Natuurlijke Taal Interfaces
Part I: Extensional Semantics

Wouter Beek & Remko Scha

October 29, 2008

Contents

1 Type Logic **5**
- 1.1 Types . . . 5
- 1.2 First-Order Logic Redefined . . . 6
- 1.3 Lambda-abstraction . . . 9
- 1.4 Higher-order variables . . . 11
- 1.5 Implementing β-conversion in Prolog . . . 13

2 Compositional Semantics **17**
- 2.1 Introduction . . . 17
- 2.2 Intransitive verbs . . . 18
- 2.3 Determiners . . . 20
- 2.4 Transitive verbs . . . 24
- 2.5 Exercises . . . 28

Introduction

This course treats some basic methods in logical semantics. We discuss how the meanings of natural language utterances may be represented by well-understood mathematical expressions, and how such meaning representations may be built up in a systematic way.

Some of the methods we discuss were developed for theoretical reasons in Philosophy of Language. At the same time, they play a crucial role in many experimental A.I. applications: they are the basis of natural language interfaces to various kinds of information repositories (e.g. relational databases, collections of logical axioms, ontologies, etc.).

Since we focus on situations where language is used to communicate factual information, we may identify the meaning of an utterance with its truth conditions. The truth conditions of many natural language sentences can be represented by means of formulas of first-order logic. However, if we want to be able to build up these formulas in a systematic way, it is useful to employ a more powerful logical language. For this purpose, we extend first-order logic with the lambda-abstraction operation. The definition of this logical language (called "Type Logic") is the topic of the first chapter of the syllabus.

The second chapter of the syllabus introduces the idea of "Compositional Semantics". It shows how logical representations of sentence meanings can be built up systematically, by means of semantic interpretation rules which are coupled to the syntactic rules which define the set of sentences and their syntactic structures. In describing this method, we assume that the reader has some basic understanding of formal syntax and parsing. (The first-year course "Taaltheorie en Taalverwerking" is a prerequisite for the present course.)

In Part II of the syllabus we shall come to treat natural language sentences that are concerned with "propositional attitudes" (for instance: beliefs of other agents). An adequate analysis of the meaning of such sentences is beyond the scope of extensional logical languages. We shall put forward two different approaches that try to cope with this kind of phenomena: an approach developed in Philosophical Logic, that makes use of so-called "intensions", and an approach that has emerged from practically oriented A.I. work, which employs "belief spaces".

In the homework for this course, you will be asked to practice the art of Compositional Semantics in some Prolog exercises.

Chapter 1

Type Logic

1.1 Types

In this chapter we develop a logical language which incorporates all familiar notions from First-Order Predicate Logic, but which is extended with lambda-abstraction and higher-order variables. As a point of departure, we reformulate the syntax and semantics of First-Order Logic from a type-theoretical perspective. (The next few sections will then introduce some extensions to this logic.)

The first step in this language definition specifies the *types* which the logical expressions may have. These types then allow us to define the expressions of the logical language in a convenient and semantically well-motivated way.

The set of type-expressions is defined recursively. We start out with the two basic types: e for entities and t for truth values. An expression of type e denotes an entity in the domain D. For example an individual constant, e.g. $John$, has type e. And so the syntactic category PN of proper names intuitively corresponds to the type e. An expression of type t denotes a truth value (either 1, which stands for truth, or 0, which stands for falsehood). For example $Walk(John)$ has type t. This type thus intuitively corresponds to the syntactic category S of sentences. From these basic types e and t, infinitely many complex types may be generated. An example of an expression of a complex type is the function $FatherOf$. It has type $\langle e, e \rangle$, which means that it is a function which takes expressions of type e and returns expressions of type e.

Definition 1.1.1 (The set of types T)**.**

- $e, t \in \mathrm{T}$
- If $a, b \in \mathrm{T}$, then $\langle a, b \rangle \in \mathrm{T}$

We use these types to define the well-formedness of expressions which are formed by means of function-application. If expression α is of type $\langle a, b \rangle$

and expression β is of type a, then α applied to β (notation: '$\alpha(\beta)$') is an expression of type b. For instance, the unary predicate $Walk$ is a function from entities to truth values (type $\langle e, t\rangle$). It may thus be applied to an argument of type e, as in $Walk(John)$, which denotes either 1 (in case the denotation of $John$ is in the positive extension of $Walk$) or 0 (in case it isn't).

1.2 First-Order Logic Redefined

We now define the syntax and semantics of the language $\mathcal{L}$, which is a first-order predicate logic.

Definition 1.2.1 (Syntax: elementary expressions)**.**

- There are infinitely many variables of type e: $\{x_1, x_2, \ldots\}$.
- The constants and their types are enumerated in the following table:

Constants	Type	
$John$, $Mary$	e	individual constants
$true$, $false$	t	
$FatherOf$, $MotherOf$	$\langle e, e\rangle$	
$Walk$, $Talk$, $Swim$	$\langle e, t\rangle$	unary predicates
Hit, $Love$, $SpitAt$	$\langle e, \langle e, t\rangle\rangle$	binary predicates
$GiveTo$, $StealFrom$	$\langle e, \langle e, \langle e, t\rangle\rangle\rangle$	ternary predicates
$\neg$	$\langle t, t\rangle$	negation
$\wedge$, $\vee$, $\rightarrow$	$\langle t, \langle t, t\rangle\rangle$	connectives

The *logical* constants (of type t, $\langle t\rangle$ and $\langle t, \langle t, t\rangle\rangle$) are the same for every first-order logic. The other constants (the *descriptive* constants)may vary in different versions of the language used for different applications.

The complex expressions are recursively defined in terms of the elementary expressions:

Definition 1.2.2 (Syntax: complex expressions)**.**

- **Function application:** If F is an expression of type $\langle \alpha, \beta\rangle$, and A is an expression of type α, then $F(A)$ (read this as: 'F applied to A') is an expression of type β.
- **Quantification:** If x is a variable (of type e), and ϕ is an expression of type t, then $\forall x(\phi)$ and $\exists x(\phi)$ are expressions of type t.

Terminology: In expressions of the form $\forall x(\phi)$ or $\exists x(\phi)$, the top level occurrence of x (after $\forall$ or $\exists$) is called a *binding* occurrence. Any occurrence of a variable which is not a binding occurrence is called an *applied* occurrence. In expressions of the form $\forall x(\phi)$ or $\exists x(\phi)$, any applied occurrence of x within ϕ is said to be *bound* by its top level binding occurrence. Any applied occurrence of a variable which is not bound by a binding occurrence is called a *free* occurrence.

