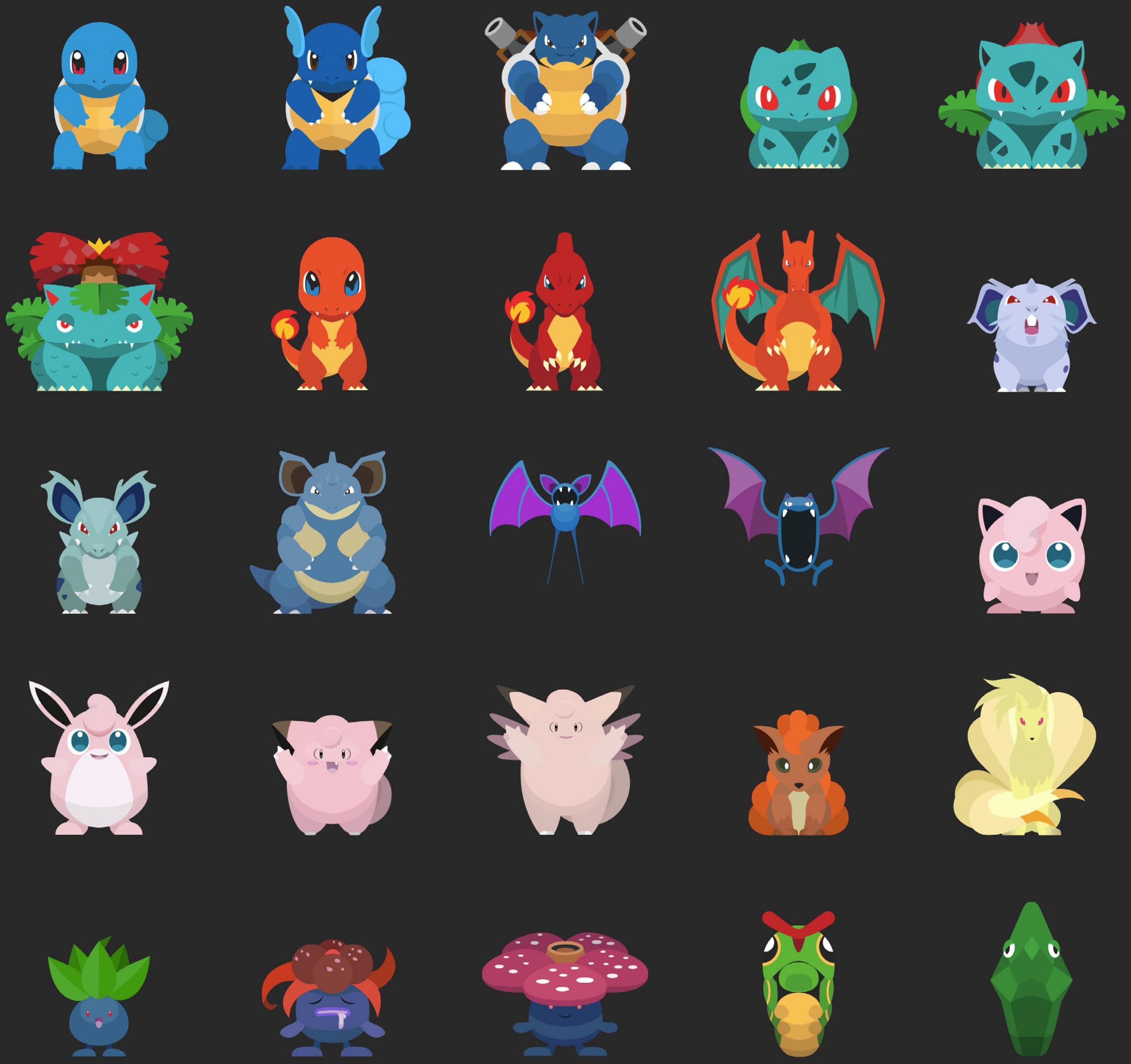


HACKERMONTHLY

Issue 53 October 2014

HACK ON YOUR SEARCH ENGINE

and help change the future of search

duckduckhack.com

Curator
Lim Cheng Soon

Contributors
Hlín Önnudóttir
Brian Guarraci
Andrew Back
Jeffrey P. Bigham
Christoffer Stjernlöf
Gershom Bazerman
Jeff LaMarche
Gustavo Duarte

Illustrator
Thong Le

Proofreaders
Emily Griffin
Sigmarie Soto

Ebook Conversion
Ashish Kumar Jha

Printer
Blurb

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format. For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.

Cover Illustration: Thong Le [weaponix.net]

Hacker Monthly is published by Netizens Media and not affiliated with Y Combinator in any way.

Contents

FEATURES

SPECIAL

PROGRAMMING

For links to Hacker News dicussions, visit *hackermonthly.com/issue-53*

Algorithm for Capturing Pokémon

Gen I Capture Mechanics

By HLÍN ÖNNUDÓTTIR

Nearly two decades after the debut of the Pokémon video game series, the first-generation games — Red, Blue and Yellow — have enjoyed a recent resurgence in popularity thanks to the Twitch Plays Pokémon phenomenon and its derivatives. In the time since their release, the games' source code — originally written in Z80 assembly — has been picked apart by enthusiastic hackers, revealing bit by bit the procedures and algorithms driving the games' mechanics. Here we will take an in-depth look at the inner workings of one of the games' most iconic features: throwing a Poké Ball at a wild Pokémon in the hope of capturing it.

The Algorithm

To determine the outcome of a thrown ball, the game executes the following procedure (cleaned up and tweaked for human presentation, of course). Note that any time it speaks of division, it means *integer division*: the result is an integer and the remainder is simply discarded. Thus, for example, if it says "Divide 5 by 2", the result is simply 2, not 2.5.

1. If the ball being thrown is a Master Ball, the Pokémon is automatically caught. Skip the rest of the procedure.
2. Generate a random number R1, with a range depending on the ball used:
   - If it's a Poké Ball, R1 ranges from 0 to 255 (inclusive).
   - If it's a Great Ball, R1 ranges from 0 to 200 (inclusive).
   - If it's an Ultra or Safari Ball, R1 ranges from 0 to 150 (inclusive).
3. Create a status variable S:
   - If the targeted Pokémon is asleep or frozen, S is 25.
   - If the targeted Pokémon is poisoned, burned or paralyzed, S is 12.
   - Otherwise, S is 0.
4. Subtract S from R1 (to avoid confusion with the original R1, I will refer to the result as R*).
5. If R* is less than zero (i.e., if the generated R1 was less than S), the Pokémon is successfully caught. Skip the rest of the procedure.
6. Calculate the HP factor F:
   1. Multiply the Pokémon's max HP by 255 and store the result in F.
   2. Divide F by
      - 8 if the ball used was a Great Ball.
      - 12 otherwise.
   3. Divide the Pokémon's current HP by 4. If the result is greater than zero, divide F by this number and make that the new F.
   4. If F is now greater than 255, make it 255 instead.
7. If the base catch rate of the Pokémon is less than R*, the Pokémon automatically breaks free. Skip to step 10.
8. Generate a second random number R2 ranging from 0 to 255 (inclusive).
9. If R2 is less than or equal to the HP factor F, the Pokémon is caught. Skip the rest of the procedure.
10. The capture fails. Determine the appropriate animation to show:
    1. Multiply the Pokémon's base catch rate by 100 and store the result in a wobble approximation variable W.
    2. Divide W by a number depending on the ball used, rounding the result down:
       - If it was a Poké Ball, divide by 255.
       - If it was a Great Ball, divide by 200.
       - If it was an Ultra or Safari Ball, divide by 150.
    3. If the result is greater than 255, the ball will wobble 3 times; skip the rest of this subprocedure.
    4. Multiply W by F (the HP factor calculated above).
    5. Divide W by 255.
    6. Add a number if the Pokémon has a status affliction:
       - If the Pokémon is asleep or frozen, add 10 to W.
       - If the Pokémon is poisoned, burned or paralyzed, add 5 to W.

7. Show the animation and message corresponding to W:
    - If W is less than 10, the ball misses ("The ball missed the Pokémon!").
    - If W is between 10 and 29 (inclusive), the ball wobbles once ("Darn! The Pokémon broke free!").
    - If W is between 30 and 69 (inclusive), the ball wobbles twice ("Aww! It appeared to be caught!").
    - Otherwise (if W is greater than or equal to 70), the ball wobbles three times ("Shoot! It was so close too!").

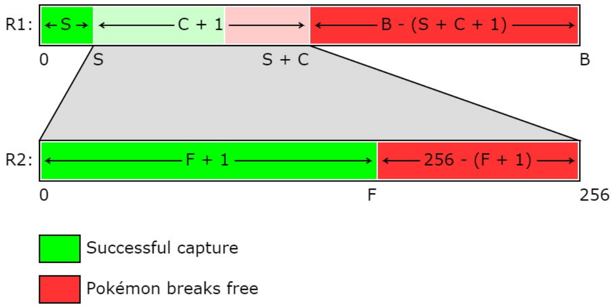
What It Means

So what, indeed, does this algorithm actually mean for capturing in R/B/Y? How is it different from the later games? In the rest of this discussion, I will assume you are not using a Master Ball; that case is trivial and it would be a bother to keep mentioning it.

Well, first, we can derive a formula for the odds of a successful capture from the algorithm. (This gets a bit mathematical, but bear with me.) Let's call the Pokémon's base catch rate C and create a ball modifier variable B to stand for the range of R1: 256 if the ball is a Poké Ball, 201 if the ball is a Great Ball and 151 if the ball is an Ultra Ball. (Note that this is one higher than the maximum numbers discussed in the algorithm above; this is because a random number between 0 and X inclusive can take on X+1 possible values.) Depending on where R1 falls in this range, three different things can happen:

- If R1 is less than the status variable S, the Pokémon is immediately caught.
- If R1 is greater than or equal to S, but less than or equal to S + C, the game calculates a second random number R2 between 0 and 255 inclusive and an HP factor F. These numbers are then compared:
    - If R2 is less than or equal to F, the Pokémon is caught.
    - If R2 is greater than F, the Pokémon breaks free.
- If R1 is greater than S + C, the Pokémon breaks free.

