Chapter 5 Predicate logic

Section 5.1 Quantifiers

Suppose U is a set (I use the letter "U" since the word "universe" is going to be used to signify this set), and R a binary relation on U. The expression

$$Rxy \tag{1}$$

is an indefinite one; when the variables x and y are given values that are from the set U, then Rxy assumes a definite *truth-value*, which is either $\top$ (*true*) or $\bot$ (*false*).

For instance, when $U=\mathbb{N}$, and R denotes the ordinary less-than relation ($<$), then

$$R(1, 3) \sim \top,$$
$$R(4, 3) \sim \bot,$$
$$R(2, 10) \sim \top .$$

We are using the symbol $\sim$ to indicate equality of truth-values. We do not want to use ordinary equality, since $\sim$ does not mean that the two expressions in question are the *same*; it only means that under the given interpretation, they *evaluate* to the same truth-value. For instance, we may say that

$$R(1, 3) \sim R(2, 10)$$

under the interpretation of R as the ordinary less-than relation; but

$$R(1, 3) = R(2, 10)$$

would suggest that the expressions $R(1, 3)$, $R(2, 10)$ "are the same" -- which we do not want to say.

Note that we did not need to write $R(x, y)$ in (1), in place of the simpler Rxy; however, we did not want to write $R12$ in place of $R(1, 2)$, or, worse, $R210$ in place of $R(2, 10)$, for obvious reasons. It would *not* be a mistake to write $R(x, y)$ for Rxy.

Now, consider the expression

$$Rxy \wedge Ryz \ . \qquad (2)$$

Under the same interpretation as above -- that is, with the universe being $\mathbb{N}$, and R being the ordinary less-than relation --, this expression also takes up a definite truth-value when the variables are given values from $U=\mathbb{N}$. E.g., when $x=1$, $y=2$ and $z=3$, the expression becomes

$$Rxy \wedge Ryz \sim R(1, 2) \wedge R(2, 3) \sim \mathsf{T} \wedge \mathsf{T} \sim \mathsf{T} \ .$$

Here we used the interpretation of $\wedge$ as an operation on truth-values as was specified in Chapter 4, Section 3, p. 122. Also remember that you can read $\wedge$ as "and"; thus $Rxy \wedge Rxz$ can be read as "Rxy and Rxz". It was explained in Chapter 4 how we can use the Boolean connectives to build compound sentences (Boolean expressions) out of simpler ones, starting with letters such as A, B, ... Here we use the connectives in the same way to build compound logical expressions (indefinite sentences) out of simpler ones, starting with *atomic* expressions such as Rxy.

Now, to the new element: *quantifiers*. Out of the expression (2), we can form, by prefixing $\exists x$ to it, the new expression

$$\exists y(Rxy \wedge Ryz) \ . \qquad (3)$$

Read "$\exists y$" as "there exists y such that". Suppose we are still using the same interpretation as before: $\mathbb{N}$ for the universe, the ordinary less-than relation for R. The role of the universe now becomes more important than before; it fixes the meaning of the *existential quantifier* $\exists y$. The actual meaning of "$\exists y$", under the given interpretation, is: "there exist $y \in \mathbb{N}$ such that". If we changed $\mathbb{N}$ to something else, the meaning of this would change. If the universe is U, the meaning of $\exists y$ is:

$\exists y$ $\sim$ "there exist $y \in U$ such that"

(we again used the symbol $\sim$ to signify "means the same under the given interpretation as").

Now, what does (3) evaluate to? First of all notice that the expression (3) really has only two "free" variables: x and z; these are the ones that we can freely give values to. Thus, when $x=1$ and $z=3$, (3) becomes

$$\exists y(Rxy \wedge Ryz) \sim \exists y(R(1, y) \wedge R(y, 3))$$
$$\sim \text{ there exists } y \in \mathbb{N} \text{ such that } R(1, y) \text{ and } R(y, 3)$$
$$\sim \text{ there exists } y \in \mathbb{N} \text{ such that } 1 < y \text{ and } y < 3,$$

and this is true, since $y=2$ is a suitable value making " $1 < y$ and $y < 3$ " true:

$$\exists y(Rxy \wedge Ryz) \sim \exists y(R(1, y) \wedge R(y, 3)) \sim \top.$$

On the other hand, when $x=1$ and $z=2$, then

$$\exists y(Rxy \wedge Ryz) \sim \exists y(R(1, y) \wedge R(y, 2))$$
$$\sim \text{ there exists } y \in \mathbb{N} \text{ such that } R(1, y) \text{ and } R(y, 2)$$
$$\sim \text{ there exists } y \in \mathbb{N} \text{ such that } 1 < y \text{ and } y < 2,$$

and this is *false* since there is no natural number (element of $\mathbb{N}$) which would be both greater than 1 and smaller than 2:

$$\exists y(Rxy \wedge Ryz) \sim \exists y(R(1, y) \wedge R(y, 2)) \sim \bot.$$

On the other hand, *it does not make sense to give a value to the variable* y in (3): it does not make sense to write

?? $\exists 6(R(x, 6) \wedge R(6, z))$

?? there exists 6 such that $R(x, 6)$ and $R(6, z)$.

If we must, then we would give the same value to this as to

$R(x, 6)$ and $R(6, z)$;

but for this, we did not need the "quantifier" $\exists 6$.

We say that the variables x and z in the above formal expression are *free variables*; and that y is a *bound variable.*

It is not a mistake to have the same variable both as a free and a bound variable in the same expression; but this can be avoided, by renaming the bound variable. For instance,

$$\exists y(Rxy \wedge Ryz) \rightarrow \exists zRxz$$

is a legitimate logical expression in which z is both free and bound; but it is the same, in essence, as

$$\exists y(Rxy \wedge Ryz) \rightarrow \exists yRxy \ ;$$

and the latter is clearer in its meaning.

Let us emphasize again that in a logical expression,

only the free variables can be freely evaluated; when all the free variables are evaluated, the whole expression assumes a definite truth-value.

Besides the existential quantifier, we also have the *universal quantifier.* With any variable, we can combine the symbol $\forall$, and prefix the result to any logical expression. For instance, we may write

$$\forall y(Rxy \rightarrow \neg Ryx) \ . \qquad (4)$$

The universal quantifier $\forall y$ is read "for all y "; or, "for every y ". Thus, the last-displayed expression is read as

"for all y , if Rxy , then *not-*Ryx ".

When we use the same U as before (U=$\mathbb{N}$), and the same R as well, the expression

evaluates to

"for all $y \in \mathbb{N}$, if $x<y$, then *not*-$y<x$".

In this expression (4), we have one free variable, x; and one bound variable, y. We see that, under the given interpretation, (4) is true no matter what the value of the free variable x is:

$$\forall y(Rxy \rightarrow \neg Ryx)$$
$$\sim \text{ "for all } y \in \mathbb{N}, \text{ if } x<y, \text{ then } \textit{not}\text{-}y<x\text{"} \sim \mathsf{T}. \qquad (4')$$

Just like Boolean connectives, quantifiers too can be used iteratively. For instance, we may write the logical expression

$$\forall x \exists y Rxy \qquad (5)$$

and this receives the value T identically, when our choice of the universe and the interpretation of R are as before:

$\forall x \exists y Rxy \sim$
for all $x \in \mathbb{N}$, there exists $y \in \mathbb{N}$ such that $x<y$
$\sim \mathsf{T}$.

Note that in (5), there is no free variable at all. As a consequence, as soon as the universe and the relation R are specified, the sentence (5) receives a definite truth-value.

Let us summarize what we have in the way of *predicate logic*, also called *first order logic*.

First of all, we have a notion of *logical expression*. Logical expressions are also called *well-formed formulas*, or simply, *formulas*, of first order logic. We will simply say "formula" now, although this is clearly a terminology that could be confusing, given the fact that in mathematics, many things are called "formulas".

A *formula* is built up out of *atomic formulas*. An atomic formula is of the form

$$Rx_1x_2\ldots x_k,$$

where the x_i's are *variables*, and R is a *k-ary relation symbol*. For instance, R above was used as a binary relation symbol ($k=2$). But we could use, profitably, S as a ternary relation symbol ($k=3$) , for instance, when we want to talk about the relation

$$Sxyz \sim x+y=z$$

on the universe U=$\mathbb{N}$ (or, for that matter, $U=\mathbb{Z}$, $U=\mathbb{Q}$, $U=\mathbb{R}$ or $U=\mathbb{C}$ -- or even many more). The language of first order logic allows k-ary relation symbols for arbitrary natural numbers $k=0, 1, 2, \ldots$. The case $k=0$ corresponds to a propositional atom: the "atomic formulas of propositional logic".

