Pulling a New Proof from Knuth's Fixed-Point Printer

(*Floating Point Formatting, Part 2*)

Russ Cox
January 10, 2026
research.swtch.com/fp-knuth

Introduction

Donald Knuth wrote his 1989 paper "A Simple Program Whose Proof Isn't" as part of a tribute to Edsger Dijkstra on the occasion of Dijkstra's 60th birthday. Today's post is a reply to Knuth's paper on the occasion of Knuth's 88th birthday.

In his paper, Knuth posed the problem of converting 16-bit fixed-point binary fractions to decimal fractions, aiming for the shortest decimal that converts back to the original 16-bit binary fraction. Knuth gives a program named *P2* that leaves digits in the array *d* and a digit count in *k*:

```
P2:
    j := 0; s := 10 * n + 5; t := 10;
    repeat if t > 65536 then s := s + 32768 − (t div 2);
        j := j + 1; d[j] = s div 65536;
        s := 10 * (s mod 65536); t := 10 * t;
    until s ≤ t;
    k := j.
```

Knuth's goal was to prove *P2* correct without exhaustive testing, and he did, but he didn't consider the proof 'simple'. (Since there are only a small finite number of inputs, Knuth notes that this problem is a counterexample to Djikstra's remark that "testing can reveal the presence of errors but not their absence." Exhaustive testing would technically prove the program correct, but Knuth wants a proof that reveals *why* it works.)

At the end of the paper, Knuth wrote, "So. Is there a better program, or a better proof, or a better way to solve the problem?" This post presents what is, in my opinion, a better proof of the correctness of *P2*. It starts with a simpler program with a trivial direct proof of correctness. Then it transforms that simpler program into *P2*, step by step, proving the correctness of each transformation. The post then considers a few other ways to solve the problem, including one from a textbook that Knuth probably had within easy reach. Finally, it concludes with some reflections on the role of language in shaping our programs and proofs.

Problem Statement

Let's start with a precise definition of the problem. The input is a fraction f of the form $n/2^{16}$ for some integer $n \in [0, 2^{16})$. We want to convert f to the shortest accurate correctly rounded decimal form.

- By 'correctly rounded' we mean that the decimal is $d = \lfloor f \cdot 10^p + ½ \rfloor \,/\, 10^p$—that is, d is f rounded to p digits—for some p.
- By 'accurate' we mean that the decimal rounds back exactly: $\lfloor d \cdot 2^{16} + ½ \rfloor \,/\, 2^{16} = f$.
- By 'shortest' we mean that any shorter correctly rounded decimal $\lfloor f \cdot 10^q + ½ \rfloor \,/\, 10^q$ for $q < p$ is not accurate.

(For this problem, Knuth assumes "round half up" behavior, as opposed to IEEE 754 "round half to even", and the rounding equations reflect this. The answer does not change significantly if we use IEEE rounding instead.)

Notation

Next, let's define some convenient notation. As usual we will write the fractional part of x as $\{x\}$ and rounding as $[x] = \lfloor x + ½ \rfloor$. We will also define $\{x\}_p$, $\lfloor x \rfloor_p$, $\lceil x \rceil_p$, and $[x]_p$ to be the fractional part, floor, ceiling, and rounding of x relative to decimal fractions with p digits:

$$\begin{aligned}
\{x\}_p &= \{x \cdot 10^p\} \,/\, 10^p \\
\lfloor x \rfloor_p &= \lfloor x \cdot 10^p \rfloor \,/\, 10^p \\
\lceil x \rceil_p &= \lceil x \cdot 10^p \rceil \,/\, 10^p \\
[x]_p &= [x \cdot 10^p] \,/\, 10^p
\end{aligned}$$

Using the new notation, the correctly rounded p-digit decimal for f is $[f]_p$.

Following Knuth's paper, this post also uses the notation $[x \mathrel{..} y]$ for the interval from x to y, including x and y; $[x \mathrel{..} y)$ is the half-open interval, which excludes y.

Initial Solution

The definitions of 'accurate' and 'correctly rounded' imply two simple observations, which we will prove as lemmas. (In my attempt to avoid accusations of imprecision, I may well be too pedantic. Skip the proof of any lemma you think is obviously true.)

Accuracy Lemma. The accurate decimals are those in the *accuracy interval* $[f{-}2^{-17} \mathrel{..} f{+}2^{-17})$.

Proof. This follows immediately from the definition of accurate.

$$\begin{aligned}
\lfloor d \cdot 2^{16} + ½ \rfloor \,/\, 2^{16} &= f && \text{[definition of accurate]} \\
\lfloor d \cdot 2^{16} + ½ \rfloor &= f \cdot 2^{16} && \text{[multiply by } 2^{16}\text{]} \\
d \cdot 2^{16} + ½ &\in f \cdot 2^{16} + [0 \mathrel{..} 1) && \text{[domain of floor; } f \cdot 2^{16} \text{ is an integer]} \\
d \cdot 2^{16} &\in f \cdot 2^{16} + [-½ \mathrel{..} ½) && \text{[subtract ½]} \\
d &\in f + [-2^{-17} \mathrel{..} 2^{-17}) && \text{[divide by } 2^{16}\text{]}
\end{aligned}$$

We have shown that d being accurate is equivalent to $d \in f + [-2^{-17} \mathrel{..} 2^{-17}) = [f{-}2^{-17} \mathrel{..} f{+}2^{-17})$.

Five-Digit Lemma. The correctly rounded 5-digit decimal $d = [f]_5$ sits inside the accuracy interval.

Proof. Intuitively, the accuracy interval has width 2^{-16} while 5-digit decimals occur at the narrower spacing 10^{-5}, so at least one such decimal appears inside each accuracy interval.

