The Division Bell Tolls for Thee: An Inquiry Into Integer Division in Haskell and C

Paul R. Potts

December 2006 and March 2025

In 2006 I published a series of blog posts called "The Division Bell Tools for Me," and a followup post called "The Divisive Aftermath." The subject was integer division, and how a seemingly simple thing could wind up being so difficult for programmers to understand and use correctly. This article combines and updates them. Note that in 2006, the platforms I was using had 32-bit integers. This is no longer true in 2025, but the original text and comments still mention 32-bit integers. I have not re-built and re-run the C code on a 64-bit platform since the way integer division works *in C has not fundamentally changed, even if* ***int*** *is bigger. Also, note that the version of GHCi I used to re-test all the Haskell code, shown in some of the logs, is most certainly not the version I used in 2006.*

> "Is it not a strange fate that we should suffer so much fear and doubt for so small a thing? So small a thing! And I have seen it only for an instant in the house of Elrond! Could I not have a sight of it again?"
>
> Frodo looked up. His heart went suddenly cold. He caught the strange gleam in Boromir's eyes, yet his face was still kind and friendly. "It is best that it should lie hidden," he answered.

— *The Fellowship fof the Ring* by J.R.R. Tolkien

Part One

With apologies to both Pink Floyd and John Donne.

I've been toying with Haskell. I wish I had enough free time for a systematic study of Richard Bird's *Introduction to Functional Programming Using Haskell, Second Edition*, but for now I have to content myself with dipping into it in small scraps of free time when both babies are asleep and the dishes are done. In addition, I'm using Haskell to model the behavior of some functions that I'm actually writing in C++.

One of the reasons I like Haskell is that with a tool like GHCi, which gives me a REPL (a Read, Evaluate, Print loop for interactive development), I can write

toy one-liners that do something useful, and since Haskell supports typing, I can test how machine types behave — with an important exception, which I will discuss.

Let's say I want to try out a function that maps the fractional values -7/7 through 7/7 to unsigned integers in the range 0..65536 (the range of values representable by 16 bits, plus one). In Haskell I can try this out using one-liners (typed into GHCi). Let's build the full function up from smaller parts. First, the range of input values. Ignore for now the possibility of using or introducing a rational number type; instead we'll generate a set of tuples. In Haskell you can express this very concisely using a list comprehension.

Note that in the logs of interactions with GHCi I have broken the output lines to fit on my web page.

```
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> [(numer, 7) | numer <- [-7..7]]
[(-7,7),(-6,7),(-5,7),(-4,7),(-3,7),(-2,7),(-1,7),(0,7),(1,7),
(2,7),(3,7),(4,7),(5,7),(6,7),(7,7)]
```

The list comprehension can be (roughly) read as "Make a list of tuples out of numer, 7 where numer takes on the values from the list -7 to 7." Think for a moment about how you could do this in your favorite language.

Now let's do our math on the resulting list. We can use **map** to apply a function to the list. We can use **fst** and **snd** to get to the values in the tuple; for example, to look at the numerators, do this:

```
ghci> map fst [(numer, 7) | numer <- [-7..7]]
[-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7]
```

Let's go ahead and do our scaling. We want to describe a simple function to pass each tuple to; in fact we want a lambda abstraction, a function that we don't bother to name.

I would normally just paste this function into GHCi as a one-liner, but the long one-liners don't fit nicely on a web page or PDF file. There's a trick we can use to paste multiple lines into GHCi and have it treat them as one line: put the lines of code between **:{** and **:}**. In the example below I copied the following:

```
:{
map (\tup -> fst tup / snd tup * 65536 )
[(numer, 7) | numer <- [-7..7]]
:}
```

I pasted this into GHCi. When GHCi saw the opening **:{** on a line, it changed the prompt to indicate it is waiting for the end of multiple lines. The session looked like this:

```
ghci> :{
ghci| map (\tup -> fst tup / snd tup * 65536 )
```

```
ghci| [(numer, 7) | numer <- [-7..7]]
ghci| :}
[-65536.0,-56173.71428571428,-46811.42857142857,-37449.142857142855,
-28086.85714285714,-18724.571428571428,-9362.285714285714,0.0,
9362.285714285714,18724.571428571428,28086.85714285714,
37449.142857142855,46811.42857142857,56173.71428571428,65536.0]
```

From here on out, I'll assume you know how to use this trick and just show the code to paste, and GHCI's output, wrapped for readability.