Of the total of B possible R1 values, S of them result in the first path (auto-capture), C + 1 of them (up to a maximum of B - S, since there are only B - S values left) result in the second path where R2 and the HP factor F are calculated, and the rest (if any) result in the third path (auto-failure). We can visualize it like this:



Now getting the formula is simple. What we want is the sum of the two possible paths leading to the Pokémon being successfully caught: first, the case where R1 is within the auto-capture range (the chance of which is S / B, since S out of B possible values of R1 give us that result), and second, the case where R1 is within that lighter range in the middle (the chance of which is min(C + 1, B - S) / B) *and* subsequently R2 <= F (the chance of which is (F + 1) / 256 - it's F + 1 because we're counting the R2 values from 0 to F *inclusive*). This directly gives us the following:

$$Chance = \frac{S}{B} + \frac{\min\left(C+1, B-S\right)}{B} \cdot \frac{F+1}{256}$$

Note that the game is not actually *performing* any of these mathematical operations; the only actual arithmetic it's doing is within the F variable, while this is a probabilistic formula derived from the structure of the algorithm. That means *these* divisions are not integer divisions, and this formula can be rearranged at will without producing rounding errors or the like. Thus we could, for instance, combine those divisions by B and get this equivalent, perhaps somewhat cleaner formula:

$$Chance = \frac{S + \min\left(C+1, B-S\right) \cdot \frac{F+1}{256}}{B}$$

You'll see that version again a bit later, but I'm sticking with the former version for the explanation, as I believe it both reflects the structure of the algorithm better and will illustrate some aspects of its behaviour more clearly. So here it is again with the full formula for F included:

$$Chance = \frac{S}{B} + \frac{\min\left(C+1, B-S\right)}{B} \cdot \frac{\min\left(255, \left\lfloor \frac{\left\lfloor \frac{M \cdot 255}{G} \right\rfloor}{\max\left(1, \left\lfloor \frac{H}{4} \right\rfloor\right)} \right\rfloor\right) + 1}{256}$$

You probably still don't have a very good idea what any of this really means, but that's okay; we're getting to that. Let's go over the values in the formula and how they ultimately affect it.

S (Status)

This is a simple variable: it is 25 if the Pokémon you're trying to catch is asleep or frozen, 12 if it's poisoned, burned or paralyzed, and 0 otherwise.

Unlike the formulas used in the more recent Pokémon games, the status is factored in not as a multiplier but an addition. This leads to two interesting conclusions. First, it means that status conditions essentially give a certain baseline chance of capturing the Pokémon, equal to S/B — regardless of the Pokémon's catch rate and HP, your final chance of capturing it will always be *at least* that baseline. Second, addition has a proportionally greater influence the *smaller* the original value is — think of how if you add 100 to 100, you're doubling it, whereas if you add 100 to 1 million, the change is barely worth mentioning. This means that status provides only a modest improvement to the odds of a successful capture if your chances are already pretty good, but makes a *massive* difference when you're trying to catch a more elusive Pokémon.

C (Capture Rate)

This is simply the base catch rate of the Pokémon species, ranging from 3 (for legendary Pokémon) to 255 (for common Pokémon like Caterpie and Pidgey).

If you look back at the visualization image above, the capture rate's role in the algorithm is to determine the size of the lighter part of the top bar: out of the B possible values R1 can take, C + 1 of them (or more accurately, min(C + 1, B - S), since the capture rate window obviously can't be bigger than all the B values that aren't in the status auto-capture window) will lead to the HP factor check happening. All values of R1 that don't fall either within the status auto-capture window or the capture rate window are auto-failures. For a legendary (with capture rate 3), there are therefore always only four (3 + 1) possible R1 values for which the game will look at the legendary's HP at all, for instance.

That capping when the capture rate is greater than B - S has some interesting consequences, which I'll go into better in a bit.

B (Ball Modifier) and G (Great Ball Modifier)

Quite unlike the ball bonus multiplier of the later games, there are two ball-related modifiers in the R/B/Y formula, both of which are primarily divisors, meaning a *lower* value for them means a *higher* chance of a successful capture. In the later games, a Poké Ball has a ball bonus multiplier of 1, with a Great Ball having a ball bonus of 1.5 and an Ultra Ball having a ball bonus of 2, forming a straightforward linear progression from worse to better balls. In R/B/Y, however, the ball modifier B is 256 for Poké Balls, 201 for Great Balls and 151 for Ultra Balls and Safari Balls, and furthermore the G value in the calculation of the HP factor (the F variable) is 8 for Great Balls but 12 for all other balls.

This makes it a little harder to see at a glance just how much more effective the better balls actually are than plain Poké Balls in R/B/Y, but let's take a good look at the formula again.

$$Chance = \frac{S}{B} + \frac{\min\left(C+1, B-S\right)}{B} \cdot \frac{F+1}{256}$$

The addition splits it into two parts: the status check (S/B), and the HP factor check (the rest). The effect of balls on the status check is simple to calculate. For a Poké Ball the status check becomes S/256, for a Great Ball it's S/201, and for an Ultra or Safari Ball it's S/151. This means that compared to a Poké Ball, a Great Ball effectively gives a 256/201 = 1.27 multiplier to the status check, and an Ultra Ball gives a 256/151 = 1.70 multiplier to it.

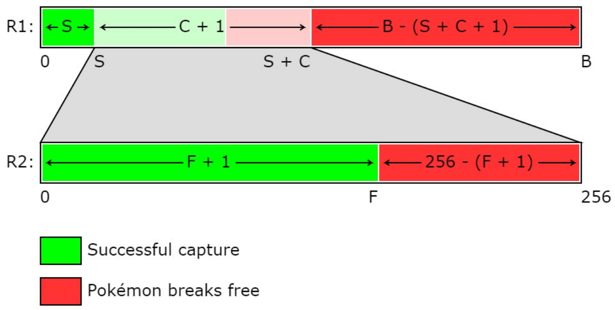
The HP factor check is a bit more complicated, since in addition to a division by B, the min(C + 1, B - S) part means it will sometimes also contain B as a multiplier instead of a divisor, and the HP factor itself contains the G value within some roundings and a cap of 255. If we ignore rounding errors and the +1, assume that C + 1 is less than B - S and the overall cap of 255 on the HP factor doesn't get tripped, however, then both B and G are basically divisors on the entire outcome, so we can compare them in a similar way as above. This tells us that compared to a Poké Ball, a Great Ball effectively yields a multiplier of around 256*12 / 201*8 = 1.91 to the HP factor check, and an Ultra Ball yields a multiplier of around 256*12 / 151*12 = 1.70.

Yes, you read that right: Great Balls actually give a *higher* boost to the HP factor check than Ultra Balls, provided our assumptions hold. Ultra Balls are still better for the status check, but if you are for instance trying to catch a Pokémon with no status condition (in which case the status check is always going to be zero), then Great Balls are actually, honest-to-God better according to this. Who would've thought?

So, when *do* those simplifying assumptions actually hold, and what happens when they don't, anyway? Well, the min(C + 1, B - S) means that when the base catch rate of the Pokémon is sufficiently high, the HP factor check formula turns into this:

$$Chance_{HP} = \frac{B-S}{B}\cdot\frac{F+1}{256} = \left(1-\frac{S}{B}\right)\cdot\frac{F+1}{256}$$

Suddenly B has vanished entirely as a divisor except in the *subtraction* of S/B. Under these circumstances, even the advantage the Ultra Ball did have thanks to its lower B value is completely canceled out, and in fact the Great Ball's disadvantage when it comes to status starts to be partly mitigated. To intuitively grasp what's going on here, let's look yet again at the visualization from earlier:



The B value determines the size of the top bar. A lower B value generally helps by making the red auto-failure portion on the right side smaller and thus also making the status auto-capture window and the capture rate window proportionally bigger. However, if the C value is so large that there are no auto-failure values on the right side either way, then a lower B just means the status auto-capture window, if any, becomes proportionally bigger at the expense of the capture rate window, where you would have had *some* chance to capture the Pokémon anyway — still an improvement, but not as much of one. And if there is no status, then so long as B is less than or equal to C + 1, the entire top bar is taken up by the capture rate window — beyond that point, a lower B won't do anything at all.

All in all, this means that the Great Ball has an *even greater* advantage over the Ultra Ball for Pokémon with high catch rates, starting at 150 and maximizing at 200+, by which time the HP factor check will be about 33% more likely to succeed for a Great Ball than an Ultra Ball.

Don't count your chickens just yet, though. The other major simplifying assumption we've been making in calculating the balls' relative strengths was ignoring the cap on the HP factor, and this will turn out to be quite significant as well. More on that in the HP factor section below.

F (HP Factor)

This is where the current health of the Pokémon you're trying to catch comes in, making it easier to capture a Pokémon that has been weakened. To recap, the F value is given by the following formula:

$$F = \min\left(255, \left\lfloor \frac{\left\lfloor \frac{M\cdot 255}{G}\right\rfloor}{\max\left(1, \left\lfloor \frac{H}{4}\right\rfloor\right)}\right\rfloor\right)$$

where M stands for the Pokémon's maximum HP, H stands for the Pokémon's current HP, G stands for the Great Ball modifier discussed above (8 for Great Balls, 12 otherwise), and ⌊x⌋ stands for rounding x down to the nearest integer. A higher F value means a greater chance of a successful capture.

First things first, let's see what sort of value is going to come out of this for both possible G values. If we plug in H = M to represent a full-health Pokémon and ignore rounding errors, we get...

$$F = \min\left(255, \left\lfloor \frac{\left\lfloor \frac{M\cdot 255}{12}\right\rfloor}{\max\left(1, \left\lfloor \frac{H}{4}\right\rfloor\right)}\right\rfloor\right) \simeq \frac{M\cdot\frac{255}{12}}{\frac{M}{4}} = 255\cdot\frac{4}{12} = \frac{255}{3} = 85$$

$$F = \min\left(255, \left\lfloor \frac{\left\lfloor \frac{M\cdot 255}{8}\right\rfloor}{\max\left(1, \left\lfloor \frac{H}{4}\right\rfloor\right)}\right\rfloor\right) \simeq \frac{M\cdot\frac{255}{8}}{\frac{M}{4}} = 255\cdot\frac{4}{8} = \frac{255}{2} = 127.5$$

Yet again, incredibly enough, the Great Ball is actually better than not just the Poké Ball but also the Ultra and Safari Balls, at least for full-health Pokémon.