More precisely,

$$\begin{aligned}
d &= \lfloor f \cdot 10^5 + ½ \rfloor \,/\, 10^5 && \text{[definition of } d\text{]} \\
&\in (f \cdot 10^5 + ½ - [0 \mathrel{..} 1)) \,/\, 10^5 && \text{[range of floor]} \\
&= (f - ½ \cdot 10^{-5} \mathrel{..} f + ½ \cdot 10^{-5}] && \text{[simplifying]} \\
&\subset [f{-}2^{-17} \mathrel{..} f{+}2^{-17}] && [½ \cdot 10^{-5} < 2^{-17}]
\end{aligned}$$

We have shown that the 5-digit correctly-rounded decimal for f is in the accuracy interval.

The problem statement and these two lemmas lead to a trivial direct solution: compute correctly rounded p-digit decimals for increasing p and return the first accurate one.

We will implement that solution in Ivy, an APL-like calculator language. Ivy has arbitrary-precision rationals and lightweight syntax, making it a convenient tool for sketching and testing mathematical algorithms, in the spirit of Iverson's Turing Award lecture about APL, "Notation as a Tool of Thought." Like APL, Ivy uses strict right-to-left operator precedence: `1+2*3+4` means `(1+(2*(3+4)))`, and `floor 10 log f` means `floor (10 log f)`. Operators can be prefix unary like `floor` or infix binary like `log`. Each of the Ivy displays in this post is executable: you can edit the code and re-run them by clicking the Play button ("▶"). A full introduction to Ivy is beyond the scope of this post; see the Ivy demo for more examples.

To start, we need to build a small amount of Ivy scaffolding. The binary operator `p digits d` splits an integer `d` into `p` digits:

```
op p digits d = (p rho 10) encode d
```

```
3 digits 123
```

```
1 2 3
```

```
4 digits 123
```

```
0 1 2 3
```

Next, Ivy already provides $\lfloor x \rfloor$ and $\lceil x \rceil$, but we need to define operators for $[x]$, $[x]_p$, $\lfloor x \rfloor_p$, and $\lceil x \rceil_p$.

```
op round x = floor x + 1/2

op p round x = (round x * 10**p) / 10**p
op p floor x = (floor x * 10**p) / 10**p
op p ceil  x = (ceil  x * 10**p) / 10**p
```

(Like APL, Ivy uses strict right-to-left evaluation; `1/2` is a rational constant literal, not a division.)

Now we can write our trivial solution.

```
op bin2dec f =
    min = f - 2**-17
    max = f + 2**-17
    p = 1
    :while 1
        d = p round f
        :if (min <= d) and (d < max)
            :ret p digits d * 10**p
        :end
        p = p + 1
    :end
```

Initial Proof of Correctness

The correctness of `bin2dec` is simple to prove:

- The implementation of `round` is the mathematical definition of p-digit rounding, so the result is correctly rounded.

- By the Accuracy Lemma, min and max are correctly defined for use in the accuracy test $min \le d < max$, so the result is accurate.
- The loop considers correctly rounded forms of increasing length until finding one that is accurate, so the result must be shortest.
- By the Five-Digit Lemma, the loop returns after at most 5 iterations, proving termination.

Therefore `bin2dec` returns the shortest accurate correctly rounded decimal for f.

Testing

Knuth's motivating example was that the decimal 0.4 converted to binary and back printed as 0.39999 in TeX. Let's define a decimal-to-binary converter and check how it handles 0.4.

```
op dec2bin f = (round f * 2**16) / 2**16
```

```
dec2bin 0.4
```

```
13107/32768
```

```
dec2bin 0.39999
```

```
13107/32768
```

```
float dec2bin 0.4
```

```
0.399993896484
```

We can see that both 0.4 and 0.39999 read as $26214/2^{16}$, or $13107/2^{15}$ in reduced form. The `float` operator converts that fraction to an Ivy floating-point value, which Ivy then prints to a limited number of decimals. We can see that 0.39999 is the correctly rounded 5-digit form.

Now let's see how our printer does:

```
bin2dec dec2bin 0.4
```

```
4
```

It works! And it works for longer fractions as well:

```
bin2dec dec2bin 0.123
```

```
1 2 3
```

(The values being printed are vectors of digits of the decimal fraction.)

We can also implement Knuth's solution, to compare against `bin2dec`.

```
op knuthP2 f =
    n = f * 2**16
    s = (10 * n) + 5
    t = 10
    d = ()
    :while 1
        :if t > 65536
            s = s + 32768 - t/2
        :end
        d = d, floor s/65536
        s = 10 * (s mod 65536)
        t = 10 * t
        :if s <= t
            :ret d
        :end
    :end
```

Compared to Knuth's original program text, the variable `d` is now an Ivy vector instead of a fixed-size array, and the variable `j`, previously the number of entries used in the array, remains only implicitly as the length of the vector. The assignment '$j := 0$' is now '`d = ()`', initializing d to the empty vector. And '$j := j + 1$; $d[j] = s$ **div** 65536' is now '`d = d, floor s/65536`', appending $\lfloor s/65536 \rfloor$ to d.

```
knuthP2 dec2bin 0.4
```

```
4
```

Now we can use testing to prove the absence of bugs. We'll be doing this repeatedly, so we will define `check desc`, which prints the result of the test next to a description.

```
)origin 0
op check desc =
    all = (iota 2**16) / 2**16
    ok = (bin2dec@ all) === (knuthP2@ all)
    print '❌✅'[ok] desc

check 'initial bin2dec'
```

```
✅ initial bin2dec
```

In this code, '`)origin 0`' configures Ivy to make `iota` and array indices start at 0 instead of APL's usual 1; `all` is $[0 \mathinner{..} 2^1 6)/2^1 6$, all the 16-bit fractions; `ok` is a boolean indicating whether `bin2dec` and `knuthP2` return identical results on all inputs; and the final line prints an emoji result and the description.

Of course, we want to do more than exhaustively check `knuthP2` against our provably correct `bin2dec`. We want to prove directly that the `knuthP2` code is correct. We will do that by incrementally transforming `bin2dec` into `knuthP2`.

