\!]_{M,g,TL}$ =

$$F_{M,TL}(CAT) = \begin{pmatrix} r\rightarrow 1 \\ p\rightarrow 1 \\ s\rightarrow 0 \end{pmatrix}$$

So whether or not you set up the semantics by stating:

$$[\![CAT(ronya)]\!]_{M,g,PL} = 1 \text{ iff } F_{M,PL}(ronya) \in F_{M,PL}(CAT)$$

or

$$[\![CAT(ronya)]\!]_{M,g,TL} = 1 \text{ iff } F_{M,TL}(CAT)(F_{M,TL}(ronya))=1$$

makes no difference.
The choice between isomorphic analyses is made by what you want the theory to do.

For instance, linguistic theories prefer the functional approach, because it makes clear that one and the same operation is putting the interpretations of expressions together.

$$F_M(\neg) = \begin{pmatrix} 1\rightarrow 0 \\ 0\rightarrow 1 \end{pmatrix} \qquad F_M(\wedge) = \begin{pmatrix} \langle 1,1\rangle\rightarrow 1 \\ \langle 1,0\rangle\rightarrow 0 \\ \langle 0,1\rangle\rightarrow 0 \\ \langle 0,0\rangle\rightarrow 0 \end{pmatrix} \qquad F_M(\vee) = \begin{pmatrix} \langle 1,1\rangle\rightarrow 1 \\ \langle 1,0\rangle\rightarrow 1 \\ \langle 0,1\rangle\rightarrow 1 \\ \langle 0,0\rangle\rightarrow 0 \end{pmatrix}$$

$[\![\neg\varphi]\!]_{M,g}$ = APPLY[$[\![\neg]\!]_{M,g}$, $[\![\varphi]\!]_{M,g}$] =
APPLY[$F_M(\neg)$, $[\![\varphi]\!]_{M,g}$] =
$F_M(\neg)$($[\![\varphi]\!]_{M,g}$) =
$\begin{pmatrix} 1\rightarrow 0 \\ 0\rightarrow 1 \end{pmatrix}([\![\varphi]\!]_{M,g})$ = 1 if $[\![\varphi]\!]_{M,g}$ = 0; 0 otherwise

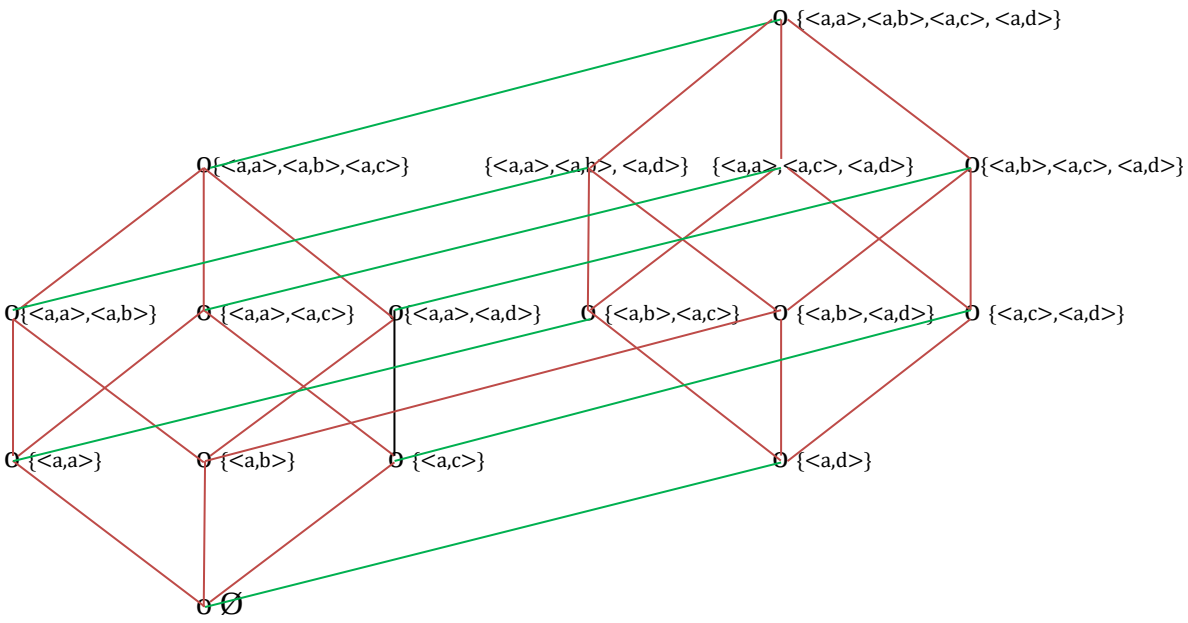
$[\![CAT(ronya)]\!]_{M,g}$ = APPLY[$[\![CAT]\!]_{M,g}$, $[\![ronya]\!]_{M,g}$] =
APPLY[$F_{M,TL}(CAT)$, $F_{M,TL}(ronya)$] =
$F_{M,TL}(CAT)$($F_{M,TL}(ronya)$) = 1 if $F_{M,PL}(ronya) \in F_{M.PL}(CAT)$;
= 0 otherwise

The functional analysis brings out the fact that the interpretation of ¬ combines with the interpretation of φ via functional application in the same way as the interpretation of CAT combines with the interpretation of pim.