An unconventional feature of the logic defined here is that it only uses one-place functions. For example, the binary predicate *Love* is a function of type $\langle e, \langle e, t\rangle\rangle$. It may thus be applied to $Mary$ (an individual constant, of type e). The function-application $Love(Mary)$ is an expression of type $\langle e, t\rangle$.

Note that this expression is itself a function. It may thus be used in yet another function application. The function-application $Love(Mary)(John)$ is an expression of type t.

The formula $Love(Mary)(John)$ does however not look like the notation that we are used to for predicates and their arguments, which would have been $Love(John, Mary)$. But the standard predicate-logical notation may be considered as mere syntactic sugar. We take the language defined above, which only employs one-place functions, as our "official" language, and introduce the usual binary predicate notation as a notational convention:

Definition 1.2.3 (Notational convention 1)**.** If an expression is of the form $(\gamma(\alpha))(\beta)$, it may also be written as: $\gamma(\beta, \alpha)$.

Similar notational conventions may be assumed for functions of arbitrary arity. We introduce an additional convention for the truthfunctional connectives (logical function constants of type $\langle t, \langle t, t\rangle\rangle$, i.e., $\wedge$, $\vee$, and $\rightarrow$), to allow the usual infix notation.

Definition 1.2.4 (Notational convention 2)**.** If C is of type $\langle t, \langle t, t\rangle\rangle$, and ϕ and ψ are of type t, then the expression $C(\psi)(\phi)$ may also be written as: $\phi C \psi$.

Semantics

We now specify the semantic part of the language definition. I.e.: we specify for every expression of the language, how its denotation is established, given a model $\mathcal{M}$ and an assignment g. A model $\mathcal{M}$ is a pair $\langle D, \mathcal{I}\rangle$, where D is a nonempty entities (called the *domain* of the language), and $\mathcal{I}$ is the *interpretation function*, which assigns denotations to all constants of the language.

Domains of types

D is the set of atomic entities that may occur in denotations of expressions of $\mathcal{M}$. Given D, every type a of the language is associated with a set

D_a, the *domain* of the type. The domain D_a of a type a is the set of things that may be denoted by an expression of type a.

The domains of all types are recursively defined as follows:

Definition 1.2.5.

- $D_e = D$
- $D_t = \{0, 1\}$
- $D_{\langle a,b\rangle}$ is the set of functions that map from D_a to D_b.

The interpretation function $\mathcal{I}$.

The interpretation function $\mathcal{I}$ assigns to every constant c in the language an entity in the domain of the type of c. E.g., $\mathcal{I}(John)$ must be an element of D_e, since $John$ has type e.

The assignment function g.

The assignment function g maps all variables (of type e) onto elements of D_e. Given an assignment function g, we can define variants of g which differ from g in only one variable under consideration. We use the notation $g[d_i/x_j]$ for the assignment function that is the same as g, except that it assigns entity d_i to variable x_j.

Given a model and an assignment function, the denotation of every expression of the language is fixed. The denotation of expression ϕ given model $\mathcal{M}$ and assignment function g is written as $\|\phi\|_{\mathcal{M},g}$. This is defined as follows:

Definition 1.2.6 (Semantics)**.**

- If x_i is a variable, then $\|x_i\|_{\mathcal{M},g} = g(x_i)$.
- $\|true\|_{\mathcal{M},g} = 1$ and $\|true\|_{\mathcal{M},g} = 0$.
- $\|\neg\|_{\mathcal{M},g} = h$, $\|\wedge\|_{\mathcal{M},g} = h$, $\|\vee\|_{\mathcal{M},g} = h$, and $\|\rightarrow\|_{\mathcal{M},g} = h$ are defined by the usual truthtables.
- For every other constant c, $\|c\|_{\mathcal{M},g} = \mathcal{I}(c)$.
- If F is an expression of type $\langle\alpha, \beta\rangle$, and A is an expression of type α, then $\|F(A)\|_{\mathcal{M},g}$ is the result of applying $\|F\|_{\mathcal{M},g}$ to $(\|A\|_{\mathcal{M},g})$.
- If x is a variable, and ϕ is an expression of type t, then $\|\forall x(\phi)\|_{\mathcal{M},g} = 1$ iff for all $d \in D_e$ we have $\|\phi\|_{\mathcal{M},g[d/x_i]} = 1$.
- If x is a variable, and ϕ is an expression of type t, then $\|\exists x(\phi)\|_{\mathcal{M},g} = 1$ iff there is a $d \in D_e$ such that $\|\phi\|_{\mathcal{M},g[d/x_i]} = 1$.

We can drop the subscript g for an expression ϕ if the denotation of ϕ is the same for every assignment function: $\|\phi\|_{\mathcal{M}} = a$ iff for all assignment functions g we have $\|\phi\|_{\mathcal{M},g} = a$. Note that this property obtains for all expressions without free variables.

If ϕ is of type t, we may be particularly interested in the case that $\|\phi\|_{\mathcal{M}} = 1$, which may also be written as: $\mathcal{M} \models \phi$. If $\mathcal{M} \models \phi$, we say that "ϕ is entailed by $\mathcal{M}$", "ϕ is satisfied by $\mathcal{M}$", or "ϕ is true in $\mathcal{M}$".

1.3 Lambda-abstraction

Introduction

In this section we are going to enrich our logic by introducing the λ-operator, which enables us to form function-expressions by abstracting over free variables. The λ-abstraction operator was first used in the lambda-calculus, developed by Alonzo Church in the 1930's as a simple and powerful logical calculus, built around the mathematical notion of a function. The expressions used in this calculus are constructed by means of just two operations. The first is the operation of applying a function to its argument: function application. The second is the operation of constructing a function by means of lambda-abstraction. This operation is the mathematical counterpart of the intuitive idea of "abstraction".

In everyday English, the word "abstract" is used in opposition to the word "concrete". Both words indicate properties of concepts. A concrete concept identifies one particular thing – e.g., the black cat that is sitting on the chair in front of me. An abstract concept identifies a property: e.g., being a black cat, being a cat, being black, being on this chair, etc. "Abstraction" is thus the operation of ignoring certain aspects of the description of a concrete thing, so that it becomes a schema for a property.