Walking Digits

We can start by simplifying the computation of decimals that are shorter than necessary. To build an inuition for that, it will help to look at the accuracy intervals of short decimals. Here are the intervals for our test case:

```
op show x = (mix 'min ' 'f' 'max'), mix '%.100g' text@ (dec2bin x) + (-1 0 1)
* 2**-17
```

```
min 0.39998626708984375
f   0.399993896484375
max 0.40000152587890625
```

That printed the exact decimal values of $f - 2^{-17}$ (*min*), f, and $f + 2^{-17}$ (*max*) for
$f = \text{dec2bin}(0.4) = \left[0.4 \cdot 2^{16}\right] / 2^{16} = 26214/65536$.

Here are a few cases that are longer but still short:

```
show 0.43
```

```
min 0.42998504638671875
f   0.42999267578125
max 0.43000030517578125
```

```
show 0.432
```

```
min 0.43199920654296875
f   0.4320068359375
max 0.43201446533203125
```

```
show 0.4321
```

```
min 0.43209075927734375
f   0.432098388671875
max 0.43210601806640625
```

These displays suggest a new strategy for `bin2dec`: walk along the digits of these exact decimals until *min* and *max* diverge at the pth decimal place. At that point, we have found a p-digit decimal between *min* and *max*, namely $\lfloor max \rfloor_p$. That result is obviously accurate and shortest. It is not obvious that the result is correctly rounded, but that turns out to be the case for $p \le 4$. Intuitively, when p is shorter than necessary, there are fewer p-digit decimals than 16-bit binary fractions, so each accuracy interval can contain at most one decimal. When the accuracy interval does contain a decimal, that decimal must be both $[f]_p$ and $\lfloor max \rfloor_p$. For the full-length case $p = 5$, the accuracy interval can contain multiple p-digit decimals, so $[f]_p$ and $\lfloor max \rfloor_p$ may differ.

We will prove that now.

Rounding Lemma. For $p \le 4$, the accuracy interval contains at most one p-digit decimal. If it does contain one, that decimal is $[\,f\,]_p$, the correctly rounded p-digit decimal for f.

Proof. By the definition of rounding, we know that $d = [\,f\,]_p$ is *at most* $½ \cdot 10^{-p}$ away from f and that any other p-digit decimal must be *at least* $½ \cdot 10^{-p}$ away from f. Since $½ \cdot 10^{-p} > 2^{-17}$, those other decimals cannot be in the accuracy interval: the rounded d is the only possible option. (The rounded d may or may not itself be in the accuracy interval, but it's our best and only hope. If it isn't there, no p-digit decimal is.)

Endpoint Lemma. The endpoints $f \pm 2^{-17}$ of the accuracy interval are never p-digit decimals for $p \le 5$, nor are they shortest accurate correctly rounded decimals.

Proof. Because $f = n \cdot 2^{-16}$ for some integer n, $f \pm 2^{-17} = (2n \pm 1) \cdot 2^{-17}$. If an endpoint were a decimal of 5 digits or fewer, it would be an integer multiple of 10^{-5}, but $(2n \pm 1) \cdot 2^{-17} / 10^{-5} = ((2n \pm 1) \cdot 5^5) \cdot 2^{-12}$ is an odd number divided by 2^{12}, which cannot be an integer. The contradiction proves that the endpoints cannot be exact decimals of 5 digits or fewer. By the Five-Digit Lemma, the endpoints must also not be shortest accurate correctly rounded decimals.

Truncating Lemma. For $p \le 4$, the accuracy interval contains at most one p-digit decimal. If it does contain one, that decimal is $\lfloor f{+}2^{-17} \rfloor_p$, the p-digit truncation of the interval's upper endpoint.

Proof. The Rounding Lemma established that the accuracy interval contains at most one p-digit decimal.

Let $min = f{-}2^{-17}$ and $max = f{+}2^{-17}$, so the accuracy interval is $[min\ ..\ max)$. Any p-digit decimal in that interval must also be in the narrower interval using p-digit endpoints

$$\big[\lceil min \rceil_p\ ..\ \lfloor max \rfloor_p\big].$$

This new interval is strictly narrower because, by the Endpoint Lemma, min and max are not themselves p-digit decimals.

If $\lceil min \rceil_p > \lfloor max \rfloor_p$, the interval is empty. Otherwise, it clearly contains its upper endpoint, proving the lemma.

The Rounding Lemma and Truncating Lemma combine to prove that when $p \le 4$ and the accuracy interval contains any p-digit decimal, then it contains the single p-digit decimal

$$\lceil f{-}2^{-17} \rceil_p = [f]_p = \lfloor f{+}2^{-17} \rfloor_p.$$

The original `bin2dec` was written like this:

```
)op bin2dec
```

```
op bin2dec f =
    min = f - 2**-17
    max = f + 2**-17
    p = 1
    :while 1
        d = p round f
        :if (min <= d) and (d < max)
            :ret p digits d * 10**p
        :end
        p = p + 1
    :end
```

By the Five-Digit Lemma, we know that the loop terminates in the fifth iteration, if not before. Let's move that fifth iteration down after the loop, written as an unconditional return. That leaves the loop body handling only the short conversions:

```
op bin2dec f =
    min = f − 2**−17
    max = f + 2**−17
    p = 1
    :while p <= 4
        d = p round f
        :if (min <= d) and (d < max)
            :ret p digits d * 10**p
        :end
        p = p + 1
    :end
    p digits (p round f) * 10**p
```

```
check 'rounding bin2dec refactored'
```

```
✅ rounding bin2dec refactored
```

Next we can apply the Truncating Lemma to the loop body:

```
op bin2dec f =
    min = f − 2**−17
    max = f + 2**−17
    p = 1
    :while p <= 4
        :if (p ceil min) <= (p floor max)
            :ret p digits (p floor max) * 10**p
        :end
        p = p + 1
    :end
    p digits (p round f) * 10**p
```

```
check 'truncating bin2dec'
```

```
✅ truncating bin2dec
```

This version of `bin2dec` is much closer to *P2*, although not yet visibly so.