Anyway, that output is not quite what we want. We want to map to only the positive values. We have to shift the domain of the incoming numerators so the results are non-negative:

```
:{
map (\tup -> (fst tup + snd tup) / snd tup * 65536)
[(numer, 7) | numer <- [-7..7]]
:}
```

The output is:

```
[0.0,9362.285714285714,18724.571428571428,28086.85714285714,
37449.142857142855,46811.42857142857,56173.71428571428,65536.0,
74898.28571428571,84260.57142857143,93622.85714285714,
102985.14285714286,112347.42857142857,121709.71428571429,131072.0]
```

But now the codomain is too big; we have to cut it in half:

```
:{
map (\tup -> (fst tup + snd tup) / snd tup * 65536 / 2)
[(numer, 7) | numer <- [-7..7]]
:}
```

The output is:

```
[0.0,4681.142857142857,9362.285714285714,14043.42857142857,
18724.571428571428,23405.714285714286,28086.85714285714,
32768.0,37449.142857142855,42130.28571428572,46811.42857142857,
51492.57142857143,56173.71428571428,60854.857142857145,65536.0]
```

That looks about right. Now let's apply a type. Because I'm eventually going to be implementing this without floating-point, I want the math to be done on machine integers. There are some exceptions, but most target architectures these days give you 32 bits for C's **int**, so Haskell's **Int**, which is defined to be at least 32 bits, should yield approximately what I want. *Note that this is no longer true in 2025, when C **int** types on most recent CPUs are 64-bit integers!*

To introduce typing, all I have to do is add a type to the list values. Type inferencing will do the rest. Oh, and since division using the forward slash "/" is not defined on **Int**, I have to change it to **‘div‘** (infix division):

```
:{
```

```
map (\tup -> (fst tup + snd tup) `div` snd tup * 65536 `div` 2)
[(numer, (7::Int)) | numer <- [(-7::Int)..(7::Int)]]
:}
```

The output is:

```
[0,0,0,0,0,0,0,32768,32768,32768,32768,32768,32768,32768,65536]
```

Whoah. Something has gone disastrously long. Take a moment to figure out what it is. I'll wait.

Did you figure it out?

The problem is that when I use integer division in an expression, the result is truncated (the fractional part is lost). If the division occurs before any of the other operations, most of the significant bits of the math are lost! I can fix this by making sure to do the division last:

```
:{
map (\tup -> (fst tup + snd tup) * 65536 `div` snd tup `div` 2)
[(numer, (7::Int)) | numer <- [(-7::Int)..(7::Int)]]
:}
```

The output is:

```
[0,4681,9362,14043,18724,23405,28086,32768,37449,
42130,46811,51492,56173,60854,65536]
```

That's better. While I'm at it, I can simplify out that factor of 2 by using half of 65536:

```
:{
map (\tup -> (fst tup + snd tup) * 32768 `div` snd tup)
[(numer, (7::Int)) | numer <- [(-7::Int)..(7::Int)] ]
:}
```

The output is:

```
[0,4681,9362,14043,18724,23405,28086,32768,37449,
42130,46811,51492,56173,60854,65536]
```

In fact, I can ask GHC to verify that the results are the same:

```
:{
map (\tup -> (fst tup + snd tup) * 65536 `div` snd tup `div` 2)
[(numer, (7::Int)) | numer <- [(-7::Int)..(7::Int)]]
==
map (\tup -> (fst tup + snd tup) * 32768 `div` snd tup)
[(numer, (7::Int)) | numer <- [(-7::Int)..(7::Int)]]
:}
```

The output is:

```
True
```

Which is, of course, what I expected, but after decades of programming, my motto is "trust, but verify." Of course, this is not a rigorous proof, but I'm satisfied that my function is correct for the given inputs, and sometimes that's enough.

Part One Comments

The Alternate Moebyus wrote…

> *quoting me: "…since Haskell supports typing, I can test how machine types behave — with an important exception, which I will discuss.."*
>
> I'd actually claim that the "machine types" are behaving exactly like they would in the C world :) Integer division is truncation, which is usually avoided by the usual trick of using the associative law.
>
> Nice to see you using Haskell, though. It's a breath of fresh air.

I replied:

> Hi Alternate Moebyus,
>
> Thanks for your comment. I am going to continue this topic and explore just how integer division in C actually behaves, what guarantees can be made about it, and what can't.

Miles wrote:

> I suspect a more Haskellish way to write the original division code would be:

```
zipWith (/) (repeat 7) [-7 .. 7]
```

> Or even:

```
[ 7 / y | y <- [-7 .. 7] ]
```

> :-)

Miles posted another suggested expression with list comprehensions but it did not have valid syntax. He may have been having trouble with Blogger swallowing characters in comments. He posted a modified version in a followup comment, but that also did not have valid syntax. I did not have any success figuring out exactly what his intended code was; it was more than a matter of missing brackets or parentheses. So, I am leaving those comments out of this edited version.

I replied:

> Miles, thanks for the suggestions… I will play with these a little later. I am very new to Haskell.

Part Two

OK. In our last installment I built up a simple Haskell function. The goal of the function was to map a set of fractions in the range -7/7 to 7/7 to the machine integers 0..65536. Here's where we left off, evaluating the function using GHCi:

```
:{
map (\tup -> (fst tup + snd tup) * 32768 `div` snd tup)
[(numer, (7::Int)) | numer <- [(-7::Int)..(7::Int)] ]
:}
```

I get back:

```
[0,4681,9362,14043,18724,23405,28086,32768,37449,
42130,46811,51492,56173,60854,65536]
```

Let's play with that function a little more, and then I want to talk about division more generally. In particular, integer division.