This common-sense idea of abstraction correlates directly with a method for defining functions, which is often used in mathematics. For instance, in arithmetic we have complex expressions which denote numbers or truth-values: "$10 + 3$" denotes 13, and "$13 < 12$" denotes false. By introducing variables in such expressions, we construct schemata whose denotation depends on the values assigned to the variables. Such schemata are then taken to represent more "abstract" concepts: "$x + 3$" represents the operation of "adding 3"; "$x < 12$" represents the property of being smaller than 12.

From a mathematical point of view, such schemata are viewed as functions. In mathematical texts we commonly find locutions like: "Let the function $Add3$ be defined as: $Add3(x) = x + 3$" or "Define the predicate $Small$ as follows: $Small(x)$ is true if and only if $x < 12$".

The intended meaning of such utterances is made formally precise by means of the operation of λ-abstraction: we introduce the notation $(\lambda x.A)$ to denote the function which assigns to any argument a the value that A

assumes if all occurrences of x in A have the same denotation as a. This notation introduces the possibility of explicit function-definitions. For instance: $Add3 = (\lambda x.x{+}3)$ or $Small = (\lambda x.x < 12)$. But, more interestingly, we may now write, not only $Add3(2) = 5$ or $Small(2)$, but also, using "anonymous function-expressions": $(\lambda x.x + 3)(2) = 5$, or $(\lambda x : x < 12)(2)$.

Extending the logic

The syntax of λ-abstraction is defined by the following rule, which is to be added to the definition of our logical language $\mathcal{L}$ in the previous section.

Definition 1.3.1. If ϕ is an expression of type b and x is a variable of type a, then $\lambda x.(\phi)$ is an expression of type $\langle a, b\rangle$. Its denotation $\|\lambda X_i(\phi)\|_{\mathcal{M},g}$ is the function that maps every element $\alpha \in D_a$ to $\|\phi\|_{\mathcal{M},g[\alpha/x]}$.

The consequences of this definition are best explained by some examples. Take the expression $Walk(x)$ which is of type t and has a free variable x (of type e). It is true whenever the entity denoted by $g(x)$ is in the positive extension of $Walk$, and false whenever it doesn't. Now we can form the new expression $\lambda x.Walk(x)$, in which the λ-operator abstracts over the free variable x. This means that the x that was free in the expression $Walk(x)$ is in the new expression bound by the λ-operator. Since x is of type e, the newly created λ-expression is of type $\langle e, t\rangle$: it maps entities onto truth-values. The example expression $\lambda x.Walk(x)$ now denotes the function that, when given an entity, yields 1 just in case that entity is in the positive extension of $Walk$, and 0 otherwise. ($\lambda x.Walk(x)$ is thus identical to $Walk$.)

A more interesting example is the function $\lambda x.Like(x, Mary)$ which intuitively stands for the property "to like Mary". It may be applied, for instance, to the individual constant $John$. The function is of type $\langle e, t\rangle$, and the argument is of type e; the function-application $(\lambda x.Like(x, Mary))(John)$ is thus of type t. It yields 1 if, assuming that $g(x)$ denotes the same entity as $John$, $Like(x, Mary)$ yields 1. The formula is thus equivalent to the formula $Like(John, Mary)$. But if we look at the English sentence "John likes Mary", we see that the expression $(\lambda x.Like(x, Mary))(John)$ may be useful: it contains the sub-expressions $John$ and $(\lambda x.Like(x, Mary))$, which correspond to the constituents of the sentence "John likes Mary" if we analyze it as consisting of a noun-phrase ("John") followed by a verb-phrase ("likes Mary").The lambda-notation may thus allow us to build up the meaning of a sentence out of the meanings of its parts.

Another example is the sentence "John walks and talks". We would like to represent the meaning of the verb-phrase "walks and talks" by a logical expression. In first-order logic this is not possible, but with lambda-abstraction it is simple: $\lambda x.Walk(x) \wedge Talk(x)$. If we apply this expression to $John$, we get the expression $(\lambda x.Walk(x) \wedge Talk(x))(John)$, which is equivalent to $Walk(John) \wedge Talk(John)$

1.4 Higher-order variables

Now that we have extended our logical language so as to incorporate λ-expressions (using only variables of type e), we make a second extension to our logical language by adding variables of arbitrary type. This makes our logic a higher-order logic. (A variable that is of a complex type is called a higher-order variable.)

A higher-order variable X of type $\langle e, t\rangle$ can be used, for instance, to construct the expression: $\lambda X.X(John)$. Since $John$ has type e,$X(John)$ is of type t. This λ-expression thus denotes a function from unary predicates to truth-values. It yields 1 for) predicates that hold for John, and yields 0 for predicates that do not hold for John.

An example of a formula in which a quantifier with such a variable is used: $\exists X(X(Peter) \wedge X(John) \wedge \neg X(Bill))$, which expresses that there is a property that Peter and John have in common, but that Bill lacks.

The higher-order variable X occurring in these examples does not belong to the logical language $\mathcal{L}$ as defined above. The language contains an infinite number of variables of type e, but no variable X of type $\langle e, t\rangle$. The addition of such variables is very important though, since it allows us to apply λ-functions not only to entities, but to expressions of any type, thus making it a very strong tool.

We thus add variables for every type to our language, and generalize the definitions of constructions that use variables.

Definition 1.4.1 (Extending first-order predicate logic: syntax)**.**

- For every type a, there is a set VAR_a which contains an infinite number of variables of type a.
- If for some type a, $X_i \in VAR_a$, and ϕ is an expression of type t, then $\forall X_i(\phi)$ is an expression of type t
- If for some types a, and b, $X_i \in VAR_a$ and β is an expression of type b, then $\lambda X_i(\beta)$ is an expression of type $\langle a, b\rangle$.

The semantics is extended accordingly. The assignment function now assigns values to variables of every type. The semantics for expressions constructed using function application is general enough to account for expressions with these higher-order variables too. The semantics for quantified expressions changes a bit.

The same general definition of the domain corresponding to a type can be used to define the semantics for λ-expressions. We will therefore add this to the semantics of our logical language too:

Definition 1.4.2 (Extending first-order predicate logic: semantics)**.**

- For every type a, if $X_i \in VAR_a$, then $\|X_i\|_{\mathcal{M},g} = g(X_i)$.

