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

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

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:

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

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

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:

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

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*:

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

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

For the first specification, $\Phi$ , we have

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

for the second, $\Psi$ ,

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

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 $\Phi$ , 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 $\Phi$ , will *answer* the spec, or, will be *a solution* of the spec, if

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

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 $\mathrm{gcd}(a, b)$ 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 $\mathrm{gcd}(a, b)$ is the so-called *Euclidean algorithm* that we are now going to describe in essence.

For now, variables $a, b, .., x, y,$ 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, $\mathrm{gcd}(a, b) = \mathrm{gcd}(c, d)$ .

(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)

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

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 $a\,\mathrm{mod}\,b$ , $\mathrm{gcd}(a,b)$ on the universe $\mathbb{N}$ with primitives $0$ , $+$ , $\cdot$ and $<$ :

$$\begin{array}{lll}
\forall a\forall b\forall r[ & (a\,\mathrm{mod}\,b=r \longleftrightarrow ((0<b \wedge \exists q(a=q\cdot b+r) \wedge r<b) & (3.1)\\
 & \vee (0=b \wedge r=a))) \quad ] & (3.2)\\
\wedge & & \\
\forall a\ [ & \mathrm{gcd}(a,0)=a \quad ] & (3.3)\\
\wedge & & \\
\forall a\forall b[ & \mathrm{gcd}(a,b)=\mathrm{gcd}(b,a\,\mathrm{mod}\,b) \quad ] & (3.4)
\end{array}$$

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

| $a$ | $b$ | $r=a \bmod b$ | $q=[a/b]$ |
|---|---|---|---|
| 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 $\text{gcd}(180,0) = 180.$ Therefore, since the gcd is constant across the table, we conclude that

$$\text{gcd}(6840, 99900) = 180.$$

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

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

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:

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

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.

$\Lambda$ = 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 $s = \langle k_0, k_1, ..., k_{n-1}\rangle$ . The empty string has length equal to $0$ .

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

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

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 $i$*th 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 $i \geq \mathrm{lh}(s)$ .

$\langle k \rangle$ = the string $s$ for which $\mathrm{lh}(s)=1$ , 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

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

as long as $s(i\text{-}2) \neq 0$ . 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 $\mathrm{lh}(s)=N+1$ . The required $\mathrm{gcd}(a, b)$ is $s(N\text{-}1)$ .

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, $N=8$ , $\mathrm{lh}(s) = N+1 = 9$ ; and

$$\begin{aligned}
&s(0) = 6840\\
&s(1) = 99900\\
&s(2) = 6840\\
&s(3) = 4140\\
&s(4) = 2700\\
&s(5) = 1440\\
&s(6) = 1260\\
&s(7) = 180\\
&s(8) = 0
\end{aligned}$$

$$\mathrm{gcd}(s(0), s(1)) = s(8\text{-}1) = 180\ .$$

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

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

$\mathtt{EUCLID}(a, b)$ , 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.

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

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: $a \,\mathtt{mod}\, b$ , $\mathtt{EUCLID}(a, b)$ , and $\mathtt{gcd}(a, b)$ ; the first and third are integer-valued, the second is string-valued.