From Proofs to Programs

Andreas Klappenecker

Primes. Let $\mathbf{N} = \{1, 2, 3, \dots\}$ be the set of natural numbers. We say that a natural number b **divides** a natural number a if and only if there exists a natural number c such that $a = bc$. A natural number p greater than 1 is called **prime** if and only if 1 and p are the only natural numbers dividing p. A natural number $n > 1$ that is not prime is called **composite**.

Our goal is to write a program that allows us to test whether a given natural number n greater than 1 is a prime. We will choose the functional programming language Haskell for the implementation.

Let us denote by $\mathbf{N}_2 = \mathbf{N} \setminus \{1\}$ the set of natural numbers greater than 1. Let $\ell \colon \mathbf{N_2} \to \mathbf{N_2}$ denote the function such that $\ell(n)$ is the least natural number greater than 1 dividing n. The function ℓ is well-defined, since there always exist a divisor of n that is greater than 1, namely n itself; thus, there must exist a smallest divisor $k = \ell(n)$ of n in the range $2 \leq k \leq n$.

Let us collect some simple observations about the function ℓ.

Proposition 1. *The value $\ell(n)$ is a prime number for all n in $\mathbf{N}_2$.*

Proof. Seeking a contradiction, we assume that there exists a number n in $\mathbf{N}_2$ such that $\ell(n)$ is not a prime. This implies that there exist numbers a and b in $\mathbf{N}_2$ such that $\ell(n) = ab$. However, this would mean that a divides n and $a < \ell(n)$, contradicting the fact that $\ell(n)$ is the smallest natural number greater than 1 dividing n. Therefore, $\ell(n)$ must be prime for all n in $\mathbf{N}_2$, as claimed. □

Therefore, if n is composite, then $\ell(n)$ must be smaller than n. In fact, the next proposition shows that $\ell(n)$ is *considerably* smaller than n when n is composite.

Proposition 2. *If n in $\mathbf{N}_2$ is a composite number, then $\ell(n)^2 \leq n$.*

Proof. Let $p = \ell(n)$. Since n is not prime, we have $n = pa$ for some a in the range $p \leq a < n$. Therefore, $\ell(n)^2 = p^2 \leq pa = n$, which proves the claim. □

The last observation alone suffices to design a simple primality test. The basic idea is to test whether any k in the range $2 \leq k \leq \sqrt{n}$ divides n. If we can find such a divisor k then n is evidently not a prime number; otherwise, n must be a prime number. Indeed, the contrapositive of Proposition 2 states that if $k^2 = \ell(n)^2 > n$, then n must be a prime number.

Haskell. We choose the programming language Haskell for the implementation. This is a functional programming language named after the logician Haskell Curry. You can find a copy of the Haskell interpreter `hugs` at the web site `www.haskell.org`. After starting hugs, you can load a program file by typing

```
:l myfilename.hs
```

where `myfilename.hs` is the filename of your program.

Implementation. We present the program in the literate programming style introduced by Donald Knuth. Our program will be extracted from this text into the file `primes.hs`; this is indicated by the first label in angled brackets. The program contained in ⟨*primes.hs*⟩ is structured into four different parts, as follows:

2a ⟨*primes.hs* 2a⟩≡

⟨*primality test* 3a⟩
⟨*least divisor function* 3b⟩
⟨*least divisor in a range* 3c⟩
⟨*division test* 2b⟩

The first part ⟨*primality test*⟩ of the program contains the primality test, and the other three parts contain the functions needed to implement this function. You can think of the terms in angled brackets as macros that will be expanded somewhere in this document.

Since you are probably not familiar with Haskell, we start by explaining the simplest function, which is contained in ⟨*division test*⟩. Recall that we can express a natural number n as a multiple of a natural number d plus a remainder r,

$$n = qd + r,$$

where $0 \leq r < d$. We call q the quotient and r the remainder. Haskell has a built-in function to calculate r, called `rem`. Haskell uses prefix notation for functions. This means that `rem n d` calculates the remainder r when n is divided by d. If the remainder is 0, then we say that d divides n. In Haskell, the test for equality is denoted by `==`. The single equality `=` is also used and should be read as "is by definition equal to". Our division test is now defined as follows:

2b ⟨*division test* 2b⟩≡ (2a)

```
divides d n = rem n d == 0
```

Here, `divides` is a function that takes two arguments, `d` and `n`. It returns `True` if and only if d divides n. The function `divides d n` is implemented by checking whether the remainder of n divided by d is 0.