- For every type a, if $X_i \in VAR_a$, and every expression ϕ of type b, we have that $\|\lambda X_i(\phi)\|_{\mathcal{M},g}$ is the function that maps every element $\alpha \in D_a$ to $\|\phi\|_{\mathcal{M},g[\alpha/X_i]}$.
- If variable X is of type a, then: $\|\forall X(\phi)\|_{\mathcal{M},g} = 1$ iff for all $d \in D_a$ we have $\|\phi\|_{\mathcal{M},g[d/x_i]} = 1$.
- If variable X is of type a, then: $\|\exists x(\phi)\|_{\mathcal{M},g} = 1$ iff there is a $d \in D_a$ such that $\|\phi\|_{\mathcal{M},g[d/x_i]} = 1$.

β-reduction

In the above we saw the example $\lambda x(Walk(x))(John)$, which turned out to be equivalent to the formula $Walk(John)$. This suggests a simplification operation which can be applied to any expression in which a λ-function is applied to an argument. In this simplification operation the λ-operator is removed and all occurrences of the λ-variable x in $Walk(x)$ are replaced by the argument expression. This operation is called **β-reduction**:

Definition 1.4.3 (β-reduction)**.** $(\lambda x.\phi)(\psi) = \phi[x/\psi]$, where $\phi[x/\psi]$ is the formula ϕ' that is the result of replacing every free occurrence of x in ϕ by ψ.

β-reduction is completely general; it allows not only individual constants (and other expressions of type e) to be substituted, but also sentences and predicates (of whatever arity).

We give a slightly more intricate example of function-application with λ-operators in order to show the process of β-reduction.

$$\begin{aligned}&\lambda U.U(mia)(\lambda(x, Snort(x))) \\ &=_\beta \lambda x.Snort(x)(mia) \\ &=_\beta Snort(mia)\end{aligned} \tag{1.1}$$

There is one last thing to keep in mind when performing β-reductions. Observe the following contrived example:

$$\begin{aligned}&\lambda U.\forall y(Man(y) \rightarrow U(y))(\lambda x.\exists y(Woman(y) \wedge Love(x,y)) \\ &=_\beta \forall y(Man(y) \rightarrow (\lambda x.\exists y(Woman(y) \wedge Love(x,y))(y) \\ &=_\beta \forall y(Man(y) \rightarrow (\exists y(Woman(y) \wedge Love(y,y))\end{aligned} \tag{1.2}$$

The problem here is that the y occurring in the functor and the y occurring in the argument are not the same. The way we defined β-conversion,

however, disregards this difference. The problem is overcome by requiring such double occurrences of variable names to be removed. This is done by a process of variable-renaming, called α-conversion.

Definition 1.4.4 (α-conversion). $\lambda x.\phi$ is α-equivalent to $\lambda y.\phi[y/x]$, as long as y does not occur freely in ϕ and there is no λy within ϕ.

In the next chapter of this syllabus, which discusses how meaning representations for sentences can be built up step by step out of the meanings of their constituents, we will see that λ-abstraction over variables of arbitrary types is a crucial tool part of our logic. But we will also see that this easily leads to large and unreadable expressions. Applying β-conversion is then a necessary step to reach formulas that can be understood by a human person. That is why the implementation of β-conversion is the first computational exercise that must be considered now.

1.5 Implementing β-conversion in Prolog

We choose `lam(X, E)` as the Prolog equivalent of a λ-expression, where `X` is the variable and `E` is the expression. We will represent a function-application by `app(L, A)`, where `L` is a λ-expression and `A` is an argument.

We wish to implement the β-reduction algorithm in such a way that all appications of λ-functions to arguments are treated, including nested ones. For instance, when provided with the input `app(app(lam(Y, lam(X, love(X, Y), mary), john)`, the algorithm should yield `love(john, mary)`. We achieve this by making use of a stack in order to keep track of the argument-expressions. The stack adheres to the LIFO-principle (last-in-first-out), and this is precisely what we need when working with function-applications, as the following example makes clear:

Table 1.1: Example of the stack in `betaConvert`

Step	Expression	Stack
1	`app(app(lam(Y, lam(X, love(X, Y), mary), john)`	`[]`
2	`app(lam(Y, lam(X, love(X, Y), mary)`	`[john]`
3	`lam(Y, lam(X, love(X, Y)))`	`[mary, john]`
4	`lam(X, love(X, mary))`	`[john]`
5	`love(john, mary)`	`[]`

We incorporate the stack in our Prolog-predicate by introducing the following wrapper clause:

```
betaConvert(Expression, Result):-
        betaConvert(Expression, Result, []).
```

If the expression we want to β-reduce is a variable, we just return that very variable (since it is impossible to further reduce a variable):

```
betaConvert(Expression, Result, []):-
        var(Expression),
        Result = Expression.
```

We need not include the case of constants here, since these will be taken care of in the last clause of `betaConvert`, where the constant `c` will be treated as a complex expression with functor `c` and argumentlist `[]`. Since the argumentlist is empty, `betaConvertList` will not make any changes to the 'arguments', and `betaConvert` will reassemble the functor `c` and the argumentlist `[]` into the constant `c`.

If the expression is a function application, we simplify the expression by focussing on the functor and adding the argument to the stack. This mirrors what happens in the move from steps 1 to 2 and from 2 to 3 in table 1.1.

```
betaConvert(Expression, Result, Stack):-
        nonvar(Expression),
        Expression = app(Functor, Argument),
        nonvar(Functor),
        betaConvert(Functor, Result, [Argument|Stack]).
```

Now we implement the Prolog-clause that treats a `lam`-operator. This process is accompanied by popping the last pushed item from the stack, and substituting it for all λ-bound occurrences in the expression at hand. It is a bit hard to see that the code below indeed manages this, but observe that the head of the stack is Prolog-matched with the λ-variable `X`.

```
betaConvert(Expression, Result, [X|Stack]):-
        nonvar(Expression),
        Expression = lam(X, Formula),
        betaConvert(Formula, Result, Stack).
```

This was the core of the algorithm (the rest will just be bookkeeping, although necessary bookkeeping). Not all expressions are λ-expressions, and yet we cannot simply skip non-λ-expressions, since an expression might be complex and contain a λ-expression nested somewhere in its structure. Examples of expressions that we will encounter here are `forall(X, app(walk, X))`, `father(john)`, and `c`.

```
betaConvert(Expression, Result, []):-
        nonvar(Expression),
        \+ (Expression = app(X, _), nonvar(X)),
        Expression =.. [Functor | SubExpressions],
        betaConvertList(SubExpressions, ResultSubExpressions),
        Result =.. [Functor | ResultSubExpressions].
```

This is also where the constants will be β-reduced. For every instantiation of `Expression` with a constant `c` we have `Expression =.. [Functor | SubExpressions]` instantiated as `c =.. [c | []]`, and therefore `betaConvertList([], ResultSubExpressions)`, which returns the empty list. The subsequent `Result` is then just `c` again.

