CHAPTER THREE: BACKWARD λ-CONVERSION

We have informally shown how you go about finding the interpretations of expressions by reasoning about the the grammar, the types and the interpretations of other exptressions. We will do that now more formally by showing how to do this via backward λ-conversion.

3.1. Finding the meaning of attributival *old*

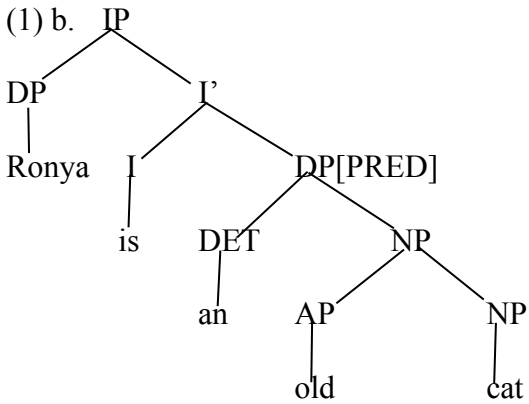
In the previous chapter I sketched how you go about finding the interpetation of a constituent.
I will be more precise about that here. Look at the example in (1):

(1) a. Ronya is an old cat

We are interested in finding the interpretation of *old*.
But we do that in the context of a grammar with syntactic and semantic assumptions.
What I am interested in is how you find an interpretation *given* your grammatical assumptions.

So look at (1). First we assume a syntactic analysis, for instance, the following:

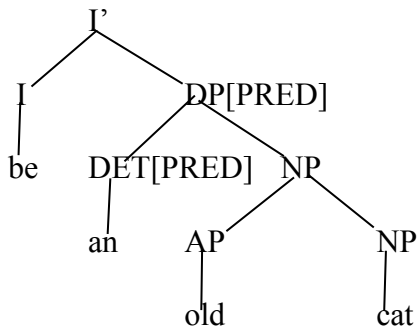
(1) b.



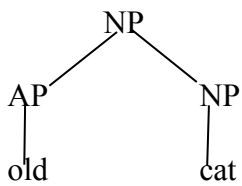
Secondly, we determine the truth conditions of (1). For instance the following:

(1) c. CAT(RONYA) ∧ OLD(RONYA)

Thirdly, we make a grammatical assumption, which is well motivated, but more importantly, useful for the present example.
And this assumption is that the following two tree structures have the same interpretation:



and



Next we make some assumptions about the relation between syntax and semantics.
These are assumptions that are grammar specific and in fact assumptions some of which we will modify later.

Assumption 1: Fixed Type Assignment

We associate with every syntactic category one and only one semantic type, and all expressions of that syntactic category are semantically interpreted as expressions of that semantic type.

Assumption 2: The interpretation of unary branching trees is identity
The interpretation of binary branching trees is functional application

Assumption 3: The grammar tells you for each binary tree what is the function and what is the argument.

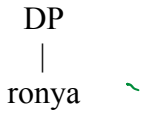
Not all semantically interpreted grammar models adhere to assumption 3, some allow flexibility in fixing which is the function and which is the argument. In these classnotes I will adhere to assumption 3, mainly for didactic reasons.

With respect to assumption 2, we will see later see other grammatical operations, in particular Function composition and Abstraction.
Also we will soon discuss type shifting operations which change not so much assumption 2 itself, but how we should interpret it.

Assumption 1 is the assumption that underlied Richard Montague's pioneering work on semantically interpreted grammars, in particular in his posthumously ublished paper PTQ, Montague 1973, The Proper Treatment of Quantification in ordinary english.
We will adhere to this assumption for the moment, again for didactic reasons, but drop it shortly when we introduce type shifting operations.

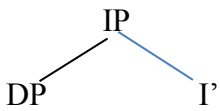
We go to the semantic interpretation of our tree.

Let us assume, for this example, that we interpret category DP as type e, and that the lexical item *ronya* is interpreted as RONYA ∈ CON_e.
Then the tree :



is also interpreted as RONYA, by assumption 2.

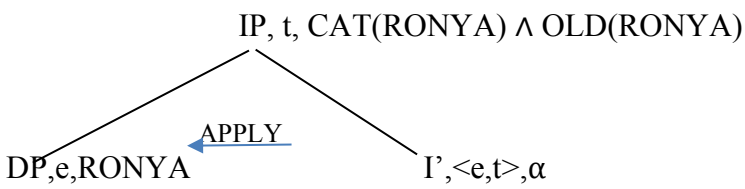
Also by assumption 2 the tree



is interpreted by function application, and we know that the interpretation of the IP is CAT(RONYA) ∧ OLD(RONYA) of type t, which fixes t as the type for IP.

From this it follows that in this tree I' is the function and DP is the argument, because type e is not a type of function.
Moreover, it also follows that the type of I' is <e,t>, because the input of the function is the argument, of type e and the output is type t.

We have derived the following information:



We are interested in determining the interpretation of I'. We know it has to be an expression of type <e,t>. Given the information that we have, we have to solve the following equation:

$$(\alpha(\text{RONYA})) = ((\text{CAT}(\text{RONYA})) \wedge (\text{OLD}(\text{RONYA})))$$

To solve this equation we need to bring the right hand side in the same form as the left hand side: an expression that applies to ronya. We do that by *backward λ-conversion*.
Backward λ-conversion is just λ-conversion:

$$\lambda x \beta(\alpha) = \beta[\alpha/x]$$

We have used this principle to simplify expressions, but it is an identity and hence can also be used in the other direction.

$$\text{CAT}(\text{RONYA}) \wedge \text{OLD}(\text{RONYA}) = [\lambda x.((\text{CAT}(x) \wedge (\text{OLD}(x))](\text{RONYA})$$

How to pull ronya out: replace the two occurrences of ronya by variable x,

CAT(RONYA) ∧ OLD(RONYA)
CAT(x) ∧ OLD(x)

abstract over x

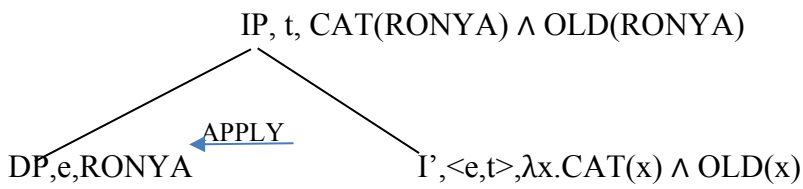
λx.CAT(x) ∧ OLD(x)

and apply to RONYA

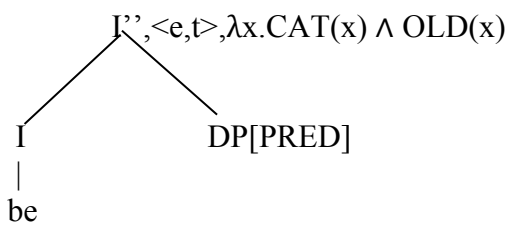
[λx.CAT(x) ∧ OLD(x)](RONYA)

We have solved our equation, because we see that:

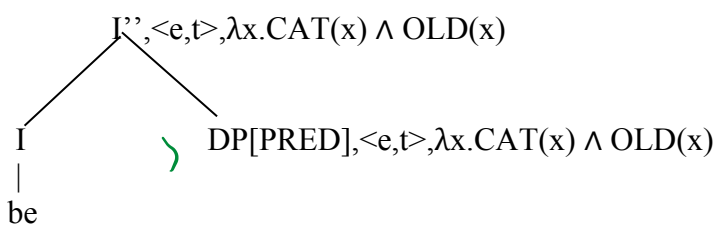
α = λx.CAT(x) ∧ OLD(x)



Now we can go down in the tree. At the next level we have:



Now we use the assumption we made above: the interpretation of the I' is the same as the interpretation of the DP[PRED]. Since I' is interpreted at type <e,t>, this means that DP[PRED] must be interpreted as type <e,t> as well. [Note that with the fixed type assumption this means that we treat the categories DP and DP[PRED] as different categories, since we will later assume that DP is interpreted as type <<e,t>,t>. We come back to these issues in later chapters.]