First, let's simplify our expressions by breaking them down and establishing some bindings. We can also get rid of the presumption that our output range starts at zero. This function will generate a range of fractions given a denominator:

```
:{
gen_nths denom = [(numer, denom :: Int) |
    numer <- [(-denom :: Int) .. (denom :: Int)]]
:}
```

I can test the function:

```
ghci> gen_nths 11
[(-11,11),(-10,11),(-9,11),(-8,11),(-7,11),(-6,11),(-5,11),
(-4,11),(-3,11),(-2,11),(-1,11),(0,11),(1,11),(2,11),(3,11),
(4,11),(5,11),(6,11),(7,11),(8,11),(9,11),(10,11),(11,11)]
```

Now we define a function called **div_map**:

```
:{
div_map m1 m2 =
    map (\tup -> (fst tup + snd tup) *
                 (((m2 :: Int) - (m1 :: Int)) `div` 2) `div`
                 snd tup + m1)
:}
```

And now we can test it. Note that we need to put parentheses around our negative number:

```
ghci> div_map (-32768) 32768 (gen_nths 11)
[-32768,-29790,-26811,-23832,-20853,-17874,-14895,
-11916,-8937,-5958,-2979,0,2978,5957,8936,11915,
14894,17873,20852,23831,26810,29789,32768]
```

OK. Now what is it we've done exactly? Well, we've generated some test values, and an algorithm, for distributing a fractional range onto a machine integer range. For our purposes, think of it as a little machine for testing division.

To see just why I chose this algorithm to play with Haskell, we have to go back and consider some C code. It's quite a headache-inducing paradigm shift to go back to C/C++, but bear with me. Let's just start with the simplest thing: we'll create the same mapping we just generated, hard-coding the values:

```
long denom = 11;
long numer;

for ( numer = -11; numer <= 11; numer++ )
{
    long val = numer * 32768 / denom;
    printf("%d\n", val);
}
```

And the results:

```
-32768
-29789
-26810
-23831
-20852
-17873
-14894
-11915
-8936
-5957
-2978
0
2978
5957
8936
11915
14894
17873
20852
23831
26810
29789
32768
```

Hmmm… wait a minute, let's compare the results:

Haskell	C
-32768	same
-29790	-29789
-26811	-26810
-23832	-23831
-20853	-20852
-17874	-17873
-14895	-14894
-11916	-11915
-8937	-8936
-5958	-5957
-2979	-2978
0	same
2978	same
5957	same
8936	same
11915	same
14894	same
17873	same
20852	same
23831	same
26810	same
29789	same
32768	same

It looks like every single one of the negative values is off by one! In the third installment I'll dive a little deeper into the semantics of integer division.

Part Two Comments

I am skipping comments about a bug in the C code, which I fixed immediately.

The Alternate Moebyus wrote:

> Out of curiosity, what is sizeof(long)? Haskell defines Data.Bits.bitSize (1::Int) to be 32 somewhere in the standard…

I replied:

> On the platforms I have access to: Mac PowerPC, an embedded PowerPC platform, Win32, and Linux on a Pentium, sizeof(long) is 4 (bytes).

Part Three

In the last installment, I pointed out a discrepancy between the way that Haskell does integer division and the way that some C code does it. Let's look at just

how integer division behaves in Haskell a little more closely.

```
GHCi, version 9.4.8: https://www.haskell.org/ghc/  :? for help
ghci> 6/5
1.2
ghci> 5/6
0.8333333333333334
ghci> (6::Int) `div` (5::Int)
1
ghci> (5::Int) `div` (6::Int)
0
```

Integer division in the positive numbers always rounds down: 1.2 rounds down to 1, and 0.8333… rounds down to 0. In fact, it looks like the fractional part is just truncated. But let's check some negative values to be sure:

```
ghci> -6 / 5
-1.2
ghci> (-6::Int) `div` (5::Int)
-2
ghci> -5 / 6
-0.8333333333333334
ghci> (-5::Int) `div` (6::Int)
-1
```

That's interesting; clearly it is not strictly truncation (rounding towards zero) that is happening, or else -5/6 would give us zero, not -1. And we clearly aren't getting "rounding" in the usual sense of rounding to the nearest integer (although there is no universal rounding algorithm to round to the nearest integer; there are lots of different variations in practice).

It looks like we have floor behavior, which in the case of negative values means rounding away from zero. To state it slightly more rigidly, if the result of the **div** operation is a whole number, we get the whole number, otherwise we get an integer that represents the exact answer minus the fractional part.

If you learned integer division the old-fashioned way, on paper, you know what remainders are. The remainder is the leftover part of the process, and represents the numeator of the fractional part of the quotient. In Haskell, **div** gives you the whole number result of integer division, and **mod** will give you the remainder. So, for example,

```
ghci> (2::Int) `div` (3::Int)
0
ghci> (2::Int) `mod` (3::Int)
2
```

In fact, what this points to is that there is a relationship between the result of **div** and the result of **mod**. It should hold true in all cases, actually; in C, it is part of the ISO standard. The relationship is this:

> For any quotient o = m ‘div‘ n where n is not zero, and the remainder p = m ‘mod‘ n, o * n + p = m.