With that we can assume functional application as a fundamental primitive of the semantic system, something that does not have to be learned.

1.5.2. Two-place relations

Suppose that D_1 = {a,} and D_2 = {a,b,c,d}, then $D_1 \times D_2$ = {<a,a>,<a,b>,<a,c>, <a,d>},
As can be calculated, **pow**({<a,a>,<a,b>,<a,c>, <a,d>}) has sixteen elements:



What are these sets of pairs? They are all the two place relations there are between the domains {a} and {a,b,c} (in that order).

This is a relational perspective, where relations are sets of ordered pairs. The notion of characteristic function is completely general. So we can define the domain:

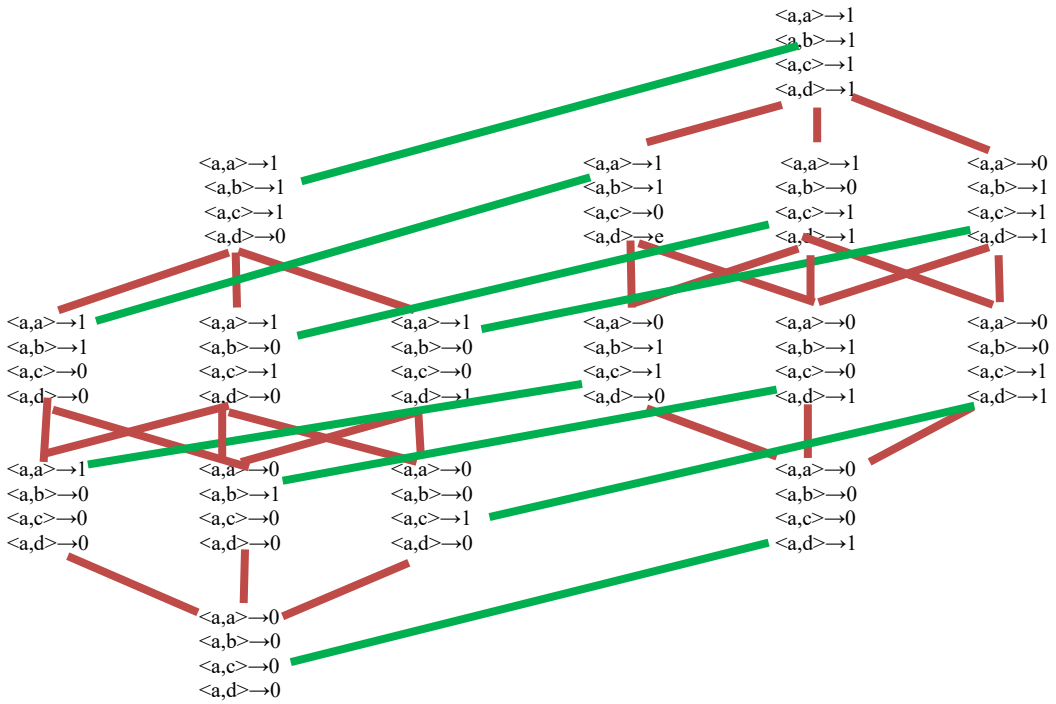
({<a,a>,<a,b>,<a,c>, <a,d>} → {0,1})

The set of all functions that map each of the pairs <a,a>, <a,b>, <a,c>, <a,d> onto truth value 0 or 1.
The functions in this domain are characteristic functions from the sets in
pow({<a,a>,<a,b>,<a,c>, <a,d>})
so the function:

$$\begin{pmatrix} <a,a> \rightarrow 1 \\ <a,b> \rightarrow 1 \\ <a,c> \rightarrow 0 \\ <a,d> \rightarrow 0 \end{pmatrix}$$

characterizes the relation {<a,a>,<a,b>}

and obviously the domains **pow**({<a,a>,<a,b>,<a,c>, <a,d>}) and
({<a,a>,<a,b>,<a,c>, <a,d>} → {0,1}) are isomorphic.
Just replace in the diagram each set by its corresponding characteristic function: the structure stays the same:



But now we go a step further. We found two perspectives: two place relations as sets of ordered pairs and two place relations as functions from ordered pairs to truth values.