This means that the copula I must be of type <<e,t>,<e,t>>, because only a function of that type can resolve a functiona argument structure where either function or argument is <e,t> and the output is <e,t> as well, and since the interpretation of I and *be* is the same, we calculate the interpretation of *be* from

BE(λx.CAT(x) ∧ OLD(x)) = λx.CAT(x) ∧ OLD(x)

We resolve this with backward λ-conversion: we need to massage λx.CAT(x) ∧ OLD(x) into an expression in which a function applies to it:

Replace λx.CAT(x) ∧ OLD(x) by a variable P, abstract over the variable and apply the result to λx.CAT(x) ∧ OLD(x)

λP.P(λx.CAT(x) ∧ OLD(x))

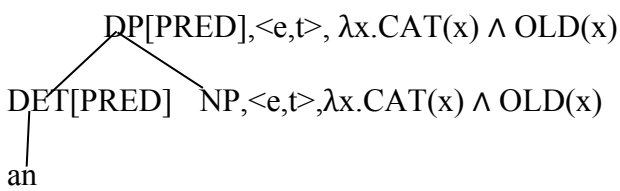
And we have found the interpretation of *be*:

be → λP.P of type <<e,t>,<e,t>>

Thus the copula is interpreted as the identity function, meaning that it doesn't have any specific semantic content (which is good).

With this out of the way we move to DP[PRED]

The assumption that we made was that DP[PRED] has the same interpretation as the NP.

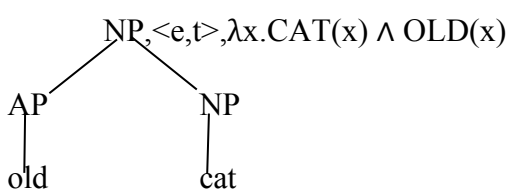


At this point we only need to observe that the semantic situation is identical to what we saw above one level up, so by exactly the same reasoning we derive tha DET[PRED] and therefore a(n) also has the interpretation λP.P

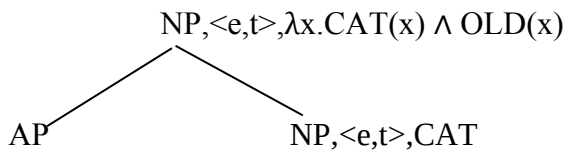
a(*n*) → λP.P of type <<e,t>,<e,t>>

So in this situation the indefinite article is also interpreted as the identity function, meaning that it doesn't have any specific semantic content (which is also good).

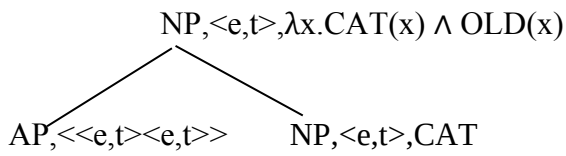
So we have derived:



Now, the UniqueType Assignment tells us that NP is interpreted at type <e,t>. We assume that *cat* is interpreted as CAT ∈ $CON_{<e,t>}$, so we derive:



We have seen this situation before: the only way we can resolve this type assignment is by assuming that AP is of type <<e,t>,<e,t>>:



And this gives us the following equation to solve:

AP(CAT) = λx.CAT(x) ∧ OLD(x)

And we solve this equation in the same way as above with backward λ-conversion:

λx.CAT(x) ∧ OLD(x)

We need to pull CAT out, so we replace it by a variable P, abstract over that variable and apply to CAT:

λx.CAT(x) ∧ OLD(x) =
[λPλx .P(x) ∧ OLD(x)](CAT)

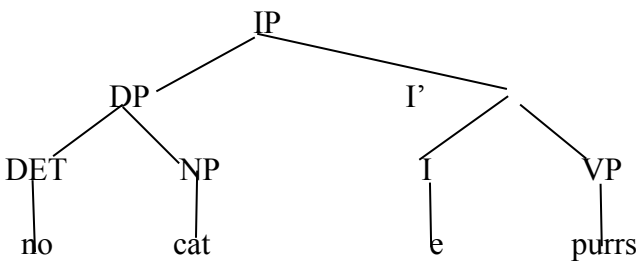
And we have solved the equation, and derive with the assumption about unary branching:

old → λPλx.P(x) ∧ OLD(x) of type <<e,t>,<e,t>>

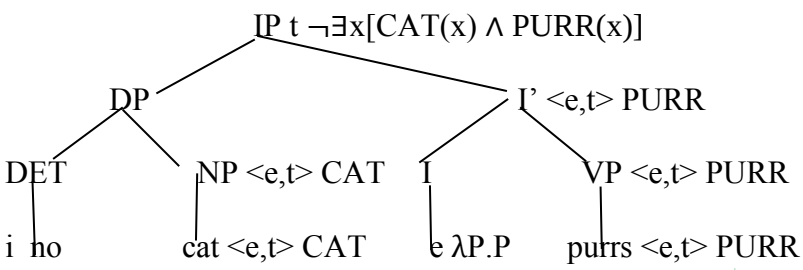
3.2. Finding the meaning of the determiner *no*

We find the meaning of the determiner *no* in (2) in the same way, but it involves setting up the grammar differently from the previous example, and hence, by the assumptions, involves a change that must be added to the analysis of (1) above as well.

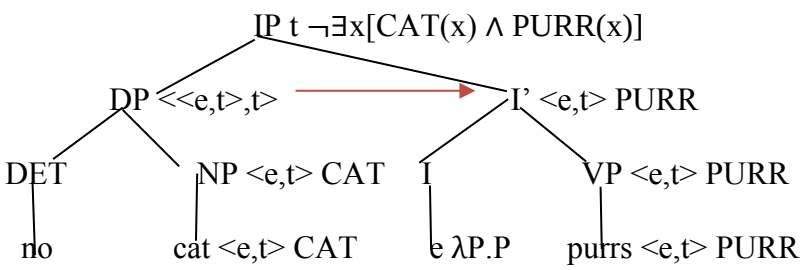
(2) No cat purrs



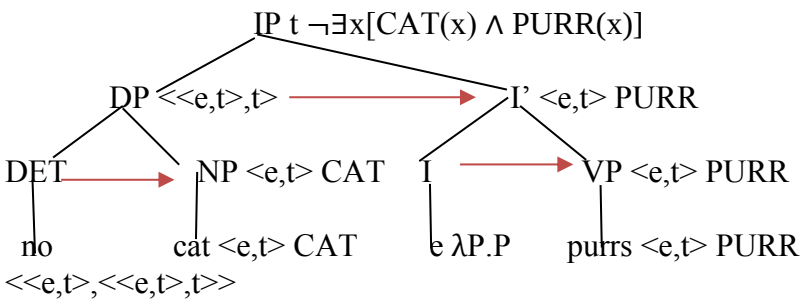
First, we will assume that the intransitive verb *purr* is interpreted as a constrant PURR ∈ $CON_{<e,t>}$, and we assume, as before that *cat* is interpreted as CAT ∈ $CON_{<e,t>}$.
And we accept the usual truth conditions for the IP of type t.
This fixes the following interpretations:



Here we know that *no cat* is not of type e.
That means that we cannot assume that the I' of type <e,t> is a function on the DP of type e.
As we have seen in our intuitive example earlier, the simplest solution is to assume the inverse function argument structure: let the interpretation of the DP be a function on the interpretation of the I', since the latter is <e,t> and the output is t, this suggests that we assign type <<e,t>,t> to the DP:



This assignment fixes also the type of the determiner. *no* cannot be the argument, because then it must be of type e and the output would be of type t, which it isn't. So *no* must be the function, and its argument is of type <e,t>, and output of type <<e,t>,t>, so *no* is of type <<e,t>,<<e,t>,t>>>, a relations between sets:



We need to find the interpretation of the DP *no cat*.
This involves solving the equation:

DP(PURR) = ¬∃x[CAT(x) ∧ PURR(x)]

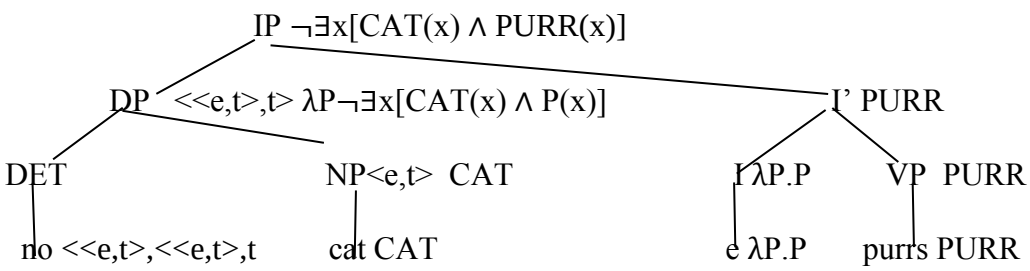
As before, we do this with backward λ-convertion:
We start with:

¬∃x[CAT(x) ∧ PURR(x)]

and we convert PURR out: replace it by a variable, abstract over the variable, and apply to PURR:

[λP.¬∃x[CAT(x) ∧ P(x)]](PURR)

λP.¬∃x[CAT(x) ∧ P(x) is of the right type of generalized quantifiers: <<e,t>,t>.
So:



Also the interpretation of the determiner is derived with backward λ-conversion:

DET(CAT) = λP.¬∃x[CAT(x) ∧ P(x)

We start with:

λP.¬∃x[CAT(x) ∧ P(x)]

We convert CAT out:

λQλP.¬∃x[Q(x) ∧ P(x)](CAT)

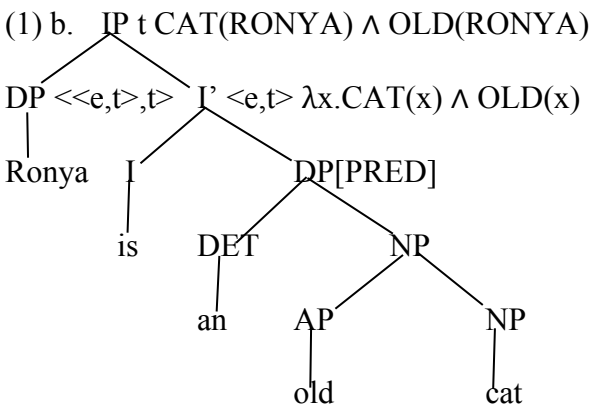
and get for DET and hence for *no*:

no → λQλP.¬∃x[Q(x) ∧ P(x)

The relation that holds between two sets if their intersection is empty.

This is, of course, exactly the relation we derived before.

But now we should go back to our earlier example:



The problem is, before we interpreted *ronya* as RONYA ∈ CON_e and the I'with interpretation at type <e,t> applied to this.
However, we cannot interpret DP *no cat* at type e, we must interpret it at type <<e,t>,t>.
But then, with the fixed type assumption, we must interpret the DP *ronya* also at type <,e,t>,t>.
But how?
The answer is: by solving the equation with backward λ-conversion:

DP(λx.CAT(x) ∧ OLD(x)) = CAT(RONYA) ∧ OLD(RONYA)

As always, we start with:

CAT(RONYA) ∧ OLD(RONYA)

We convert RONYA out, replace it by a variable x, abstract and apply to RONYA:

[λx.CAT(x) ∧ OLD(x)](RONYA)

However, this is not enough, because it is not of the form DP(λx.CAT(x) ∧ OLD(x)): the DP material is the argument, but it should be the function.
How do we solve that?
The answer is: by solving the equation with backward λ-conversion:
We convert the <e,t> predicate λx.CAT(x) ∧ OLD(x)] out by replacing it by a variable P

P(RONYA)

abstracting over P

λP.P(RONYA)

and applying the result to λx.CAT(x) ∧ OLD(x):

λP.P(RONYA)(λx.CAT(x) ∧ OLD(x))

With this we have found:

ronya → λP.P(RONYA) of type <<e,t>,t>

The set of all properties that RONYA has.

So, while we associate with the lexical item *ronya* a constant RONYA ∈ CON_e, we don't interpret *ronya* as that constant (at type e), but as the expression λP.P(RONYA).

Notice that we did exactly the same with *old*: we associated with *old* a constant OLD ∈ $CON_{<e,t>}$, but translated it as λPλx.P(x) ∧ OLD(x).
The case is not quite the same, because it is reasonable to argue that when *old* is a predicative adjective, AP[PRED] (as in *Ronya is old*) it *is* interpreted as OLD.

But this is Montague's interpretation strategy, it is called *Generalize to the Worst Type*: because some DPs (like *no cat*) need to be interpreted at the higher type <<e,t>,t>, all DPs need to be interpreted there. We will soon step away from this strategy, but it is good to point out that it has some excellent consequences. For instance for DP conjunction: *Ronya and every kitten purr*.

We have given the interpretation for *and* at type DP as:

λUλTλP.T(P) ∧ U(P)

If we interpret *ronya* as λP.P(RONYA), we can unproblematically enter the two DP interpretations into the schema:

λUλTλP.T(P) ∧ U(P)(λZ.∀x[KITTEN(x) → Z(x)]) =

λTλP.T(P) ∧ λZ.∀x[KITTEN(x) → Z(x)](P) $=_{\lambda\text{-conversion}}$

λTλP.T(P) ∧ ∀x[KITTEN(x) → P(x)] $=_{\lambda\text{-conversion}}$

λTλP.T(P) ∧ ∀x[KITTEN(x) → P(x)](λZ.Z(RONYA)) $=_{\lambda\text{-conversion}}$

λP.λZ.Z(RONYA))(P) ∧ ∀x[KITTEN(x) → P(x)] $=_{\lambda\text{-conversion}}$

λP.P(RONYA) ∧ ∀x[KITTEN(x) → P(x)] $=_{\lambda\text{-conversion}}$

The set of properties that Ronya shares with every kitten

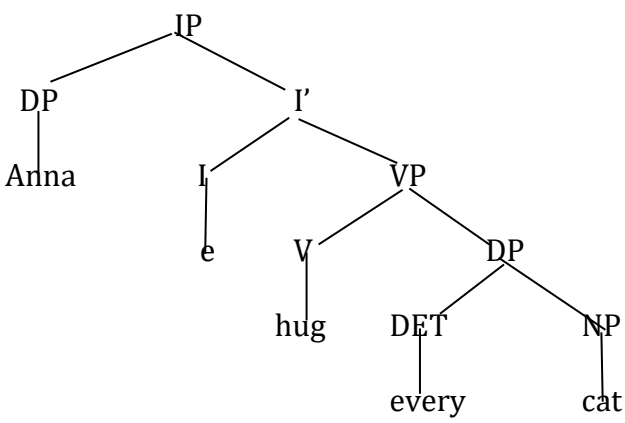
If we interpret *ronya* as RONYA we cannot use this scheme and would have to assume four schemas for DP conjunction (for the Boolean, distributive interpretations):