In other words, you can reverse the integer division by multiplication, as long as you add the remainder back in.

This relationship should be obviously true for any non-negative **n** and **m**. But I'll go further and say that while the exact meaning of **mod** for negative numbers varies from language to language, and perhaps from implementation to implementation, this relationship should still hold true, with the possible exception of cases involving overflow (which I'll discuss further below).

We can represent the relationship in Haskell like this:

```
ghci> 10 `div` 3
3
ghci> 10 `mod` 3
1
ghci> undivide n o p = o * n + p
ghci> undivide 3 (10 `div` 3) (10 `mod` 3)
10
```

In fact, let's test it for a few values of n and m. Copy and paste this definition into GHCi:

```
:{
check_divide m n =
    (undivide n (m `div` n) (m `mod` n)) == m
:}
```

Then this expression:

```
:{
[check_divide m n | m <-
    [(-5::Int)..(5::Int)],
    n <- [(-5::Int)..(-1::Int)]]
:}
```

I get:

```
[True,True,True,True,True,True,True,True,True,True,
True,True,True,True,True,True,True,True,True,True,
True,True,True,True,True,True,True,True,True,True,
True,True,True,True,True,True,True,True,True,True,
True,True,True,True,True,True,True,True,True,True,
True,True,True,True,True]
```

Check it out for yourself! Again, this is not a rigorous proof, but suggestive.

But what about the overflow cases? Let's examine them. If Haskell's **Int** type is a 32-bit signed int, then we should be able to represent the values

−2,147,483,648 (corresponding to 10000000000000000000000000000000 in binary or 0x80000000 in hexadecimal), to +2,147,483,647 (corresponding to 01111111111111111111111111111111 in binary or 0x7FFFFFFF in hexadecimal). Let's ask Haskell what happens when we try to exceed our largest positive number.

Note: these are the results I got when evaluating these expressions in 2006 on a 32-bit PowerPC system; if you're testing with a 64-bit machine, I'll leave it as an exercise for the reader to find the equivalent 64-bit values.

```
ghci> (2147483647::Int) + (1::Int)
-2147483648
```

Yep, we overflow; we get the largest negative number. Conversely, we can underflow:

```
ghci> (-2147483648::Int) - (1::Int)
2147483647
```

We don't get run-time error checking (and probably don't want it, for the sake of efficiency, when using mahine types). Of course, division by zero is still considered a run-time error:

```
ghci> (2147483648::Int) `div` (0::Int)
*** Exception: divide by zero
```

What about overflow or underflow in the results of division? With multiplication, it is very easy to overflow; multiplying two large numbers can easily exceed the width of a 32-bit signed value. You can come pretty close with:

```
ghci> (46341::Int) * (46340::Int)
2147441940
```

because 46340 is the next-lowest integer value to the square root of 2,147,483,647. Generate a result just slightly higher and you will roll over to a large negative:

```
ghci> (46341::Int) * (46341::Int)
-2147479015
```

With multiplication there are lots of combinations of values that will overflow, but with division it is a little bit harder to trigger overflow or underflow. There is one way, though. While 2,147,483,647 divided by any lesser value will be a lesser value, and divided by itself will equal one, recall that twos-complement representation gives us a slightly *asymmetric* set of values: the largest representable negative number is one larger, in absolute magnitude, than the largest representable positive value. Thus:

```
ghci> (214483647::Int) `div` (1::Int)
214483647
ghci> (214483647::Int) `div` (-1::Int)
-214483647
ghci> (-214483648::Int) `div` (1::Int)
```

```
-214483648
```

In 32 bit twos-complement, -2,147,483,648 is the "weird number." As a general rule, if you take a twos-complement number, flip all the bits, and add one, you get the negative of that number. The only exception is the weird number. Let's see what happens when we try to divide the weird number by -1:

```
ghci> (-2147483648::Int) `div` (-1::Int)
```

Now, I would like to tell you what GHCi says when I ask it to divide the weird number by -1. I'd also like to see what it says about taking the weird number **mod** -1. But I can't. GHCi crashes when I type in those expressions.

So, I can't tell you what Haskell says. But I can tell you what the C standard says. Here's a hint: not only is it not defined, but in fact the result of *any* overflow that occurs with arithmetic operations on signed integral types *is not defined* by the language standard. Only the results of arithemtic operations on unsigned integers have defined semantics for overflow, making them portable — assuming that the size of the type in question is the same. More on that next time.

Part Three Comments

docmach wrote:

> What platform did GHCi crash on? I tried **(-2147483648::Int)** `div` **(-1::Int)** and **(-2147483648::Int)** `mod` **(1::Int)** and both produced 0 for me. I'm using Mac OS X 10.4.8 PPC.