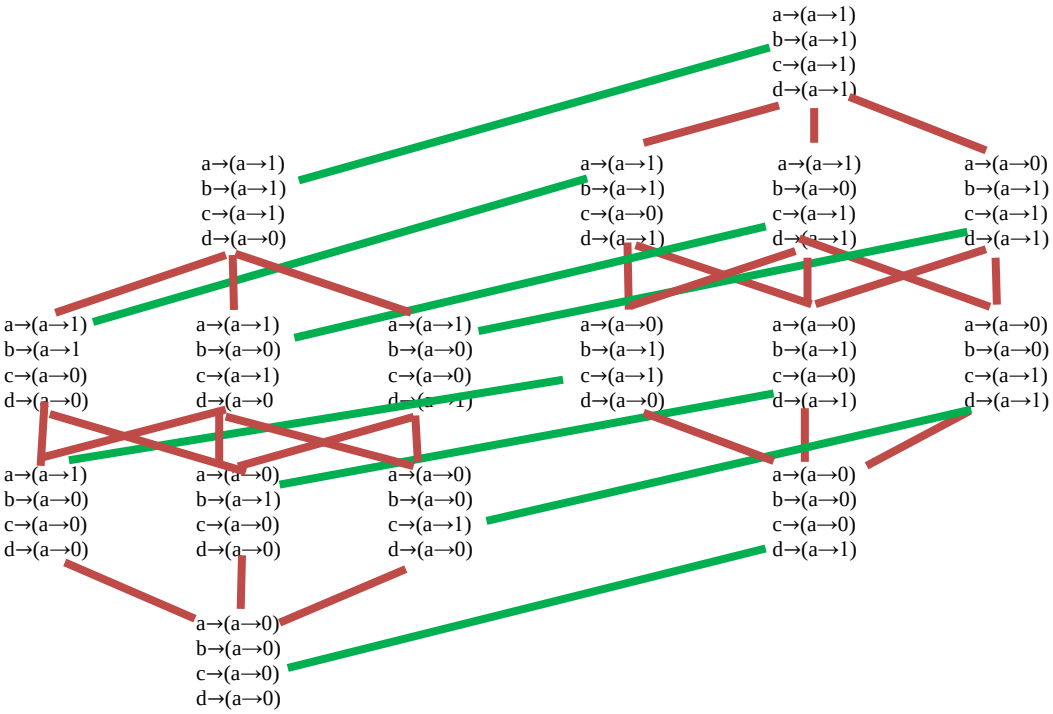
But look again at the domains {a} and {a,b,c,d}. Instead of taking the two arguments in {a} and {a,b,c,d} together and map them onto a truth value, you could also take them one at a time:
first take an argument in {a,b,c,d}, *then* take the argument in {a}, and get a truth value {1,0}.

n-place functions that take their n argument one at a time are called *curried functions* (after Haskill B. Curry, in whose Combinatory Logic from the 1030s such functions play a fundamental role, but the technique was invented by Moses Schönfinkel in the 1920s, (although Frege apparently already used it...well, he would, of course)).

Technically, a curried function is a one-place function f from domain{a,b,c,d} mapping each element x of the domain onto a function f(x), which is itself a function from domain {a} into {0,1}.

In the particular case we are dealing with, ({a}→{0,1}) contains only two functions, namely a→1 and a→0. We will then be looking at functions that map the four elements in {a,b,c,d} onto one of those two functions.

How many such functions are there? Well, 16.



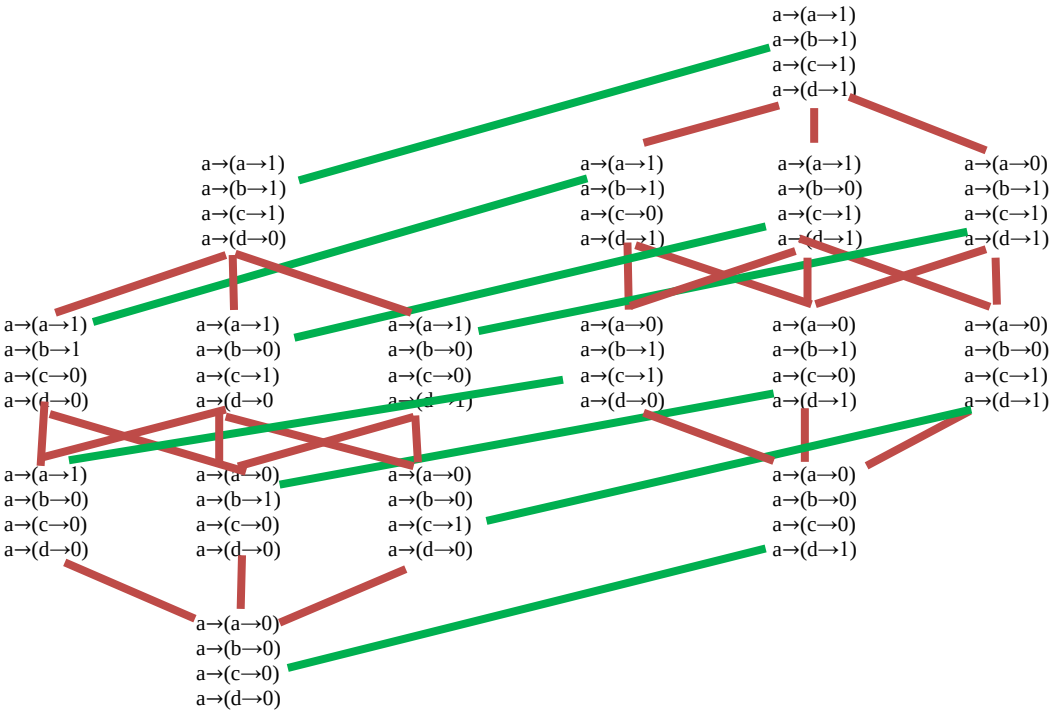
The principle underlying this is:

<RONYA,SANDER> ∈ $F_{M,PL}$(CHASE) iff ($F_{M,TL}$(CHASE)(SANDER))(RONYA) = 1

Currying is a way of encoding a 2-place function via 1-place functions. $F_{M,TL}$(CHASE) is a 1-place function from individuals to 1-place functions from individuals to truth values. If you feed it an argument you get $F_{M,TL}$(CHASE)(SANDER), which is the *chase sander* function, a function that maps individuals onto truth values, for instance, it maps Ronya onto 1, because Ronya chases Sander, but it maps Pim onto 0, because Pim does not chase Sander.

This form of currying takes the arguments in the pair with the second one first. Technically, you can also do it the other way round: map a in {a} onto a function that maps each element in {a,b,c,d}onto one of the truth values in {0,1}.

How many functions is that? Well, 16.



Here the underlying principle is:

$\langle X,Y \rangle \in F_{M,GQ}(\text{EVERY})$ iff $(F_{M,TL}(\text{EVERY})(X))(Y) = 1$

Here too $F_{M,TL}$(EVERY) is a 2 place function encoded via 1-place functions. It is a function that takes say, the set of cats, CAT and maps it onto a function from sets to truth values, the *every cat* function. This function with take the set of smart individuals SMART and maps it onto 0 because not every cat is smart. It takes the set of mammals MAMMAL and maps it onto 1 because every cat is a mammal.

1.5.3. General principles:

1. Cantor's theorem: $|\mathbf{pow}(A)| = 2^{|A|}$
$|(A \rightarrow B)| = |B|^{|A|}$

There is an important class of Boolean algebras called the complete atomic Boolean algebras, ca Boolean algebras.

2. The ca Boolean algebras are, up to isomorphism, exactly the powerset Boolean algebras.

3. The cardinality of the domain of ca Boolean algebras is always a power of 2 (this includes infinite powers).

4. If k is a power of 2, there is, up to isomorphism exactly one ca-Boolean algebra of cardinality k.

5. {0,1} is a ca Boolean algebra.

6. Lifting theorem: If A is a set and B a ca Boolean algebra then $(A \rightarrow B)$ is a ca Boolean algebra.

Isomorphisms.
7. Let A be a set.
h
$(A \rightarrow \{0,1\})$ is the set of all functions from A into {0,1}
$(A \rightarrow \{0,1\})$ is isomorphic to $\mathbf{pow}(A)$.
A function from A-entities to truth values is, up to isomorphism, the same as a set of A-entities.

The following domains are all isomorphic:

The set of all relations between A-entities and B-entities:

$\mathbf{pow}(A \times B)$
The set of all sets of ordered pairs with the first element in A and the second element in B

$\mathbf{pow}(B \times A)$
The set of all sets of ordered pairs with the first element in B and the second element in A

$(A \times B \rightarrow \{0,1\})$
The set of all functions that map each ordered pair in $A \times B$ onto a truthvalue

$(B \times A \rightarrow \{0,1\})$
The set of all functions that map each ordered pair in $B \times A$ onto a truthvalue

$(B \rightarrow (A \rightarrow \{0,1\}))$
The set of all functions that yield a truth value by taking a B-entity into a function that takes any A-entity into a truth value.

$(A \rightarrow (B \rightarrow \{0,1\}))$
The set of all functions that yield a truth value by taking an A-entity into a function that takes any B-entity into a truth value.

The key to the relation between the theory of Boolean algebras and the functional type theory lies in the lifting theorem:

The Lifting Theorem:
Lifting theorem: If A is a set and B a ca Boolean algebra then $(A \rightarrow B)$ is a ca Boolean algebra.

Thus, if B is a Booelan algebra, you can lift the Boolean structure from B onto the set of all functions from A into B.