λyλxλP.P(x) ∧ P(y)	Ronya and Pim
λUλxλP.P(x) ∧ U(P)	Ronya and every kitten
λyλTλP.T(P) ∧ P(y)	Every kitten and ronya
λUλTλP.T(P) ∧ U(P)	Every kitten and some old cat

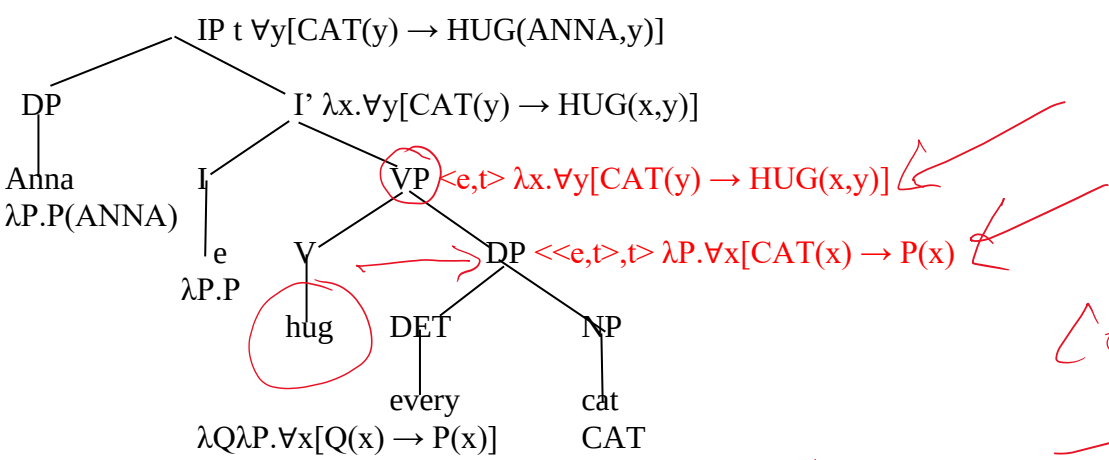
3.3. Finding the meaning of *hug every cat*

To complete Montague's analysis of verbs and their arguments we need to solve one more equation. Look at (3):

(3) Anna hugged every cat



The theory that we have given so far, gives us the following information:



Just to repeat the top bit:

We convert ANNA out in :

∀y[CAT(y) → HUG(ANNA,y)]

which gives:

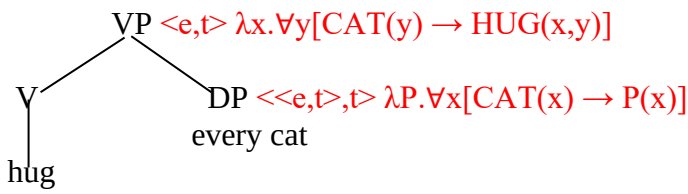
$\lambda x.\forall y[CAT(y) \rightarrow HUG(x,y)](ANNA)$

Then we convert $\lambda x.\forall y[CAT(y) \rightarrow HUG(x,y)]$ out and get:

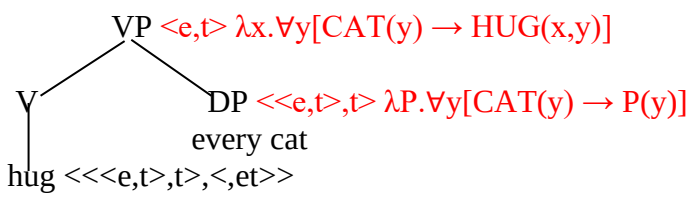
$P(\lambda P.ANNA)(\lambda x.\forall y[CAT(y) \rightarrow HUG(x,y)])$

Which gives $\lambda x.\forall y[CAT(y) \rightarrow HUG(x,y)]$ for the I', as indicated.

Ignoring all the irrelevant material, we have to find the interpretation for the following structure:



Now so far, we assumed that *hug* was interpreted as HUG ∈ $CON_{<e,<e,t>>}$, and is the function on the complement DP. That was ok when the DP was *ronya* of type e, but is a problem here, where the complement is *every cat* of type <<e,t>,t>.
It won't do to change the function-argument structure, because that doesn't fit the input and output. So we have no choice but to assume that *hug* is interpreted as the function, and that makes it a function of type <<<e,t>,t>,<e,t>> :



The interpretation of *hug* is a function that gobbles up a generalized quantifier, the set of properties that every cat has, and spits out a one place property, the property that you have if you hug every cat. And out task is to find out what function that is.
How do we do that?
The answer is: by solving the equation with backward λ-conversion:

$V(\lambda P.\forall y[CAT(y) \rightarrow P(y)]) = \lambda x.\forall y[CAT(y) \rightarrow HUG(x,y)]$

Solving this equation is more difficult than any of the ones we have seen before.
Let us start by writing HUG(x,y) into its TL form:

$V(\lambda P.\forall y[CAT(y) \rightarrow P(y)]) = \lambda x.\forall y[CAT(y) \rightarrow [HUG(y)](x)]$

The problem is best addressed from the back: we want to get to an expression in which the V applies to the generalized quantifier:

$V(\lambda P.\forall y[CAT(y) \rightarrow P(y)])$

We get there if we can massage λx.∀y[CAT(y) → [HUG(y)](x)] into a form that has the generalized quantifier as a subexpressions:

.....[λP.∀y[CAT(y) → P(y)]]....

because, then we can convert it out and get the right form:

λT...[T]...(λP.∀y[CAT(y) → P(y)])

i.e. then V = λT...[T]...

Now we can massage λx.∀y[CAT(y) → [HUG(y)](x)] into .....[λP.∀y[CAT(y) → P(y)]].... if we can massage into an expression that has for some α of type <e,t>
∀y[CAT(y) → α(y)] as subformula:

.....[∀y[CAT(y) → α(y)]]....

Because then we can convert α out at the level of that subformula and get:

.....[λP.∀y[CAT(y) → P(y)]](α)....

which is exactly the right form to continue.

So we look at:

λx.∀y[CAT(y) → [HUG(y)](x)]

and are looking for:

...∀y[CAT(y) → α(y)]

This is, what we call *close, but no sigar*.
We want an expression of the form α(y), but what we have is an expression of the form α(x).
And that is not good enough.
What do we do?
The answer is: we solve the equation with backward λ-convertion.

α(y) = [HUG(y)](x)

Convert y out: replace it by variable z, abstract over z, and apply to y:

[HUG(y)](x)
[λz.HUG(z)](x)](y)

In fact, we can write this in relational form to make it more legible:

[λz.HUG(x,z)](y)

So λz.HUG(x,z) is the α we are looking for.