Premultiplication

Next we need to apply a basic optimization. The expressions `p ceil min`, `p trunc max`, and `p round f` hide repeated multiplication and division by 10^p. We can avoid those by multiplying *min*, *max*, and *f* by 10 on each iteration instead.

As an intermediate step, let's write the multiplications by 10^p explicitly, changing `p ceil x` to `(ceil x * 10**p) / 10**p` and so on:

```
op bin2dec f =
    min = f - 2**-17
    max = f + 2**-17
    p = 1
    :while p <= 4
        :if ((ceil min * 10**p) / 10**p) <= ((floor max * 10**p) / 10**p)
            :ret p digits ((floor max * 10**p) / 10**p) * 10**p
        :end
        p = p + 1
    :end
    p digits ((round f * 10**p) / 10**p) * 10**p
```

```
check 'multiplied bin2dec'
```

```
✅ multiplied bin2dec
```

Next, we can simplify away all the divisions by 10^p. In the comparison, not dividing both sides by 10^p leaves the result unchanged, and the other divisions are immediately re-multiplied by 10^p.

```
op bin2dec f =
    min = f - 2**-17
    max = f + 2**-17
    p = 1
    :while p <= 4
        :if (ceil min * 10**p) <= (floor max * 10**p)
            :ret p digits (floor max * 10**p)
        :end
        p = p + 1
    :end
    p digits (round f * 10**p)
```

```
check 'simplified'
```

```
✅ simplified
```

Finally, we can replace the multiplication by 10^p with multiplying f, min, and max by 10 each time we increment p:

```
op bin2dec f =
    min = f - 2**-17
    max = f + 2**-17
    p = 1
    f = 10 * f
    min = 10 * min
    max = 10 * max
    :while p <= 4
        :if (ceil min) <= (floor max)
            :ret p digits floor max
        :end
        p = p + 1
        f = 10 * f
        min = 10 * min
        max = 10 * max
    :end
    p digits round f
```

```
check 'premultiplied'
```

```
✅ premultiplied
```

Collecting Digits

At this point the only important difference between Knuth's *P2* and our current `bin2dec` is that *P2* computes one digit per loop iteration instead of computing them all from a single integer when returning. As we saw above, `bin2dec` is walking along the decimal form of min, f, and max until they diverge, at which point it can return an answer. Intuitively, since the walk continues only while the digits of all decimals in the accuracy interval agree, it is fine to collect one digit per step along the walk.

To help prove that intuition more formally, we need the following law of floors, which Knuth also uses. For all integers a and b with $b > 0$:

$$\left\lfloor \frac{\lfloor x \rfloor + a}{b} \right\rfloor = \left\lfloor \frac{x + a}{b} \right\rfloor.$$

Now we are ready to prove the necessary lemma.

Digit Collection Lemma. Let $min = f - 2^{-17}$ and $max = f + 2^{-17}$. For $p \geq 1$, the p+1-digit decimal $\lfloor max \rfloor_{p+1}$ has $\lfloor max \rfloor_p$ as its leading digits:

$$\left\lfloor \lfloor max \rfloor_{p+1} \right\rfloor_p = \lfloor max \rfloor_p.$$

Furthermore, for $p = 4$, if $\lceil min \rceil_p > \lfloor max \rfloor_p$ then the p+1-digit decimal $[f]_{p+1}$ has $\lfloor max \rfloor_p$ as its leading digits:

$$\left\lfloor [f]_{p+1} \right\rfloor_p = \lfloor max \rfloor_p.$$

Proof. For the first half, we can prove the result for any $p \geq 1$ and any $x \geq 0$, not just $x = max$:

$$\begin{aligned}
\left\lfloor \lfloor x \rfloor_{p+1} \right\rfloor_p &= \left\lfloor \left(\left\lfloor x \cdot 10^{p+1} \right\rfloor / 10^{p+1} \right) \cdot 10^p \right\rfloor / 10^p && [\text{definition of } \lfloor\,\rfloor_{p+1} \text{ and } \lfloor\,\rfloor_p] \\
&= \left\lfloor \left\lfloor x \cdot 10^{p+1} \right\rfloor / 10 \right\rfloor / 10^p && [\text{simplifying}] \\
&= \left\lfloor x \cdot 10^p \right\rfloor / 10^p && [\text{law of floors}] \\
&= \lfloor x \rfloor_p && [\text{definition of } \lfloor\,\rfloor_p]
\end{aligned}$$

For the second half, $\lceil min \rceil_p > \lfloor max \rfloor_p$ expands to $\left\lceil f - 2^{-17} \right\rceil_p > \left\lfloor f + 2^{-17} \right\rfloor_p$, which implies f is 2^{-17} away in both directions from an exact p-digit decimal: $2^{-17} \leq \{f\}_p < 10^{-p} - 2^{-17}$, or equivalently
$10^p \cdot 2^{-17} \leq \left\{f \cdot 10^p\right\} < 1 - 10^p \cdot 2^{-17}$. Note in particular that since $10^p \cdot 2^{-17} = 10^4 \cdot 2^{-17} > 1/20$,
$\left\lfloor f \cdot 10^p \right\rfloor = \left\lfloor f \cdot 10^p + 1/20 \right\rfloor = \left\lfloor max \cdot 10^p \right\rfloor$.