So what happens as you lower the Pokémon's HP? It's not hard to see that the basic interaction between the Pokémon's maximum and current HP involved here is M/H (M appears above the line in the fraction while H appears below the line), which at a glance makes sense: at full HP you'll get 1 out of that, and then as H drops, the value will rise. (This means the above full-health values are the *minimum* possible values the HP factor can take for their respective balls.) But if left unchecked, it would just rise ever faster and faster, which would lead to absurd results — the value for 4 HP would be twice as high as for 8 HP. That's where the HP factor cap comes in: once you've whittled the Pokémon's HP down enough to make the HP factor value 255, lowering its HP more will no longer have any effect upon its catch rate. Given the full-health values above, this means lowering the Pokémon's HP can at most double your chances for the HP factor check if you're using a Great Ball, or triple them when you're using another ball - which in turn means that despite the Great Ball's initial advantage, the Ultra Ball can catch up, what with the maximum being the same for both of them.

So, just when is this cap tripped? Again, if we ignore rounding errors, we can calculate that easily:

$$255 = \frac{M \cdot 255}{G} \Big/ \frac{H}{4} \Leftrightarrow 1 = \frac{M}{G} \Big/ \frac{H}{4} \Leftrightarrow H = \frac{4}{G} \cdot M$$

which means the HP factor caps when the Pokémon is at roughly 1/2 (4/8) of its total HP for Great Balls, or around 1/3 (4/12) of its total HP for all other balls.

...*Wait*, you say. *Did you just say what I think you just said?*

Yes, I'm afraid so. With or without status effects, regardless of catch rates, *lowering your Pokémon's HP below one third, or one half if you're using a Great Ball, does absolutely nothing to help you catch it*. Continuing to painstakingly deal tiny slivers of damage beyond that is simply a waste of your time and effort.

For a low-health Pokémon whose HP factor has capped, the final capture formula (regardless of the ball used) simplifies beautifully to:

$$Chance_{LowHP} = \frac{S + \min\left(C + 1, B - S\right)}{B}$$

W (Wobble Approximation)

But what about the wobbling? What's all that weird calculation the game is doing just to figure out how many times the ball is going to wobble?

Well. Let's analyze just what the game is doing there, shall we? Most of the variables involved in the wobble calculation are the same or basically the same as in the actual capture formula, so I'll use the same variable names, but since the status factor in the wobble formula is *not* the same (it's 10 for sleep/freezing and 5 for poisoning/burns/paralysis instead of 25 and 12 respectively) I'll call that S2. With this in mind, here's the formula the game is evaluating:

$$W = \left\lfloor \frac{\left\lfloor \frac{C \cdot 100}{B-1} \right\rfloor \cdot F}{255} \right\rfloor + S_2$$

Hmm. Doesn't this formula look just the slightest bit familiar? No? How about if we ignore the roundings and rearrange it just a bit...

$$W = 100 \cdot \frac{\frac{(B-1) \cdot S_2}{100} + C \cdot \frac{F}{255}}{B - 1}$$

...and note that for Poké Balls in particular, $\frac{(B-1) \cdot S_2}{100}$ gives a result uncannily close to the S variable from the success formula...

$$W = 100 \cdot \frac{S^{*} + C \cdot \frac{F}{255}}{B - 1}$$

...doesn't it look just a little bit like simply a hundred times a variation of another formula we know, with some off-by-one errors and a cap removed?

$$Chance = \frac{S + \min\left(C + 1, B - S\right) \cdot \frac{F+1}{256}}{B}$$

Now, I can't claim I know what Game Freak were thinking when they programmed this calculation. But I would bet money that what the game is trying to do here with the wobbles is to *calculate a percentage approximation of its own success rate*. If the ball misses, the game estimates your chances are less than 10%, whereas if it wobbles once it's guessing 10-29%, twice means 30-69% and three times means 70% or more.

The main factors to introduce serious errors in this approximation are, firstly, that they failed to account for the status bonus in the real formula being affected by the ball modifier (as noted above, ((B - 1) * S2)/100 only approximates S for Poké Balls); secondly, the one that should be added to the C value (which has some significance for very low catch rates); and thirdly, the C + 1 value potentially hitting a cap. What were the programmers thinking? I'm rather inclined to think it was simply an honest oversight. That or, more charitably, they didn't want to waste the additional overhead it would take to be more accurate about it for something so insignificant.

Notice anything interesting? This is a completely static calculation with no random factor to it whatsoever, and that means (unlike the later games) *the number of wobbles in R/B/Y is always the same given the Pokémon's HP and status and the type of ball*. Thus, if you're throwing balls at something and you've seen it break out on the first wobble, you can be certain you're catching it for real the moment you see the ball start to wobble a second time, unless its HP or status were changed in between.

In Plain English

Confused by all the math talk? All right; here's a plain, summarized, as-few-numbers-as-possible version of the unexpected conclusions of the algorithm. Remember, this is all stuff that applies to *R/B/Y only*; it does *not* work this way in the later Pokémon games.

First of all, *regardless of anything else, if the targeted Pokémon has a status affliction, you get a set extra chance to capture it depending on the status and the Pokéball you're using, before the game even starts checking the Pokémon's HP and whatnot*. These chances are listed in the following table:

Ball	PSN/PAR/BRN	SLP/FRZ
Poké Ball	12/256 = 4.69%	25/256 = 9.77%
Great Ball	12/201 = 5.97%	25/201 = 12.44%
Ultra Ball	12/151 = 7.95%	25/151 = 16.56%

Your overall chance of capturing the Pokémon if it has a status affliction will never be less than the chance stated above, even if it's a legendary at full health — this is a check the game applies before it even looks at the HP or catch rate, after all. This makes status by far the most viable way of increasing your chances of getting a legendary — the improvements to be made by lowering their HP are frankly negligible in comparison. (The chance of catching an average full-HP sleeping Mewtwo in an Ultra Ball is 17.45%; the chance of catching an average 1-HP sleeping Mewtwo in an Ultra Ball is 19.21%.)

Second of all, *lowering a Pokémon's HP below one third has no effect at all on its catch rate*. If you're using Great Balls, in fact, that cutoff point is at one half rather than one third. Painstakingly shaving off slivers until the HP bar is one pixel wide is and always has been a waste of time in R/B/Y. When a Pokémon is down to the cutoff point, you are twice as likely to capture it as if it were at full health if you're using a Great Ball, and three times as likely if you're using any other ball.

Third, *Great Balls are actually better than Ultra Balls for Pokémon with no status afflictions or that have an intrinsic catch rate of 200+*, provided they're at around half of their health or more. The difference is slight (but still there) for non-statused Pokémon with low catch rates, such as legendaries, but at high catch rates, Great Balls are very noticeably superior to Ultra Balls, even with status afflictions. Ultra Balls, meanwhile, truly shine at catching statused Pokémon with low intrinsic catch rates, and for low-health Pokémon they're always at least as good as Great Balls.

Fourth, *wobbles are a loose indicator of the game's approximation of your chances of capturing the Pokémon at the current status and HP with the current ball*. This approximation can be significantly flawed, especially when status or high catch rates are involved, but I would guess accuracy wasn't particularly a priority for the programmers, considering it's just determining how many times you'll see a ball wobble on the screen before a breakout. Roughly:

Wobbles	Message	Approximated chance of capture
0	"The ball missed the Pokémon!"	< 10%
1	"Darn! The Pokémon broke free!"	10-30%
2	"Aww! It appeared to be caught!"	30-70%
3	"Shoot! It was so close too!"	>= 70%

Check out the catch rate calculator for the first-generation games here: *hn.my/catchrate*

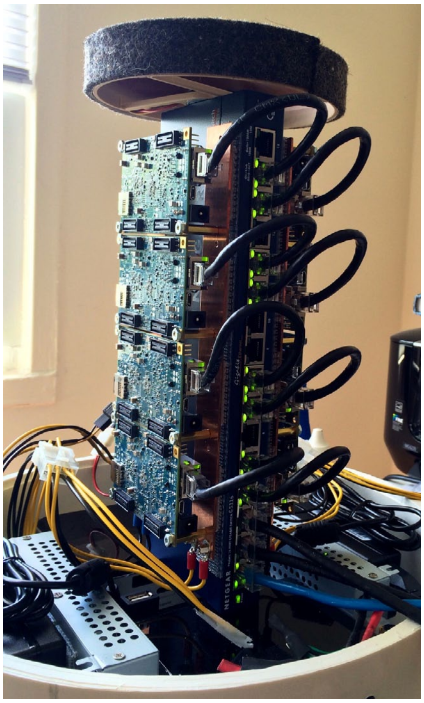
Hlín Önnudóttir is an Icelandic programmer with a bachelor's degree in computer science and has been a fan of the Pokémon series since she was ten. In her free time she runs a website dedicated to the series at *dragonflycave.com*

Reprinted with permission of the original author. First appeared in *hn.my/pokemon* (dragonflycave.com)

My Name is Brian and I Build Supercomputers in My Spare Time

Interviewed by ANDREW BACK

Brian Guarraci is a software engineer at Twitter and in his spare time he's building a Parallella cluster with a design that was inspired by two of the most iconic supercomputers ever made.

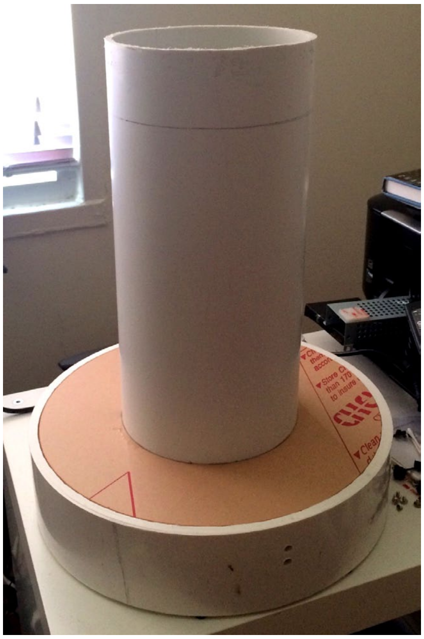


When we saw pictures of Brian's cluster, we were impressed, and when we shared these with the community, it became apparent that we were not the only ones! It didn't take long before curiosity got the better of me and I decided to get in touch with Brian to find out more.