`betaConvertList` is defined in the obvious way:

```
betaConvertList([H | T],[SolH | SolT]):-
        betaConvert(H, SolH),
        betaConvertList(T, SolT).
betaConvert(X, Y):-
        betaConvert(X, Y, []).
```

Normally, we would now have to implement the algorithm for α-conversion, and relate it to `betaConvert`. But Prolog solves the problem of unintended double occurrence of the same variable name all by itself.

Exercise 1.5.1. Implement the β-conversion algorithm. Create a number of test expressions that show that the algorithm is valid. Make sure that all of the intermediary β-reduction steps are shown in the program's output.

Exercise 1.5.2. Why can't we use default Prolog-unification in order to implement β-reduction? For example: `app(lam(Argument, Result), Argument, Result)`. **Hint:** Think about cases in which the same substitution will have to be performed multiple times, as in cases of coordination (e.g "John and Mary dance").

Chapter 2

Compositional Semantics

2.1 Introduction

Now that we have a logic with λ-expressions and higher-order variables, we can start to work on a system for building meaning representations for natural-language sentences. The techniques that we will use are inspired by the approach of the philosopher Richard Montague, which is known as "Montague grammar". In the present chapter we only treat a subtheory which is limited to extensional terms only. The other, intensional aspects will be introduced in the second part of this course.

Observe that the syntactic structure of a sentence must be taken into account whenever we want to treat its semantic contents. We should not directly assign meaning to a natural language string, since then syntactically ambiguous sentences would be assigned the same meaning. So, for example, the following two variants must be distinguished when assigning a meaning to the sentence "John walks to the friend of his brother and the moon."

John walks to [[the friend of his brother] and [the moon]] (2.1)

John walks to [the friend of [[his brother] and [the moon]]] (2.2)

Since a syntactic analysis is needed anyway and the compositionality principle requires us to derive the meaning of complex expressions on the basis of the meanings of its constituent components, it seems intuitive to assign a meaning to every lexical expression (i.e. the leaves of the syntactic analysis tree) and to pair each syntactic construction rule with a semantic rule. The Montague approach follows this intuition in positing a complete homomorphism between syntax and semantics, according to which each syntactic entity (of whatever complexity) has to be accompanied by a semantic one.

2.2 Intransitive verbs

Our strategy for constructing a formal semantics for a natural language will be to first focus on a (very) small subset of the language and to search for a semantic theory that sufficiently satisfies our intuitions with respect to the meanings of that particular subset. Subsequently, we will incrementally broaden the set of sentences the theory gives meanings for. At each step we make sure that the meanings assigned to new sentences are in line with our intuitions, while the meanings of the old sentences are left intact.

First of all, we may notice that many syntactic categories correspond to semantic types in an intuitively obvious way. Some of these intuitions are summarized in the following table.

Words	Category	Type
John, Mary	proper name (PN)	e
walk, talk, swim	intransitive verb (IV)	$\langle e,t\rangle$
red, big, slow	adjective (ADJ)	$\langle e,t\rangle$
man, boy, cat	noun (N)	$\langle e,t\rangle$
hit, love, see	transitive verb (TV)	$\langle e,\langle e,t\rangle\rangle$
give, steal	ditransitive verb (DV)	$\langle e,\langle e,\langle e,t\rangle\rangle\rangle$
not	adverb (ADV)	$\langle t,t\rangle$
en, of	"conjunctions"	$\langle t,\langle t,t\rangle\rangle$

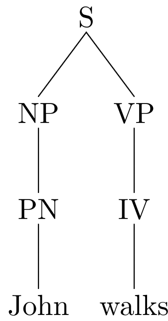
We will begin with a very small subset of English, characterized by the following grammar.

```
S --> NP, VP
NP --> PN
VP --> IV
PN --> John
PN --> Mary
IV --> walks
IV --> talks
```

We shall take the following sample sentence in order to illustrate our approach to the construction of meaning representations.

$$\text{John walks} \tag{2.3}$$

The syntactic analysis of sentence (2.3), given the grammar rules, is



In specifying the semantic rules we make use of the notation $[[A]]$ to indicate (if A is a syntactic category) the meaning of the constituent with that category, or (if A is a word) the meaning of the word A.

First we specify semantic rules for the lexical elements.

Semantics 2.2.1. The meaning of the proper noun 'John' is the individual constant $John$.
The meaning of the proper noun 'Mary' is the individual constant $Mary$.

Semantics 2.2.2. The meaning of the intransitive verb 'walks' is the predicate $Walk$.
The meaning of the intransitive verb 'talks' is the predicate $Talk$.

We can illustrate this second rule by an example. If the walking entities are d_2, d_3 and d_8, then the meaning representation of 'walks', i.e. the predicate $Walk$, must denote the functor which assigns 1 to just these three entities (i.e. $\mathcal{I}(Walk) = \{\langle d_2, 1\rangle, \langle d_3, 1\rangle, \langle d_8, 1\rangle, \langle d_1, 0\rangle, \langle d_4, 0\rangle, \ldots\}$).

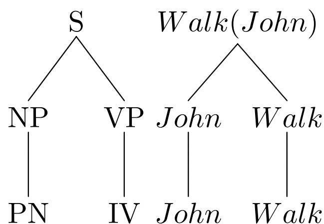
For every syntactic rule we define a corresponding semantic rule:

Semantics 2.2.3. If a NP consists of a PN, then $[[NP]] = [[PN]]$.

Semantics 2.2.4. If a VP consists of an IV, then $[[VP]] = [[IV]]$.

Semantics 2.2.5. If a S consists of $[NP, VP]$, then $[[S]] = [[VP]]([[NP]])$.

The semantic rules 2.2.3 and 2.2.4 simply percolate the meanings that were already established on the level of the lexical elements (rules 2.2.1 and 2.2.2). Semantic rule 2.2.5 introduces the first use of function application. Here the meaning of the verb phrase is applied to the meaning of the noun phrase. We can now draw the semantic tree that accompanies the syntactic tree of sample sentence (2.3).



The meaning of (2.3) is thus $Walk(John)$.