Out of atomic formulas, we build compound formulas by using the Boolean connectives

$$\wedge , \vee , \neg , \rightarrow , \leftrightarrow , \top , \bot ,$$

and the quantifiers

$$\exists x , \qquad \forall x$$

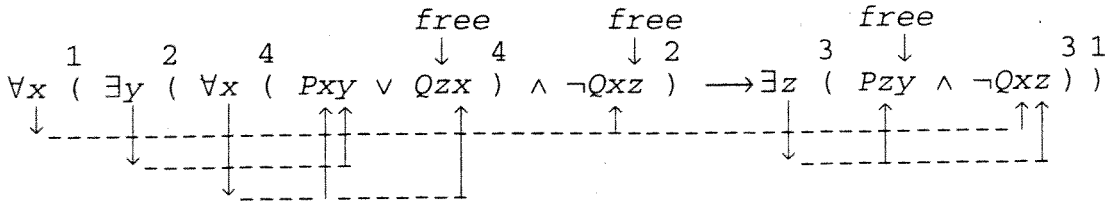
with arbitrary variables x . The connectives and the quantifiers are jointly called *logical operators.* (Note that it is $\exists x$, rather than $\exists$, what we call a quantifier.) Of course, these operators must be applied "meaningfully". For instance, $\wedge$ is always applied to two formulas: of Φ and Ψ are formulas, then so is $\Phi\wedge\Psi$; when Φ and Ψ are compound formulas, we have to use parentheses: $(\Phi)\wedge(\Psi)$. Note that we listed $\top$ and $\bot$ as connectives; they are 0-ary connectives; they do not apply to given formulas, but they are, by themselves, formulas. Thus, for instance, if Φ is a formula, then $\Phi\rightarrow\bot$ is also a well-formed formula.

All formulas are obtained by the iterated application of the logical operators, starting with the atomic formulas (and $\top$ and $\bot$).

Formulas may have *free* variables and *bound* variables in them.

Consider the following formula and an annotated version of it:

$$\forall x(\exists y(\forall x(Pxy \vee Qzx) \wedge \neg Qxz) \longrightarrow \exists z(Pzy \wedge \neg Qxz))$$



There are four quantifiers in the formula: $\forall x$ occurring twice, $\exists y$ and $\exists z$. Each quantifier has a definite *scope*: the part of the formula it has effect over. The scope of the first $\forall x$ is enclosed by the pair of parentheses marked by $\overset{1}{(}\ \overset{1}{)}$, the scope of $\exists y$ by $\overset{2}{(}\ \overset{2}{)}$; the scope of $\exists z$ by $\overset{3}{(}\ \overset{3}{)}$; the scope of the second $\forall x$ by $\overset{4}{(}\ \overset{4}{)}$.

For a quantifier Qu (where Q may be $\forall$ or $\exists$),

Qu *binds each occurrence of the variable* u *in its scope unless the occurrence is already bound by another copy of* Qu *inside the scope of the first one;*

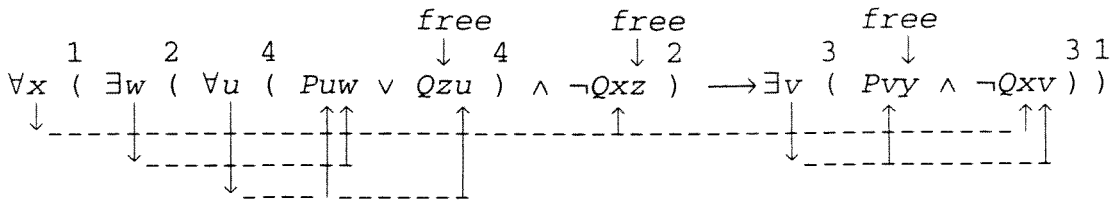
Qu does not bind any other occurrence of u, and any variable other than u. When we talk about "occurrences" of a variable, we disregard those copies of the variable that are inside a quantifier. Thus, in the example, the copies of the variables that have the arrows coming out of them (these are the ones that are inside quantifiers) do not count as "occurrences" in what we are saying here.

For instance, in our example, the first $\forall x$ binds two occurrences of x; it does not bind two other occurrences of x. All occurrences of x (there are four of them) are in the scope of $\forall x$. However, two of them are bound by the second $\forall x$, which is inside the scope of the first $\forall x$. The binding relations are shown by arrows connecting occurrences of variables with variables inside quantifiers.

A *bound variable occurrence* is one that is bound by a quantifier; a *free variable occurrence* is one that is not bound by any quantifier. A *free variable* of a formula is one that has at least one free occurrence in the formula; similarly, for "bound variable".

Bound variables are "dummy"; they can be replace by others provided one does not change the binding pattern of the formula. In our example, the formula may be changed, without changing the meaning of the formula, to this:

$$\forall x(\exists y(\forall u(Puy \vee Qzu) \wedge \neg Qxz) \longrightarrow \exists v(Pvy \wedge \neg Qxv))$$

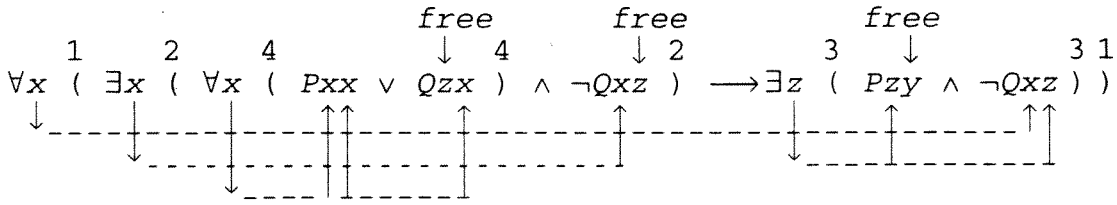


We have changed $\exists y$ to $\exists w$ and the y bound by $\exists y$ to w. We have changed the second $\forall x$ to $\forall u$, and we changed the two x's bound by the second $\forall x$ to u's. We have changed $\exists z$ to $\exists v$, and changed the two z's bound by $\exists z$ to v's. The binding pattern of the formula has not changed.

(Changing *free* variables is not allowed; it changes the meaning of the formula!)

The new version we obtained above of our formula has the advantages that (1) no variable is both free and bound in it (in the original version, all three variables x, y, z were both free and bound variables in the formula); and (2) no quantifier appears twice (in the original formula, $\forall x$ appeared twice). These features make for better readability. However, it should be emphasized that the original formula was perfectly well-formed as it was.

Let us emphasize that changing of bound variables is allowed as long as it does not alter the binding pattern of the formula. In our example, changing $\exists y$ to $\exists x$, with changing the y bound by $\exists y$ to x, would be inadmissible; it would result in



which has a different binding pattern. However, we can always make a change of bound

variables if we use a *brand new* variable: one that is not used elsewhere in the formula at all. In fact, this is what we did three times in our first series of changes of bound variables. Sometimes, however, one can repeat a variable without harm (that is, without altering the binding pattern). For instance, in the formula that we obtained before, we could change v to either u or w without harm.

Note that it is not necessary that a variable appear anywhere else in the formula, other than the quantifier. For instance,

$$\exists y Rxz$$

is a meaningful formula. In the previously considered interpretation, it means that "there is $y \in \mathbb{N}$ such that $x<z$ ". x and z are free variables in this. Although meaningful, $\exists y Rxz$ is of the same meaning as Rxz, and thus, the use of $\exists y$ was superfluous in this case.

As we noted, it may happen that a formula has the same variable both free and bound; on the other hand, this can always be avoided, and in fact, it is advisable to avoid it.

Formulas can be *interpreted.* This takes place upon the specification of a universe of discourse, or simply, *universe*, and the specification of a *relation corresponding to* each relation symbol in the formula. A specification of a universe, together with specific relations corresponding to the relation symbols is called a *structure*, or an *interpretation.*

We stipulate that *the universe be always a non-empty set.* It would not be meaningless to consider an empty universe -- but if we did not exclude the empty universe, certain things would become more complicated (e.g., what we said about $\exists y Rxz$ above would become incorrect).

When R is a k-ary relation symbol, its interpretation has to be a *k-ary relation.* For instance, a ternary ($k=3$) relation on the set U is a subset $R \subseteq U \times U \times U = U^3$, that is, R is a set of ordered triples (a, b, c) of elements a, b, c of U. An example was given above: with $U=\mathbb{N}$ (for instance), we could interpret the ternary relation symbol R as the relation $\{(a, b, c) \in \mathbb{N}^3 : a+b=c\}$; this is the same thing as to stipulate that

$$Rxyz \sim x+y=z\,,$$

with the understanding that the variables x, y, z *range over* $\mathbb{N}$, which phrase is just another way of saying that the universe is $\mathbb{N}$.