Now:

$$\begin{aligned}
\left\lfloor [f]_{p+1} \right\rfloor_p &= \left\lfloor [f]_{p+1} \cdot 10^p \right\rfloor / 10^p && [\text{definition of } \lfloor\,\rfloor_p] \\
&= \left\lfloor \left(\left\lfloor f \cdot 10^{p+1} + ½ \right\rfloor / 10^{p+1} \right) \cdot 10^p \right\rfloor / 10^p && [\text{definition of } [\,]_{p+1}] \\
&= \left\lfloor \left\lfloor f \cdot 10^{p+1} + ½ \right\rfloor / 10 \right\rfloor / 10^p && [\text{simplifying}] \\
&= \left\lfloor (f \cdot 10^{p+1} + ½) / 10 \right\rfloor / 10^p && [\text{law of floors}] \\
&= \left\lfloor f \cdot 10^p + 1/20 \right\rfloor / 10^p && [\text{simplifying}] \\
&= \left\lfloor max \cdot 10^p \right\rfloor / 10^p && [\text{noted above}] \\
&= \lfloor max \rfloor_p && [\text{definition of } \lfloor\,\rfloor_p]
\end{aligned}$$

(As an aside, this result is not a fluke of 16-bit binary fractions and $p = 4$. For any b-bit binary fraction, there is an accurate, correctly rounded p+1-digit decimal for $p+1 = \left\lceil \log_{10} 2^b \right\rceil$, because $10^{p+1} > 2^b$. That implies
$10^p \cdot 2^{-(b+1)} > 1/20$.)

The Digit Collection Lemma proves the correctness of saving one digit per iteration and using that sequence as the final result. Let's make that change. Here is our current version:

```
)op bin2dec
```

```
op bin2dec f =
    min = f - 2**-17
    max = f + 2**-17
    p = 1
    f = 10 * f
    min = 10 * min
    max = 10 * max
    :while p <= 4
        :if (ceil min) <= (floor max)
            :ret p digits floor max
        :end
        p = p + 1
        f = 10 * f
        min = 10 * min
        max = 10 * max
    :end
    p digits round f
```

Updating it to collect digits, we have:

```
op bin2dec f =
    min = f - 2**-17
    max = f + 2**-17
    p = 1
    f = 10 * f
    min = 10 * min
    max = 10 * max
    d = ()
    :while p <= 4
        d = d, (floor max) mod 10
        :if (ceil min) <= (floor max)
            :ret d
        :end
        p = p + 1
        f = 10 * f
        min = 10 * min
        max = 10 * max
    :end
    d, (round f) mod 10
```

```
check 'collecting digits'
```

```
✅ collecting digits
```

This program is very close to *P2*. All that remains are straightforward optimizations.

Change of Basis

The first optimization is to remove one of the three multiplications in the loop body, using the fact that *min*, f, and *max* are linearly dependent. If it were up to me, I would keep *min* and *max* and derive $f = (min + max)/2$ as needed, but to match *P2*, we will instead keep $s = max$ and $t = max - min$, deriving $max = s$, $min = s - t$, and $f = s - t/2$ as needed.

Let's make that change to the program:

```
op bin2dec f =
    s = f + 2**-17
    t = 2**-16
    p = 1
    s = 10 * s
    t = 10 * t
    d = ()
    :while p <= 4
        d = d, (floor s) mod 10
        :if (ceil s-t) <= (floor s)
            :ret d
        :end
        p = p + 1
        s = 10 * s
        t = 10 * t
    :end
    d, (round s - t/2) mod 10
```

```
check 'change of basis'
```

```
✓ change of basis
```

Discard Integer Parts

The next optimization is to reduce the size of *s* (formerly *max*). The only use of the integer part of *s* is to save the ones digit on each iteration, so we can discard the integer part with `s = s mod 1` each time we save the ones digit. That lets us optimize away the two uses of `mod 10`:

```
op bin2dec f =
    s = f + 2**-17
    t = 2**-16
    p = 1
    s = 10 * s
    t = 10 * t
    d = ()
    :while p <= 4
        d = d, floor s
        s = s mod 1
        :if (ceil s-t) <= (floor s)
            :ret d
        :end
        p = p + 1
        s = 10 * s
        t = 10 * t
    :end
    d, round s - t/2
```

```
check 'discard integer parts'
```

```
✓ discard integer parts
```

After the new `s = s mod 1`, $\lfloor s \rfloor = 0$, so the `if` condition is really $\lceil s - t \rceil \le 0$, which simplifies to $s \le t$. Let's make that change too:

```
op bin2dec f =
    s = f + 2**-17
    t = 2**-16
    p = 1
    s = 10 * s
    t = 10 * t
    d = ()
    :while p <= 4
        d = d, floor s
        s = s mod 1
        :if s <= t
            :ret d
        :end
        p = p + 1
        s = 10 * s
        t = 10 * t
    :end
    d, round s - t/2
```

```
check 'simplify condition'
```

```
✓ simplify condition
```

Refactoring

Next, we can inline `round` on the last line:

```
op bin2dec f =
    s = f + 2**-17
    t = 2**-16
    p = 1
    s = 10 * s
    t = 10 * t
    d = ()
    :while p <= 4
        d = d, floor s
        s = s mod 1
        :if s <= t
            :ret d
        :end
        p = p + 1
        s = 10 * s
        t = 10 * t
    :end
    s = (s - t/2) + 1/2
    d, floor s
```

```
check 'inlined round'
```

```
✓ inlined round
```

Now the two uses of '`d, floor s`' can be merged by moving the final return back into the loop, provided (1) we make the `while` loop repeat forever, (2) we apply the final adjustment to s when $p = 5$, and (3) we ensure that when $p = 5$, the `if` condition is true, so that the return is reached. The `if` condition is checking for digit divergence, and we know that *min* and *max* will always diverge by $p = 5$, so the condition $s \le t$ will be true. We can also confirm that arithmetically: $t = 10^5 \cdot 2^{-16} > 1 > s$.

