Introduction

Motivation

During decades, classical introductory textbooks presenting logic for undergraduate computer science students have been focused on the syntax and semantics of propositional and predicate calculi and related computational properties such as decidability and undecidability of logical questions. This kind of presentations, when given with the necessary formal details, are of great interest from the mathematical and computational points of view and conform a *sin equa non* basis for computer students interested in theory of computing as well as in the development of formal methods for dealing with robust software and hardware.

In addition to the unquestionable theoretical importance of these classical lines of presentation of the fundaments of logic for computer science, nowadays, it is of essential relevance for computer engineers and scientists the precise understanding and mastering of the mathematical aspects involved in several deductive methods that are implemented and available in well-known modern proof assistants and deductive frameworks such as Isabelle, Coq, HOL, ACL2, PVS, among others. Only through the careful and precise use of this kind of computational tools, it is possible to assure the *mathematical correctness* of software and hardware that is necessary in order to guarantee the desired robustness of computer products.

Today, it is accepted that the software and hardware that is being applied in critical systems such as (sea, earth, air and space) human-guided or autonomous navigation systems, automotive hardware, medical systems and others, should have *mathematical certificates* of quality. But also it is clear for the computer science community, that in all other areas in which computer science is being applied, this kind of formal verification is of fundamental

importance (to try) to eliminate any possibility of harming or even offering a disservice to any user. These areas include financial, scheduling, administrative systems, games, among others; areas of application of computational systems in which users might be negatively affected by computational bugs. It is unnecessary to stress here, that nowadays, the society, that is formed by the users of all these so called *non critical* systems, is ready to complain in a very organized manner against any detected bug; and, the computer engineers and scientists, who are involved in these developments, will be identified as those directly responsible. Indeed, nowadays, nobody accepts the old standard excuse given some years ago: "sorry, the issue was caused by an error of the computer".

The current presentation of logic for computer science and engineering focuses on the mathematical aspects subjacent to the deductive techniques applied to build proofs in both propositional and predicate logic. Derivations or proofs will be initially presented in the style of natural deduction and subsequently in the style of Gentzen's sequent calculus. Both these styles of deduction are implemented in several proof assistants and knowing how deductions are mathematically justified will be of great importance for the application of these tools in a very professional and effective manner. The correspondence between both styles of deduction will be discussed and simple computational applications in the PVS proof assistant will be given.

Examples

In order to explain the necessity of knowledge on formal deductive analysis and technologies for the adequate development of robust computer tools, we consider a classical piece of mathematics and computation that is implemented in the arithmetic libraries of the majority of modern computer languages. The problem is to compute the *greatest common divisor*, gcd for brevity, of pairs of integers that are not simultaneously zero.

In first place, the mathematical object that one wants to capture by an implementation is defined as below.

Definition 1 (Greatest Common Divisor)**.** *The greatest common divisor of two integer num-*

bers i and j, that are not simultaneously equal to zero, is the greatest number k that divides both i and j.

Several observations are necessary before proceeding with the implementation of a simple program that effectively might correspond to the Definition 1. For instance, it is necessary to observe that the *domain* of the defined function gcd is $\mathbb{Z} \times \mathbb{Z} \setminus \{(0,0)\}$, while its *range* is $\mathbb{N}$. **Why?**

The first naive approach to implement gcd could be an *imperative* and *recursive algorithm* that checks, in a decreasing order, whether natural numbers divide both the integers i and j. The starting point of this algorithm is the natural number given by the smallest of the absolute values of the inputs: $min\{|i|, |j|\}$. But more sophisticated mathematical knowledge can be used in order to obtain more efficient solutions. For instance, one could apply a classical result that is attributed to Euclid and was developed more than two millennia ago.