When an *interpretation* is fixed, *and* the free variables are assigned specific values (which have to be elements of the universe), the formula evaluates to a definite truth-value. We saw some examples for this above. We also saw that if the formula has no free variable, then it evaluates directly to a truth-value, without our having to specify any additional value of a (free) variable.

The word "predicate" signifies the entity that is obtained from a formula when an interpretation (structure) is given, but the values of the free variables are left undetermined. Predicates are almost the same as relations, except that their places have been filled by variables, and the identity of these variables is important. For instance, from a relation R (not just a relation-symbol!), we can form the predicates Rxy, Ruv, Ryx, ...; these are all *distinct* predicates.

It should not be thought that the evaluation of a formula is always an easy matter. Let us use relation symbols R (binary), S, P (both ternary), $\mathbf{1}$, $\mathbf{2}$ (both unary, $k=1$), and $=$ (binary; "equality"), and write down the formulas

$$\Phi(x) \equiv \neg\mathbf{1}(x) \wedge \forall y\forall z(Pyzx \rightarrow (y=x \vee z=x))\,; \qquad (5')$$

and

$$\Psi \equiv \forall u\exists x\exists w(Rux \wedge \Phi(x) \wedge \Phi(w) \wedge \exists v(\mathbf{2}(v) \wedge S(x, v, w)))\,.$$

Here, the second formula is indicated in an abbreviated manner; the previous formula Φ is used in it twice, once with x, and once with w as the free variable. By the way, writing $\Phi(x)$ is a way of indicating that the formula Φ has x as its sole free variable. The unabbreviated way of writing Ψ is

$$\Psi \equiv \forall u\exists x\exists w(Rux \wedge \neg\mathbf{1}(x) \wedge \forall y\forall z(Pyzx \rightarrow (y=x \vee z=x)) \wedge \neg\mathbf{1}(w) \wedge \forall y\forall z(Pyzw \rightarrow (y=w \vee z=w))$$

$$\wedge \exists v(\mathbf{2}(v) \wedge S(x, v, w))) \ .$$

Now, note that, when we put $U=\mathbb{N}$,

$$\begin{aligned} Rxy &\sim x<y, \\ Sxyz &\sim x+y=z, \\ Pxyz &\sim x\cdot y=z, \\ \mathbf{1}(x) &\sim x=1, \\ \mathbf{2}(x) &\sim x=2, \end{aligned}$$

and, by $x=y$ we mean ordinary equality (as we always do in first order logic), then the formula Φ becomes

$\Phi(x) \sim x\neq 1$ and for all $y, z\in\mathbb{N}$,
if $y\cdot z=x$, then either $y=x$ or $z=x$

$\sim$ x is a prime number ;

and the sentence Ψ becomes

$\Psi \sim$ for all $u\in\mathbb{N}$, there are $x, w\in\mathbb{N}$ such that $u<x$,
both x and w are prime numbers, and $x+2=w$

$\sim$ there are infinitely many pairs of prime numbers x and w
such that $x+2=w$

$\sim$ there are infinitely many twin primes

(x and w are "twin primes" if they are both prime numbers and $x+2=w$. Note that 3 and 5, 17 and 19, are twin primes. Also note that

$\exists v(\mathbf{2}(v) \wedge S(x, v, w)) \sim$ there is $v\in\mathbb{N}$ such that $v=2$ and $x+v=w$

$\sim x+2=w$.)

Nobody knows whether Ψ evaluates to $\top$, or to $\bot$ under the given interpretation, because nobody knows whether there are infinitely many twin primes.

Many mathematical matters can be expressed in first order logic. In fact, if we use the universe of all sets, *everything* mathematical can be expressed in first order logic -- or at least, experience seems to show that this is the case. (Note that such a statement is not subject to proof; its truth depends on what we deem to belong to mathematics; and this cannot be done by any means other than agreement.)

Let us use another piece of notation. Given a *sentence* Φ, meaning a formula without any free variable, and a structure $(U; R, S, \ldots)$ where the universe is U, and the relation R is meant to interpret the relation-symbol R (here, we employ a double meaning of the same letter R; but this kind of thing is done a lot in mathematics; one tries to be careful not to get confused ...), the relation S interprets the relation symbol S, etc, we write

$$(U; R, S, \ldots) \models \Phi$$

to mean that Φ is true, $\Phi \sim \top$ under the given interpretation $(U; R, S, \ldots)$. The symbol $\models$ is read as "satisfies": the whole phrase reads "the structure $(U; R, S, \ldots)$ satisfies the sentence Φ".

Let us consider a few examples for expressing properties of a (binary) relation in first order logic. We have in mind a binary relation (A, R) ; $R \subseteq A \times A$. We also use the *equality relation* $x=y$; a binary relation (symbol). Since $x=y$ is always interpreted as ordinary equality, it does not have to be listed in an interpretation.

Then

(A, R) is reflexive $\iff$ $(A, R) \models \forall x Rxx$;

(A, R) is symmetric $\iff$ $(A, R) \models \forall xy(Rxy \rightarrow Ryx)$

($\forall xy$ abbreviates $\forall x \forall y$);

(A, R) is transitive $\iff$ $(A, R) \models \forall xyz((Rxy \wedge Ryz) \rightarrow Rxz)$;

(A, R) is irreflexive $\iff$ $(A, R) \models \forall xy(Rxy \rightarrow x \neq y)$;

(A, R) is antisymmetric $\iff$ $(A, R) \models \forall xy((Rxy \wedge Ryx) \rightarrow x=y)$;

(A, R) is strictly antisymmetric $\iff$ $(A, R) \models \forall xy \neg(Rxy \wedge Ryx)$;

(note that, following (4'), another way of expressing strict antisymmetry is

(A, R) is strictly antisymmetric $\iff$ $(A, R) \models \forall xy(Rxy \rightarrow \neg Ryx)$)

(A, R) satisfies dichotomy $\iff$ $(A, R) \models \forall xy(Rxy \vee Ryx)$;

(A, R) satisfies trichotomy $\iff$ $(A, R) \models \forall xy(Rxy \vee x=y \vee Ryx)$.

In applications of predicate logic, one wants to use, besides *relations*, also *operations*. As a matter of fact, the formal use of operations is not necessary; it is something that may be called "syntactic sugar" on predicate logic. We saw above how to express arithmetical matters by relations only.

Each operation symbol comes with a definite *arity*, a natural number; if the arity is n , we talk about an *n-ary*, or *n-place* operation symbol. We use the letters f, g, h for operation symbols -- but also, special symbols such as $+$, $\cdot$, etc.

The syntax of operations is this. We define a grammatical category of entities called *terms* as follows:

every variable is a *term*;

if f is an n-ary operation symbol, and t_1, t_2, ..., t_n are *terms*, then the expression $f(t_1, t_2, \ldots, t_n)$ is a *term*;

the only *terms* are obtained by the previous two clauses.

For instance, if f is ternary, g is unary, and c is a zero-ary operation symbol (denoting, in any interpretation, a distinguished element of the universe), and x , y are variables, then

$$f(c, g(f(x, c, y)), g(y))$$

is a term. The special binary symbols $+$, $\cdot$ and some others are used with infix notation: instead of $+(t_1, t_2)$, one writes t_1+t_2 .

Terms are used to define *atomic formulas*:

an *atomic formula* is an expression of the form $R(t_1, t_2, \ldots, t_n)$, where R is an n-ary relation symbol, or an expression of the form $t_1=t_2$, with each t_i being a term.

Finally, the general definition of *formula* is as follows:

every atomic formula is a *formula*;
if Φ and Ψ are formulas, then
$\neg(\Phi)$, $(\Phi)\wedge(\Psi)$, $(\Phi)\vee(\Psi)$, $(\Phi)\longrightarrow(\Psi)$, $(\Phi)\longleftrightarrow(\Psi)$ are *formulas*;
if Φ is a *formula*, and x is a variable, then $\forall x(\Phi)$, $\exists x(\Phi)$ are *formulas*;
the only *formulas* are obtained by the previous three clauses.

Interpretations of operation symbols are operations on the given universe. For instance, if f is a ternary operation symbol, then it may be interpreted by a three-place operation, also denoted by f , with $f: U\times U\times U\longrightarrow U$.

The interpretation of terms, based on that of the operation symbols, is self-explanatory: it follows the usual algebraic practice of using compound algebraic expressions.

The important thing to emphasize is that all operations must be everywhere defined on U . This is necessary for the general rules of logic (see section 5.3) to remain valid in the presence of the operations.