We define:

BOOL, the set of **Boolean types** is the smallest set of types such that:

1. $t \in$ **BOOL**
2. if $a \in$ **TYPE** and $b \in$ **BOOL** then $\langle a,b \rangle \in$ **BOOL**

The lifting theorem tells us that the domain of every Boolean type has the structure of a Boolean algebra.
It tells us more actually. t is not just a Boolean type, but D_t is finite, hence the domain D_t forms a complete atomic Boolean algebra.
And this means that the domain of every type in **BOOL** forms a complete atomic Boolean algebra.

With that we have proved that every Boolean type is isomorphic to a powerset Boolean algebra.

1.6. Meet the cast

<e,t> is the type of one-place predicates.

$D_{M,<e,t>} = (D_{M,e} \rightarrow D_{M,t}) = (D_M \rightarrow \{0,1\})$.
The set of functions from individuals to truth values.

If CAT ∈ $CON_{<e,t>}$ and RONYA ∈ CON_e then (CAT(RONYA)) ∈ EXP_t
$[\![(CAT(RONYA))]\!]_{M,g} = 1$ iff
$[\![CAT]\!]_{M,g}([\![RONYA]\!]_{M,g}) = 1$ iff
$F_M(CAT)(F_M(RONYA)) = 1$

While <e,t> is officially the type of functions from individuals to truth values, we have seen that we can just as well think of it as the type of sets of individuals.

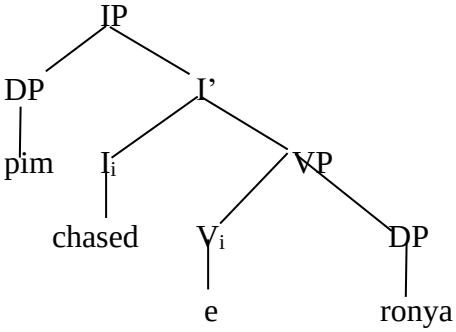
<<e,t>,<e,t>> is the type of predicate modifiers.

$D_{M,<<e,t>,<e,t>>} = ((D_M \rightarrow \{0,1\}) \rightarrow (D_M \rightarrow \{0,1\}))$
We find that too complicated to read aloud, but we can simplify directly.
Since <e,t> is the type of sets of individuals, <<e,t>,<e,t>> is the type of functions from sets of individuals to sets of individuals.
Attributival adjectives are of assumed to be of this type:
SMART ∈ $CON_{<<e,t><e,t>>}$, CAT ∈ $CON_{<e,t>}$, hence (SMART(CAT)) ∈ $EXP_{<e,t>}$

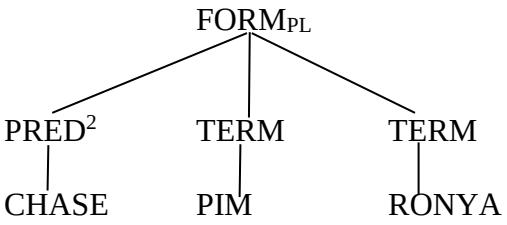
Attributival adjectives are of a modifier type:
A modifier type is a type of the form <a,a>, which input and output type the same.
The idea is that SMART denotes a function that takes the set of cats as input and gives the set of smart cats as output.
As we will see, we can define this meaning with help of the λ-operator, but we postpone that discussion to later.

<e,<e,t>> is the type of (curried) two place predicates

Look at the following tree:



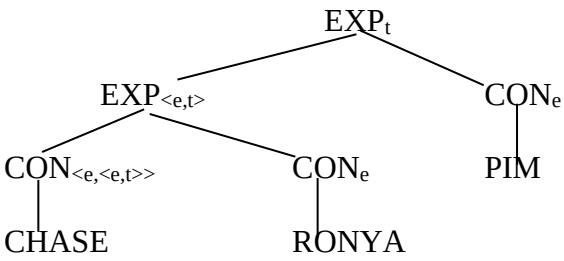
We ignore for the moment the semantics of the past tense. The syntax of predicate logic assumed a 2-place predicate CHASE which combined with the arguments pim and ronya as in the following structure tree:



CHASE(PIM,RONYA)

$[\![\text{CHASE(PIM,RONYA)}]\!]_{M,g}$ = 1 iff
$\langle F_M(\text{PIM}), F_M(\text{RONYA})\rangle \in F_M(\text{CHASE})$

In type logic we can give the following analysis:



((CHASE(RONYA))(PIM))

(CHASE(RONYA))$_{<e,t>}$ PIM$_e$

CHASE$_{<e,<e,t>>}$ RONYA$_e$

What we see is the following:
Predicate logic represents the *thematic structure*: the first argument of the verb CHASE is the agent, the second argument of the verb is the theme.

Type logic represents the *constituent structure* (ignoring left-right order).
CHASE combines *first* with the object *ronya*, because that gives an interpretation for VP *chase ronya*:

(CHASE(RONYA)) of type <e,t>,
denoting the set of individuals that chase ronya,

The interpretation of the VP at type <e,t>, then combines with the subject *pim*, expressing that pim chases ronya.