Hi, Brian. Can you tell me about the Parallella cluster you are building?

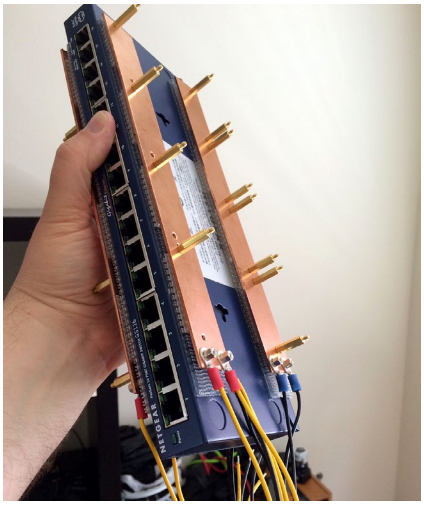
I'm building a low-power general purpose compute cluster. I want it to be able to take advantage of standard distributed system packages so that there's a familiar developer model. The Parallella boards are great for computation, but since they have relatively limited storage and memory, I added two Intel NUCs. Each NUC has 1x Intel i3, 16GB RAM, 120GB SSD, 802.11ac WiFi and are also pretty low-power. The NUCs run Ubuntu server and are storage hosts and the primary interface to the external world. The system has 8x Parallella boards and a shared gigabit Ethernet switch, giving a peak performance of around 208 GFLOPs.

The physical assembly is inspired by the Cray-1 and Connection Machine supercomputers, also the new Mac Pro. It is 16“ high, with a 12” diameter and 3" high base. Starting with the original motivation of creating a cooling tower for the 8 Parallella boards (Mac Pro style), the design expanded to also include power transformers, the two NUCs, a gigabit Ethernet switch, an Arduino, and LED strips. In the spirit of the Connection Machine (CM-5), there will be 8x Adafruit Neopixel LED strips mounted on the outside of the tower and each with 16 LEDs, which will show the status of the Parallella compute nodes.

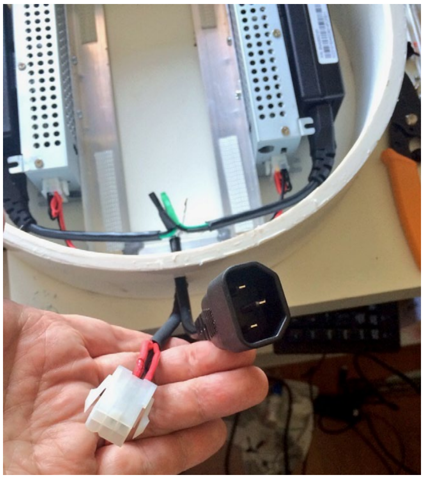


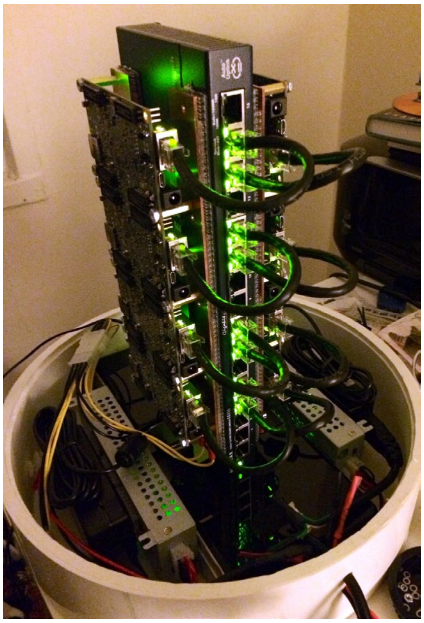
Power-wise, the system has 3 transformers: the two stock NUC 19V power supplies and an external 180W 12V power supply for everything else. The entire expected power consumption is about 120W on average: 30W * 2 (NUC) + 5W * 8 (Parallella) + 20W (LEDs).

A 12" x 3" PVC tube is used to form the base. The tube has a milled edge to support the laser-cut disc that is the tower base. The tower is a 13" x 6" PVC tube with a 140MM fan mounted at the top. Aluminum rails are mounted inside both the base and tower using high-temperature epoxy. The base itself includes the two stock AC-19V NUC transformers, the NUCs themselves, and 2 19v-5V DC-DC converters providing power for the Parallella boards and LEDs.



I spent a lot of time figuring out how best to mount the Parallella boards while satisfying the space and airflow requirements. The solution I ended up with was to use a gigabit Ethernet switch as a kind of "spine." In doing so, it was easy to minimize wire lengths, provide power, and simplify mounting. On each side of the switch are 4x Parallellas mounted to oxygen-free copper bus bars. The bus bars are then mounted to the switch using 3M industrial Velcro, which is easy to work with, very strong, and serves to insulate the bus bars from the metal switch case. The bus bars are subsequently connected to the base power supplies via standard Molex connectors.





Although building a system with a round case is much harder than a rectangular system, it is definitely more satisfying. To my surprise, when I first fired up the tower I was amazed how awesome it looked and started to think about using a clear acrylic tower tube instead.

Will the cluster have any custom hardware extensions or modifications?

Not at the moment. I think that the possibility of removing the HDMI controller from the FPGA and instead having custom logic for specialized hardware compute support is interesting, though.

What applications do you have in mind?

My current plan is to build a familiar compute cluster setup with host HDFS, Redis, and ZooKeeper on the NUCs, and then farm out tasks to the Parallella boards. I think there's a lot of opportunity to try different styles of parallel computing in the system, so I will use it as a platform for experimenting. Practically speaking, I will use the system for machine learning and Hadoop style tasks.

Why is parallel computing important?

For the most part, CPUs aren't getting faster, and the best way to get more done is to compute in parallel. One of the things that's cool about a Parallella cluster is that you can perform multiple heterogeneous parallel computing tasks all at the same time. Some nodes can be doing image processing while other nodes are doing machine learning. It's a very flexible setup.

How can people follow the progress of your project?

I call this machine the Parallac, and I'm planning on documenting any specific details at *parallac.org* ■

Andrew Back is Parallella's Community Manager. He leads the UK Open Source Hardware group and is a former Open Source evangelist for BT.

Reprinted with permission of the original author.
First appeared in *hn.my/briansuper* (parallella.org)

My Half Workday as a Turker

By JEFFREY P. BIGHAM

Last Tuesday, I spent 4 hours, half a work day, as a Mechanical Turk worker.

Over the past 6-7 years, I've spent thousands of dollars as a requester on Mechanical Turk. I've also done some small-scale turking myself. My lifetime earnings were only around $25 as of that morning. While not much money, this almost certainly puts me near the higher end of researchers who do work on Mechanical Turk.

I conducted a very informal poll during a talk at the CVPR (top computer vision conference) human computation workshop last week. Almost everyone there had used Mechanical Turk, and about half had even done a job on Mechanical Turk. When I asked how many had earned $2, most hands went down, a couple remained at $5, and none were still up at $10. I've worked on Mechanical Turk for a variety of reasons, but mostly to get a better sense of how Mechanical Turk works and understand how I could best leverage it.

I think this has worked out. For a while now, our HITs have included clear indications of what we expected the hourly wages of workers to be. We've given incremental feedback on our HITs, allowed workers to keep working when we still had more for them to do, without bouncing them back through the dreadful Mechanical Turk interface to continue.

When I taught Crowd Programming [programthecrowd.com] this past Spring at CMU, the first assignment for my class was to earn $2 on Mechanical Turk. This turned out to be surprisingly difficult, time-consuming, and frustrating for some of the top undergraduate students in the world (by at least one metric, they're destined for the highest average starting pay in a couple years). I think it moves the conversion in a positive direction. Students are less likely to refer to turkers as "lazy" and more likely to think about the usability and transparency of their tasks.

I wanted to go a step further and experience what it might be like if I expected to earn a living from Amazon Mechanical Turk. This report details my experience. I don't necessarily think any of this is new. I would even venture to guess that it's all contained on the forums at *turkopticon.com* or *turkernation.com*, and to some extent in academic papers.

My Experience

I started out pretty optimistically. I'm a pretty smart guy, a pretty computer-savvy guy, and I have a fair amount of experience around crowdsourcing. I also have had a handful of really good experiences on Mechanical Turk that I thought I might be able to generalize. For instance, I once was paid ~$5 for writing a 500-word article about Cisco Networking Certification — I don't know much about that, but I leveraged my experience as a professor to wax on about it for 500 words anyway.

Hour One

Because of my positive experience bullshitting an essay, I started out looking for HITs that paid more than $1.00 and had the word "write" in them.

I found a handful of HITs that I was qualified for, but none were offering $5. I found one that offered $2 but wanted me to write about a golf course I knew nothing about, and it also wanted me to upload a document about it to Dropbox and then paste the link into the form field on Mechanical Turk. That sounded complicated. I figured by the time I did actual research and uploaded the document, it wouldn't possibly be worth my time for only $2. Looking back, I'm not sure if I was right about that.

I found a job that was about editing, which is close to writing, so I took that one. It pointed me at a 2.5 page Google Doc, which I was supposed to edit. The instructions said to mark my edits in red type, which turned out to be really awkward (time-consuming) to do. The document was about how to run some Mormon fellowship meeting, and had a bunch of errors in it. I made a **whole bunch** of changes, which took me maybe 10 minutes. Only after I clicked submit did I remember that it only paid $0.25. I had accidentally become an eager beaver.

Not looking good so far!

Then I "read and responded to messages," which was a survey for $1.01. This advertised 5-10 minutes and took me 8 minutes. That's actually pretty good.