I replied:

> Hi docmach,
>
> I am using GHCi 6.6, the binary distribution, on Win32 (Windows XP Pro SP2) on Intel (Pentium 4 3.6 GHz).
>
> I just tested the Sep 2006 version of WinHUGS and it did the same thing.
>
> I will have to try it at home; there I can test it on Windows 2000 and Fedora Core 5 on a different PC, and on a Mac Mini (G4) also running MacOS X 10.4.8.
>
> Interesting that you got zero. The answer for the `div` is +2147483648, but of course that is not representable.

Antti-Juhani Kaijanaho wrote:

> I trust you are aware of the quot-rem pair of functions in Haskell, in addition to the div-mod pair? :)

I replied:

> Hi antti-juhani,
>
> No, I was not! I am very new to Haskell, and just learning. I will look them up.
>
> Thanks for adding my blog to Planet Haskell!
>
> Before anyone asks – I have just filed a bug report on the GHC bug tracker for the "weird number" div -1 crash.

gaurang wrote:

> Hey, I am just a passerby, and came via google searches. Basically I wanted to tell you that I believe that PowerPC processor does not segfault when dividing by zero, but instead gives you some other answer like zero or a very high number (I don't know which). On Intel, the same thing generates an exception, and crashes.
>
> Anyway this is what I beleive right now, am googling for this thing itself, when I found your page.

I replied:

> Hi gaurang,
>
> Thanks for your comment. I discovered the existence of SIGFPE to handle division overflow on Pentium CPUs. I had not previously been aware that overflow could give that result; it seems like a strange choice, since overflow from multiply does not.
>
> I guess it comes down to what guarantees, if any, GHC makes (or can make) about the behavior of the code. Unfortunately I suspect they will probably just decide that this case has to be left undefined, since there probably isn't much they can do that would not ruin the efficiency of the compiled code with runtime checks.

Part Four

Well, this was initially going to be a three-part article, but things have become more interesting. The situation is not unlike picking up a log in the forest and finding all kinds of nifty crawlies living under it. "Interesting" to certain geeky people like myself, at least! It's about to get geekier. If you are allergic to ANSI or ISO standards, you'd better stop reading now. But I claim the payoff is worth it, if you can make it through to the end. At least, for some value of "worth it."

In the last installment we were exploring the behavior of integer division in Haskell. I want to get back to C now. We learned that the behavior of integer

division involving negative results is not strictly defined for C. The behavior of % (mod) is also not very strictly defined.

Let's start with Kernighan and Ritchie's *The C Programming Language, Second Edition* (I'm not even going to *try* to figure out rules for pre-ANSI/ISO C). K&R tell us on p. 10 that

> …integer division *truncates*; any fractional part is discarded.

That isn't very detailed, but on p. 41 they tell us

> the direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as is the action taken on overflow or underflow.

OK, but that doesn't describe exactly what the options are. Things get a little bit more detailed on p. 205, where we read that no guarantees are made about the sign of the remainder!

> …if the second operand is zero, the result is undefined. Otherwise, it is always true that **(a/b)*b + a%b** is equal to **a**. If both operands are non-negative, then the remainder is non-negative and smaller than the divisor; if not, it is guaranteed only that the absolute value of the remainder is smaller than the absolute value of the divisor.

This isn't actually internally consistent, because if the sign of the remainder is not correct, then the identity for division doesn't work for negative values unless the sign of the quotient is allowed to be incorrect! I've never seen it suggested anywhere else that division could be *that* loosely defined. But K&R isn't the formal standard, so let's move on.

The reason for the "looseness" that exists in the C standard, of course, is that the standard was originally written to, as much as possible, codify existing practice, and C has a strong bias towards efficiency. Since (apparently) different hardware division algorithms had different behavior, the standards committee did not feel that it was appropriate to require existing implementations to change behavior. Doing so might have required existing architectures to have to replace hardware division with software division, which could have inflicted an enormous efficiency cost.

According to ISO/IEC 9899:1990 (known as C90):

> When integers are divided and the division is inexact, if both operands are positive the result of the / operator is the largest integer less than the algebraic quotient and the result of the % operator is positive. If either operand is negative, whether the result of the / operator is the largest integer less than or equal to the algebraic quotient or the smallest integer greater than or equal to the algebraic quotient is implementation-defined, as is the sign of the result of the % operator. If the quotient **a/b** is representable, the expression **(a/b)*b + a%b** shall equal **a**.

So we know that there are two options for rounding. In order to maintain the identity involving integer division and mod, though, the value of mod has to be consistent with the rounding strategy, even if the sign does not have to be consistent.

There is a back door that lets us get to defined behavior. Apparently the **div** function *does* have strictly defined rounding. In section 7.10.6.2 of the standard, we read:

> If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient.

That's interesting — this is rounding towards zero. The **div** function returns a structure of type **div_t** that contains both the integer quotient and integer remainder. It is provided in the header, available in C++ as . In his book *The Standard C Library*, P. J. Plauger provides an implementation for **div**. It looks like this:

```
div_t (div)(int numer, int denom)
    {        /* compute int quotient and remainder */
    div_t val;
    val.quot = numer / denom;
    val.rem = numer - denom * val.quot;
    if (val.quot < 0 && 0 < val.rem)
    {  /* fix remainder with wrong sign */
    val.quot += 1;
    val.rem -= denom;
    }

    return (val);
    }
```

We can see what he is doing: for negative quotients that are inexact answers to the division problem, he shifts the quotient up towards zero and the remainder get its sign reversed. Note that he does not use the built-in % facility, presumably since its behavior does not have very strong guarantees placed on it.