We chose the specific curried interpretation which applies first to the theme and then the agent for linguistic reasons: that is the way the constituent structure works, and this way we get a theory that compositionally interprets a complex sentence in terms of its constituent parts.

Type logic introduces a lot of brackets, we get rid of some of them, by introducing relational notation (i.e. predicate logical notation) to simplify our expressions:

Let $\beta \in \text{EXP}_{<a,<b,t>>}$ and $\alpha_1 \in \text{EXP}_b$ and $\alpha_2 \in \text{EXP}_a$, then we define:

$\beta(\alpha_1, \alpha_2) =_{df} ((\beta(\alpha_2))(\alpha_1)$

Thus, with the notation convenion we can write:

((CHASE(RONYA))(PIM) in relational notation as:
CHASE(PIM, RONYA)

But note that this is not itself a generated expression of type logic: we generate and interpret ((CHASE(RONYA))(PIM), but read this, for simplicity, as: CHASE(PIM, RONYA).

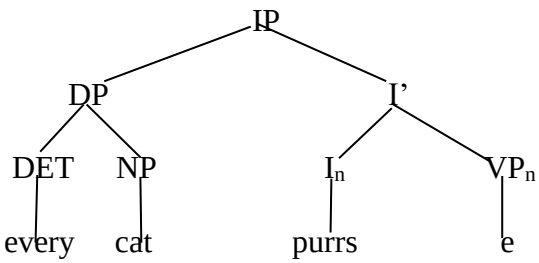
There is nothing deep about this: the logical language is there to help you.
Since you are used to writing this in relational notation, you may as well make the logical language useful and define a form that is easiest to read for you.
So the notation convention is for convenience.

The choices of the the type theory are made for grammatical reasons, though:
We choose curried functions, because that is easiest for the interpretations of syntactic trees.
We choose curried functions that work from the back because that is easiest for the interpretation of constituent structures.

<<e,t>,<<e,t>,t>> is the type of determiners

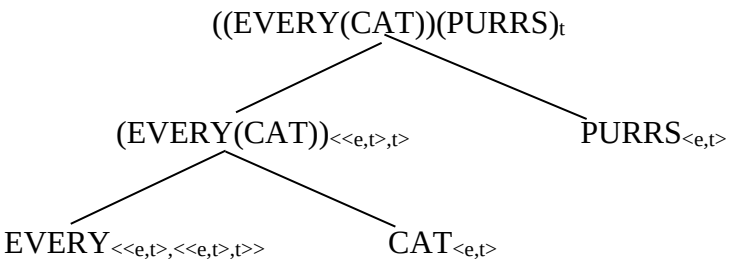
This type is of the form: <a, <a,t>>, with a = <e,t>, the type of sets of individuals.
Since <a,<a,t>> is the type of 2-place relations between a-entities, <<e,t>,<<e,t>,t>> is the type of 2-place relations between sets of individuals.
In Generalized Quantifier Theory this is, as we have seen, exactly what determiners denote.
So we take <<e,t>,<<e,t>,t>> to be the type of determiners.

Now, once again look at a syntactic tree:



We assume: EVERY ∈ $EXP_{<<e,t>,<<e,t>,t>>}$, CAT ∈ $CON_{<e,t>}$, PURRS ∈ $CON_{<e,t>}$

Obviously, we follow in the semantics the constituent structure and derive:



In the domain <<e,<e,t>> we relied on the isomorphism which curries ordered pairs from the back, as expressed in the notation convention:

$$\beta(\alpha_1, \alpha_2) =_{df} ((\beta(\alpha_2))(\alpha_1)$$

The *second* element of $<\alpha_1,\alpha_2>$ is the *first* argument in in the curried function.
The reason we chose this isomorphism is constituent structure: the transitive verb forms a constituent with the object DP (α_2).

In the nominal domain, the constituent structure works the other way round.
In Generalized Quantifier Theory we write:
EVERY[CAT, PURRS] for what we derive in TL:
((EVERY(CAT)) (PURRS))
If we use currying from the back, we should write:
EVERY(PURRS, CAT), meaning that every cat purrs.

This is a terrible notation, because it is not mnemonic.
So we don't chose this notation and we don't choose this isomorphism.
Instead we chose the isomorphism that involves currying from the front,
and we introduce a second notational convention, distinguished from the first by square brackets:

Let $\beta \in EXP_{<a,<b,t>>}$ and $\alpha_1 \in EXP_a$ and $\alpha_2 \in EXP_b$, then we define:

$$\beta[\alpha_1, \alpha_2] =_{df} ((\beta(\alpha_1))(\alpha_2)$$

Now indeed we can write ((EVERY(CAT)) (PURRS)) in the mnemonic notation:
EVERY[CAT, PURRS].