This is the missing link. Now we can write down the whole derivation:

λx.∀y[CAT(y) → HUG(x,y)]	= [convert y out by abstraction over z ∈ VAR_e]
λx.∀y[CAT(y) → ([λz.HUG(x,z)](y)])	= [convert λz.HUG(x,z) out by abstraction over P ∈ $VAR_{<e,t>}$]
λx.([λP.∀y[CAT(y) → P(y)]](λz.HUG(x,z))	=[convert λP.∀y[CAT(y) → P(y)] out] by abstraction over T ∈ $VAR_{<<e,t>,t>}$]
[λTλx.T(λzHUG(x,z))](λP.∀y[CAT(y) → P(y)])	

Hence we derive as the interpretation of *hug*:

hug → λTλx.T(λz.HUG(x,z))

Let's check that the type is right:

λz.HUG(x,z)	is of type <e,t>
T(λz.HUG(x,z))	is of type t
λx.T(λz.HUG(x,z))	is of type <e,t>
λTλx.T(λz.HUG(x,z))	is of type <<<e,t>,t>,<e,t>>

λTλx.T(λy.HUG(x,y)) is
the relation that holds between individual x and set of properties T iff
being hugged by x is one of the properties in T

So Anna stands in this relation to the set of all properties that every cat has
if Being Hugged by Anna is one of the properties in that set,
which is the case if Being Hugged by Anna is a property that every cat has
which is the case if every cat is being hugged by Anna
which is the case if Anna hugs every cat.

λTλx.T(λy.R(x,y)) is the correct interpretation at the type <<<e,t>,t>,<e,t>> for all extensional- non-collective transitive verb interpretations R.

The intuition is: you want to hug an individual, but you feed *hug* a generalized quantifier. You can't hug a generalized quantifier, so how do you express this in terms of hugging individuals. The relation you want at type <<<e,t>,t>,<e,t>> is the relation that is as close to the hug-relation at type <e,<e,t>> as possible.
What does this mean?

1. **Extension:** Individual RONYA at type e corresponds to generalized quantifier λP.P(RONYA) at type <<e,t>,t>.
   The interpretation of $HUG_{<<<e,t>,t>,<e,t>>}$ should do with λP.P(RONYA) what $HUG_{<e,<e,t>>}$ does with RONYA:

$$HUG_{<<<e,t>,t>,<e,t>>}(\lambda P.P(RONYA) = \lambda x.HUG_{<e,<e,t>>}(x,RONYA)$$

This is of course the case:
$\lambda T\lambda x.T(\lambda y.HUG(x,y))(\lambda P.P(RONYA)) =$
$\lambda x.[\lambda P.P(RONYA)(\lambda y.HUG(x,y))] =$
$\lambda x.[\lambda y.HUG(x,y)(RONYA)] =$
$\lambda x.HUG(x,RONYA)$

2. **Homomorphism**: for all the other generalized quantifiers, you want their behaviour to be lifted from the behaviour for generalized quantifiers corresponding to individuals:

Again this is the case:

$\lambda T\lambda x.T(\lambda y.HUG(x,y))(\lambda P.\forall y[CAT(y) \rightarrow P(y)]) = \lambda x.\forall y[CAT(y) \rightarrow HUG(x,y)]$
$= \lambda x.HUG(x,CAT_1) \wedge HUG(x,CAT_2) \wedge ...$

$\lambda T\lambda x.T(\lambda y.HUG(x,y))(\lambda P.\exists y[CAT(y) \wedge P(y)]) = \lambda x.\exists y[CAT(y) \wedge HUG(x,y)]$
$= \lambda x.HUG(x,CAT_1) \vee HUG(x,CAT_2) \vee ...$

$\lambda T\lambda x.T(\lambda y.HUG(x,y))(\lambda P.\neg\exists y[CAT(y) \wedge P(y)]) = \lambda x.\neg\exists y[CAT(y) \wedge HUG(x,y)]$
$= \lambda x.\neg HUG(x,CAT_1) \wedge \neg HUG(x,CAT_2) \wedge ...$

Theorem: for every relation R of type <e,<e,t>> (= extensional, non-collective, i.e. not *seek* and not *combine*) there is **exactly one** relation, the *homomorphism extending* R, that satisfies both these conditions, and that relation is $\lambda T\lambda x.T(\lambda y.R(x,y))$.

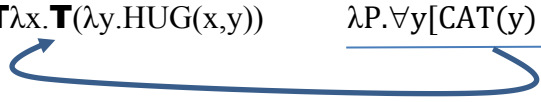
Connection with scope and distributivity:
Syntactic tree of *John kissed every girl:*
every girl is in the syntactic scope of *kissed*.

Frege's theory of quantifiers and relations (*Begriffsschrift*, 1879,introduction: do not interpret the object *in situ*, but give it semantic scope *over* the verb:
$\forall y[girl(y) \rightarrow kiss(j,y)]$

Montague's observation: we can do this *without* raising, the translation $\lambda T\lambda x.T(\lambda y.R(x,y))$ does *exactly* that. You see that in the TL expression: If we apply this to the interpretation of the object, the generalized quantifier α, then α lands in the T position, taking semantics scope over relation R:

Mismatch between syntactic scope and semantic scope:

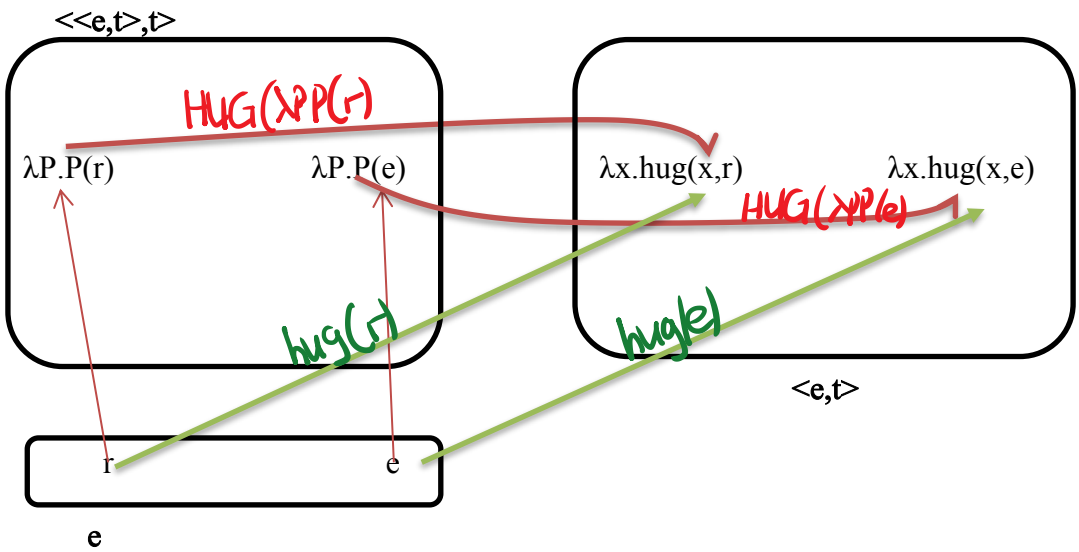
[*hug* *every cat*]
$\lambda \mathbf{T}\lambda x.\mathbf{T}(\lambda y.HUG(x,y))$ $\lambda P.\forall y[CAT(y) \rightarrow P(y)]$



the object *every cat* is in the syntactic scope of the verb *hug*, but its interpretation gets converted into a 'semantic position' where it takes scope *over* the interpretation of the verb.