To implement these semantical rules in Prolog we start off with the CFG for the syntax.

```
S --> NP, VP
NP --> PN
VP --> IV
PN --> [john]
PN --> [mary]
IV --> [walks]
IV --> [talks]
```

Now we add the semantics to our syntactical rules. The straightforward way to do this is to introduce an argument for the meaning of the complex, left-hand expression, based upon the meanings of the right-hand, simpler expressions.

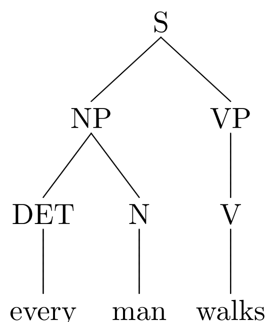
```
s(app(VP, NP)) --> np(NP), vp(VP).
np(PN) --> pn(PN).
vp(IV) --> iv(IV).
pn(john) --> [john].
pn(mary) --> [mary].
iv(walks) --> [walks].
iv(talks) --> [talks].
```

2.3 Determiners

The semantics introduced in the previous section works for sentences with the same structure as sample sentence (2.3). This is a very limited subset of the English language of course, and we are therefore going to add semantic rules for more syntactic constructs. We shall now first consider noun phrases that consist of a determiner and a noun. This extends the set of sentences we are concerned with so as to allow, for instance, the following sentence:

Every man walks. (2.4)

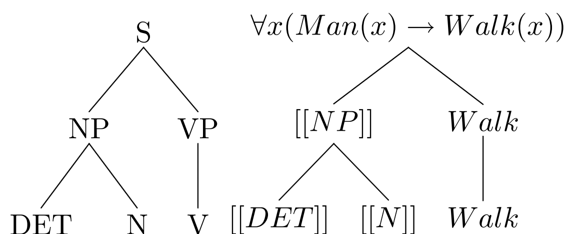
The syntax of this sentence is straightforward and the additions to the grammar are implicit from the parse tree.



The problem of extending our semantics in order to account for the meaning of this sentence starts with the question what meanings should be assigned to the words 'every' and 'man'. To answer this question, we enter a process of backwards engineering that most of Montague grammar construction consists of. As competent language users we already know the meaning of sentence (2.4):

$$\forall x(Man(x) \rightarrow Walk(x)) \tag{2.5}$$

Since we know the meaning of the total expression, we are able to assign meanings to the constituents in a top-to-bottom fashion, so that these sub-meanings will yield the desired meaning for the top expression. Let us do this for the example at hand. When we start off with the rules for meaning-assignment we defined in the previous section, we get the following partial semantical tree.



But here our semantics breaks down, since regardless of what $[[NP]]$ is, when we use the semantical rule for S we will get $Walk([[NP]])$. It is clear that no substitution for the meaning for NP could result in (2.5). We therefore need to introduce an altogether different rule for the semantics of NP's, and change the rule for the semantics of S accordingly.

How to proceed? Let us think for a moment. Whenever we apply the meaning of VP to the meaning of NP, the meaning of NP gets stuck inside the meaning of VP. This is the situation which we would like to overcome. This might be done by swapping the order of function application in the semantical rule for S.

Semantics 2.3.1. If a S consists of $[NP, VP]$, then $[[S]] = [[NP]]([[VP]])$.

Now we need an appropriate meaning for a noun phrase consisting of a determiner and a noun. And we must choose our meanings in such a way that the total semantics works for the old NP's too (i.e. those consisting of a proper noun).

We need the meaning of the NP to be such that we only need to fill in the constant $Walk$ in order to arrive at (2.5). This is aptly expressed by the lambda-statement $\lambda P.\forall x(Man(x) \to P(x))$. For this is just the expression we had in mind, only stripped from its intransitive verb. It is easy to see that for any intransitive verb whatsoever, the S-rule now coughs up the right interpretation. For our sample sentence this would be $\lambda P.\forall x(Man(x) \to P(x))(Walk)$. And then by β-reduction we indeed get (2.5).

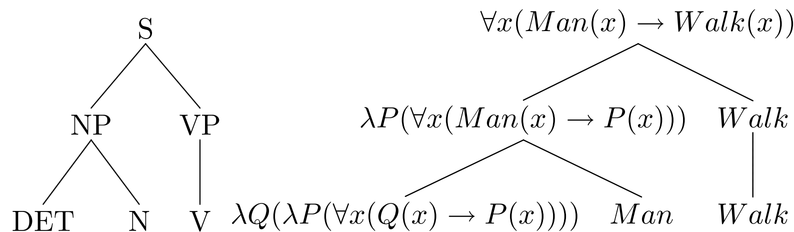
As a final step we need to construct the semantic rules for the determiner and the noun in such a way that a rule for the meaning of the NP, resulting in the above introduced λ-statement, can be defined in terms of these. When we look at $\lambda P.\forall x(Man(x) \to P(x))$, we see that just as we stripped off the constant $Walk$ from the sentence meaning, we might as well strip off the constant Man from the meaning of the NP. Once we have gained these insights the right rules for the semantics of DET, N and NP seamlessly follow.

Semantics 2.3.2. If DET = 'every', then $[[DET]] = \lambda Q.\lambda P.\forall x(Q(x) \to P(x))$.

Semantics 2.3.3. The meaning of the common noun 'man' is the predicate Man.
The meaning of the common noun 'woman' is the predicate $Woman$.

Semantics 2.3.4. If NP consists of $[DET, N]$, then $[[NP]] = [[DET]]([[N]])$.

2.3.2 and 2.3.3 are rules for lexical semantics, 2.3.4 is a compositional semantical rule involving function application. Each time a compositional semantic rule is used, we have to β-reduce the outcome, thus yielding a simpler expression. We can then draw the full semantic tree for sentence (2.4):



But we aren't finished yet. The new sentence (2.4) is now analyzed correctly, but the old sentence (2.3) from the previous section no longer is. It is now analyzed as $John(Walk)$, which is an illegitimate and meaningless expression. (In this expression a constant of type e is applied to a constant of type $\langle e,t\rangle$, but this is not allowed according to the definition of function application.) We might solve this problem by differentiating between the two kinds of NP's in the following way.

(Bad idea)
If a sentence S' consists of $[NP'_1, VP]$, then $[[S']] = [[VP]]([[NP'_1]])$.
If S' consists of $[NP'_2, VP]$, then $[[S']] = [[NP'_2]]([[VP]])$.
$[[NP'_1]] = [[PN]]$.
$[[NP'_2]] = [[DET]]([[N]])$.