Next, I summarized a couple of linked articles in 50 words or more. These paid $0.35. Some of these articles didn't even have 50 words in them to start with, so that required me to embellish a little. I installed a "Word Count" Chrome extension so I would know how many words I had typed.

I wrote a few 350+ word articles for $1 each about "Home Remodelling Ideas," "Owning a Pet," and "Travel Insurance" to round out the hour. These were fine. I made more than $6/hr while working on them, although I was working pretty hard at it to get that much typing done in just 10 minutes.

Hour 2

In the next hour, I started off answering 10 really long and confusing questions about web sites and how and where they were ranked on Google. I had to follow many of the links, and fill in about 40-50 answers in total. It was really cognitively taxing to switch between the various browser windows, select and copy-paste information, and understand the instructions. I thought $1.00 would be a good pay rate for this when I started, but in the end I made less than $6/hr on the task and I was trying pretty hard.

I realize now that it really bugged me when I didn't understand the point of a task, or why someone was having a person do it instead of scripting it themselves. This task in particular seems like it would be a great candidate for writing a simple little browser script to harvest the information they were interested in. If I had some confidence that this HIT would be around for very long, I might even write that script myself.

After that, I did a study about social networks from Stanford University. The study only paid $0.30, which I was pretty down about because it took ~10 minutes, but then this morning I got an email saying that I had received a $1.65 bonus from them. That makes it one of the better-paying tasks. I can't say that I read the instructions too carefully (who has the time?), but I don't remember seeing anything about a bonus. Had I seen the notice about the bonus, then I might have paid more attention… although maybe not.

Then I did a few HITs transcribing text from images (human OCR). I was only getting paid $0.01 for doing these tasks, but the HITs promised me a bonus if I did a good job. I got $0.07 late last night. Turns out that's not nearly enough to make it worth it. To their credit, this was the first task that I saw that gave me clear feedback and incremental updates on how much I would make.

Then I wrote an outline for an article for $0.20. I participated in an economic experiment for $0.10. The HIT was pretty confusing. I didn't actually have to do anything, but it looked like someone they're going to randomly pair me with can decide to give me more money if they want. Three days later I received a bonus of $0.45. I guess I got paired with a sucker.

Hour 3

At hour three, I started sorting by least number of HITs. My goal here was to find academic studies because I thought maybe they would pay reasonably, and probably wouldn't check too carefully whether the HIT was done perfectly. It wasn't clear how to search for these generically, so I eventually started searching for university names. Most of what came up were surveys, though, and almost all of them paid extremely poorly.

I started doing a survey and after about 4 minutes it kicked me out saying that I didn't qualify. It warned me not to submit the HIT because if I did so I would be rejected. However, I was pretty mad about this, and so I submitted the HIT anyway with the survey code: "I did 4 minutes of the survey before it told me that I didn't qualify. I expect you to approve this HIT." I wasted some time and found that this survey was posted by a somewhat shady looking market research company headed by a guy named Dmitry. At the time, I was hoping Dmitry would reject my HIT so I could go on a tirade about how unfair it was, and try to get him banned from MTurk or whatever. Turns out he actually accepted my HIT and I earned $1.00 for those 4 minutes. However, I suspect many workers wouldn't risk it.

Hour 4

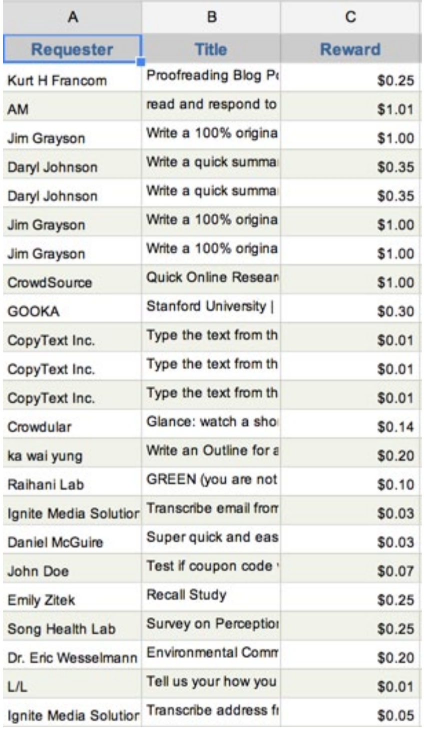
At some point I got super frustrated with all the broken or misleading HITs. I found a video-labeling task from the MIT Media Lab that paid $0.07 that I could fairly reliably do in ~30-35 seconds, and just kept doing that one. I did almost exclusively that task for the last hour or so.

For most of hour 4 I just did the video labeling HITs. I was too tired to go looking for something better. And, I got pretty good at doing them quickly. Unfortunately, as of almost 2 days later I've yet to actually be paid for doing these HITs, although I also haven't been rejected.

Discussion

In the end, I submitted HITs totaling $17.29, and received $2.17 in bonuses. This works out to an effective hourly wage of $4.87. As of one week later, none of my HITs have been rejected, but only about half have been accepted. I've only received $8.50 of the $17.29 (~49%).

I can't decide if my hourly wage is good or bad. Also, while I think I did a pretty good job, some of my work very well might be rejected anyway. I had expected Mr. Dmitry to reject my survey (on which I just entered “I did 4 minutes of the survey before it told me that I didn't qualify. I expect you to approve this HIT.” as my survey code), but he didn't.



A	B	C
Requester	Title	Reward
Kurt H Francom	Proofreading Blog Po	$0.25
AM	read and respond to	$1.01
Jim Grayson	Write a 100% origina	$1.00
Daryl Johnson	Write a quick summa	$0.35
Daryl Johnson	Write a quick summa	$0.35
Jim Grayson	Write a 100% origina	$1.00
Jim Grayson	Write a 100% origina	$1.00
CrowdSource	Quick Online Resear	$1.00
GOOKA	Stanford University |	$0.30
CopyText Inc.	Type the text from th	$0.01
CopyText Inc.	Type the text from th	$0.01
CopyText Inc.	Type the text from th	$0.01
Crowdular	Glance: watch a sho	$0.14
ka wai yung	Write an Outline for a	$0.20
Raihani Lab	GREEN (you are not	$0.10
Ignite Media Solutior	Transcribe email fron	$0.03
Daniel McGuire	Super quick and eas	$0.03
John Doe	Test if coupon code	$0.07
Emily Zitek	Recall Study	$0.25
Song Health Lab	Survey on Perceptio	$0.25
Dr. Eric Wesselmann	Environmental Comm	$0.20
L/L	Tell us your how you	$0.01
Ignite Media Solutior	Transcribe address fi	$0.05

My pay was pretty low, but that wasn't the most frustrating part of it. For me, it was the lack of transparency regarding how well I was doing, and poorly designed HITs that were the most frustrating. I had no idea which jobs would be good jobs, which meant that I wasted a lot of time on duds. When I found a reasonable job that I could keep doing, I kept doing it even if there might be something better out there, because experience taught me that most of what was out there would stink.

I started so many surveys, which informed me some questions into the survey that I didn't qualify. Some of these claimed to have IRB approval, but my experience is that generally participants are to be told that they can quit at any time and be compensated for the time spent. Some surveys were conducted by shady market research businesses. In either case, I could have chosen to complain, but I assume that at best I would get my $0.15 and maybe have the survey taken down. I don't personally benefit too much from that.

So many HITs were just broken. I'd see a post on mturk.com that seemed promising, click it, and then the page just wouldn't load. Or, sometimes they'd say that the experiment was over, even though they hadn't removed their HITs from Mechanical Turk. One time I saw a fail-whale equivalent turking mascot that said they had enough turkers right now and didn't need any more.

On the one hand, I could see these problems pretty quickly and move on. But, the 20 seconds it cost me to do that — from deciding to click on that page, to clicking on it, to waiting to see if it would load — ate into my working time. If I'm targeting $6/hr, the hourly wage on HITs that I could work on has to go up by enough to make up for the $0.033 cents I just lost in opportunity cost while chasing a dud.

I found myself developing heuristics about how long I thought tasks would take. That made for some weird tradeoffs. For instance, HITs that paid less than $0.10 just weren't worth looking at unless they came with a clear time estimate. There's a high chance of any new HIT being a dud, and so it wasn't worth the expected payoff to check on low-paying HITs.

I took tasks that included a time estimate in the title, e.g. “takes 10 minutes,” because at least then I had some estimate of how long it should take me.

I'll also admit to occasionally being the so-called lazy turker. When I saw my pay rate dipping below $6/hour, I just tried to finish the task as quickly as I could. I tried to pay just enough attention so I'd catch obvious gold standard questions and otherwise ward off probable rejection, but I wasn't doing a good job anymore. Probably some machine learning researcher is using me as motivation for a new statistical correction technique as we speak.

The problem is that by the time I realized these HITs weren't worth doing, I had sunk so much into these tasks that I felt like I couldn't just abandon them. I was often already 10 minutes into a task before I realized that I wouldn't be making my $6/hr. Very few HITs provided reliable feedback regarding my progress.

And, holy shit, requesters don't know how to use qualifications. If I click through to a task only to find some equivalent of "this job is only meant for Bob, if you're not Bob we'll reject you," I really might punch someone. Variations include, "only people who took a survey on X two months ago are eligible for this HIT." Presumably, the people who posted the tasks emailed those who were eligible to let them know (how else would they find them?). If so, it takes only one extra step to assign a qualification to the HIT so only the workers you want to do the task can do it, and, importantly, only those workers see it in the list of available jobs.

These problems could be easily fixed by requesters, although educating the many requesters on how to avoid these problems sounds like a pretty daunting task. In my other research life I work on web accessibility. It turns out to be really difficult to get web developers to create accessible web pages because (i) they don't know they should do it, (ii) they don't know how to do it, and (iii) the benefits to them are somewhat invisible. I think these same root causes explain the problems on Mechanical Turk.

It seems like Mechanical Turk could reasonably introduce changes that could help with these problems. For instance, it would be great if there was a better flagging mechanism, and if Mechanical Turk took it more seriously. Perhaps they could consider charging requesters who post bad HITs more on subsequent HITs. I think they could justify this because it would likely keep costs lower for everyone else.