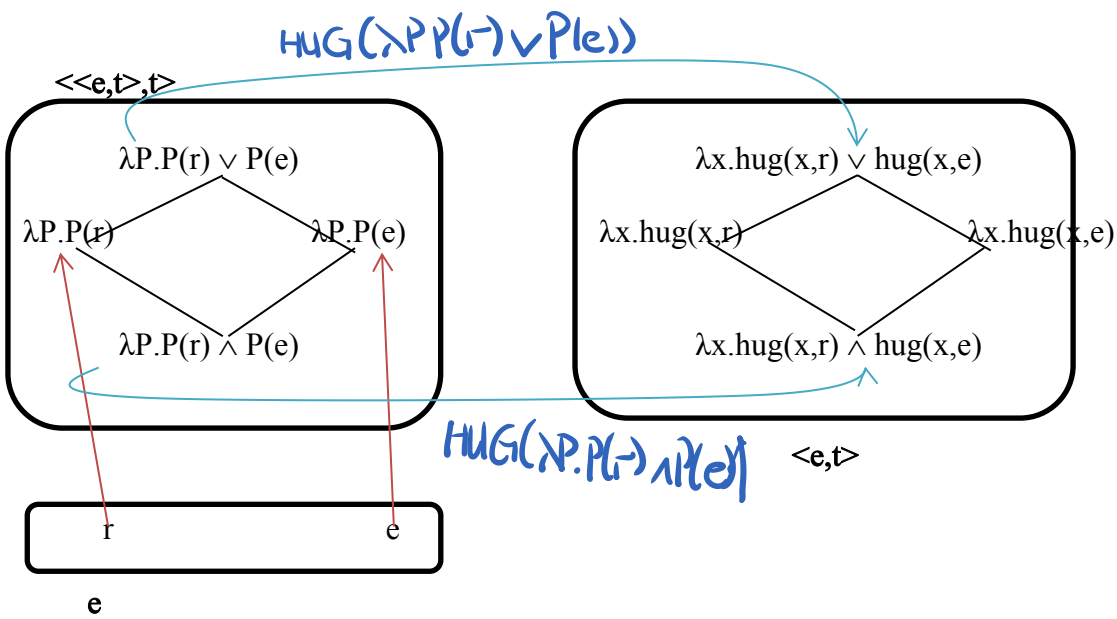
Let hug be $HUG_{<e,<e,t>>}$ and let HUG be $HUG_{<<<e,t>,t>,<e,t>>}$

1. HUG **extends** hug
   For every d ∈ D_e: (HUG(λP.P(d)) = λx.hug(x,d)



2. Homomorphism:

   HUG(λP.P(RONYA) ∨ P(EMMA)) = λx.hug(x,RONYA) ∨ hug(x,EMMA)
   HUG(λP.P(RONYA) ∧ P(EMMA)) = λx.hug(x,RONYA) ∧ hug(x,EMMA)
   HUG(λP.¬P(RONYA)) = λx.¬HUG(x,RONYA)



The theorem is a consequence of a deep theorem in Universal Algebra: the Fundamental Theorem for *Free* Algebras. It so happens that $D_{<<e,t>,t>}$ is not just a Boolean algebra, but a free, freely generated, Boolean algebra, and so is, in general any domain $D_{<<a,t>,t>}$, with a any type. Free Algebras are (relative to their cardinality) the most general of their type. This means that homomorphic extensions exist from any type<a,b> to type <<<a,t>,t>,b>.
It is this structure that underlies Montague's (and our more modern type shifting) analysis of relations and their arguments.

3.4. Cross categorial negation, conjunction and disjunction.

The Boolean operators *not*, *and* and *or* are cross-categorial, we find them across different syntactic categories, so we can conjoin sentences, VPs, DPs, but also, say, prepositions, as in *drinks are served* ***before and after*** *the show*. We can in fact in some circumstances conjoin *non-constituents* if they are of the same semantic type, as in ***Ronya likes but Pim hates*** *tuna*.

When I say Boolean operations, I mean *not*, *and*, *or* with their standard Boolean interpretation. I mean something very specific by that.

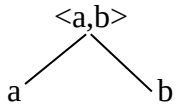
The Lifting Theorem tells us that if A is a set and B a Boolean algebra with operations $\neg_B, \wedge_B, \vee_B$, then the set of all functions form A into B, $(A \rightarrow B)$ forms a Boolean algebra under the operations $\neg_{(A \rightarrow B)}, \wedge_{(A \rightarrow B)}, \vee_{(A \rightarrow B)}$ that are lifted pointwise from B onto $(A \rightarrow B)$. We have seen that the interpretation domains of all Boolean types in BOOL form Boolean algebras in this way.

BOOL is the smallest subset of TYPE such that:
1. $t \in BOOL$
2. If $a \in TYPE$ and $b \in BOOL$ then $<a,b> \in BOOL$

Since e is not a Boolean type, but t is, the recursive definition of BOOL tells you that the Boolean operations on any Boolean type $b \in BOOL$, are ultimately lifted from $\neg, \wedge, \vee$, i.e. $\neg_t, \wedge_t, \vee_t$.

We will later in this class see a different notion of conjunction, sum conjunction, which is not Boolean in this sense (because it is in essence related to $\cup$ and not to $\cap$, while Boolean conjunction is related to $\cap$, as we have seen). But here we are only concerned with the connectives and their Boolean interpretation.

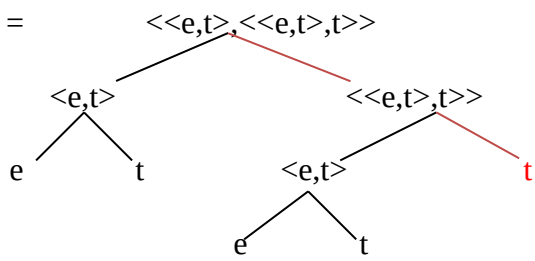
For type $a \in TYPE$, let TREE[a] be the decomposition tree of a
where: <a,b>



Fact: For any type $a \in TYPE$: $a \in BOOL$ iff the right branch in TREE[a] ends in t.

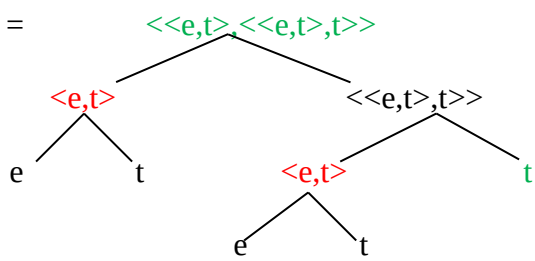
Example:

TREE[<<e,t>,<<e,t>,t>>] =



What the Boolean structure tells us is that the Boolean interpretation of *not*, *and*, *or* at any Boolean type is, in essence, just ¬, ∧, ∨, but adjusted to the higher type.
Given this, we may expect to be able to define a procedure telling us, for any Boolean type what the interpretation derived for ¬, ∧, ∨ are at that type. In fact, there are various such procedures. We will give a particularly simple one here.
Look at the example:

TREE[<<e,t>,<<e,t>,t>>] =



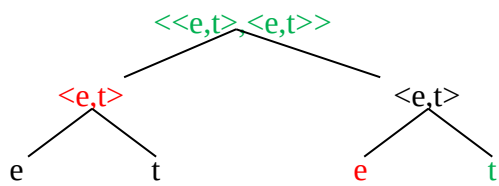
We ask: If we have a functional expression of type <<e,t>,<<e,t>,t> how do we get from there to t? the answer is, by feeding it an expression of type <e,t>, and then feeding it another expression of type <e,t>, and presto, you are at t.

Let us call this sequence of types the ***argument sequence*** of <<e,t>,<e,t>,t>>:

ARG[<<e,t>,<<e,t>,t>>] = [<e,t> — <e,t>]

Rather than define it, I give an other example:

TREE[<<e,t>,<e,t>>] =



ARG[<<e,t>,<e,t>>] = [<e,t> — e]

We can now specify the algorithm.
Rather than giving the general form, I will give it for ARG[<<e,t>,<e,t>>] = [<e,t> — e]

ARG[<<e,t>,<e,t>>] = [<e,t> — e]

Step 1: For ¬ choose a variable of type <<e,t>,<e,t>>.
For ∧ and ∨ choose two different variables of type <<e,t>,<e,t>>.
For each type in ARG[<<e,t>,<e,t>>] choose a variable of that type.
In general, choose different variables if a type occurs more than once in ARG[a].