C: A Reference Manual, Fifth Edition by Harbison and Steele seems to indicate that the semantics are a little more rigorously defined. We read there that

> For integral operands, if the mathematical quotient of the operands is not an exact integer, then the fractional part is discarded (truncation toward zero). Prior to C99, C implementations could choose to truncate toward or away from zero if either of the operands were negative. The **div** and **ldiv** library functions were always well defined for negative operands.

That seems pretty unambiguous, but then on page 419, when describing the **div** and **ldiv** function, the authors write

> The returned quotient **quot** is the same as **n/d**, and the remainder **rem** is the same as **n%d**.

But that ignores the possibility of a pre-C99 implementation where / rounds away from zero for negative values, as does GHCi in my tests. So again we have a lack of rigorous internal consistency. It is worth noting here that K&R don't make any extra statements about the required rounding behavior of **div** and **ldiv**, and since K&R is still considered the "bible" by many C programmers, the guarantees on the **div** and **ldiv** function may not be very well-known — or properly implemented.

Does C99 clarify this at all? ISO/IEC 9899:1999(E), Second Edition 1999-12-01, tells us that "the result of the / operator is the algebraic quotient with any fractional part discarded" and a footnote indicating that this is often called "truncation towards zero."

How about C++? ISO/IEC 14882, Second Edition (2003-10-15) does not actually say how rounding is performed, although it says that "If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined." There is a footnote indicating that "According to work underway toward the revision of ISO C, the preferred algorithm for integer division follows the rules defined in the ISO Fortran standard" (rounding towards zero). When we try to look up information about **div** and **ldiv**, we find that the C++ standard just refers us to the Standard C library.

OK, all the standards language is giving me a headache; let's take a look at some code. To begin with, let's confirm the way my C implementation does rounding of integer quotients. We took a look at 6/5, -6/5, 5/6, and -5/6 in Haskell; let's look at the same quotients in C:

```
int val_1 = 6;
int val_2 = 5;
int val_3 = -6;
int val_4 = -5
int result_1 = val_1 / val_2;
int result_2 = val_3 / val_2;
int result_3 = val_2 / val_1;
int result_4 = val_4 / val_1;
printf("result_1 = %d, result_2 = %d, result_3 = %d, result_4 = %d\n",
    result_1, result_2, result_3, result_4);
```

This gives me:

```
result_1 = 1, result_2 = -1, result_3 = 0, result_4 = 0
```

Yep, I'm getting rounding towards zero for both positive and negative values. Next, some Haskell. Let's bring back our algorithm from part 2:

```
:{
gen_nths denom =
```

```
    [(numer, denom::Int) |
    numer <- [(-denom::Int)..(denom::Int)]]
div_map m1 m2 =
    map (\tup -> (fst tup + snd tup) *
    (((m2::Int) - (m1::Int)) `div` 2)
    `div` snd tup + m1)
:}
```

Now apply it:

```
ghci> div_map (-10) 10 (gen_nths 3)
[-10,-7,-4,0,3,6,10]
```

And now let's write up my C algorithm again. This time we'll use **int** instead of **long** since we don't need big numbers (although on my target, ints are actually 32 bits long).

```
unsigned int denom = 3;
int numer;
for (numer = -3; numer <= 3; numer++)
{
    int quot = numer * 10 / denom;
    printf("quotient %d\n", quot);
}
```

Go ahead and try it on your platform.

Whoah! What happened? You might have noticed something very odd about the first few quotients. They are monstrously large! On my target I get:

```
quotient 1431655755
quotient 1431655758
quotient 1431655762
quotient 0
quotient 3
quotient 6
quotient 10
```

What happened is that I injected a bug to point out yet another possible way your C code can fail catastrophically. This behavior can surprise even experienced programmers. Note that since my denominator is always a positive value, I declared it as an **unsigned int** instead of an **int**. So what went wrong?

Let's look at the first example: -3/3 yielding 1431655755. The behavior you are looking at is actually mandated by the C standard. When mixing signed and unsigned types, the compiler is required to promote (that is, widen) the actual type of the calculation to an unsigned type. So, internally, if we are performing the calulation -22 (signed) / 7 (unsigned), the compiler is required to notice that both a signed and unsigned long value are presented to the integer division operator. The 32-bit twos-complement representation of -3 is 0xFFFFFFFD

(the conversion to an unsigned intermediate value changes the meaning, not the representation). I multiply this value by 10, yielding 0xFFFFFFE2, and then divide it by 3, yielding 0x5555554B, or a decimal value of 1431655755. The compiler sticks this unsigned value right into the signed result variable. The signed or unsigned status of the destination variable has absolutely no effect; it does not dissuade the compiler from deciding to treat the values as unsigned. (It is a common misconception to think that the destination value influences the type of the calculation in C).