The moral is: what types we choose for our expressions depends on the syntax and the semantics of the constructions we are studying.
We use the isomorphism, and sometimes encode this in the notation conventions to make the TL expressions we use more readable.
The motto here is: your functional domain is isomorphic to many other structures.
Use anything that helps you to think about your expressions in the simplest way.
Sometimes, as we have seen, that means imposing a relational perspective, sometimes, it doesn't.

For instance, modifiers like *smart* are of type <<e,t>,<e,t>>, functions from sets to sets.
They are also, of course, 2-place relations between individuals and sets of individuals.
So instead of writing
((SMART(CAT))(RONYA)),
we *could* also use relational notation and write:
SMART(RONYA, CAT).

There is nothing technically wrong with that, but it doesn't help us:
it is *more* insightful to think of *smart* as denoting a function from cats to smart cats (mapping <e,t> onto <e,t>).
So we go for the way of thinking about the type that helps us most.

<<e,t>,t> is the type of Generalized Quantifiers, DP denotations

We see this in the above denotation:
we have derived an interpretation for the DP *every cat*, namely (EVERY(CAT)) at type <<e,t>,t>.

<<e,t>,t>, the type of generalized quantifiers plays a central, pivotal role in the semantic applications of type theory.
There are mathematical reasons for why that is so that will be discussed briefly later.

We saw that some noun phrases can have an interpretation at type e.
It has also been argued that some noun phrases can have an interpretation at the predicate type <e,t> (we will see that later).
As has been argued in the work of Montague, Lewis, and most specifically Barwise and Cooper 1982, all noun phrases can have an interpretation at the type <<e,t>,t>.

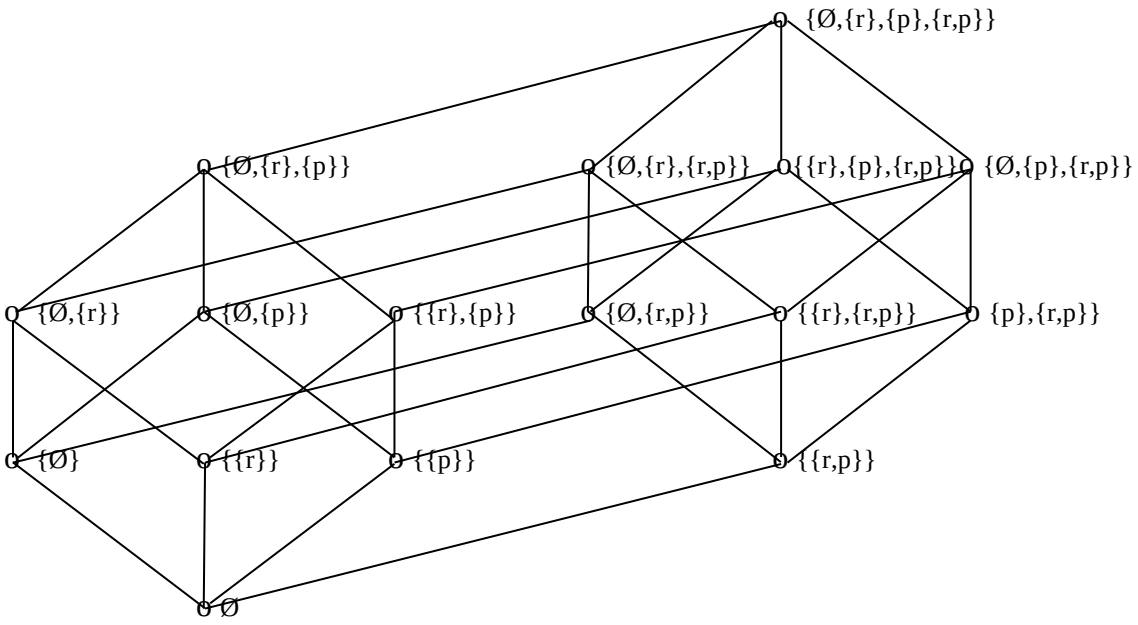
Since <e,t> is the type of sets of individuals, <<e,t>,t> is the type of functions from sets of individuals into truth values.
These are, of course characteristic functions of sets again, hence, the type <<e,t>,t> is, up to isomorphism, the type of sets of sets of individuals.