ARG[<<e,t>,<e,t>>] = [<e,t> — e]
V P x
W P x

Step 2: Apply each variable to its neighbour, left to right:

ARG[<<e,t>,<e,t>>] = [<e,t> — e]
V P x ⇒ ((V(P))(x)) of type t
W P x ⇒ ((W(P))(x)) of type t

Step 3: Apply ¬, ∧, ∨ to this at type t.

ARG[<<e,t>,<e,t>>] = [<e,t> — e]
V P x ⇒ ¬(V(P))(x)
W P x (V(P))(x) ∧ (W(P))(x)
(V(P))(x) ∨ (W(P))(x)

Step 4: Abstract over the all the variables in inverse order
ARG[<<e,t>,<e,t>>] = [<e,t> — e]
V P x ⇒ λV.λPλx¬(V(P))(x)
W P x λWλV.λPλx(V(P))(x) ∧ (W(P))(x)
λWλV.λPλx.(V(P))(x) ∨ (W(P))(x)

The intuition is:
¬ takes an input of type <<e,t>,<e,t>>: V and maps it onto an output of the same type
∧,∨ take two inputs of type <<e,t>,<e,t>>: V, W and maps it onto an output of the same type

You get the contents of the output, by bringing the input variables down to type t by application to variables: (V(P))(x) and (W(P))(x).
To those you can apply the Boolean connectives at type t,
and by abstracting over the variables used, you get the correct output of type <<e,t>,<e,t>>.

Let us apply it to a familiar type, to see that indeed we get the right result:

ARG[<e,<e,t>>] = [e — e]
R y x ⇒ (R(y))(x)
S y x (S(y))(x)
We form: λRλyλx.¬(R(y))(x)
λSλRλyλx.(R(y))(x) ∧ (S(y))(x)
λSλRλyλx.(R(y))(x) ∨ (S(y))(x)

or in relational notation:
λRλyλx.¬R(x,y)
λSλRλyλx.R(x,y) ∧ S(x,y)
λSλRλyλx.R(x,y) ∨ S(x,y)

3.5. Tarski's Trick: Defining in type logic all logical operations in terms of application, abstraction and identity

Theorem: You can, without loosing anything restrict the type logical language to include only expressions built from constants and variables with functional application, functional abstraction and identity. Everything else can be defined in this language.

Proof:

There are two objects of type t, namely 1 and 0, and there are four functions of type <t,t>, namely the identity function ID = {<0,0>, <1,1>}, the constant function on 1 C_1 = {<0,1>, <1,1>}, the constant function on 0, C_0 = {<0,0>, <1,0>} and negation ¬ = {<0,1>, <1,0>}.

We start by finding expressions of type logic denoting each of these.
Of course, these should all be expressions that only uses variables, application, abstraction and identity.

Let $\alpha \in VAR_t$.

We define:

ID := $\lambda\alpha.\alpha$

This is an expression of type <t,t>, and it denotes the identity function at type <t,t>, i.e. function {<1,1>,<0,0>}.

Next we define:

1 := $(\lambda\alpha.\alpha = \lambda\alpha.\alpha)$

This is an expression of type t, and it is a tautology, so indeed it denotes 1.

Note now that since we have defined 1 in the logical language, we can use 1 to define the others (replace it by its definition if you want to see only application, lambdas and identity).

Next is the constant function on 1:

C_1 := $\lambda\alpha.1$

This is an expression of type <t,t>, and it denotes the constant function on 1: it maps every α onto a tautology, which means that it maps 0 and 1 onto 1.

Now we can use what we have so far to define 0:

0 := $(\lambda\alpha.\alpha = \lambda\alpha.1)$

Since $\lambda\alpha.\alpha$ denotes the identity function and $\lambda\alpha.1$ denotes the constant function on 1, they are different function, and stating that they are the same function is a contradiction, 0.

$\neg$:= $\lambda\alpha.(\alpha = 0)$

The claim is that $\lambda\alpha.(\alpha = 0)$ denotes negation.
This can be seen with λ-conversion:

$\lambda\alpha.(\alpha=0)(0)$ $=_{\lambda\text{-conversion}}$
$(0 = 0)$ =
1

$\lambda\alpha.(\alpha=0)(1)$ $=_{\lambda\text{-conversion}}$
$(1 = 0)$ =
0

So indeed, $\lambda\alpha.(\alpha=0)$ maps 0 onto 1 and 1 onto 0, so it is negation.

Finally:

C_0 := $\lambda\alpha.0$

This is, by now, obvious.

So indeed we have defined 0, 1 and the four truth functions by using only variables, application, abstraction, and identity.

Our next step is to define the universal quantifier.

If $x \in VAR_a$ and $\varphi \in EXP_t$ and φ is definable with constants, variables, application, abstraction and identity, we want to define $\forall x\varphi$ with variables, application, abstraction and identity.
We do this by first defining an expression that for any type a denotes D_a.

Let $x \in VAR_a$.

D_a := $\lambda x.1$

$[\![\lambda x.1]\!]_{M,g}$ = that function $h:D_a \rightarrow \{0,1\}$ such that for every $d \in D_a$: $h(d)=1$.

The set characterized by this function is, of course, indeed D_a.

Now assume that φ is an expression that is definable with only constants, variables, application, abstraction and identity. We define:

$\forall x\varphi$:= $(\lambda x.\varphi = \lambda x.1)$

$[\![\lambda x.\varphi = \lambda x.1]\!]_{M,g} = 1$ iff $\{d \in D_a: [\![\varphi]\!]_{M,g^d_x} = 1\} = D_a$
iff for every $d \in D_a$: $[\![\varphi]\!]_{M,g^d_x} = 1$

The idea is: the set of entities in domain D_a that have property φ is the whole domain D_a iff every entity in D_a has property φ.

With this, we, of course define $\exists x\varphi$ in the usual way:

$\exists x\varphi := \neg\forall x\neg\varphi$.

All of these definitions are relatively straightforward.
And we are almost there: if we can define conjunction, $\varphi \wedge \psi$, by using only variables, application, abstraction and identity, we are done, because then we can define disjunction in the usual way as:

$(\varphi \vee \psi) := \neg(\neg\varphi \wedge \neg\psi)$

So we have only conjunction left.

And that is much more difficult, in fact, difficult enough that Tarski in essence got his MA in 1923 for solving this. Here is the definition that in essence goes back to Tarski's MA thesis (although the setting was not functional type theory, because that didn't exist yet).

Assume that φ and ψ are expressions of type t, defined only with constants, variables, application, abstraction and identity.

Let $f \in VAR_{<t,t>}$ ⟦

We define:

$(\varphi \wedge \psi) := \forall f[\ (f(\varphi) = f(\psi)) = \psi\ ]$

We want to show that the formula $\forall f[\ (f(\varphi) = f(\psi)) = \psi\ ]$ has exactly the same truth conditions as $(\varphi \wedge \psi)$ does.