If you have a target with 16-bit ints, your rolled-over values will be different (0xFFFD, 0xFFE2, and 0x554B, or decimal 21835). So the mixing of signed and unsigned values has not only given us the wrong answer, but made the code that generates the wrong answer produce inconsistent results depending on the size of **int**, even when the values we start with are not actually too big to fit into 16 bits.

If that wasn't perfectly clear, and even if it was, if you are really trying to write portable numeric code in C that has to be correct, I urge to consider purchasing the MISRA (Motor Industry Software Reliability Association) book *MISRA-C:2004, Guidelines for the Use of the C language in Critical Systems*. It has the most detailed explanation of C's arithmetic type conversions, and more importantly, the implications of these type conversions in real code. The crux of the biscuit is that C is hard to get right, even for good programmers.

OK, so let's change my denominator back to a signed long and try it again. This time the results look more reasonable: -10, -6, -3, 0, 3, and 6.

Are things any different if we use **div**?

```
int denom = 3;
int numer;
for (numer = -3; numer <= 3; numer++)
{
    div_t val = div(numer * 10, denom);
    printf("quotient %d\n", val.quot);
}
```

In my case, no, since on my compiler and target, `/` already seems to round towards zero (asymmetrically) while Haskell rounds symmetrically downwards (floor behavior).

By the way, while we're at it, let's see what C has to say about the "weird" number divided by -1:

```
long most_negative_long = -2147483648;
long minus_one = -1;
long result = most_negative_long / minus_one;
ldiv_t ldiv_result =
    ldiv(most_negative_long, minus_one);
printf("results: %lX, %lX\n", result, ldiv_result.quot);
```

What do you get? I get zero as the result of both the / and **ldiv**. But I *also* get GCC telling me "warning: this decimal constant is unsigned only in ISO C90."

What does *that* mean? The compiler is apparently warning us that in C99 we won't be getting an implicitly unsigned value out of the constant -2147483648, in case we might have wanted to use this constant in an expression involving signed types to force the calculation to be promoted to an unsigned type. But why would we get an unsigned value in the first place? Apparently in C90, the number's magnitude is interpreted first, and the value is negated. 2147483648 is too large (by one) to fit into a signed long, so the compiler promotes the type to an unsigned long, then negates it.

I have been trying to come up with an example of an expression that behaves differently when I use -2147483648 as a constant, as opposed to 0x80000000, but so far I haven't been able to come up with one. This may be because my compiler is already operating under C99 rules and so I never get the implicit promotion to an unsigned value.

Anyway, be that as it may, some parting advice and conclusions:

1. GHCi (on the PC and Mac) and C (Visual C++ 6.0 and GCC on the PC and Mac) yield distinctly different rounding behavior for integer division. C90 allows this behavior. Does Haskell? Would it make sense for Haskell to define integer division more strictly, the way that C99 does?
2. Division by zero is never OK; neither is taking % of zero. Both are a fatal error in GHCi. The results are undefined in C (but a fatal error is generated by most implementations).
3. Overflow (which, in the case of integer division, occurs only when the "weird number" is divided by -1) produces undefined results, and your code should avoid it, both in Haskell and in C.
4. Operations on unsigned values are guaranteed to produce results that are consistent from platform to platform, assuming the integer size is the same. Operations on signed values don't have this guarantee.
5. The rounding (truncation) of inexact quotients may be either towards zero in all cases, or towards zero for positive quotients and away from zero when the results are negative. If your implementation follows the newer C99 standard rounding should always be towards zero.
6. Mixing signed and unsigned values in C expressions is very dangerous. If you do so, your implementation may have errors far more severe than differences in rounding behavior.
7. Consider avoiding signed division of negative values in C altogether, if your algorithm allows it. Either implement the algorithm using entirely unsigned integral types, or shift the domain of your function so that you are operating on non-negative signed values, and shift it back (using subtraction, which has well-defined semantics) when your division is done.
8. Algorithms which feed a range of values which cross the origin to integer division may vary between GHCi and C implementations. Because GHCi does not provide guaranteed rounding towards zero as C99 and the **div**

and **ldiv** functions require, it is difficult to prototype in GHCi and expect to get the same results in C.

9. And, finally, make sure your implementation actually works the way it is supposed to. Only testing can truly accomplish this.

Interesting, isn't it, how studying one language can enrich your understanding of another! At least, it works for me!

Part Four Comments

The Alternate Moebyus wrote:

> Paul, this is really really useful. Thank you.
>
> It explains the common saying you hear after hanging around people who code numerical stuff: "never, ever, mix unsigned and signed types."

I replied:

> Thanks. They say that all open source software projects start out as someone trying to scratch a personal itch. I generally start writing things like this in an attempt to codify what I've learned from bugs in my own code, in the hopes that someone else will be able to learn something from my process as well.

Carl wrote:

When I try your "weird" number divided by -1 (using the C code from your post), I get "Floating point exception" and a program crash. This is on a Pentium-M; on what platform do you get zero?