Let's make that change:

```
op bin2dec f =
    s = f + 2**-17
    t = 2**-16
    p = 1
    s = 10 * s
    t = 10 * t
    d = ()
    :while 1
        :if p == 5
            s = (s - t/2) + 1/2
        :end
        d = d, floor s
        s = s mod 1
        :if s <= t
            :ret d
        :end
        p = p + 1
        s = 10 * s
        t = 10 * t
    :end
```

```
check 'return only in loop'
```

```
✓ return only in loop
```

Next, since $t = 10^p \cdot 2^{-17}$, we can replace the condition $p = 5$ with $t > 1$, after which p is unused and can be deleted:

```
op bin2dec f =
    s = f + 2**-17
    t = 2**-16
    s = 10 * s
    t = 10 * t
    d = ()
    :while 1
        :if t > 1
            s = (s - t/2) + 1/2
        :end
        d = d, floor s
        s = s mod 1
        :if s <= t
            :ret d
        :end
        s = 10 * s
        t = 10 * t
    :end
```

```
check 'optimize away p'
```

```
✓ optimize away p
```

Next, note that the truth of $s \leq t$ is unchanged by multiplying both s and t by 10 (and the return does not use them) so we can move the conditional return to the end of the loop body:

```
op bin2dec f =
    s = f + 2**-17
    t = 2**-16
    s = 10 * s
    t = 10 * t
    d = ()
    :while 1
        :if t > 1
            s = (s - t/2) + 1/2
        :end
        d = d, floor s
        s = s mod 1
        s = 10 * s
        t = 10 * t
        :if s <= t
            :ret d
        :end
    :end
```

```
check 'move conditional return'
```

```
✓ move conditional return
```

Finally, let's merge the consecutive assignments to s and t, both at the top of `bin2dec` and in the loop:

```
op bin2dec f =
    s = 10 * f + 2**-17
    t = 10 * 2**-16
    d = ()
    :while 1
        :if t > 1
            s = (s - t/2) + 1/2
        :end
        d = d, floor s
        s = 10 * s mod 1
        t = 10 * t
        :if s <= t
            :ret d
        :end
    :end
```

```
check 'merge assignments'
```

```
✓ merge assignments
```

Scaling

All that remains is to eliminate the use of rational arithmetic. which we can do by scaling *s* and *t* by 2^{16}:

```
op bin2dec f =
    s = (10 * f * 2**16) + 5
    t = 10
    d = ()
    :while 1
        :if t > 2**16
            s = (s - t/2) + 2**15
        :end
        d = d, floor s/2**16
        s = 10 * s mod 2**16
        t = 10 * t
        :if s <= t
            :ret d
        :end
    :end
```

```
check 'no more rationals'
```

```
✅ no more rationals
```

If written in a modern compiled language, this is a very efficient program. (In particular, `floor s/2**16` and `s mod 2**16` are simple bit operations: `s >> 16` and `s & 0xFFFF` in C syntax.)

And we have arrived at Knuth's *P2*!

```
)op knuthP2
```

```
op knuthP2 f =
    n = f * 2**16
    s = (10 * n) + 5
    t = 10
    d = ()
    :while 1
        :if t > 65536
            s = s + 32768 - t/2
        :end
        d = d, floor s/65536
        s = 10 * (s mod 65536)
        t = 10 * t
        :if s <= t
            :ret d
        :end
    :end
```

We started with a trivially correct program and then incrementally modified it, proving the correctness of each non-trivial step, to arrive at *P2*. We have therefore proved the correctness of *P2* itself.

Simpler Programs and Proofs

We passed a more elegant iterative solution a few steps back. If we start with the premultiplied version, apply the change of basis $\varepsilon = max - f = f - min$, scale by 2^{16}, and change the program to return an integer instead of a digit vector, we arrive at:

```
op shortest f =
    p = 1
    f = 10 * f * 2**16
    ε = 5
    :while p < 5
        :if (ceil (f-ε)/2**16) <= floor (f+ε)/2**16
            :ret (floor (f+ε)/2**16) p
        :end
        p = p + 1
        f = f * 10
        ε = ε * 10
    :end
    (round f/2**16) p
```

The program returns the digits as an integer accompanied by a digit count. Any modern language can print an integer zero-padded to a given number of digits, so there is no need for our converter to compute those digits itself.

We can test `shortest` by writing `bin2dec` in terms of `shortest` and `digits`:

```
op bin2dec f =
    d p = shortest f
    p digits d
```

```
check 'simpler'
```

```
✅ simpler
```

The full proof of correctness is left as an exercise to the reader, but note that the proof is simpler than the one we just gave for *P2*. Since we are not collecting digits one at a time, we do not need the Digit Collection Lemma.

This version of `shortest` could be made faster by using *P2*'s s, t basis, but I think this form using the f, ε basis is clearer, and the cost is only one extra addition in the loop body. The cost of that addition is unlikely to be important compared to the two multiplications and other arithmetic (two shifts, one subtraction, one comparison, one increment).

One drawback of this simpler program is that it requires f to hold numbers up to $10^5 \cdot 2^{16} > 2^{32}$, so it needs f to be a 64-bit integer, and the system Knuth was using probably did not support 64-bit integers. However, we can adapt this simpler program to work with 32-bit integers by observing that each multiplication by 10 adds a new always-zero low bit, so we can multiply by 5 and adjust the precision b:

```
op shortest f =
    b = 16
    p = 1
    f = 10 * f * 2**b
    ε = 5
    :while p < 5
        :if (ceil (f-ε)/2**b) <= floor (f+ε)/2**b
            :ret (floor (f+ε)/2**b) p
        :end
        p = p + 1
        b = b - 1
        f = f * 5
        ε = ε * 5
    :end
    (round f/2**b) p
```

```
check '32-bit'
```

```
✓ 32-bit
```

All modern languages provide efficient 64-bit arithmetic, so we don't need that optimization today.