For instance, using the arithmetic division $/$ in formal logic is admissible only if we artificially agree that $a/0$ is something specific, say 0 itself. Then, of course, one still has to worry whether $a/b = 0$ means that $a=0$ (the "genuine" case), or $b=0$ (the "artificial" case). It will *not* be true that

? $\forall a\forall b((a/b)\cdot b=1)$;

instead,

$$\forall a \forall b (b \neq 0 \longrightarrow (a/b) \cdot b = 1) \ .$$

Section 5.2 Formal specification of computer programs

One of the most important areas where quantifier logic is used is *formal specification of computer programs*.

Specification takes place on several levels in software development. The most obvious event of specification happens at the very start: we have to state what we want the software to do. Very often, at this stage, specification is *informal* and incomplete. Even if we know precisely what we want -- for instance, a program to calculate the gcd (greatest common divisor) of two integers --, the task is far from being well-specified, since there may be very different computational ways of achieving the stated goal, some more efficient than others. After the initial statement of the main goal, one may specify further things that the program should involve: one *refines* the specification. The refinement of the specification may take several successive steps.

In short, a specification tells us *what* to do; and only indirectly, *how* to do it. However, if one is told *what* to do in *sufficient detail*, it is likely to become clear how to do it.

It is important to stress that specification is something separate from actually writing the program. The most obvious sign of this fact is that specification takes place in a *different language* from the code-language of the program. For instance, we will use predicate logic for specification; the code for the program answering the specification would then be written in any one of the usual languages such as PASCAL or C.

The idea is that the writing of the code for a *well-specified* program is easy -- or at least easier than without the specification. It should be emphasized that the advantages of good program specification, preceding the actual writing of the code, will become really clear when the program is large. Of course, here where we see only a first introduction to specification -- a large subject of computer science -- , we will deal only with simple computational tasks when the code is fairly easy to write already without specification.

Here, we will see how predicate logic can be used for the *formal* specification of programs.

Why *formal* specification? The answer is clear: the formal way of stating what we want should make sure that there is certainty about the matter. For instance, it should ensure that there is no

misunderstanding on the part of the code-writer about the intentions of the specifier. But also, sometimes it happens we think we know precisely what we want, and that what we want makes sense -- and upon further analysis, it turns out that our ideas were not well-formed. The discipline of formal specification forces the specifier to be specific and consistent.

Complete formal specification should be something that *uniquely determines* the coded program: ideally, something that can be *automatically* compiled into a code in the target language. Completeness in this sense may be a commendable ideal; however, one may argue that it is contrary to the spirit of the task of specification. In fact, the natural role of the specification is the production of a relatively short statement of the goals and main characteristics of the program. The actual code may then include tricks and features that are specific to the code language, one that may not even be known to the specifier.

Let us return to our first example already mentioned above: the calculation of $\gcd(a, b)$ for $a, b \in \mathbb{N}$. (We might regard the set $\mathbb{Z}$ as the proper "universe" for the purposes of the gcd function; however, to keep the example simple, we stick to $\mathbb{N}$.) When we want to tell somebody to write a program to calculate $\gcd(a, b)$ for $a, b \in \mathbb{N}$, the first thing we have to tell the person *what* the gcd is. Here is a formal specification of the function $\gcd(a, b)$:

$$d = \gcd(a, b) \iff d \mid a \wedge d \mid b \wedge \forall z((z \mid a \wedge z \mid b) \longrightarrow z \mid d) \ .$$

We have to understand what the *basis* of this (and of every) specification is. What happens here is that we have in mind a *universe* U -- which in this case is $\mathbb{N}$ -- , and we have given *primitive* predicates and operations on this universe that are *assumed understood* -- in this case, the single *primitive* predicate is the binary predicate "divisibility", $\mid$ ($a \mid b$ meaning " a divides b "). Of course, the variables in the specification are ranging over the given understood universe. In short, the basis of the present specification is the structure $(\mathbb{N}; \mid)$.

What should be understood very clearly is that nothing can be gotten from nothing: every specification has to assume certain things already known and understood. However, it is not at all fixed once and for all *what* these known and understood *primitive* notions are. For instance, our example can be *refined* by incorporating a specification of the divisibility relation, to get the two-line specification:

$$x \mid y \iff \exists z(x \cdot z = y)$$

$$d = \gcd(a, b) \iff d|a \wedge d|a \wedge \forall z((z|a \wedge z|b) \longrightarrow z|d) \ .$$

As a matter of fact, one can eliminate the use of the symbol | by using its definition (the first line) inside the body of the specification of the gcd function:

$$d = \gcd(a, b) \iff$$
$$\exists x(d \cdot x = a) \wedge \exists y(d \cdot y = b) \wedge \forall z((\exists u(z \cdot u = a) \wedge \exists v(z \cdot v = b)) \longrightarrow \exists w(z \cdot w = d)) \ .$$

However, this way of writing the thing should be seen as inferior to the first, two-line, version: the body-formula is less transparent in the second version.

The basis of the new specification of the gcd function, in either the one-line or in the two-line form, is the structure $(\mathbb{N}; \cdot)$: the universe $\mathbb{N}$, with the binary operation $\cdot$ (multiplication). We also use equality (=) now; but equality is understood to be part of the general arsenal of logic, something that need not be enumerated when the basis of the specification is stated.

Let us try to give a general notion of "specification", even if it is "too general" at first sight. (The famous example of a notion that was first considered "too general", but which has turned out to be fundamental, is the modern notion of *function* in mathematics.) I first reformulate the two specifications we have seen *as single sentences*:

Φ :=:
$$\forall a \forall b \forall d[\quad d = \gcd(a, b) \longleftrightarrow (d|a \wedge d|b \wedge \forall z((z|a \wedge z|b) \longrightarrow z|d))] \ .$$

Ψ :=:
$$\forall x \forall y[\quad x|y \longleftrightarrow \exists z(x \cdot z = y)]$$
$$\wedge$$
$$\forall a \forall b \forall d[\quad d = \gcd(a, b) \longleftrightarrow (d|a \wedge d|a \wedge \forall z((z|a \wedge z|b) \longrightarrow z|d)))] \ .$$

For the first specification, Φ, we have

$$(\mathbb{N}; |; \gcd) \models \Phi \ ;$$

for the second, Ψ,

$$(\mathbb{N}; \cdot, |, \mathtt{gcd}) \models \Phi$$

Note the placing of the two semicolons ";" in the two cases. In both cases, $\mathbb{N}$ is the universe; it is separated from the rest of the data by a semicolon. Next comes the list of the *basis* of the spec, the relations and/or operations assumed to be known. In the first case, this is the single relation $|$; in the second, it is the single operation $\cdot$. Finally, we have the list of the relations and/or operations that are being specified. In the first case, this is just the operation $\mathtt{gcd}$; in the second, the system of two components, the relation $|$ and the operation $\mathtt{gcd}$.

Note that in both cases we have the simple statement of a sentence being true (satisfied) in a structure. This suggests the following general concept.

A *first order specification* consists of:

(i) a sentence Φ , called the *body* of the spec, in which we distinguish two kinds of relation/operation symbols: one kind is the *primitives* (say, $P, \ldots, f, \ldots$) ; the other the *unknowns* (say, $Q, \ldots, g, \ldots$);

(ii) some *universe* U , and known relations/operations $P, \ldots, f, \ldots$ on U corresponding to the primitive symbols.

A system of relations/operations $Q, \ldots, g, \ldots$ on the given universe U , and corresponding to the "unknowns" in Φ , will *answer* the spec, or, will be *a solution* of the spec, if

$$(U; P, \ldots, f, \ldots; Q, \ldots, g, \ldots) \models \Phi \ .$$

Note the analogy with solving equations, in particular, a system of differential equations. In the latter case, we have some given functions that figure in the formulation; these are the ones that corresponds to our primitives; and there are the unknown functions which we seek. This analogy suggests the following definition.

We say that a specification is *exact* if it has a *unique solution*: in the above notation, it has a uniquely determined system of relations/operations $Q, \ldots, g, \ldots$ for which (1) holds.

Of course, both of the two specs we have looked at are exact: the descriptions of the unknowns in the body of the spec are in fact *explicit definitions* of them in terms of the primitives.

It may be argued that the specifications of the `gcd` function given so far are not telling us how to *calculate* the gcd of any given pair of numbers. Indeed, a good specification should, by its very form, and by what auxiliary items it contains, point to a computation of the target function. We will now consider *better* specifications of the `gcd` function.

The well-known way of calculating `gcd(a, b)` starts with factoring the numbers into the product of prime numbers, after which the gcd is obtained by comparing the two factorizations. A very different, and very much more efficient way of calculating `gcd(a, b)` is the so-called *Euclidean algorithm* that we are now going to describe in essence.

For now, variables $a, b, \ldots, x, y,$ range over $\mathbb{Z}$, the set of all (positive, negative, zero) integers. A *linear combination* of a and b is any integer of the form $ax+by$ (remember that x, y must also be integers). Note the following obvious fact:

any common divisor of a *and* b *is a divisor of* $ax+by$.