Its domain is, up to isomorphism: $D_{M,<<e,t>,t>}$ =
$((D_M \rightarrow \{0,1\}) \rightarrow \{0,1\}))$ =
pow(**pow**(D_M)),
and its cardinality is $2^{2^{|D_M|}}$. This means that if there are two individuals in the domain, then there are 16 different noun phrase interpretations:

{r,p}

pow({r,p}) = {Ø, {r}, {p}, {r,p}}

pow(**pow**({r,p})) =



Which set of sets is denoted by which noun phrase depends, of course, on the denotation of the noun *cat* and the denotation of the determiner. We haven't fixed those yet here, but they can be derived from the denotations of the determiners given in Generalized Quantifier Theory, in terms of sets of sets:

$[\![\text{EVERY(CAT)}]\!]_{M,g} = \{X \in D_{<e,t>}: F_M(\text{CAT}) \subseteq X\}$

The set of all properties that every cat has = the set of all sets that contain every cat

$[\![\text{SOME(CAT)}]\!]_{M,g} = \{X \in D_{<e,t>}: F_M(\text{CAT}) \cap X \neq \emptyset\}$

The set of all properties that some cat has = the set of all sets that contain some cat

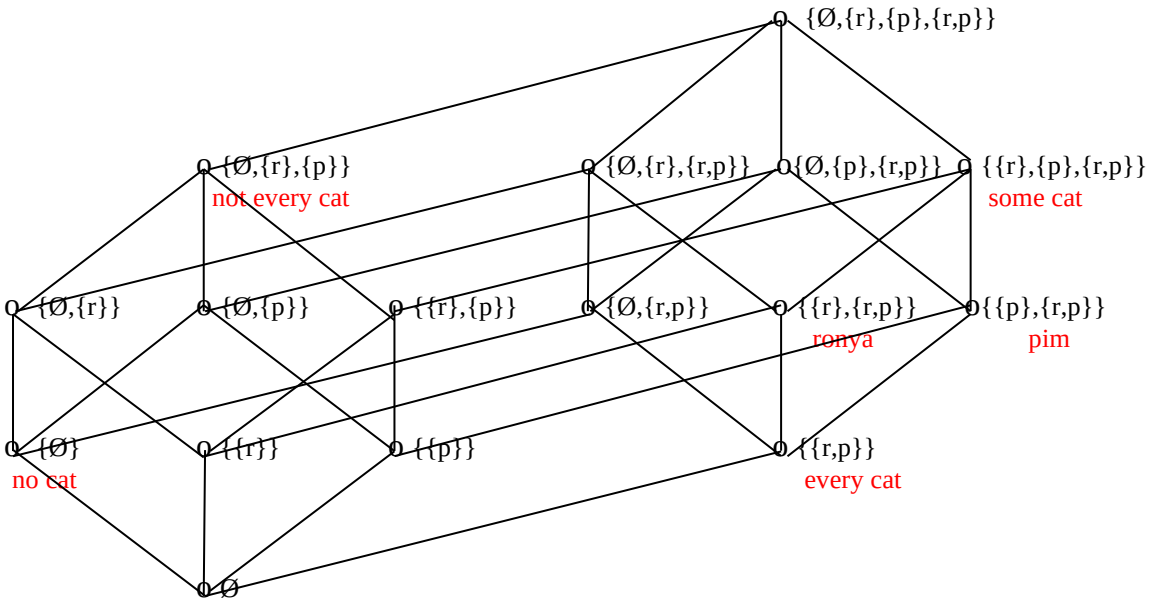
$[\![\text{NO(CAT)}]\!]_{M,g} = \{X \in D_{<e,t>}: F_M(\text{CAT}) \cap X = \emptyset\}$

The set of all properties that no cat has = the set of all sets that contain no cat

$[\![\text{RONYA}]\!]_{M,g} = \{X \in D_{<e,t>}: F_M(\text{RONYA}) \in X\}$

The set of all properties that ronya has = the set of all sets that contain ronya

If we assume that CAT = {r, p} , then on that domain *every cat* has the same interpretation as *everybody*, since there are only cats. We can take the above picture for that.



Explanation:
The set of all sets that contain ronya is {{r}, {r,p}}
The set of all sets that contain pim is {{p}, {r,p}}

Since ronya and pim are the only two cats *every cat* has the same denotation as *ronya and pim*, which should be the set of all sets that contain both ronya and pim, which is indeed {{r,p}}, and is indeed the set of all sets that every cat is in.

Similarly, *some cat* on this doman has the same denotation as *ronya or pim*, which should be the set of all sets that ronya is in, or pim is in, or both, which is {{r},{p},{r,p}}, and, of course, that indeed is the set of all sets that contain one or more cats.

Other DP denotations collapse on a domain with two elements into some of the given ones.

So, *not every cat* and *at most one cat* have the same denotation here, because, the set of all sets that do not contain every cat are the sets Ø, {r} and {p},
so *not every cat* = {Ø,{r},{p}}, but that is, of course, on this domain also the set of all sets that contain zero or one cat, i.e. at most one cat.

You could illustrate the distinctions better if you started out with three cats and a dog, {r,e,p,s}. The problem with that is the cardinality of $D_{<<e,t>,t>}$ in that case is 256, which is a bit cumbersome to draw and fit on the page in a readable format.

With this we are finished with functional application in type logic and come to the other even more salient feature of functional type theory: λ-abstraction and λ-conversion.

But first the ultimate summary of functional type theory:

A cat, a rat and a bat