IP Lecture 6: Printing Numbers in Decimal

Joe Pitt-Francis

—with thanks to Mike Spivey & Gavin Lowe—

Problem statement

Given a positive integer `t`, we want to calculate its decimal digits, and store them in an array `d[0..n)`, with the least significant digit in `d(0)`.

For example, given `t = 12345`, we will set `n=5` and

```
d(0) = 5,  d(1) = 4,  d(2) = 3,  d(3) = 2,  d(4) = 1
```

Let's write $\mathtt{t}@i$ for the digit of `t` that should be put into $\mathtt{d}(i)$:

$$\mathtt{t}@i = (\mathtt{t} \text{ div } 10^i) \text{ mod } 10$$

Our correctness condition will be that we end up with the correct values:

$$\textbf{pre:} \quad \mathtt{t} > 0$$

$$\textbf{post:} \quad (\forall i \in [0..\mathtt{n}) \bullet \mathtt{d}(i) = \mathtt{t}@i) \wedge \mathtt{t} < 10^{\mathtt{n}}$$

First program

Our first program will calculate the digits from right-to-left.

Looking at the correctness condition, it seems sensible to have an invariant including

$$I_1 \mathrel{\widehat{=}} \forall i \in [0..\mathtt{n}) \bullet \mathtt{d}(i) = \mathtt{t}@i$$

i.e., all the digits calculated so far are correct.

Since `t` is an `Int`, $\mathtt{t} < 2^{31} < 10^{10}$, so 10 digits is enough:

```
val N = 10
val d = new Array[Int](N)
```

We will also have the invariant

$$0 \leq \mathtt{n} \leq \mathtt{N}$$

First program

Here's a very simple program following that invariant.

```
// Invariant: I = (for all i in [0..n), d(i) = t@i) && 0<=n<=N
// Variant: N-n
var n = 0
while(n < N){
  var x = 1; for(j <- 1 to n) x *= 10 // Set x = 10^n
  d(n) = (t/x)%10                      // d(n) = t@n
  n = n+1                              // I
}
// I && n=N, so all digits calculated
```

The `main` method

```
def main(args:Array[String]){
  // Get the argument from the command line
  require(args.size>0); val t = args(0).toInt; require(t>0)

  // Initialise array.  2^31 < 10^10, so 10 digits is enough to
  // represent an Int
  val N = 10; val d = new Array[Int](N)

  ... // Above code to calculate the entries for d

  // Print out the digits
  for(i <- n-1 to 0 by -1) print(d(i))
  println
}
```

But the calculation of the digits is rather inefficient, as it calculates `10^n` from scratch on each iteration; and it also calculates leading `0`s (if `t < 10^9`).

Second program

It would be better to store the value of `10^n` in a variable `x` from one iteration to another. That is, we strengthen the invariant by adding a clause:

```
x = 10^n
```

This will also allow us to stop once `t < x`:

```
// Invariant: I = (for all i in [0..n), d(i) = t@i) && x = 10^n
var n = 0; var x = 1
while(t >= x){
  d(n) = (t/x)%10 // d(n) = t@n
  x *= 10          // x = 10^(n+1)
  n = n+1          // I
}
// I && t < x = 10^n, so all digits calculated
```

Termination

To prove termination, we could take the variant to be `10*t-x`; but then we need to add the clause `x <= 10*t` to the invariant.

Alternatively, we could take the variant to be $1 + \lfloor \log_{10} \mathtt{t} \rfloor - \mathtt{n}$; again we need to add the clause `x <= 10*t` to the invariant.

Next program

The previous program used three multiplications/divisions on each iteration. We can do better.

Rather than dividing by $\mathtt{x} = 10^{\mathtt{n}}$ on each iteration, we can use a variable `u` to store the value of $\mathtt{t}\ \mathrm{div}\ 10^{\mathtt{n}}$ from one iteration to the next. That is, we strengthen the invariant with the clause

$$\mathtt{u} = \mathtt{t}\ \mathrm{div}\ 10^{\mathtt{n}}$$

Note that the termination condition $\mathtt{t} < 10^{\mathtt{n}}$ is equivalent to $\mathtt{u} = 0$.