Our main function will be called `prime` and it takes one argument `n`. Thus, `prime n` is supposed to return `True` if n is a prime number, otherwise it should return `False`. It is assumed that the input n is an integer greater than 1. The primality test takes advantage of the function ℓ that we have defined above. Recall that $n \geq 2$ is a prime number if and only if $n = \ell(n)$. In our Haskell

program, the function ℓ will be called `ld`. Thus, `prime n` simply checks whether `ld n` is equal to `n`. In other words, the primality test is implemented as follows:

3a ⟨*primality test* 3a⟩≡ (2a)
```
prime n = ld n == n
```

The function ℓ, or `ld` is Haskell, will be implemented in terms of a function `ldf` that takes two arguments, `k` and `n`, and returns the least divisor of n in the range from k to n, assuming that no divisor in the range $[2, k-1] = \{2, \ldots, k-1\}$ has been found.

3b ⟨*least divisor function* 3b⟩≡ (2a)
```
ld n = ldf 2 n
```

All the work will be done in the function `ldf`. It is clear that `ldf k n` should return the value k if k divides n. If k^2 exceeds n, then n does not have any divisors m in the range $k \leq m \leq \sqrt{n}$, so it should return n. Otherwise, `ldf k n` is the same as `ldf (k+1) n`. In other words, we would like to define `ldf` using three different cases.

Haskell allows us to do such case-by-case definitions of functions with the help of so-called guarded equations. One writes

```
myfunction arg1 arg2 | mycondition = mydefinition
```

This means that the function `myfunction` has two arguments `arg1` and `arg2` and is defined in terms of `mydefinition` given that the condition `mycondition` holds. The condition is called a *guarded equation* in Haskell.

Our function `ldf k n` that returns the smallest divisor m of n in the range $k \leq m \leq n$ is defined as follows:

3c ⟨*least divisor in a range* 3c⟩≡ (2a)
```
ldf k n | divides k n = k
        | k^2 > n = n
        | otherwise = ldf (k+1) n
```

You will immediately recognize the three different cases in the implementation of the function `ldf`. The last case is a recursive call to `ldf`. This is called a *tail recursion*, and it is a simple way to implement the equivalent of a loop in a functional programming language.

This completes our implementation of the primality test. If you load the file `primes.hs` into Haskell and type `prime 1111` then the Haskell interpreter will respond with `False`, since 11 is the least divisor of 1111.

Literate Programming. Even though we have not specified the literate programming macros in order, the extracted program follows precisely the outline given above. The content of the file `primes.hs` is:

```
prime n = ld n == n
ld n = ldf 2 n
ldf k n | divides k n = k
        | k^2 > n = n
        | otherwise = ldf (k+1) n
divides d n = rem n d == 0
```

This document was created with noweb by Norman Ramsey and LaTeX by Donald Knuth and Leslie Lamport. You should get familiar with LaTeX, since it is the best way to produce documents that are nicely typeset.

The literate programming paradigm is that programs should be written in a form that is a pleasure to read for humans, rather than in a form that is easily understood by a computer. Of course, you do not need literate programming to understand a tiny program with 6 lines of code. However, there is little hope to understand a scarcely documented program consisting of thousands of lines of code. In my experience, I insert more explanations when using literate programming, simply because I want to make the LaTeX document look better.

Norman Ramsey's noweb scripts are a simple way to use literate programming for any programming language. The documentation for noweb consists of a single page, so you can easily learn it.

One of the most striking examples of literate programming is illustrated in [D.E. Knuth, *Computers & Typesetting, Volume B: TeX: The Program*, Addison-Wesley, Reading, MA, 1986, xviii+600pp.]. This book contains the full implementation of the typesetting system TeX, on which LaTeX is based. Knuth offers $\$327.68=2^{15}¢$ for every error found in this program. Talk about confidence!

You can learn more about Haskell and its usage in discrete mathematics in [K. Doerts, J. van Eijck, *The Haskell Road to Logics, Maths, and Programming*, King's College Publications, London, 2004]. I have taken the example discussed here from the first chapter of this book. You can find more details on Haskell there.

Exercise 1. Download and install `hugs` on your computer.

Exercise 2. Experiment with the program `primes.hs`. Modify it so that the function arguments are of type integer. Make it robust so that it reports an error if the integer provided is less than 2.

Exercise 3. Use Proposition 1 and design a Haskell program to factor a natural number into primes numbers. The program should return a list of prime numbers such that their product is equal to the input.