The problem with this approach is that, since the semantic rules are coupled one-to-one with syntactic rules, we need to rewrite part of our grammar. So our familiar DCG fragment

$$\begin{aligned} S &\rightarrow NP, VP \\ NP &\rightarrow PN|(DET, N) \end{aligned} \tag{2.6}$$

would be replaced by

$$\begin{aligned} S' &\rightarrow (NP'_1|NP'_2), VP \\ NP'_1 &\rightarrow PN \\ NP'_2 &\rightarrow DET, N \end{aligned} \tag{2.7}$$

But we do not want this, since we generally do not want to rewrite our syntax just in order to patch up our semantics. This is not to say that semantical considerations should never lead us into reconsidering some aspects of our syntax, but at least these semantically-induced changes should be reduced to a bare minimum.

A more elegant solution is to change the semantics of noun phrases that consist of a proper name. To bring such NP's in line with the quantifier-NP's, it is necessary that they have the form $\lambda P.(\ldots P \ldots)$, so that the semantic content of the VP can be inserted at the P-spot. The possibilities for $(\ldots P \ldots)$ are narrowed down by the necessary occurrence of $[[PN]]$ and by the added requirement that whatever is substituted for P must be applied to the meaning of the proper noun. We therefore get the following altered rule for the meanings of noun phrases:

Semantics 2.3.5 (Replacing the **Bad idea** above)**.** If NP consists of PN, then $[[NP]] = \lambda P.P([[PN]])$. If NP consists of $[DET, N]$, then $[[NP]] = [[DET]]([[N]])$.

It is clear that we need not alter any syntactic rule in our DCG. The process of changing the semantics of our language so as to accommodate different sorts op NP's has been somewhat lengthy, but this was done in order to show how the process of elaborating the semantics of a natural language proceeds. The process of looking for low-level meanings that might support the intuitively fixed top-meanings of sentences, and investigating possible conflicts with other parts of the semantics, is the general pattern which is followed whenever we try to broaden the coverage of a semantic system.

The changes and additions to our Prolog-code are straightforward:

```
s(app(NP, VP) --> np(NP), vp(VP).
np(lam(P, P(PN))) --> pn(PN).
np(app(DET, N)) --> det(DET), n(N).
vp(IV) --> iv(IV).
det(lam(P, lam(Q, forall(X, and(app(P, X), app(Q, X))))))
 \hspace*{40 mm}--> [every].
iv(walk) --> [walks].
pn(john) --> [john].
```

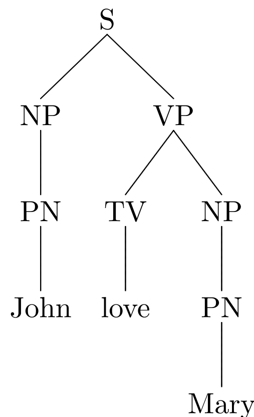
Since the formulas that this semantics gives rise to can be β-reduced, you will have to invoke the `betaConvert` clause.

2.4 Transitive verbs

We shall yet again widen the subset of English for which our theory churns out meaning-representations by including transitive verbs. We shall use the following sample sentence:

$$\text{John loves Mary} \tag{2.8}$$

This has the following syntactic analysis:



This sample sentence, and any sentence like it, is easily incorporated by adding the following rules:

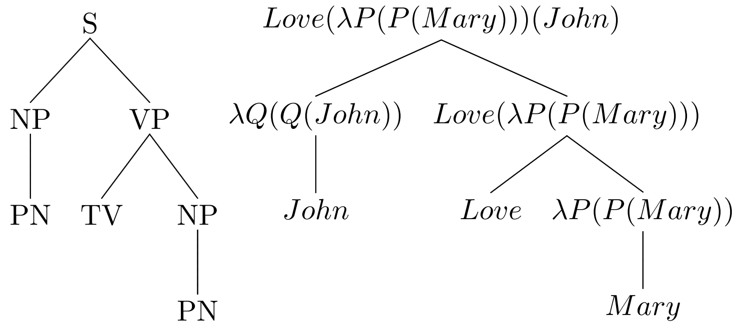
Semantics 2.4.1. The meaning of the transitive verb 'loves' is *Love*, with type $\langle e, \langle e, t\rangle\rangle$.
The meaning of the transitive verb 'hates' is $Hate$, with type $\langle e, \langle e, t\rangle\rangle$.

Semantics 2.4.2 (Replacing 2.2.4)**.** If a VP consists of an IV, then $[[VP]] = [[IV]]$. If a VP consists of $[TV, NP]$, then $[[VP]] = [[TV]]([[NP]])$.

This corresponds to alterations in the Prolog code:

```
vp(app(TV, NP)) --> tv(TV), np(NP).
```

For the sample sentence (2.8) this results in the following semantic tree:



The *Love*-relation now holds between the individual $John$ and the second-order property of being a property of $Mary$. How are we to interpret this second argument? $\lambda P(P(Mary))$ is a function from unary predicates to truth values. It is true for properties that hold of Mary, and it is false for those properties that do not hold of Mary. The reason for this more complex semantics is that in the general case we do not want a transitive verb to hold between entities. An example of such a sentence that gets a false meaning under such a semantic analysis would be "John seeks Santa Claus". Since there is no entity that corresponds to the proper noun 'Santa Claus', the meaning of this sentence cannot be $Seek(John, SantaClaus)$. The reason why some sentence meanings hold between entities while others hold between entities and λ-expressions will be made clear in section **??**.

Right now we need a way to transform meanings like $Love(John, \lambda P(P(Mary)))$ into $Love^*(John, Mary)$, where $Love^*$ is the predicate that represents the intuitive meaning of the verb 'love', and thus has the type $\langle e, \langle e, t\rangle\rangle$. We do this by introducing a meaning postulate which states that precisely in all models in which the formula $Love(John, \lambda P(P(Mary)))$ holds, the formula $Love^*(John, Mary)$ holds too. The meaning postulate that achieves this is the following:

Definition 2.4.1 (Meaning postulates)**.** For every extensional verb δ we have $\forall x, \forall X(\delta(x, X) \leftrightarrow X(\lambda y.\delta^*(x, y)))$, where variable x is of type e and variable X is of type $\langle\langle e, t\rangle, t\rangle$.