We could also pretty easily create a browser extension that would automatically solve some of the annoying problems. Disabling the forms of any HIT that hasn't been accepted would be really easy and help a lot. It could also simply check to see if there is content on the HIT page links, so I wouldn't have to manually click on each one. Fixing these problems aren't exactly grand leaps toward the future of work, but I think they'd impact ordinary people quite a lot.

Of course, much of what stood out to me about this experience was really just the failings of Amazon Mechanical Turk. Other platforms don't have these issues, at least not to the same scale. Amazon doesn't seem to like to update Mechanical Turk very often, however, so I'm not holding out too much hope. As I've heard Rob Miller say, "The great thing about Mechanical Turk is that they so rarely update things that I can still use my slides about them from 2008."

Conclusions

I got a lot out of spending 4 hours working on Mechanical Turk. Not because the experience of working itself was all that positive, but because I feel that I learned a lot about the worker experience. Much of what I learned justified what we're already doing. I'm going to make especially sure now that we advertise our hourly wage in the HIT title, and that we continue to give incremental feedback about payment. I hope we've been avoiding the obvious usability problems, but I want to take special care to avoid them in the future.

Most of the problems I experienced seem fairly trivial to fix. I think we need to make certain going forward with research that we're not fixing issues with the site that Amazon or requesters could fix overnight if they were incentivized to do so. A cautionary note is that I'm by no means an expert worker, so it's possible that more experienced workers have found ways around the problems I've experienced, possibly with tools or approaches that overcome them, or more likely by having found ways to avoid the worst of the HITs that are out there. ■

Jeffrey P. Bigham is an Associate Professor in the Human-Computer Interaction Institute at Carnegie Mellon University. He uses clever combinations of on-demand crowds and computation to build the interactive systems from science fiction.

Reprinted with permission of the original author. First appeared in *hn.my/turker* (cs.cmu.edu)

A Gentle Introduction to Monad Transformers

or, Values as Exceptions

By CHRISTOFFER STJERNLÖF

Either Left or Right

Before we break into the mysterious world of monad transformers, I want to start with reviewing a much simpler concept, namely the `Either` data type. If you aren't familiar with the `Either` type, you should probably not jump straight into monad transformers — they do require you to be somewhat comfortable with the most common monads.

With that out of the way:

Pretend we have a function that extracts the domain from an email address. Actually checking this properly is a rather complex topic which I will avoid, and instead I will assume an email address can only contain one `@` symbol, and everything after it is the domain.

I'm going to work with `Text` values rather than `String`s. This means if you don't have the text library, you can `either` work with `String`s instead, or `cabal install text`. If you have the Haskell platform, you have the text library.

We need to import `Data.Text` and set the `OverloadedStrings` pragma. The latter lets us write string literals (such as "Hello, world!") and have them become `Text` values automatically.

```
λ> :module +Data.Text
λ> :set -XOverloadedStrings
```

Now, figuring out how many `@` symbols there are in an email address is fairly simple. We can see that

```
λ> splitOn "@" ""
[""]

λ> splitOn "@" "test"
["test"]

λ> splitOn "@" "test@example.com"
["test", "example.com"]

λ> splitOn "@" "test@example@com"
["test", "example", "com"]
```

So if the split gives us just two elements back, we know the address contains just one `@` symbol, and we also as a bonus know that the second element of the list is the domain we wanted. We can put this in a file.

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Text

-- Imports that will be needed later:
import qualified Data.Text.IO as T
import Data.Map as Map
import Control.Applicative
```

```
data LoginError = InvalidEmail
  deriving Show

getDomain :: Text -> Either LoginError Text
getDomain email =
  case splitOn "@" email of
    [name, domain] -> Right domain
    _              -> Left InvalidEmail
```

This draws on our previous discoveries and is pretty self-explanatory. The function returns `Right domain` if the address is valid, otherwise `Left InvalidEmail`, a custom error type we use to make handling the errors easier later on. (Why this is called `LoginError` will be apparent soon.)

This function behaves as we expect it to.

```
λ> getDomain "test@example.com"
Right "example.com"

λ> getDomain "invalid.email@example@com"
Left InvalidEmail
```

To deal with the result of this function immediately, we have a couple of alternatives. The basic tool to deal with `Either` values is pattern matching, in other words,

```
printResult' :: Either LoginError Text -> IO ()
printResult' domain =
  case domain of
    Right text        -> T.putStrLn (append
"Domain: " text)
    Left InvalidEmail -> T.putStrLn "ERROR:
Invalid domain"
```

Testing in the interpreter shows us that

```
λ> printResult' (getDomain "test@example.com")
Domain: example.com

λ> printResult' (getDomain "test#example.com")
ERROR: Invalid domain
```

Another way of dealing with `Either` values is by using the `either` function. `either` has the type signature

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

In other words, it "unpacks" the `Either` value and applies one of the two functions to get a c value back. In this program, we have an `Either LoginError Text` and we want just a `Text` back, which tells us what to print. So we can view the signature of `either` as

```
either :: (LoginError -> Text) -> (Text -> Text)
-> (Either LoginError Text -> Text)
```

and writing `printResult` with the help of `either` yields a pretty neat function.

```
printResult :: Either LoginError Text -> IO ()
printResult = T.putStrLn . either
  (const "ERROR: Invalid domain")
  (append "Domain: ")
```

This function works the same way as the previous one, except with the pattern matching hidden inside the call to `either`.

Introducing Side-Effects

Now we'll use the domain as some sort of "user token" — a value the user uses to prove they have authenticated. This means we need to ask the user for their email address and return the associated token.

```
getToken :: IO (Either LoginError Text)
getToken = do
  T.putStrLn "Enter email address:"
  email <- T.getLine
  return (getDomain email)
```

So when `getToken` runs, it'll get an email address from the user and return the domain of the email address.

```
λ> getToken
Enter email address:
test@example.com
Right "example.com"
```

and, importantly,

```
λ> getToken
Enter email address:
not.an.email.address
Left InvalidEmail
```

Now, let's complete this with an authentication system. We'll have two users who both have terrible passwords:

```
users :: Map Text Text
users = Map.fromList [("example.com",
"qwerty123"), ("localhost", "password")]
```

With an authentication system, we can also run into two new kinds of errors, so let's change our LoginError data type to reflect that.

```
data LoginError = InvalidEmail
                | NoSuchUser
                | WrongPassword
  deriving Show
```

We also need to write the actual authentication function. Here we go...

```
userLogin :: IO (Either LoginError Text)
userLogin = do
  token <- getToken

  case token of
    Right domain ->
      case Map.lookup domain users of
        Just userpw -> do
          T.putStrLn "Enter password:"
          password <- T.getLine

          if userpw == password
             then return token

             else return (Left WrongPassword)
        Nothing -> return (Left NoSuchUser)
    left -> return left
```

This beast of a function gets the email and password from the user, checks that the email was processed without problems, finds the user in the collection of users, and if the passwords match, it returns the token to show the user is authenticated.

If anything goes wrong, such as the passwords not matching, there not being a user with the entered domain, or the getToken function failing to process, then a Left value will be returned.

This function is not something we want to deal with. It's big, it's bulky, it has several layers of nesting: it's not the Haskell we know and love.

Sure, it's possible to rewrite it using function calls to either and maybe, but that wouldn't help very much. The real reason the code is this ugly is that we're trying to mix both Either and IO, and they don't seem to blend well.

The core of the problem is that the IO monad is designed for dealing with IO actions, and it's terrible at handling errors. On the other hand, the Either monad is great at handling errors, but it can't do IO. So let's explore what happens if you imagine a monad that is designed to both handle errors and IO actions.

Too good to be true? Read on and find out.

We Can Make Our Own Monads

We keep coming across the IO (Either e a) type, so maybe there is something special about that. What happens if we make a Haskell data type out of that combination?

```
data EitherIO e a = EitherIO {
    runEitherIO :: IO (Either e a)
}
```

What did we get just by doing this? Let's see:

```
λ> :type EitherIO
EitherIO :: IO (Either e a) -> EitherIO e a

λ> :type runEitherIO
runEitherIO :: EitherIO e a -> IO (Either e a)
```

So already we have a way to go between our own type and the combination we used previously! That's gotta be useful somehow.

Implementing Instances for Common Typeclasses

This section might be a little difficult if you're new to the language and haven't had a lot of exposure to the internals of how common typeclasses work. You don't need to understand this section to continue reading the article, but I strongly suggest you put on your to-do list to learn enough to understand this section. It touches on many of the core components of what makes Haskell Haskell and not just another functional language.

But before we do anything else, let's make EitherIO a functor, an applicative, and a monad, starting with the functor, of course.

```
instance Functor (EitherIO e) where
  fmap f ex = wrapped
    where
      unwrapped = runEitherIO ex
      fmapped   = fmap (fmap f) unwrapped
      wrapped   = EitherIO fmapped
```

This may look a little silly initially, but it does make sense. First, we "unwrap" the EitherIO type to expose the raw IO (Either e a) value. Then we fmap over the inner a, by combining two fmaps. Then we wrap the new value up in EitherIO again, and return the wrapped value. If you are a more experienced Haskell user, you might prefer the following, equivalent, definition instead.

```
instance Functor (EitherIO e) where
  fmap f = EitherIO . fmap (fmap f) .
runEitherIO
```

In a sense, that definition makes it more clear that you are just unwrapping, running a function on the inner value, and then wrapping it together again.

The two other instances are more of the same, really. Creating them is a mostly mechanical process of following the types and unwrapping and wrapping our custom type. I challenge the reader to come up with these instances on their own before looking below how I did it, because trying to figure these things out will improve your Haskell abilities in the long run.

In any case, explaining them gets boring, so I'll just show you the instances as an experienced Haskell user might write them.

```
instance Applicative (EitherIO e) where
  pure    = EitherIO . return . Right
  f <*> x = EitherIO $ liftA2 (<*>) (runEitherIO
f) (runEitherIO x)

instance Monad (EitherIO e) where
  return  = pure
  x >>= f = EitherIO $ runEitherIO x >>= either
(return . Left) (runEitherIO . f)
```

If your definitions look nothing like these, don't worry. As long as your definitions give the correct results, they are just as good as mine. There are many ways to write these definitions, and none is better than the other as long as all are correct.

Using EitherIO

Now that our `EitherIO` type is a real monad, we'll try to put it to work! If we change the type signature of our `getToken` function, we'll run into problems quickly, though.