(why?). The following is a consequence:

Assume that the pairs (a, b) and (c, d) of integers are such that

both c and d are linear combinations of a and b,

and vice versa:

both a and b are linear combinations of c and d.

Then the common divisors of a and b are the same as the common divisors of c and d. In particular, `gcd(a, b) = gcd(c, d)` .

(Verify!; note that here we take the *existence* of the gcd for granted.)

Now let $b>0$, and $a\geq 0$. We can write

$$a=qb+r \text{ with } 0\leq r<b \qquad (1)$$

[remember that all variables denote integers!]; this is division-with-remainder; q is the quotient, r is the remainder. As usual, we write $a \bmod b$ for this r. Also, we make the convention that $a \bmod 0 = a$, to ensure that $a \bmod b$ is well-defined for all $a, b \in \mathbb{N}$.

Now, consider that under (1), we have that

both a and b are linear combinations of b and r;
both b and r are linear combinations of a and b.

For the first:

$$a=b\cdot q+r\cdot 1\,, \qquad b=b\cdot 1+r\cdot 0\,;$$

for the second:

$$b=a\cdot 0+b\cdot 1\,, \qquad r=a\cdot 1+b\cdot(-q)\,.$$

It follows that, under (1)

$$\gcd(a,\,b)\;=\;\gcd(b,\,r)\,. \tag{2}$$

We are ready to give our improved spec for $\gcd$, based on the fact (2).

The following is a first-order specification of the "unknown" functions $a\mathrm{mod}b$, $\gcd(a,\,b)$ on the universe $\mathbb{N}$ with primitives 0, $+$, $\cdot$ and $<$:

$$\forall a\forall b\forall r[\quad (a\mathrm{mod}b=r \longleftrightarrow ((0<b \wedge \exists q(a=q\cdot b+r) \wedge r<b) \tag{3.1}$$

$$\vee\ (0=b \wedge r=a))\ ] \tag{3.2}$$

$$\wedge$$

$$\forall a\ [\quad \gcd(a,\,0)=a\ ] \tag{3.3}$$

$$\wedge$$

$$\forall a\forall b[\quad \gcd(a,\,b)\;=\;\gcd(b,\,a\mathrm{mod}b)\ ] \tag{3.4}$$

I **claim** that in fact the displayed sentence is an exact specification of the (true) functions $a\mathrm{mod}b$ and $\gcd(a,\,b)$: that is, the description given in the displayed sentence determines the functions $a\mathrm{mod}b$ and $\gcd(a,\,b)$ uniquely.

Before we substantiate this claim, we make several remarks.

Firstly, I want to draw attention to the essentially more sophisticated nature of the present specification, in comparison to the previous ones. The present one is *not* what we would usually accept as an (explicit) definition of the `gcd` function. The key line (3.4) contains the symbol `gcd` on both sides of the equation; it does not tell outright what `gcd(a, b)` is, but it says that it is equal to another value of the same "unknown" function!

In fact, what we have here is a *recursive* specification. This term refers to the fact that the specification, instead of giving a direct way of determining a value of the function at hand, it gives a way how to *reduce* the calculation of the value to another value (or in some cases, several other values), which latter value may then be given directly, or else may be subjected to another reduction; and so on. In our case, in (3.4), we *reduce* the value at the arbitrary `(a, b)` to the value at `(b, amodb)` . To be sure, some values should be given directly: in our case, the ones in line (3.3) are so given.

In using a recursive definition, the first question whether it is *consistent*: whether the said reductions will stop at a unique value that is given directly. What we have here is a consistent recursion; we will remark on why that is the case below when we look at an example.

Consider the example when we want to calculate `gcd(6840,99900)` . I construct the following table:

a	*b*	*r*=amod*b*	*q*=[*a*/*b*]
6840	99900	6840	0
99900	6840	4140	14
6840	4140	2700	1
4140	2700	1440	1
2700	1440	1260	1
1440	1260	180	1
1260	180	0	7
180	0	180	undefined

In the first row of the table, the choice of the values of a and b is given by our input: 6840 and 99900.

In each subsequent row, we follow the hint under (3.4): with any given pair (a, b) in a row already completed, the next row has the pair $(b, a \bmod b)$ for the positions under the headings a and b, unless $b=0$. This means that in row $i+1$, the first and second items are the same as the second and third items in row i, unless the second item is 0. Once we find that the second item is 0, we stop, and do not generate any more rows.

In each row except the last one, we have the results of a calculation corresponding to line (3.1) of the spec. That is, we have the values of a and b at which we instantiate the universal quantifiers $\forall a$ and $\forall b$ in line (3.1), the value of r which is taken to be the value of $a \bmod b$ (this is the one value that makes the part $a \bmod b = r$ true), and the corresponding witness q for $\exists q$ (this is the same as the integer part of a/b).

In the last row, we have an application of line (3.2).

Looking at line (3.4), we see that

the gcd of the first and second numbers in each row remains constant throughout.

Finally, by line (3.3), we get that $\gcd(180, 0) = 180$. Therefore, since the gcd is constant across the table, we conclude that

$$\gcd(6840, 99900) = 180.$$

It is clear what is going on in general. Perhaps the main point is that in the second column the numbers must be strictly decreasing, by the inequality $r<b$ in line (3.1), and therefore they must come to 0 sooner or later; but at that point, the gcd is the number in the first column to the left of that 0, which is the same as the number just above the 0 in the second column.

What we see is that our new specification is suggestive, to say the least, on how to calculate the gcd at given arguments, unlike the first, purely theoretical (however *exact*) specification. But, it seems, we can do better yet, in the sense that we can make explicit what the calculation of the above table involves.

To deal with recursive definitions formally, it is very useful to introduce *strings*, in particular, strings of integers, and take as our basis a certain stock of primitives dealing with strings. From the logical point of view, we take a universe that includes both integers (elements of $\mathbb{N}$; we still restrict integers to non-negative ones) and strings of integers. Writing $\mathbb{N}^*$ for the set of all strings of integers, our universe becomes $U = \mathbb{N} \cup \mathbb{N}^*$. Actually, $\mathbb{N}^*$ would be enough, since individual integers could be identified with one-term strings -- but we will keep both $\mathbb{N}$ and $\mathbb{N}^*$.

With this universe adopted, the first two primitive predicates are the unary predicates $\mathbb{N}$ and $\mathbb{N}^*$, subsets of U. These are needed since we want to say for $x \in U$, whether it is an integer or a string. In the first case $\mathbb{N}(x)$ is true (and $\mathbb{N}^*(x)$ is false); in the second case $\mathbb{N}^*(x)$ is true (and $\mathbb{N}(x)$ is false). When we want to say: "for all integers x, $P(x)$", where $P(x)$ is any statement involving x and possibly other variables, we write

$$\text{"for all integers } x\text{, } P(x)\text{"} \quad \equiv \quad \forall x(\mathbb{N}(x) \longrightarrow P(x)) \ ;$$

also,

$$\text{"there exists an integer } x \text{ such that } P(x)\text{"} \quad \equiv \quad \exists x(\mathbb{N}(x) \wedge P(x)) \ .$$

Note the differences in the two expressions.

Of course, similar expressions can be written for phrases involving strings rather than integers; these use $\mathbb{N}^*(x)$ rather than $\mathbb{N}(x)$.

Another, and our preferred, way of dealing with the two kinds of entities (integers and strings) is to use *variable declarations* that fix the meaning of certain variables as to range over one or the other kind of entity. For instance, it is customary to declare i, j, k, l, m, n to be integers; we may declare r, s, t to range over strings. Variable declarations assign *types* to variables:

$$i, j, k, l, m, n : \mathbb{N} \qquad r, s, t : \mathbb{N}^* \ ;$$

the *type* of the variables in the first set is $\mathbb{N}$; that of those in the second is $\mathbb{N}^*$.

The use of typed variables in quantifiers will have the expected meaning -- which, however, has to be clearly kept in mind. Now, $\forall i.P(i)$ means $\forall x(\mathbb{N}(x) \longrightarrow P(x))$; and $\exists i.P(i)$ means $\exists x(\mathbb{N}(x) \wedge P(x))$; similarly for quantification of variables of type $\mathbb{N}^*$.

Another effect of the use of typed variables is that operations may now have variables that are restricted in meaning -- and so, the operation may not be defined on the whole universe. This is the case with the important operations related to strings. The problem with the operations not being everywhere defined is not serious. We may adopt the convention that they are always equal to 0 when they are not otherwise defined, and thereby making them defined everywhere. We will have to keep track of this convention by remembering that the value of the operation at certain arguments being zero may mean either that we either have a "genuine" 0-value, or that we have a "conventional" 0-value.