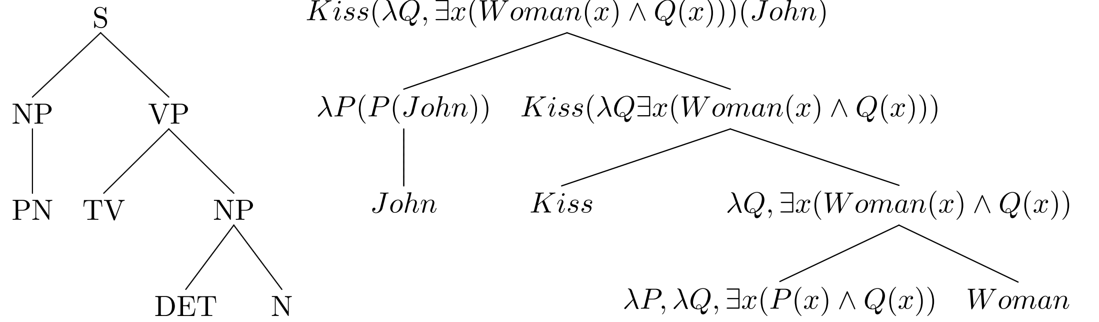
Filling in *Love* for δ, *John* for x, and $\lambda Q(Q(Mary))$ for X, we get $(\lambda Q(Q(Mary)))(\lambda y(Love^*(John, y)))$. After β-reduction we get:

$$\begin{aligned} &(\lambda Q(Q(Mary)))(\lambda y(Love^*(John, y))) \\ &=_\beta \lambda y(Love^*(John, y))(Mary) \\ &=_\beta Love^*(John, Mary) \end{aligned} \tag{2.9}$$

Let us derive the meaning of another, slightly more complex sentence, just to practice these concepts a bit:

John kisses a woman (2.10)

When we apply the syntactic and semantic rules and get the following trees:



The meaning of the sentence can be rewritten into

$$Kiss(John, \lambda Q(\exists x(Woman(x) \wedge Q(x)))) \tag{2.11}$$

The *Love*-relation once again holds between an individual *John* and a second-order property of being a property of a woman. This is once more not what we want for the transitive verb 'kiss'. We instantiate the scheme of meaning postulate 2.4.1 once more: δ is *Love*, x is *John*, and X is $\lambda Q(\exists x(Woman(x) \wedge Q(x)))$. This λ-expression is indeed of type $\langle\langle e, t\rangle, t\rangle$, as the meaning postulate requires. The outcome of using the meaning postulate is $\lambda Q.\exists x(Woman(x) \wedge Q(x))(\lambda y.\delta^*(x, y))$. And although this might look a bit daunting at first sight, it can be considerably simplified by several applications of β-reduction, where we first rename the variable x in

$\lambda y.\delta^*(x,y)$ to z in order to avoid confusion with the completely different variable x in $\lambda Q.\exists x(Woman(x) \wedge Q(x))$ (this is just α-conversion of course).

$$\begin{aligned} &\lambda Q(\exists x(Woman(x) \wedge Q(x)))(\lambda y(Love^*(z,y))) \\ &=_\beta \exists x(Woman(x) \wedge (\lambda y(Love^*(z,y)))(x)) \\ &=_\beta \exists x(Woman(x) \wedge Love^*(x,y)) \end{aligned} \tag{2.12}$$

We now have to add the Prolog code for the meaning postulate:

```
mp(Old, New) :-
        Old =.. [Delta, X, XX],
        ext(Delta),
        NN =.. [Delta, X, Y],
        betaConvert(app(XX, lam(Y, NN)), New).
```

In the code δ is `Delta`, x is `X`, X is `XX`. `XX` is the λ-expression, which gets β-reduced in the last rule. The use of the `ext`-predicate, which checks whether a verb is extensional or not, will become clear in the next section. For now it suffices to state that the verbs 'love' and 'kiss' are indeed extensional, i.e. `ext(love)` and `ext(kiss)` are both part of the knowledge base.

The attentive reader has recognized that whereas in the meaning postulate there is a distinction between the two predicates that are filled in for δ and δ^* respectively, this distinction is not made in the Prolog code (`Delta` is used in both cases). We make use here of the notion of *type overloading*. I.e. we suppose that the two 'versions' of a predicate can be distinguished simply by looking at their types; and in the context of a Prolog semantics a difference in types amounts to a difference in how a predicate occurs within a formula.

This concludes our introduction to extensional Montague grammar. The system that was described here can be extended in many ways. Some attempts at this will be made in the exercises. Another, more fundamental, extension will be made in the second part of this course.

2.5 Exercises

Exercise 2.5.1. For the following sentences (a) construct the syntactical parse tree, (b) add to each node in the syntactical tree the corresponding meaning, and (c) reduce the meaning of the sentence using β-conversion and the meaning postulates.

- Every man loves a woman.
- The man talks to Robin.[1]

Exercise 2.5.2. Implement a semantic analysis algorithm in Prolog that processes, among others, the sentences from the previous exercise. Demonstrate some sample sentences. Make sure that the output shows the functioning of your program in a perspicuous way.

Exercise 2.5.3. Add adjectives to your computational semantics, so that iterations of arbitrary length are allowed. Demonstrate some sample sentences. Make sure that the output shows the functioning of your program in a perspicuous way.

Exercise 2.5.4. Add computational semantics for coordination in both NP's and VP's, so that sentences like "John and Mary walk" and "John walks and talks" are assigned the correct meaning. Demonstrate some sample sentences. Make sure that the output shows the functioning of your program in a perspicuous way.

Exercise 2.5.5. Add the category of ditransitive verbs (e.g. 'give') to your computational semantics, by translating them as tertiary predicates. Is it necessary to introduce yet another notational convention for the third argument to this predicate, or is the convention that we defined for transitive predicates sufficient?

[1] The meaning of 'the' could be rendered $\lambda P, \lambda Q.\exists x(\forall y(P(x) \leftrightarrow x = y) \wedge Q(x))$. Here a singular entity is assumed to be in the extension of the P-predicate, whereas there may be multiple entities within the extension of the Q-predicate. This neatly captures Russell's classical analysis of definite descriptions. (Compare it with the meaning of 'one', which requires the extension of the intersection of both predicates to be a singleton set, i.e. $\lambda P, Q.\exists x, \forall y((P(y) \wedge Q(y)) \leftrightarrow x = y)$.) Note that from a discourse perspective, Russell's analysis is obviously inadequate, but those issues are the topic of a different course.