Theorem 1 (Euclid 320-275 BC)**.** $\forall m > 0, n \geq 0, \gcd(m,n)$ *equals to* $\gcd(m, n \ MOD \ m)$*, if* $n > 0$*,* m *otherwise; where* $n \ MOD \ m$ *denotes the remaining of the integer division of* n *by* m*.*

Observe that this result only will provide us a partial solution because, in this theorem, the domain is restricted to $(\mathbb{N} \setminus \{0\}) \times \mathbb{N}$, that is $\mathbb{N}^+ \times \mathbb{N}$, instead $\mathbb{Z} \times \mathbb{Z} \setminus \{(0,0)\}$, that is the domain of the mathematical object that we want to capture. Despite this drawback, we will proceed treating to capture a restricted version of the mathematical object gcd restricting its domain. In the end, $\gcd(i,j) = \gcd(|i|, |j|)$. **Why?**

A first easy observation is that Euclid's theorem does not provide any progress when $m > n$, because in this case n MOD $m = n$. Thus, the theorem is of computational interest when $m \leq n$. The key point, in order to apply Euclid's theorem, is to observe that the remainder of the integer division between naturals n and m, n MOD m can be computed decreasing n by m, as many times as possible, whenever the result of the subtraction remains greater than or equal to m. This is done until a natural number smaller than m is reached. For instance, 27 MOD 5 equals to $((((27-5)-5)-5)-5)-5 = 2$. This procedure is possible in general, since for any integer k, $\gcd(m,n) = \gcd(m, n + k\,m)$. **Why?**

Once the previously suggested procedure stops, one will have as first argument m, and as second argument a natural number, say $n - km$, that is less than m. The procedure can be repeated if one interchanges these arguments, since in general $\gcd(i, j) = \gcd(j, i)$. **Why?**

Following the previous observations, a first *attempt* to compute gcd restricted to the domain $\mathbb{N}^+ \times \mathbb{N}$ may be given by the *procedure* $\gcd_1$ presented in Algorithm 1.

procedure $\gcd_1(m : \mathbb{N}^+, n : \mathbb{N}) : \mathbb{N}$;
if $m > n$ **then**
 | $\gcd_1(n, m)$
else
 | $\gcd_1(m, n - m)$
end

Algorithm 1: First attempt to specify gcd: procedure $\mathbf{gcd_1}$

The careful reader will notice that this first attempt fails because the restriction of the domain is not preserved by this specification; that is, the first argument of this function may become equal to zero. For instance, for inputs 6 and 4 infinite recursive calls are generated by $\gcd_1$:

$$\gcd_1(4, 6) \to \gcd_1(4, 2) \to \gcd_1(2, 4) \to \gcd_1(2, 2) \to \gcd_1(2, 0) \to \gcd_1(\underline{0}, 2) \to \cdots$$

In the end $\gcd_1$ fails because it is specified in such a manner that it never returns a natural number as answer, but instead recursive calls to $\gcd_1$.

Formally, the problem can be detected when trying to prove that the "function" specified as $\gcd_1$ is *well-defined*; that is, to prove that the function is defined for all possible inputs of its domain. The attempt to prove well-definedness of $\gcd_1$ might be by nested induction on the first and second parameters of $\gcd_1$ as sketched below.

Induction Basis: Case $m = 1$. Notice that we start the induction from $m = 1$ since the type of m is $\mathbb{N}^+$. Trying to conclude by induction on n, two cases are to be considered: either $1 > n$ or $1 \leq n$. The case $1 > n$ gives rise to the recursive call $\gcd_1(0, 1)$ that has *ill-typed* arguments, since the first argument does not belong to the set $\mathbb{N}^+$ of positive naturals. The case $1 \leq n$ gives rise to the recursive call $\gcd_1(1, n - 1)$, that is correctly typed since $n - 1 \geq 0$. But the attempt to conclude by induction on n fails.

Induction Step: Case $m > 1$.

Induction Basis: Case $n = 0$. $\gcd_1(m, 0) = \gcd_1(0, m)$ which is undefined, according to the analysis in the induction basis for m. To correct this problem, one needs to specify $\gcd(m, 0) = m$.

Induction Step: Case $n > 0$. This is done by analysis of cases:

Case $m > n$, $\gcd_1(m, n) = \gcd_1(n, m)$, that is well-defined by induction hypothesis, since $n < m$; that is, the first argument of $\gcd_1$ decreases.

Case $m \leq n$, $\gcd_1(m, n) = \gcd_1(m, n - m)$, that is well-defined by induction hypothesis, since the first argument remains the same and the second one decreases. Notice that in fact $n - m < n$, since in this step of the inductive proof m is assumed to be greater than zero. In addition, notice that $n - m$ has the correct type ($\mathbb{N}$), since $n - m \geq 0$.