A More Direct Solution

Raffaello Giulietti's Schubfach algorithm avoids the iteration entirely. Applied to Knuth's problem, we can let $p = \lceil \log_{10} 2^b \rceil$ (= 5) and compute the exact set of accurate p-digit decimals $\left[\lceil min \rceil_p \mathinner{..} \lfloor max \rfloor_p \right]$. That set contains at most 10 consecutive decimals (or else p would be smaller), so at most one can end in a zero. If one of the accurate decimals d ends in zero, we can use d after removing its trailing zeros. (The Truncating Lemma guarantees that this shortest accurate decimal will be correctly rounded.) Otherwise, we should use the correctly rounded $[\,f\,]_p$.

```
op shortest f =
    b = 16
    p = ceil 10 log 2**b
    f = (10**p) * f * 2**b
    t = (10**p) * 1/2
    min = ceil  (f - t) / 2**b
    max = floor (f + t) / 2**b
    :if ((d = floor max / 10) * 10) >= min
        p = p - 1
        :while ((d mod 10) == 0) and p > 1
            d = d / 10
            p = p - 1
        :end
        :ret d p
    :end
    (round f / 2**b) p
```

```
check 'schubfach'
```

```
✓ schubfach
```

This program still contains a loop, but only to remove trailing zeros. In many use cases, short outputs will happen less often than full-length outputs, so most calls will not loop at all. Also, all modern compilers implement division by a constant using multiplication, so the loop costs at most $p-1$ multiplications. Finally, we can shorten the worst case, at the expense of the best case, by using a ($\log_2 p$)-iteration loop: divide away $\lfloor p/2 \rfloor$ trailing zeros (by checking $d \bmod 10^{\lfloor p/2 \rfloor}$), then $\lfloor p/4 \rfloor$, and so on.

A Textbook Solution

All of this raises a mystery. Knuth started writing TeX82, which introduced the 16-bit fixed-point number representation, in August 1981 (according to "The Errors of TeX", page 616), and the change to introduce shortest outputs was made in February 1984 (according to tex82.bug, entry 284). The mystery is why Knuth, working in the early 1980s, did not consult a recently published textbook that contains the answer, namely *The Art of Computer Programming, Volume 2: Seminumerical Algorithms (Second Edition)*, by D. E. Knuth (preface dated July 1980).

Before getting to the mystery, we need to detour through the history of that specific answer. The first edition of Volume 2 (preface dated October 1968), contained exercise 4.4-3:

1. [25] (D. Taranto.) The text observes that when fractions are being converted, there is in general no obvious way to decide how many digits to give in the answer. Design a simple generalization of Method (2a) [incremental digit-at-a-time fraction conversion] which, given two positive radix b fractions u and ε between 0 and 1, converts u to a radix B equivalent U which has just enough places of accuracy to ensure that $|U - u| < \varepsilon$. (If $u < \varepsilon$, we may take $U = 0$, with zero "places of accuracy.")

The answer at the back of the book starts with the notation "[CACM 2 (July 1959), 27]", indicating Taranto's article "Binary conversion, with fixed decimal precision, of a decimal fraction". Taranto's article is about converting a decimal fraction to a shortest binary representation, a somewhat simpler problem than Knuth's exercise poses. Written in Ivy, with variable names chosen to match this post, and scaling to accumulate bits in an integer during the loop, Taranto's algorithm is:

```
op taranto (f ε) =
    fb = 0
    b = 0
    :while (ε < f) and (f < 1-ε)
        f = 2 * f
        fb = (2 * fb) + floor f
        f = f mod 1
        ε = 2 * ε
        b = b + 1
    :end
    :if (fb & 1) == 0
        fb = fb + 1
    :end
    fb / 2**b
```

If we write f_0 and ε_0 for the initial f and ε passed to `shortest`, then the algorithm accumulates a binary output in f_b, and maintains $f = (f_0 - f_b) \cdot 2^b$, the scaled error of the current output compared to the original input. When $f \leq \varepsilon$, f_b is close enough; when $f \geq 1 - ε$, $f_b + 1$ is close enough. Otherwise, the algorithm loops to add another output bit.

After the loop, the algorithm forces the low bit to 1. This handles the "when $f \geq 1 - ε$, $f_b + 1$ is close enough" case. It would have been clearer to write $f \geq 1 - ε$, but testing the low bit is equivalent. If the last bit added was 0, the final iteration started with $ε < f$ and did $f = 2f, ε = 2ε$, in which case $ε < f$ must still be true and the loop ended because $f \geq 1 - ε$. And if the last bit added was 1, the final iteration started with $f < 1 - ε$ and did $f = 2f - 1, ε = 2ε$, in which case $f < 1 - ε$ must still be true and the loop ended because $f \leq ε$.

(In fact, Taranto's presentation took advantage of the fact that the low bit tells you which half of the loop condition is possible; it only checked the possible half in each iteration. For simplicity, the Ivy version omits that optimization. If you read Taranto's article, note that the computation of the loop condition is correct in step B at the top of the page but incorrect in the IAL code at the bottom of the page.)

For the answer to exercise 4.4-3, Knuth needed to generalize Taranto's algorithm to non-binary outputs ($B > 2$), which he did by changing the final condition to $f > 1 - ε$. For decimal output, the implementation would be:

```
op shortest f =
    ε = 2**-17
    d = 0
    p = 0
    :while (ε < f) and (f < 1-ε)
        f = 10 * f
        d = (10 * d) + floor f
        f = f mod 1
        ε = 10 * ε
        p = p + 1
    :end
    :if f > 1-ε
        d = d + 1
    :end
    d (1 max p)
```

Knuth's presentation generated digits one at a time into an array. I've accumulated them into an integer *d* here, but that's only a storage detail. When incrementing the low digit after the loop, Knuth's answer also checked for overflow from the low digit into higher digits. That check is unnecessary: if the increment overflows the bottom digit to zero, that implies the bottom digit can be removed entirely, in which case the loop would have stopped earlier.

It turns out that this code is not an answer to our problem:

```
check 'Knuth Volume 2 1e'
```

```
❌ Knuth Volume 2 1e
```

The problem is that while it does compute a shortest accurate decimal, it does not guarantee correct rounding:

```
shortest dec2bin 0.12344
```

```
12345 5
```

For binary output, shortest and correctly rounded are one and the same; not so in other bases. As an aside, Taranto's forcing of the low bit to 1 mishandles the corner case $f = 0$. Knuth's updated condition fixes that case, but it doesn't correctly round other cases. All that said, Knuth's exercise did not ask for correct rounding, so the answer is still correct for the exercise.