I replied:

> Hi Carl,
>
> I have tried the C code on MacOS X 10.4 using the freely available XCode toolchain (which uses GCC).
>
> The C standard says that the behavior of division under these circumstances is undefined. All environments I've worked with produce some kind of a program-terminating crash when dividing by zero, but I generally would not have expected overflow in integer division to produce a crash. It certainly doesn't occur in multiplication or addition which overflows, or subtraction which underflows. However, according to Wikipedia the IA-32 produces an exception when the "weird number" is divided by -1; see
>
> http://en.wikipedia.org/wiki/SIGFPE

> I guess the question then becomes what the OS and/or the program does with that exception. I don't really know what the answers to those questions are for any given OS and application, although it might be interesting to find out.

The Divisive Aftermath

Antti-Juhani pointed out to me that Haskell provides the **quot** and **rem** pair of operations in addition to **div** and **mod**. The difference is that while both satisify the identity involving division, multiplication, and the remainder, **div** and **mod** round down (towards -infinity), while **quot** and **rem** round towards zero (truncating the fractional part).

That's great! (Besides the fact that I displayed my ignorance in public, that is). It means I really can use GHCi to model my functions for testing! And I was getting worked up to write a C version of Plauger's **div** which rounded away from zero for negative quotients so I could have some guaranteed-consistent behavior. Instead I can use the C standard function **div** and Haskell's **quot** and **rem**.

So, this sort of begs the question: is there a good reason to prefer one method of integer division rounding to another? Yes. It comes down to this question: do you want your algorithms to behave *smoothly across* the origin, or to behave *symmetrically around* the origin?

Let's briefly consider some Haskell code for mapping a range of fractions onto a range of integers again.

```
:{
gen_nths denom =
    [(numer, denom::Int) |
    numer <- [(-denom::Int)..(denom::Int)]]
div_map m1 m2 =
    map (\tup -> (fst tup + snd tup) *
    (((m2::Int) - (m1::Int)) `div` 2)
    `div` snd tup + m1)
:}
```

Now apply it:

```
ghci> div_map (-10) 10 (gen_nths 3)
[-10,-7,-4,0,3,6,10]
```

Notice something about those values: the distances between them follow a consistent pattern. Reading the list left to right, we have distances of 3, 3, 4, 3, 3, 4. This pattern repeats. It is *smooth across* the origin.

Now let's make a version of our **div_map** function that uses **quot**:

```
:{
```

```
quot_map m1 m2 =
    map (\tup -> fst tup *
    (((m2::Int) - (m1::Int))
    `quot` 2) `quot` snd tup)
:}
```

Now apply it:

```
ghci> quot_map (-10) 10 (gen_nths 3)
[-10,-6,-3,0,3,6,10]
```

Looking at the differences between values, from left to right, we find that they are 4, 3, 3, 3, 3, 4. The function is not *smooth across* the origin, but *symmetric around* the origin.

Does this matter in practice? Of course! Let's say that we have a mapping using a non-origin-crossing version of our **quot_map** function, and a new function to test it:

```
:{
quot_map_2 m1 m2 =
    map (\tup -> fst tup *
    ((m2::Int) - (m1::Int))
    `quot` snd tup )
gen_non_negative_nths denom = [(numer, denom::Int) |
    numer <- [(0::Int)..(denom::Int)]]
:}
```

Now apply it:

```
ghci> quot_map_2 0 20 (gen_non_negative_nths 6)
[0,3,6,10,13,16,20]
```

Shiny. The intervals go 3, 3, 4, 3, 3, 4. But if we shift the values:

```
:{
gen_non_positive_nths denom = [(numer, denom::Int) |
    numer <- [(-denom::Int)..(0::Int)]]
:}
```

And apply it:

```
ghci> quot_map_2 0 20 (gen_non_positive_nths 6)
[-20,-16,-13,-10,-6,-3,0]
```

The intervals go 4, 3, 3, 4, 3, 3 — they are backwards. In other words, the effect of integer rounding on the values of the codomain changes if you shift the domain of the denominator across the origin: the ordering of the pattern is reversed.

If we do the same thing using div:

```
:{
div_map_2 m1 m2 =
```

```
    map (\ tup -> fst tup *
    ((m2::Int) - (m1::Int)) `div` snd tup)
:}
```

And apply it:

```
ghci> div_map_2 0 20 (gen_non_negative_nths 6)
[0,3,6,10,13,16,20]
ghci> div_map_2 0 20 (gen_non_positive_nths 6)
[-20,-17,-14,-10,-7,-4,0]
```

we get a pattern of intervals (the effect of rounding) which remains the same: 3, 3, 4, 3, 3, 4.

Now, smoothness across the origin is what I want, at least for the kinds of functions I am working on now. But your problem domain may lend itself to writing functions which involve rotations, or mapping values across the origin: in that case, you're going to want the symmetry. The important thing is to know what strategy you are using and apply it consistently. I'm really impressed that Haskell give me so many interesting options.