Despite $\gcd_1$ is undefined for $m = 0$, one has that an eventual recursive call of the form $\gcd_1(0, n)$ produces a recursive call of the form $\gcd_1(0, n - 0)$, that as observed before might generate an infinite loop! Thus, a correct procedure should be specified in such a way that it takes care of this abuse on the restricted domain of the function gcd (restricted to the domain $\mathbb{N}^+ \times \mathbb{N}$) avoiding any possible recursive call with ill-typed arguments.

In the end this attempt to prove well-definedness of the procedure $\gcd_1$ fails, but it provides valuable pieces of information that are useful to correct the procedure. Several elements of logical deduction were applied in the analysis, among them:

- Application of the principle of mathematical induction;
- Contradictory argumentation (undefined arguments are defined by the specified function);

- Analysis of cases: $\gcd_1$ is well-defined for positive values of m and n because it is proved well-defined for a *complete* set of cases for m and n; namely, the case in which $m > n$ and the contrary case, that is the case in which $m > n$ does not hold, or equivalently, the case in which $m \leq n$.

Some of the mathematical aspects of this kind of logical analysis will be made precise through the next chapters of this book.

From the corrections made when attempting to prove well-definedness of $\gcd_1$ a new specification of the function gcd, called $\gcd_2$, is proposed in Algorithm 2.

```
procedure gcd_2(m : ℕ⁺, n : ℕ) : ℕ ;
if n = 0 then
    m
else
    if m > n then
        gcd_2(n, m)
    else
        gcd_2(m, n − m)
    end
end
```

Algorithm 2: Second attempt to specify gcd: procedure $\mathbf{gcd_2}$

A thoughtful revision of the attempt to proof well-definedness of the procedure $\gcd_1$ will provide a verification that this property is owned by the new specification $\gcd_2$. As before, the proof follows a nested induction on the first and second parameters of $\gcd_2$.

Induction Basis: Case $m = 1$. $\gcd_2(1, n)$ gives to cases according to whether $1 > n$ or $1 \leq n$, which are treated by nested induction on n.

Induction Basis: Case $n = 0$. Since $1 > n$, the answer is 1.

Induction Step: Case $n > 0$. This is the case in which $1 \leq n$, that gives rise to the recursive call $\gcd_2(1, n-1)$, that is well-defined by induction hypothesis for n.

Induction Step: Case $m > 1$.

Induction Basis: Case $n = 0$. $\gcd_2(m, 0) = m$, which is correct.

Induction Step: Case $n > 0$. This is done by analysis of cases:

Case $m > n$, $\gcd_2(m, n) = \gcd_2(n, m)$, that is well-defined by induction hypothesis, since $n < m$; that is, the first argument of $\gcd_2$ decreases. Also, observe that since it is supposed that $n > 0$, this interchange of the arguments respects the type restriction on the parameters of $\gcd_2$.

Case $m \leq n$, $\gcd_2(m, n) = \gcd_2(m, n - m)$, that is well-defined by induction hypothesis, since the first argument remains the same and the second one decreases. Notice that in fact $n - m < n$, since in this step of the inductive proof m is supposed to be greater than zero. Also, since $m \leq n$, $n - m \geq 0$, thus $n - m$ has the correct type $\mathbb{N}$.

The moral of this example is that the correction of the specification, from the point of view of well-definedness of the specified function, relies on the proof that it is defined for all possible inputs. Therefore, in order to obtain correct implementations, it is essential to know how to develop proofs formally. Of course, it is much more complex to prove that in fact, $\gcd_2$ for inputs $(m, n) \in \mathbb{N}^+ \times \mathbb{N}$, correctly computes $\gcd(m, n)$.