Λ = the *empty string* (this is a constant, or zero-ary operation)

$\mathrm{lh}(s)$ = the *length* of the string s ; the number n if the string is $s = \langle k_0, k_1, \ldots, k_{n-1} \rangle$. The empty string has length equal to 0 .

$s\hat{}t$ = *concatenation* of s and t ; if $s = \langle k_0, k_1, \ldots, k_{n-1} \rangle$ and $t = \langle \ell_0, \ell_1, \ldots, \ell_{m-1} \rangle$, then

$$s\hat{}t = \langle k_0, k_1, \ldots, k_{n-1}, \ell_0, \ell_1, \ldots, \ell_{m-1} \rangle .$$

The length of $s\hat{}t$ is $\mathrm{lh}(s\hat{}t) = n+m = \mathrm{lh}(s)+\mathrm{lh}(t)$.

For s a string, and $i<\mathrm{lh}(s)$,

$s(i)$ = the *ith component* of the string; here we start counting with 0 , rather than 1 , and end *just before* the length. For instance, when $s=\langle 1, 3, 10, 8 \rangle$, then $s(0)=1$, $s(1)=3$, $s(2)=10$, $s(3)=8$; $lh(s)=4$.

Now, the symbol $s(i)$ actually denotes a *binary operation* whose first argument is s , a string-variable, and whose second argument is i , an integer variable. To make $s(i)$ meaningful for all i , and not just for $i<\mathrm{lh}(s)$, we declare, conventionally, that $s(i)=0$

every time $i \geq lh(s)$.

$\langle k \rangle$ = the string s for which $lh(s)=1$, and $s(0)=k$.

This is how we use strings to make our recursive specification of the gcd "explicit".

Given a and b (for instance, the numbers 6840 and 99900), we create a string s of integers for which

$$s(0)=a\ ,\ s(1)=b\ ,\ \text{and}\ s(i) = (s(i-1)) \bmod s(i-2)$$

as long as $s(i-2) \neq 0$. We do this until we get $s(i)=0$ with $i>0$; we stop incrementing i ; we denote the last i by N, and put $lh(s)=N+1$. The required $gcd(a, b)$ is $s(N-1)$.

The string s is basically the table shown above in the example, except that we suppress the systematic repetitions in that table.

In our example, $N=8$, $lh(s) = N+1 = 9$; and

$s(0) = 6840$
$s(1) = 99900$
$s(2) = 6840$
$s(3) = 4140$
$s(4) = 2700$
$s(5) = 1440$
$s(6) = 1260$
$s(7) = 180$
$s(8) = 0$

$gcd(s(0), s(1)) = s(8-1) = 180$.

Here is the specification that describes this procedure. The part of it relating to the definition of $amodb$ is unchanged. We are specifying the system of functions

$a\mathrm{mod}b$, $\mathrm{EUCLID}(a, b)$, $\mathrm{gcd}(a, b)$.

$\mathrm{EUCLID}(a, b)$, so named after Euclid who wrote down the algorithm in question more than two thousand years ago, is the string s given in the above description.

$$
\begin{array}{ll}
\forall a \forall b \forall r[& (a\mathrm{mod}b = r \longleftrightarrow (0<b \wedge \exists q(a = q \cdot b + r) \wedge r<b) \\
 & \vee (0 = b \wedge r = a))) \qquad] \\
\wedge & \\
\forall a \forall b \forall s[& \mathrm{EUCLID}(a, b) = s \longleftrightarrow s(0) = a \wedge s(1) = b \wedge \\
 & 1 < \mathrm{lh}(s) \wedge \\
 & \forall i(1<i<\mathrm{lh}(s) \longrightarrow s(i) = s(i-2) \bmod s(i-1)) \wedge \\
 & \forall i(0<i<\mathrm{lh}(s)-1 \longrightarrow s(i) \neq 0) \wedge \\
 & s(\mathrm{lh}(s)-1) = 0 \qquad] \\
\wedge & \\
\forall a \forall b[& \mathrm{gcd}(a, b) = (\mathrm{EUCLID}(a, b))(\mathrm{lh}(\mathrm{EUCLID}(a, b))-2) \qquad]
\end{array}
$$

The basis of the specification is the universe $U = \mathbb{N} \dot{\cup} \mathbb{N}^*$, together with the usual arithmetic relations and operations on $\mathbb{N}$, and the string-manipulation primitives listed above. The specification specifies three operations: $a\mathrm{mod}b$, $\mathrm{EUCLID}(a, b)$, and $\mathrm{gcd}(a, b)$; the first and third are integer-valued, the second is string-valued.

Section 5.3 Entailment in predicate logic

The fundamental problem of quantifier logic is to decide if certain formulas *entail* another formula or not:

$$? \qquad \Phi_1, \Phi_2, \ldots \Phi_\ell \vdash \Psi . \tag{1}$$

We say that (1) is the case, $\Phi_1, \Phi_2, \ldots \Phi_\ell$ *entail* Ψ, if the following holds: no matter how we specify a (non-empty) universe, and interpretations of the relation-symbols in all of the formulas $\Phi_1, \Phi_2, \ldots \Phi_\ell, \Psi$, and furthermore, no matter how we give values to all the free variables in all the formulas involved, every time when all the *premisses* $\Phi_1, \Phi_2, \ldots \Phi_\ell$ evaluate to T, the *conclusion* Ψ also evaluates to T.

Let us note that, in (1), the possibility $\ell=0$ is allowed. This means that Ψ is *identically true*, without any premiss, no matter what interpretation, and what values of the free variables in Ψ (if any) we take. We say that Ψ is *logically valid*, and write $\vdash \Psi$, if this is the case.

Of course, there are two possibilities with given premisses $\Phi_1, \Phi_2, \ldots \Phi_\ell$ and a given conclusion Ψ. Either (1) does hold, or it does not. The basic question is how we recognize if one or the other is the case.

To recognize that (1) does not hold,

$$? \qquad \Phi_1, \Phi_2, \ldots, \Phi_\ell \nvdash \Psi ,$$

what we need is an example (*counter-example*) consisting of an interpretation, and values of the free variables that make *all the premisses *true** and the *conclusion *false**.

For instance,

$$\forall x \exists y Rxy \nvdash \exists y \forall x Rxy , \tag{2}$$

as the following very simple example shows:

$$U = \{1, 2\}$$
$$R = \{(1, 2), (2, 1)\} .$$

Now, $\forall x \exists y Rxy \sim \top$: for every $x \in U$, there is $y \in U$ such that xRy : for $x=1$, we can take $y=2$, and for $x=2$, we can take $y=1$; but *it is not the case* that there is a $y \in U$ such that for all x, xRy : if $y=1$, then *not*-$1Ry$, and if $y=2$, then *not*-$2Ry$.

Another example showing (2) uses familiar material: we let $U=\mathbb{N}$, and $Rxy \sim x<y$. Then, clearly, $\forall x \exists y Rxy \sim \top$, and $\exists y \forall x Rxy \sim \bot$.

It must be said that it could be *very difficult* to find a counter-example to an entailment, even if such exists.

On the other hand, we have

$$\exists y \forall x Rxy \vdash \forall x \exists y Rxy .$$

This can be shown by a direct "logical argument", a simple mathematical proof, as follows.

We argue in any fixed, but otherwise unspecified, interpretation. Suppose that $\exists y \forall x Rxy \sim \top$. Then there is $y \in U$ such that $\forall x R(x, y) \sim \top$. Let us fix this y . Now, to show that $\forall x \exists y Rxy \sim \top$, let $x \in U$ be arbitrary. Using y specified above, we have $R(x, y) \sim \top$. This means that $\exists y R(x, y) \sim \top$. Since x was arbitrary in U, we have that $\forall x \exists y Rxy \sim \top$.

This proof was very easy; in fact, it was trivial. However, once again, it could be *very difficult* to prove that a given entailment holds true, even if it is in fact true.

There is a systematic way of showing of an entailment is true -- if it *is* true. In fact, if an entailment holds true, we can *deduce* it in a specific way. We will explain how this is done using the framework of Boolean algebras.

First of all, note that the fact that there are more than one premiss on the left of (1) is not really essential; (1) is equivalent to

$$\Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_\ell \vdash \Psi ,$$

where we took the conjunction of the premisses to be the new, single, premiss. Also, $\vdash \Psi$ iff $\top \vdash \Psi$ (why?). Let us fix a certain set of relation symbols and variables, and consider the set of all formulas using these only; let $\mathcal{F}$ be the set of all formulas. Therefore, what we really have is a *binary relation* $\vdash$ *called entailment on the set* $\mathcal{F}$. Now, we observe, very directly, that the relation $\vdash$ is reflexive, and transitive:

$\Phi \vdash \Phi$;
$\Phi \vdash \Psi$ and $\Psi \vdash \Lambda$ imply that $\Phi \vdash \Lambda$.