```
getToken :: EitherIO LoginError Text
getToken = do
  T.putStrLn "Enter email address: "
  input <- T.getLine
  return (getDomain input)
```

We get three type errors from this function alone now! In order:

1. `T.putStrLn "email"` returns `IO ()`, but we want `EitherIO LoginError ()`
2. T.getLine returns `IO Text`, we want `EitherIO LoginError Text`.
3. `getDomain input` returns `Either LoginError Text`, we want `EitherIO LoginError Text`.

Converting `Either` e a to `EitherIO e a` isn't terribly difficult. We have two functions to help us with that.

```
return   :: Either e a -> IO (Either e a)
EitherIO :: IO (Either e a) -> EitherIO e a
```

With both of those, we can take the `getDomain` function call and fit it in the new `getToken` function, like so:

```
EitherIO (return (getDomain input))
```

Remember this line, because we're going to put it into the function soon enough. But first, let's find out how to convert the two `IO a` values to an `EitherIO e a`. Again, we will use the `EitherIO` function. For the rest, it's useful to know your functors. If you do, you'll realize that

```
fmap Right :: IO a -> IO (Either e a)
```

so our `IO` actions would both be

```
EitherIO (fmap Right (T.putStrLn "email"))
EitherIO (fmap Right (T.getLine))
```

With these three lines, the `getToken` function is now written as

```
getToken :: EitherIO LoginError Text
getToken = do
  EitherIO (fmap Right (T.putStrLn "Enter email
address:"))
  input <- EitherIO (fmap Right T.getLine)
  EitherIO (return (getDomain input))
```

But this looks even more horrible than it was before! Relax. We'll take a detour to clean this up slightly.

Do You Even Lift?

The more general pattern here is that we have two kinds of functions: those that return `IO` something, and those that return `Either` something. We want to use both of those in our `EitherIO` monad. Converting a “lesser” monad to a “more powerful” one, like we want to do here, is often called lifting the lesser operation into the more powerful monad.

We can define two lift operations to do exactly this for our case.

```
liftEither :: Either e a -> EitherIO e a
liftEither x = EitherIO (return x)

liftIO :: IO a -> EitherIO e a
liftIO x = EitherIO (fmap Right x)
```

With these two functions, the `getToken` function in turn is a little more clean.

```
getToken :: EitherIO LoginError Text
getToken = do
  liftIO (T.putStrLn "Enter email address:")
  input <- liftIO T.getLine
  liftEither (getDomain input)
```

If you try to run this in the interpreter, you’ll get a nasty type error. The reason is that the interpreter expects something of type `IO a`, but `getToken` has type `EitherIO e a`, so we need to convert it back to `IO a` when we run it in the interpreter. Fortunately, we had a function that does just that — `runEitherIO`.

```
λ> runEitherIO getToken
Enter email address:
test@example.com
Right "example.com"
```

We’ll also want to do the same conversion for `userLogin`, of course. First we change the type signature, and then we fix the values that are of the wrong type. If you haven’t been paying 100% attention, the new look of `userLogin` might surprise you.

```
userLogin :: EitherIO LoginError Text
userLogin = do
  token  <- getToken
  userpw <- maybe (liftEither (Left NoSuchUser))
                 return (Map.lookup token users)
  password <- liftIO (T.putStrLn "Enter your
password:" >> T.getLine)
  if userpw == password
     then return token
     else liftEither (Left WrongPassword)
```

Where did all the nesting go? It’s gone. All the nesting was there simply because we had to handle a bunch of error cases in the `IO` monad. The `IO` monad is not meant to handle error cases, so you have to do it manually. Our `EitherIO` monad on the other hand, is built both to handle errors and to perform `IO` actions, so we get the best of both worlds. We just have to signal when errors occur, and the `EitherIO` monad takes care of the rest.

If we want to, say, print the result of this, we’ll need a print function similar to the one we had previously.

```
printResult :: Either LoginError Text -> IO ()
printResult res =
  T.putStrLn $ case res of
    Right token        -> append "Logged in with
token: " token
    Left WrongPassword -> "Invalid email address
entered."
    Left NoSuchUser    -> "No user with that
email exists."
    Left InvalidEmail  -> "Wrong password."
```

This function, just like the previous one, takes an `Either` value, so we’ll need to “unwrap” our result before we send it over.

```
λ> runEitherIO userLogin >>= printResult
Enter email address:
test@example.com
Enter your password:
qwerty123
Logged in with token: example.com

λ> runEitherIO userLogin >>= printResult
Enter email address:
test@127.0.0.1
No user with that email exists.
```

As you can see, we’ve got a lot of convenience already, being able to just signal errors and let our `EitherIO` monad take care of them just like it takes care of `IO` actions.

Signaling Errors

But how are we signaling errors, really? It turns out that to signal something like `WrongPassword`, we have to return

```
liftEither (Left WrongPassword)
```

and that's not very tidy. We can easily make a function

```
throwE :: e -> EitherIO e a
throwE x = liftEither (Left x)
```

When we have this function, `userLogin` gets even better.

```
userLogin :: EitherIO LoginError Text
userLogin = do
  token      <- getToken
  userpw     <- maybe (throwE NoSuchUser)
                  return (Map.lookup token
users)
  password   <- liftIO $ T.putStrLn "Enter your
password:" >> T.getLine

  if userpw == password
     then return token
     else throwE WrongPassword
```

throwE? What Is This, Java?

No, of course not. But I did choose that name deliberately. What we have with our `EitherIO` monad is looking more and more like exceptions in languages like Java, Python, C++ and so on. And that's not a bad way to view it.

However, there are some differences. One big difference is that our "exceptions" are just normal values that are being returned from the function, while more traditional (Java, Python) exceptions are interruptions of normal control flow. Another difference is that our "exceptions" are checked by the type system, so we can't forget to catch our exceptions.

ExceptIO

But let's entertain that idea further. What happens if we just say goodbye to `Either` and talk about exceptions instead? First, we'll need to rename our `EitherIO` monad:

```
data ExceptIO e a = ExceptIO {
    runExceptIO :: IO (Either e a)
}
```

It still works the same as before, it's just been renamed. The names need to be changed throughout the code, but other than that, the code still works.

Gotta Catch 'Em All

So if we can *throw* what basically amounts to exceptions...

Yes! Keep going!

Can we also...

Yes?

...catch them?

Oh, I'm so happy you asked! Of course we can. And it's really simple, too! We need to define a function that does the catching. Call it `catchE`, so it looks similar to `throwE`.

```
catchE :: ExceptIO e a -> (e -> ExceptIO c a) ->
ExceptIO c a
catchE throwing handler =
  ExceptIO $ do
    result <- runExceptIO throwing
    case result of
      Left failure -> runExceptIO (handler fail-
ure)
      success      -> return success
```

This unwraps the `throwing` computation and inspects its result. If it was successful, it just returns it right back without doing anything. If it was a failure, it runs the `handler` with the error as an argument, and returns the result of that instead.

We can use this to spice up our application a little. We'll start by writing two exception handlers — one that catches just a single exception, and one that catches all exceptions.

```
wrongPasswordHandler :: LoginError -> ExceptIO
LoginError Text
wrongPasswordHandler WrongPassword = do
  liftIO (T.putStrLn "Wrong password, one more
chance.")
  userLogin
wrongPasswordHandler err = throwE err
```

The `wrongPasswordHandler` handles only the `WrongPassword` exception, and rethrows everything else. It responds to a `WrongPassword` exception by running the `userLogin` function again to give the user a second chance.

The other exception handler will respond to exceptions by printing an error message and then re-throwing the exception to abort the current execution.

```
printError :: LoginError -> ExceptIO LoginError
a
printError err = do
  liftIO . T.putStrLn $ case err of
    WrongPassword -> "Wrong password. No more
chances."
    NoSuchUser -> "No user with that email
exists."
    InvalidEmail -> "Invalid email address
entered."

  throwE err
```

We'll create a third function where we use these exceptions.