$\forall f[\ (f(\varphi) = f(\psi)) = \psi\ ]$ is true relative to M,g

iff

$f(\varphi) = f(\psi)) = \psi$ is true relative to M,g^{d}_{f}, for each $d \in D_{<t,t>}$

iff

$f(\varphi) = f(\psi)) = \psi$ is true relative to M,g^{ID}_{f} and
$f(\varphi) = f(\psi)) = \psi$ is true relative to $M,g^{C_1}_{f}$ and
$f(\varphi) = f(\psi)) = \psi$ is true relative to $M,g^{C_0}_{f}$ and
$f(\varphi) = f(\psi)) = \psi$ is true relative to $M,g^{\neg}_{f}$

Since we have defined these four operations in the language, we get the following equivalence:

$f(\varphi) = f(\psi)) = \psi$ is true relative to M,g_f^{ID} and
$f(\varphi) = f(\psi)) = \psi$ is true relative to $M,g_f^{C_1}$ and
$f(\varphi) = f(\psi)) = \psi$ is true relative to $M,g_f^{C_0}$ and
$f(\varphi) = f(\psi)) = \psi$ is true relative to $M,g_f^{\neg}$

iff

$[(ID(\varphi) = ID(\psi)) = \psi]$ is true relative to M,g and
$[(C_1(\varphi) = C_1(\psi)) = \psi]$ is true relative to M,g and
$[(C_0(\varphi) = C_0(\psi)) = \psi]$ is true relative to M,g and
$[(\neg(\varphi) = \neg(\psi)) = \psi]$ is true relative to M,g

Now we can fill in the definitions of these four functional expressions:

$[(ID(\varphi) = ID(\psi)) = \psi]$ and
$[(C_1(\varphi) = C_1(\psi)) = \psi]$ and
$[(C_0(\varphi) = C_0(\psi)) = \psi]$ and
$[(\neg(\varphi) = \neg(\psi)) = \psi]$ are true relative to M,g

iff

$[(\lambda\alpha.\alpha(\varphi) = \lambda\alpha.\alpha(\psi)) = \psi]$ and
$[(\lambda\alpha.1(\varphi) = \lambda\alpha.1(\psi)) = \psi]$ and
$[(\lambda\alpha.0(\varphi) = \lambda\alpha.0(\psi)) = \psi]$ and
$[(\neg\varphi = \neg\psi) = \psi]$ are true relative to M,g

And we can simplify this with λ-conversion:

$[(\lambda\alpha.\alpha(\varphi) = \lambda\alpha.\alpha(\psi)) = \psi]$ and
$[(\lambda\alpha.1(\varphi) = \lambda\alpha.1(\psi)) = \psi]$ and
$[(\lambda\alpha.0(\varphi) = \lambda\alpha.0(\psi)) = \psi]$ and
$[(\neg\varphi = \neg\psi) = \psi]$ are true relative to M,g

iff

$[(\varphi = \psi) = \psi]$ and
$[(1 = 1) = \psi]$ and
$[(0 = 0) = \psi]$ and
$[(\neg\varphi = \neg\psi) = \psi]$ are true relative to M,g

Observation 1: $(\varphi = \psi)$ and $(\neg\varphi = \neg\psi)$ are logically equivalent
Hence $(\varphi = \psi) = \psi$ and $(\neg\varphi = \neg\psi) = \psi$ are logically equivalent

so:

$[(\varphi = \psi) = \psi]$ and
$[(1 = 1) = \psi]$ and
$[(0 = 0) = \psi]$ and
$[(\neg\varphi = \neg\psi) = \psi]$ are true relative to M,g

iff

$[(\varphi = \psi) = \psi]$ and
$[(1 = 1) = \psi]$ and
$[(0 = 0) = \psi]$ are true relative to M,g

Observation 2: (1 = 1) is logically equivalent to 1
(0 = 0) is logically equivalent to 1
This means that both get simplified to $(1 = \psi)$, and one drops out,

$[(\varphi = \psi) = \psi]$ and
$[(1 = 1) = \psi]$ and
$[(0 = 0) = \psi]$ are true relative to M,g

iff

$[(\varphi = \psi) = \psi]$ and
$[1 = \psi]$ are true relative to M,g

Observation 3: $(1 = \psi)$ is logically equivalent to ψ:
$(1 = \psi)$ is true (denotes 1) if ψ denotes 1, $(1 = \psi)$ is false (denotes 0) if ψ denotes 0.
So:

$[(\varphi = \psi) = \psi]$ and
$[1 = \psi]$ are true relative to M,g

iff

$[(\varphi = \psi) = \psi]$ and
ψ are true relative to M,g

This is as far as we have gotten. We have shown that:

$\forall f[\ (f(\varphi) = f(\psi)) = \psi\ ]$ is true relative to M,g

iff $[(\varphi = \psi) = \psi]$ is true relative to M,g and ψ is true relative to M,g

Now look at the truth table for $(\varphi = \psi) = \psi$:

φ	ψ	$(\varphi = \psi)$	$(\varphi = \psi) = \psi$
1	1	1	1
1	0	0	1
0	1	0	0
0	0	1	0

We see that $[(\varphi = \psi) = \psi]$ and φ are equivalent and hence we have proved that:

$\forall f[\ (f(\varphi) = f(\psi)) = \psi\ ]$ is true relative to M,g iff
φ is true relative to M,g and ψ is true relative to M,g

So indeed we can define conjunction as:

$(\varphi \wedge \psi) := \forall f[\ (f(\varphi) = f(\psi)) = \psi\ ]$

And with that we have completed our proof: all type logical expressions can be defined by type logical expressions that only use constants, variables, functional application, functional abstraction and identity.

It should be clear that identity plays a crucial role in these considerations: the power of the theory lies in the combined power of abstraction and identity, and identity itself cannot be defined with just application and abstraction.

3.5. Undefinedness

I will end this chapter by introducing one thing that isn't part of classical type theory, but that we had in Foundations and that will be useful here too: the possibility that the denotation of wellformed expressions is undefined.

1. We add to the model a set of undefined objects $\{\perp_a: a \in TYPE\}$ with the conditions:
   1_a: if a, b ∈ TYPE and a ≠ b then $\perp_a \neq \perp_b$
   1_b: if a, b ∈ TYPE then $\perp_a \notin D_b$

2. For every type a ∈ TYPE, $D_a^{\perp} = D_a \cup \{\perp_a\}$

TL is a theory of total function:
$D_{<a,b>} = (D_a \rightarrow D_b)$, the set of all total function from domain D_a into D_b.
$(D_a \rightarrow D_b^{\perp})$ is the set of all partial functions from domain D_a into D_b, functions that are allowed to be undefined for certain arguments.

Jansen 1982 shows that the proper way of extending type theory to include partial functions is to include undefined objects and operate on the domains $D_a^{\perp}$ in general.
Jansen shows that this way of extending TL preserves the principle of λ-conversion (while dealing with undefinedness without having undefined objects does not).

I will not formulate the more general type theory, but allow functions to be undefined. In particular, I will add to the type theory the definiteness operation α from Foundations, first for singular predicates, later redefined for plurality.

For definiteness we add to the theory the following:

Definiteness:
For every type a ∈ TYPE: $\sigma^a \in CON_{<<a,t>,t>}$
$F_M(\sigma^a)$ is $\sigma^a: D_{<a,t>} \rightarrow D_a^{\perp}$

where σ^a is the function such that for every $X \in D_{<a,t>}$

$$\sigma^a(X) = \begin{cases} x & \text{if } char(X) = \{x\} \\ \perp_a & \text{otherwise} \end{cases}$$

We will mostly be concerned with σ^e and drop the superscript.

If CAT ∈ $D_{<e,t>}$ then σ^e(CAT) = RONYA iff $\{d \in D_e: CAT(d) = 1\} = \{RONYA\}$