Once well-definedness of $\gcd_2$ is guaranteed, becomes interesting proving that indeed this specification computes correctly the function gcd as given in the definition. For doing this one will require the application of Euclid's theorem as well as previously properties of gcd (that were highlighted with questions "Why?'s"):

1. For all integers i, j that are not simultaneously equal to zero, that is for all $(i, j) \in \mathbb{Z} \times \mathbb{Z} \setminus \{(0, 0)\}$, $\gcd(i, j) \in \mathbb{N}$;

2. For all $(i, j) \in \mathbb{Z} \times \mathbb{Z} \setminus \{(0, 0)\}$, $\gcd(i, j) = \gcd(|i|, |j|)$;

3. For all $(m, n) \in \mathbb{N}^+ \times \mathbb{N}$, and $k \in \mathbb{Z}$, $\gcd(m, n) = \gcd(m, n + km)$;

4. For all $(i, j) \in \mathbb{Z} \times \mathbb{Z} \setminus \{(0, 0)\}$, $\gcd(i, j) = \gcd(j, i)$.

Exercise 1. *Prove these four properties.*

Notice that the third property is the one that justifies the second nested **else** case in the specification $\gcd_2$, since for the case in which $k = -1$, one has

$$\gcd(m, n) = \gcd(m, n - m)$$

Also, this justifies as well Euclid's theorem.

Exercise 2. (*) *Design an algorithm for computing the function* gcd *in its whole domain:* $\mathbb{Z} \times \mathbb{Z} \setminus \{(0, 0)\}$. *Prove that your algorithm is well-defined and that is correct.*
Hint: assuming the four properties and Euclid's theorem prove that $\gcd_2$ *is algebraically correct in the following sense:*

For all $(i, j) \in \mathbb{Z}^* \times \mathbb{Z}$, *where* $\mathbb{Z}^*$ *denotes the non zero integers,* $\gcd_2(|i|, |j|)$ *computes a number* $k \in \mathbb{N}$, *such that*

- k *divides* i,
- k *divides* j *and*
- *For all* $p \in \mathbb{Z}$ *such that* p *divides both* i *and* j, *it holds that* $p \leq k$.

The "**(*)**" in this exercise means that it is of a reasonably high level of complexity or that will require additional knowledge. But the readers do not need to worry if they cannot answer this exercise at this point, since the objective of the course notes is to bring the required theoretical background and minimum practice to be able to formally build mathematical certificates of computational objects, such as the inquired in this exercise.

Structure of the book

The technology of computational formalization and verification is a mature area both of computer science and mathematical logic. This technology involves a great deal of precise

mathematical knowledge about logical deduction and proof theory. These areas are originally placed in the setting of formalisms of mathematics and have been studied in detail since the earlier years of the last century by well-known mathematicians, from whom perhaps the most famous were David Hilbert, Luitzen Brouwer, Kurt Gödel, Alan Turing and Alonso Church, but since then, other researchers have provided well-known related results in computer science that are very useful and have been inspiring the development of proof assistants and formal methods tools (e.g., Gerhard Gentzen, Haskell Curry, Robert Floyd, Corrado Böhm, Robin Milner, Nicolaas de Bruijn, among others).

In this book we will focus on the mathematical technology of logical deduction for the most elementary logics that are the propositional logic and the logic of predicates. The focus will be on the calculi of deduction for these two logical systems according to two styles of deduction; namely, natural deduction and sequent calculus, both deductive systems are contributions of the German mathematician Gerhard Gentzen. The motivation for restricting our attention to these two deduction styles and these two logical systems is that they are in the basis of all modern proof assistants.

In the first Chapter, we will present the *propositional logic* and its calculus in the style of *natural deduction*. We will present the syntax and semantics of this logical system and then we will prove that the deductive calculus in this natural style is *correct* and *complete*. *Correctness*, is inherent to well specified computer systems, in general, and in this setting it means that the deductive calculus deduces correct mathematical conclusions. *Completeness* means that all correct mathematical conclusions in the setting of this logical system can be deduced by the deductive calculus.

In the second Chapter, the propositional calculus is enriched with first-order variables which can be existentially and universally quantified by giving rise to the *logic of predicates* or first-order logic. This logical system provides a much more elaborated and expressive language, that corresponds to the basic logical language applied in most computational environments. To this end, the natural deductive calculus of the propositional logic will be enriched with rules for dealing with quantifiers and variables and, as for the propositional calculus, *correctness* and *completeness* will be considered too.