(as before, we use the upper-case Greek letters Φ , Ψ , Λ , ... to denote formulas, elements of $\mathcal{F}$). In other words, $\vdash$ is a *preorder* on $\mathcal{F}$. However, it is not an order: it is possible that $\Phi \vdash \Psi$ and $\Psi \vdash \Phi$, but $\Phi \neq \Psi$. For instance, when $\Phi = \forall xRxx$ and $\Psi = \forall yRyy$, this is the case. However, we feel that if $\Phi \vdash \Psi$ and $\Psi \vdash \Phi$ both hold then Φ and Ψ are "essentially the same".

The way we make this precise is that we define a relation, denoted $\equiv$, on formulas, by writing

$$\Phi \equiv \Psi \quad \overset{\text{def}}{\Longleftrightarrow} \quad \Phi \vdash \Psi \text{ and } \Psi \vdash \Phi \ .$$

We say that Φ and Ψ are *logically equivalent* if $\Phi \equiv \Psi$ holds. For instance, $\forall xRxx$ and $\forall yRyy$ are logically equivalent. There are logically equivalent formulas that look entirely different; in fact, it is in general *very difficult* to decide if two formulas are logically equivalent or not.

We can see (in fact, just by using that $\vdash$ is a preorder) that $\equiv$ is an *equivalence relation.* Next, we consider the set of all equivalence classes $[\Phi]$ of this equivalence relation. Let us denote this set by $\mathcal{F}/\equiv$. An element of $\mathcal{F}/\equiv$ is a set of the form $[\Phi]$ with Φ an arbitrary element of $\mathcal{F}$; here, $[\Phi] = \{\Psi \in \mathcal{F} : \Phi \equiv \Psi\}$. Intuitively, we "identify" two formulas when they are logically equivalent. This is very similar to modular arithmetic, in which, after a fixed modulus n is given, one *identifies* any two integers a and b , one pretends that they are equal, if $a \equiv b \pmod{n}$.

It is easy to see that the above move turns the preorder $(\mathcal{F}, \vdash)$ into a (partial) order. That is, we can define $[\Phi] \leq [\Psi]$ to mean that $\Phi \vdash \Psi$, and, under this definition, $(\mathcal{F}/\equiv, \leq)$ is an order. More is true. $(\mathcal{F}/\equiv, \leq)$ *is a Boolean algebra*; in fact, in a natural way, because we

have

$$[\Phi] \wedge [\Psi] = [\Phi \wedge \Psi] \ ;$$
$$[\Phi] \vee [\Psi] = [\Phi \vee \Psi] \ ;$$
$$-[\Phi] = [\neg\Phi]$$

the top element of $(\mathcal{F}/\equiv, \leq)$ equals $[\top]$;

the bottom element of $(\mathcal{F}/\equiv, \leq)$ equals $[\bot]$.

The Boolean algebra $(\mathcal{F}/\equiv, \leq)$ is called the *Lindenbaum-Tarski algebra of quantifier logic*. We will refer to it briefly as the *L-T-algebra*.

When we "do" Boolean algebra in the L-T-algebra, we do not write the symbols [,] for equivalence class; on the other hand, instead of $\leq$, we write $\vdash$, and instead of $=$, we write $\equiv$. For instance, we have, as a special case of a general fact in Boolean algebras, that

$$\Phi \vdash \Psi \iff \Phi \wedge \neg\Psi \equiv \bot \ ;$$

this is just the rule

$$x \leq y \iff x \wedge -y = \bot$$

that we know is true in any Boolean algebra.

However, the L-T-algebra has more structure than just the Boolean structure; this additional structure is related to the quantifiers. To explain this, we must explain *substitution*.

Given a formula Φ , and variables x and y , we can *substitute* y for x in Φ . Let us look at this in an example. In fact, we saw an example for this before. With the formula (5') in section 5.1:

$$\Phi(x) = \neg\mathbf{1}(x) \wedge \forall y \forall z (Pyzx \rightarrow (y=x \vee z=x)) \ , \qquad (3)$$

we also considered

$$\Phi(w) = \neg\mathbf{1}(w) \wedge \forall y \forall z (Pyzw \rightarrow (y=w \vee z=w)) \ .$$

Here, $\Phi(w)$ is obtained by substituting w for x in $\Phi(x)$. In general, we write $\Phi[y/x]$ for the result of substituting y for x in Φ (Φ may have more free variables than just x; it is also allowed that x is not a free variable in Φ at all, in which case $\Phi[y/x] = \Phi$). However, we make certain exclusions in the definition of substitution.

First of all, when forming $\Phi[y/x]$, we substitute y *only for the free occurrences of the variable* x. For instance, if, for some reason, we had

$$\Phi = \neg\mathbf{1}(x) \wedge \forall y \forall z (Pyzx \rightarrow (y=x \vee z=x)) \wedge \exists x(x=x),$$

(which uses a superfluous conjunct $\exists x(x=x)$, and is logically equivalent to the original Φ), then

$$\Phi[w/x] = \neg\mathbf{1}(w) \wedge \forall y \forall z (Pyzw \rightarrow (y=w \vee z=w)) \wedge \exists x(x=x);$$

note that we *did not* substitute for the bound occurrences of x. Secondly, and more importantly,

the substitution may not alter the binding pattern of the formula; it may not introduce new bound occurrences.

This is the same as to say that

if in Φ, x *occurs as a free variable in the scope of a quantifier* $\exists y$ *or* $\forall y$ *with the same variable* y *that we want to substitute for* x, *the substitution is not allowed;* $\Phi[y/x]$ *is not defined at all.*

For instance, consider the formula $\exists yRxy$. The substitution $(\exists yRxy)[y/x]$ is not defined; in $\exists yRxy$, the free variable x is in the scope of $\exists y$, which is the formula Rxy. To take a more complicated example, with Φ as in (3), $\Phi[y/x]$ is not defined: x is a free variable in the scope of $\forall y$, which is the formula $\forall z(Pyzx \rightarrow (y=x \vee z=x)$. By the way, this is the reason why we wrote $\Phi(w)$ rather than $\Phi(y)$ in our original use of Φ.

We can explain this exclusion by pointing out that an "illegal" substitution has a meaning that does not correspond to the intention of substitution. Let's look at the case of $\Phi = \exists yRxy$. For illustration, take the interpretation $U=\mathbb{N}$, and R = ordinary $<$. Φ says "there is y such

that $x<y$". When we do $(\exists yRxy)[z/x]$, we get $\exists yRzy$, expressing "there is y such that $z<y$", which is all right: just as the original formula $\exists yRxy$, $(\exists yRxy)[z/x]$ is also identically true in the given interpretation. Here, we used a *legal* substitution: z is not free in the scope of $\exists y$. However, if we did $(\exists yRxy)[y/x]$, which is *illegal* as we said above, the result wold be $\exists yRyy$, which means "there is y such that $y<y$", a *false* statement in the given interpretation, something that is not intended by the substitution.

We can also substitute an arbitrary term t for a variable x in a formula Φ; the result is written $\Phi[t/x]$. This is defined (*legal*) only if by the substitution

no variable y appearing in t gets into the scope of a quantifier $\forall y$ or $\exists y$ with the same variable y.

However, we never substitute anything *for* a term if that term is not a variable. Similarly, one cannot use "quantifiers" $\forall t$, $\exists t$ unless t is a variable.

We can now express, first in an abstract and succinct way, later in some more detail, the main fact about quantifiers in the L-T-algebra.

For any formula Φ, and any variable x, we have

$$\forall x\Phi \equiv \bigwedge_t \Phi[t/x],$$

and

$$\exists x\Phi \equiv \bigvee_t \Phi[t/x];$$

here, t ranges over all terms such that the substitution $\Phi[t/x]$ is *legal*; $\bigwedge$ means meet of the set of all formulas after it, $\bigvee$ means join.

Remember that the elements of the L-T-algebra are, really, equivalence classes $[\Phi]$; in fact, what we have is

$$[\forall x \Phi] \;=\; \bigwedge_{t} [\Phi[t/x]] \,,$$

and

$$[\exists x \Phi] \;=\; \bigvee_{t} [\Phi[t/x]] \;;$$

but the way we wrote these relations first is nicer.