Guy L. Steele and Jon L. White, working in the 1970s on fixed-point and floating-point formatting, consulted Knuth's first edition and adapted this answer to round correctly. They wrote a paper with their new algorithms, both for the fixed-point case we are considering and for the more general case of floating-point numbers. That paper presents a correctly-rounding extension of Knuth's first edition answer as the algorithm '$(FP)^3$'. The change is tiny: replace $f \geq 1 - ε$ with $f \geq ½$.

```
op shortest f =
    ε = 2**-17
    d = 0
    p = 0
    :while (ε < f) and (f < 1-ε)
        f = 10 * f
        d = (10 * d) + floor f
        f = f mod 1
        ε = 10 * ε
        p = p + 1
    :end
    :if f >= 1/2
        d = d + 1
    :end
    d (1 max p)
```

```
check 'Steele and White (FP)³'
```

```
✅ Steele and White (FP)³
```

In the paper, Steele and White described the $(FP)^3$ algorithm as "a generalization of the one presented by Taranto [Taranto59] and mentioned in exercise 4.4.3 of [Knuth69]." They shared the paper with Knuth while he was working on the second edition of Volume 2. In response, Knuth changed the exercise to ask for a "*rounded* radix B equivalent" (my emphasis), updated the answer to use the new fix-up condition, removed the unnecessary overflow check, and cited Steele and White's unpublished paper. Knuth introduced the revised answer by saying, "The following procedure due to G. L. Steele Jr. and Jon L. White generalizes Taranto's algorithm for $B = 2$ originally published in *CACM* **2**, 7 (July 1959), 27." The attribution 'due to [Steele and White]' omits Knuth's own substantial contributions to the generalization effort.

Steele and White published their paper in 1990, and Knuth cited it in the third edition of Volume 2 (preface dated July 1997). At first glance, this seems to create a circular reference in which both works credit the other for the algorithm, like a self-justifying out-of-thin-air value. The cycle is broken by noticing that Steele and White carefully cite the *first edition* of Volume 2.

In 1984, as Knuth was writing TeX82 and needed code to implement this conversion, the second edition had just been published a few years earlier. So why did Knuth invent a new variant of Taranto's algorithm instead of using the one he put in Volume 2? I find it comforting to imagine that he made the same mistake we've all made at one time or another: perhaps Knuth simply forgot to check *Knuth*.

But probably not. Knuth's "Simple Program" paper names Steele and White and cites the answer to exercise 4.4-3. The wording of the citation suggests that Knuth did not consider it an answer to his question "Is there a better program?" Why not? I don't know, but we just saw that it does work. I plan to send Knuth a letter to ask.

(Detour sign photos by Don Knuth.)

Conclusion

This post shows what I think is a better way to prove the correctness of Knuth's *P2*, as well as a few candidates for better programs. More generally, I think the post illustrates that the capabilities of our programming tools affect the programs we write with them and how easy it is to prove those programs correct.

If you are programming a 32-bit computer in the 1980s using a Pascal-like language in which division is expensive, it makes perfect sense to compute one digit at a time in a loop as Knuth did in *P2*. Going back even further, the iterative digit-at-a-time approach made sense on Von Neumann's computer in the 1940s (see p. 53). Today, a language with arbitrary precision rationals makes it easy to write simpler programs.

The choice of language also affects the difficulty of the proof. Using Ivy made it natural to break the proof into pieces. We started with a simple proof of a nearly trivial program. Then we proved the correctness of the "truncated max" version. Finally we proved the correctness of collecting digits one at a time. It was easier to write three small proofs than one large proof. Ivy also made it easy to isolate the complexity of premultiplication by 10^p and scaling by 2^{16}; we were able to treat those steps as optimizations instead of fundamental aspects of the program and proofs. Now that we know the path, we could write a direct proof of *P2* along these lines. But I wouldn't have seen the path without having a capable language like Ivy to light the way.

It is also worth noting how the capabilities of our programming tools affect our perception of what is important in our programs. After writing his 1989 paper, Knuth optimized '$s := s + 32768 - (t$ **div** $2)$' in the production version of TeX to '$s := s - 17232$', because at that point t is always 100000. On the IBM 650 at Case Institute of Technology where Knuth got his start (and to which he dedicated his *The Art of Computer Programming* books), removing a division and an addition might have been important. In 1989, however, a good compiler would have optimized 't **div** 2' to a shift, and the shift and add would hardly matter compared to the eight multiplications that preceded them, not to mention the I/O to print the result, and the code was probably clearer the first way. But old habits die hard for all of us. I spent my formative years programming 32-bit systems, and I have not broken my old habit of worrying about '32-bit safety', as evidenced by the discussion above!

Knuth wrote in his paper that "we seek a proof that is comprehensible and educational" and then added:

> Even more, we seek a proof that reflects the ideas used to create the program, rather than a proof that was concocted ex post facto. The program didn't emerge by itself from a vacuum, nor did I simply try all possible short programs until I found one that worked.

This post is almost certainly not the proof Knuth sought. On the other hand, I hope that it is comprehensible and educational. Also, Taranto's short article doesn't include any proof at all, nor even an explanation of how it works. If I had to prove Taranto's algorithm correct, I would probably proceed as in the initial part of this post. Then, if you accept Taranto's algorithm as correct, the main changes on the way to *P2* are to nudge f up to max at the start and then nudge it back down on the final iteration. The Truncating Lemma and the Digit Collection Lemma prove the correctness of those changes. Maybe that does match what Knuth had in mind in 1984 when he adapted Taranto's algorithm. Maybe the difficulty arose from having to prove Taranto's algorithm correct simultaneously. This post's incremental approach avoids that complication.

In any event, Happy 88th Birthday Don!

P.S. This is my first blog post using my blog's new support for embedding Ivy code and for typesetting equations. For the latter, I write TeX syntax like inline `` `$s ≤ t$` `` or fenced ```` ``` ````eqn blocks. A new TeX macro parser that I wrote executes the TeX input and translates the expanded output to MathML Core, which all the major browsers now support. It is only fitting that this is the first post to use the new TeX-derived mathematical typesetting.