Code

$$I \mathrel{\widehat{=}} (\forall i \in [0..\mathtt{n}) \bullet \mathtt{d}(i) = \mathtt{t}@i) \wedge \mathtt{u} = \mathtt{t}\ \mathrm{div}\ 10^{\mathtt{n}}$$

```
var n = 0; var u = t  // I
while(u != 0){
  d(n) = u%10 // d(n) = (t div 10^n)%10 = t@n
  u = u/10      // u = t div 10^(n+1)
  n = n+1       // I
}
// I && t<10^n, so all digits have been calculated
```

Final program

The final program will calculate the digits from left-to-right.

Thinking about the correctness condition, it seems sensible to have an invariant including

$$I_1 \mathrel{\widehat{=}} \Big(\forall i \in [\mathtt{k}..\mathtt{n}) \bullet \mathtt{d}(i) = \mathtt{t}@i\Big) \wedge 0 \le \mathtt{k} \le \mathtt{n}$$

i.e. we've calculated the `n-k` most significant digits correctly.

We'll need to work out the value of `n` initially, and set `k = n`.

At each iteration, we'll calculate the value for `d(k-1)` and decrease `k`. This will continue until `k = 0`.

Final program

We need to calculate the value for `d(k-1)`, i.e.

$$\texttt{t@k-1} = (\texttt{t} \text{ div } 10^{\texttt{k}-1}) \text{ mod } 10 = (\texttt{t} \text{ mod } 10^{\texttt{k}}) \text{ div } 10^{\texttt{k}-1}$$

[Exercise: prove this equality.]

Bearing in mind that we'll later have to calculate `t@k-1`, `t@k-2`, ..., `t@0`, it makes sense to use variables `u` and `x` such that

$$I_2 \mathrel{\widehat{=}} \texttt{u} = \texttt{t} \text{ mod } 10^{\texttt{k}} \land \texttt{x} = 10^{\texttt{k}}$$

Code

```
// Find the number of digits (n); calculate 10^n at the same time.
// (Ignore overflow.)

var n = 1; var x = 10 // Invariant x = 10^n
while(t >= x){
  n = n+1; x = 10*x
}
// t < x = 10^n

// Invariant:
// (for all i in [k..n), d(i) = t@i) && u = t%(10^k) && x = 10^k
var k = n; var u = t
while(k > 0){
  k = k-1; x = x/10   // u = t % 10^(k+1), x = 10^k
  d(k) = u/x          // d(k) = (t%10^(k+1))/10^k = (t/10^k)%10 = t@k
  u = u%x             // u = (t % 10^(k+1)) % 10^k = t % 10^k
}
```

The second program

The code from the previous slide can be embedded into the `main` function, as on slide 5.

Alternatively, we could print out the digits as we calculate them.

Choosing invariants

A good invariant will explain how the program works.

In the final program above, the clause

$$\forall i \in [\mathtt{k}..\mathtt{n}) \bullet \mathtt{d}(i) = \mathtt{t}@i$$

explains what we have achieved to far.

The clauses

$$\mathtt{u} = \mathtt{t} \bmod 10^{\mathtt{k}} \wedge \mathtt{x} = 10^{\mathtt{k}}$$

explain the roles of `u` and `x`. These clauses are necessary to justify (or prove) that the value written into `d(k)` was correct. Having a clear statement of the values held in these variables helped come up with correct code, and avoided errors such as off-by-one (OBO) errors.

If you have some state that is carried forward from one iteration to the next, then the invariant should explain that.

Choosing invariants

Finally the clause

$$0 \leq \texttt{k} \leq \texttt{n}$$

helped document the range of the control variable `k`.

Summary

- Augmenting the state with extra variables to make the program more efficient;
- Using the invariant to explain the role of variables, and to help produce correct code.
- Next time: Binary search.