Now, we will write these relations in a more explicit way:

Rule of Universal Specification (US):

$$\forall x \Phi \;\vdash\; \Phi[t/x] \;;$$

Rule of Universal Generalization (UG):

if $\Psi \vdash \Phi$, and x is not free in Ψ, then $\Psi \vdash \forall x \Phi$:

$$\frac{\Psi \;\vdash\; \Phi}{\Psi \;\vdash\; \forall x \Phi} \qquad \text{provided } x \text{ is not free in } \Psi \,.$$

Rule of Existential Generalization (EG):

$$\Phi[t/x] \;\vdash\; \exists x \Phi \;;$$

Rule of Existential Specification (ES):

if $\Phi \vdash \Psi$, and x is not free in Ψ, then $\exists x \Phi \vdash \Psi$:

$$\frac{\Phi \;\vdash\; \Psi}{\exists x \Phi \;\vdash\; \Psi} \qquad \text{provided } x \text{ is not free in } \Psi \,.$$

It is understood that only legal substitutions can be taken. t denotes an arbitary term.

(The observant student will see that the second, more detailed, statement of, say, the rule for the universal quantifier is not obviously equivalent to what we had in the succinct form. In fact, universal generalization would, if translated directly from the succinct statement, look like this:

$$\frac{\Psi \vdash \Phi[t/x] \quad \text{for all} \quad t}{\Psi \vdash \forall x \Phi} \text{ (such that the substitution is legal)}$$

It turns out that this is an equivalent formulation, if we also take into account Universal Specification).

The main result is that that the above rules for the quantifiers completely describe the L-T-algebra:

if $\Phi \vdash \Psi$, *then this fact can be deduced using Boolean algebra, and the four rules for quantifiers*,

-- provided the formulas involved do not contain = (equality); if they do, additional "axioms of equality" are needed.

This fact is a deep theorem, whose proof cannot be indicated here; it is Kurt Gödel's *completeness theorem for first order logic.*

Let us look at some examples for using the above rules, and Boolean algebra.

1. $\forall x \Phi \equiv \forall y (\Phi[y/x])$;

here, $\Phi[y/x]$ is assumed to be a legal substitution, and y is assumed to be *not free or bound in* $\forall x \Phi$.

(Example for 1.:

(i) $\forall x Rxz \equiv \forall y Ryz$;

but we are *not* asserting that $\forall x Rxy \equiv \forall y Ryy$, which is actually false; in this case, $\Phi = Ryz$, the second proviso, " y is not free in Φ ", is violated. Also:

(ii) $\forall x \exists u (Rxz \wedge Rux) \equiv \forall y \exists u (Ryz \wedge Ruy)$.)

Proof of 1. By US,

$$\forall x \Phi \vdash \Phi[y/x] \ ;$$

therefore, since y is not free in $\forall x \Phi(x)$, by UG:

$$\forall x \Phi \vdash \forall y (\Phi[y/x]) \ . \qquad (4)$$

Also, we have that

$$\Phi[y/x][x/y] = \Phi : \qquad (10)$$

this is because y is not bound in $\Phi[y/x]$, the substitution $\Phi[y/x][x/y]$ is legal, and all it does is to restore x where we had x before in Φ . Therefore, applying to $\Phi[y/x]$ what we already proved for Φ , with the roles of x and y exchanged, we get

$$\forall y (\Phi[y/x]) \vdash \forall x (\Phi[y/x][x/y])$$

which is, by (10), the same as

$$\forall y (\Phi[y/x]) \vdash \forall x \Phi \qquad (11)$$

By (4) and (6), we have

$$\forall x \Phi \equiv \forall y (\Phi[y/x])$$

as desired.

2. $\forall x (\Lambda \vee \Gamma) \equiv \Lambda \vee \forall x \Gamma$ provided x is not free in Λ

(Example for 2.: $\forall x (\exists y Ryz \vee Rxz) \equiv \exists y Ryz \vee \forall x Rxz$.)

Proof. 1 $\Gamma \vdash \Lambda \vee \Gamma$ by Boolean algebra;

2 $\forall x\Gamma \vdash \Gamma$ by US ($\Gamma[x/x] = \Gamma$);

3 $\forall x\Gamma \vdash \Lambda \vee \Gamma$ by 1 and 2 ($\vdash$ is transitive);

4 $\forall x\Gamma \vdash \forall x(\Lambda \vee \Gamma)$ by UG applied to 3 (x is not free in $\forall x\Gamma$);

5 $\Lambda \vdash \Lambda \vee \Gamma$ by Boolean algebra;

6 $\Lambda \vdash \forall x(\Lambda \vee \Gamma)$ by UG (x is not free in Λ , by assumption);

7 $\Lambda \vee \forall x\Gamma \vdash \forall x(\Lambda \vee \Gamma)$ by 4 and 6 , and Boolean algebra ($x \leq z$ and $y \leq z$ imply $x \vee y \leq z$);

8 $\forall x(\Lambda \vee \Gamma) \vdash \Lambda \vee \Gamma$ by US;

9 $\forall x(\Lambda \vee \Gamma) \wedge \neg\Lambda \vdash \Gamma$ by 8 and Boolean algebra: $x \leq y \vee z$ implies $x \wedge -z \leq y$: $x \leq y \vee z$ implies $x \wedge -z \leq (y \vee z) \wedge -z$, and $(y \vee z) \wedge -z = (y \wedge -z) \vee (z \wedge -z) = (y \wedge -z) \vee \top = y \wedge -z$; thus, $x \wedge -z \leq y \wedge -z \leq y$;

10 $\forall x(\Lambda \vee \Gamma) \wedge \neg\Lambda \vdash \forall x\Gamma$ by 9 and UG: x is not free in the premiss, since it is not free in $\forall x(\Lambda \vee \Gamma)$, and it is not free in $\neg\Lambda$ by assumption;

11 $\forall x(\Lambda \vee \Gamma) \vdash \Lambda \vee \forall x\Gamma$ by 10 and Boolean algebra: $x \wedge -z \leq y$ implies $x \leq z \vee y$;

12 $\forall x(\Lambda \vee \Gamma) \equiv \Lambda \vee \forall x\Gamma$ by 7 and 11.

3. $\exists x\Phi \equiv \neg\forall x\neg\Phi$

Proof. We prove that

$$\exists x\Phi \vee \forall x\neg\Phi \equiv \top \qquad (12)$$

and that

$$\exists x\Phi \wedge \forall x\neg\Phi \equiv \bot . \qquad (13)$$

By the definition of $\neg$ as complement in a Boolean algebra, this will mean that $\neg\forall x\neg\Phi \equiv \exists x\Phi$.

To see that $\exists x\Phi \vee \forall x\neg\Phi \equiv \top$, note that x is not free in $\exists x\Phi$; therefore, we may apply **2.** with $\Lambda = \exists x\Phi$, to get that $\exists x\Phi \vee \forall x\neg\Phi \equiv \forall x(\exists x\Phi \vee \neg\Phi)$. Now, $\Phi \vdash \exists x\Phi$ by ES. But, in any Boolean algebra, $x \leq y$ implies $y \vee -x = \top$. Applying this to Φ as x and $\exists x\Phi$ as y , we get $\exists x\Phi \vee \neg\Phi \equiv \top$. But also, $\forall x\top \equiv \top$, since UG can be applied to $\top \vdash \top$ to get $\top \vdash \forall x\top$, which is $\forall x\top \equiv \top$. Therefore,

$$\exists x\Phi \vee \forall x\neg\Phi \equiv \forall x(\exists x\Phi \vee \neg\Phi) \equiv \forall x\mathsf{T} \equiv \mathsf{T} .$$

This proves (12).

To show (13), note that $\forall x\neg\Phi \vdash \neg\Phi$, thus $\Phi\wedge\forall x\neg\Phi \vdash \Phi\wedge\neg\Phi \equiv \bot$, $\Phi\wedge\forall x\neg\Phi \vdash \bot$. This implies $\Phi \vdash \neg\forall x\neg\Phi$, since in a Boolean algebra, $x\wedge y=\bot$ iff $x\leq -y$. By EG, since x is not free in $\neg\forall x\neg\Phi$, we get that $\exists x\Phi \vdash \neg\forall x\neg\Phi$. By the same Boolean principle backwards, we get that $\exists x\Phi \wedge \neg\forall x\neg\Phi \vdash \bot$, what we wanted.

In the above deductions the steps taken were not suggested by plausible rules. We have found deductions of the entailments in question, but there was no explanation given *how* these deductions were found. There is a deep reason for this. As a matter of fact, there is a fundamental theoretical limitation here: *there cannot be any* completely algorithmic method of finding deductions to entailments. This is Church's Theorem on the undecidability of logical validity, a fundamental limitative result of Mathematical Logic.

The examples shown above are meant as illustrations of what we get when our search for a deduction is successful, but not of a method of finding a deduction.

On the other hand, there are various ways of organizing the search for a deduction that help in many practically important cases to find one. These ways belong to the subject area of Computer Science called Automatic Theorem Proving.