```
loginDialogue :: ExceptIO LoginError ()
loginDialogue = do
  let retry =  userLogin `catchE` wrongPassword-
Handler
  token     <- retry `catchE` printError
  liftIO $ T.putStrLn (append "Logged in with
token: " token)
```

Note in particular how we "wrap" exception handlers around each other. In the innermost layer is the `userLogin` computation, which gets wrapped by the `wrongPasswordHandler` handler, and then that entire package gets wrapped by the `printError` handler. You need to wrap your handlers in the order you expect them to catch exceptions from underneath each other.

Going General

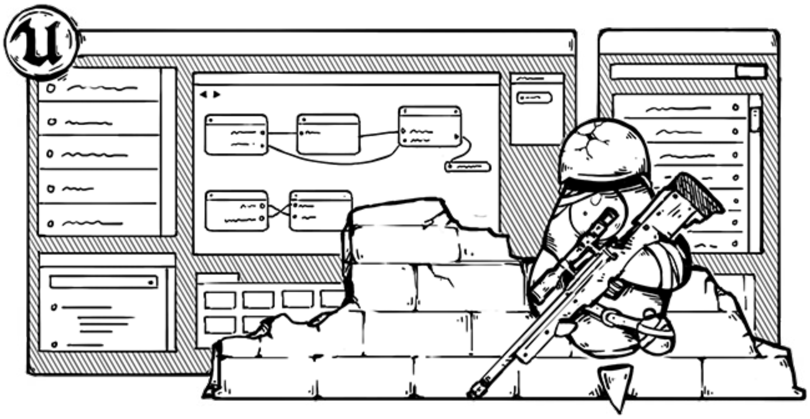
There is just one, tiny, little thing I want to change in our `ExceptIO` type. Currently, we're stuck with only being able to combine `IO` and exceptions. What if we wanted to combine exceptions with database transactions, or lists, or some other kind of monad? We can make the other monad an argument to our exception monad.

Then we'll also have to change the name from `ExceptIO` to `ExceptT`. Why the `T`, you ask? Because it's a monad transformer.

Congratulations! You made it all the way. You've created and used your first monad transformer. That's no small feat.

Christoffer is a 22 year old ICT student in Stockholm, Sweden. His passions are mainly programming and teaching, which is why he enjoys writing about programming concepts. Three years ago, Christoffer picked up Haskell as the primary language for personal projects, and after the initial hump he has not looked back.

Reprinted with permission of the original author.
First appeared in *hn.my/monad* (github.com)

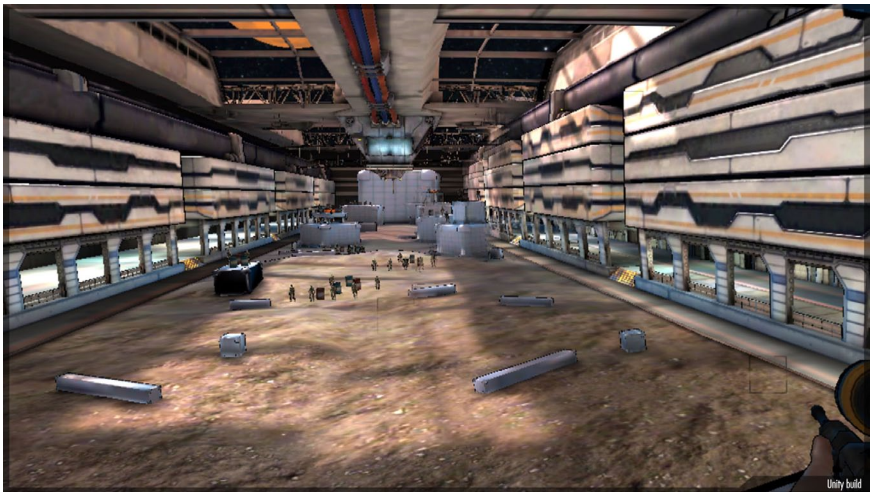


An Unreal Decision

By JEFF LAMARCHE

TWO MONTHS AGO, I made the decision to throw out months of work on one of our game development projects. For Republic Sniper [republicsniper.com], we made a fresh start, turning our back on months of work and switching to a new game engine: Unreal Engine 4 [unrealengine.com]. It wasn't an easy decision to make. None of the code or shaders we wrote for Unity could be brought over to UE4, and there's a fair bit of work involved in bringing over the 3D models and image assets we've created.

We've now been using UE4 in anger for a couple of months, and one thing has become very clear: we made the right decision. The pace of development has increased substantially since making the switch, and the latest builds look much better. Much more importantly, the Republic Sniper team is happy with their tools and once again feel good about what we're creating.



A WIP in-engine screenshot of the last Unity version of Republic Sniper before we made the switch

Our Blue Period

By June of this year, the team had become frustrated with our Unity workflow and wanted to explore other options. Once we started, it didn't take long to see that the best alternative — and possibly the only viable alternative for us — was the Unreal Engine. After a few weeks of experimenting, reading documentation, and test driving the latest version (UE4), we were convinced that we needed to make the switch.

That doesn't mean the decision was made without trepidation. We had a lot invested in the Unity version of Republic Sniper, and walking away from it was more than a little scary.

A WIP in-engine shot of the current Republic Sniper training range in UE4

Fear of the Sunk Cost

In a past life, I spent about a decade as a consultant working in Enterprise computing. I traveled constantly and did programming, database, and integration work for very large corporate and governmental entities.

In that role, I repeatedly saw bad decisions — often to the tune of tens of millions of dollars — being made simply because a large investment of time and money had already been made pursuing that bad decision. On many projects, I watched people stick by decisions long after everyone on the project knew it was a bad one.

The funny thing about the sunk cost fallacy is that when you're an outsider — with no skin in the game — it's really obvious when it's time to cut losses and try something new. When it's not your money, reputation, or job on the line, the tough choices are a hell of a lot easier.

I didn't have the luxury of being an outsider with Republic Sniper. When I started to realize that, perhaps, Unity wasn't the right choice for us, I didn't really want to accept it. It took a couple of months and required exuberant lobbying by certain members of the team for me to authorize the switch.

The Unity Fit Problem

Unity is a very capable tool, and a lot of really good games have been made with it. We absolutely could have finished Republic Sniper using Unity. We could have pushed through our frustrations and delivered a respectable game.

Only…nobody on the team would've been happy making a *respectable* game.

The First Sign of Trouble

I first started questioning our use of Unity early this year. As we started to explore just what we could accomplish in a mobile game, we started having significant problems. The Unity Editor is still — in 2014 — a 32-bit application. While the theoretical memory limit of a 32-bit Mac App is 4 GB, on a machine with dual GPUs, you actually only get an application heap of about 2.7 GB of RAM in practice. When a 32-bit app hits that memory limit, you start getting erratic behavior and crashes that give you very little information about what happened. While 2.7 GB may sound like quite a lot of memory, we found ourselves hitting the limit regularly.

I wasted weeks of time figuring out the problem. In the process, I lost a substantial amount of work to memory crashes. I filed dozens of bug reports, but none of them yielded a satisfactory response from Unity's tech support. Nearly every ticket was closed with no acknowledgement that maybe a 32-bit application in 2014 — 8 years after Apple stopped shipping 32-bit-only computers — might be anything less than perfectly reasonable. No matter how many times I asked, I got no hint of when a 64-bit editor might be coming out. All of my interactions with Unity tech support left me frustrated and feeling, frankly, like they didn't care all that much.

The Real Problem

There were other technical issues along the way, though none as severe as the constant memory-related crashes. Yet, the technical issues were not what made us start looking for alternatives. Nor was it Unity's weak customer service responses to our legitimate frustrations. Our team just wasn't finding it easy to collaborate. We weren't gelling as a cohesive team, and we often felt like the tools were working against us.

WIP shot of the shooting stalls on the training level

An in-engine shot from the catwalks

Every time we pulled code updates from source control or switched target platforms, Unity would take at least forty-five minutes to open our project, and considerably longer on less powerful hardware. The solution to this problem — a problem that screams design flaw — was for us to use the Unity Cache Server at an additional $500 per seat.

Being able to use version control effectively or to switch platform targets on a large project without having to wait for an hour is an add-on feature, not included with the Professional license?

Yes.

But really, that was just one symptom of a larger problem. The real issue was that Unity seems to have been built for very small development teams. While Republic Sniper isn't a huge project, we have seven regular project members and a number of other contributors.

There are upsides to Unity's approach: It allows tiny teams to create things that used to be impossible for small teams to build. Unity has enabled some truly phenomenal indie games, many of which were created by a single developer.

Unity is also a very developer-centric toolset. Our game artists honestly hated working in the Unity Editor and felt like second-class participants in the game development process. They were constantly waiting on a developer to write a shader or component for them, and the tools felt alien. We had constant bottlenecks and were never happy with the look and feel we could achieve, despite a lot of time and effort invested into the art design and production.

Enter Unreal

I've known about the Unreal Engine for a very long time. Indeed, many of the games that inspired me to create games display the Unreal Engine logo when launched. Despite that, we didn't even consider using Unreal when we started Republic Sniper. The UDK (the previous version of the Unreal Engine dev tools) took 25% of your gross income after the first $50,000. When you added that 25% to the 30% taken by the app store, it meant we'd be losing more than half of our gross income. The other big obstacle was that the UDK tools were Windows only. MartianCraft is a shop of (mostly) dyed-in-the-wool Mac folk who'd rather not spend their days using Windows.

When UE4 came out with a license cost of $19.95 per month per seat plus 5% of gross, a Mac version of the tools, and full access to the underlying source code, we decided to take a long, hard look at it.

UE4's physically-based rendering is gorgeous. It runs well on iOS devices and looks absolutely fantastic on the latest generation of hardware. The editor and tools are very artist-friendly. More importantly, UE4 has two options for programming: C++ and a visual programming language called Blueprint. [hn.my/blueprint]

Blueprint

I'd never been a fan of visual programming languages before, so I didn't pay much attention to Blueprint at first. But when one of our code-phobic artists sent me a small game level, complete with programmed interactions, that he had put together after a couple of days of tinkering, I was curious.

The way Epic has designed Blueprint and C++ in UE4 to work together is pretty amazing. You can write a C++ class and then subclass it in Blueprint. You can expose C++ methods to use as Blueprint nodes. Almost everything you can do in C++, you can do in Blueprint. Although there is a small performance hit with Blueprint, it's rarely enough of a hit to be an issue. Both

An in-engine shot of one of the rifles available in the game: the LT-4 “Hellfire” Tactical Sniper Rifle

languages are first-class citizens in the engine, and there are very few tasks that require C++. For some tasks (especially tasks with complex algorithms), C++ will likely be the better choice for experienced programmers, but almost nothing has to be done in C++.

And that is incredibly empowering.

For visual thinkers, Blueprint just makes a lot more sense than lines of text-based code. It’s also a lot less intimidating. In fact, Blueprint has enabled our game artists to accomplish many tasks by themselves that would’ve required a developer when we were using Unity. This has made us much more efficient and faster as a team. While people still specialize, most tasks can be handled by anyone on the team. The logic for opening a door, for example, might be written by a programmer, but it might also be put in place by a level designer or modeler. It can even be started by a level designer then polished and tweaked later by a developer who can also make it more efficient.

Goodbye Shading Language

UE4’s physically-based material system produces great results, and the way you build shaders for it is similar to Blueprint. UE4 uses a visual node-based material system much like those familiar to our 3D artists from other applications. The artists never have to see cryptic GLSL or HLSL code. There’s no ShaderLab or CG or a need to wait for a shader programmer’s help to get the look they want. The whole material system is visual, it all makes sense to artists, and the results are jaw-droppingly fantastic in the hands of a good artist.

Miscellany

Although we’ve barely scratched the surface of what UE4 can do in the few months we’ve been using it on Republic Sniper, we’re constantly finding little unexpected treats — things that are easier than we expected. Building an iOS app, for example, doesn’t require exporting our project to Xcode and then compiling and code signing. We can build within the Unreal Editor and deploy test builds directly to a device for testing. Heck, even those black sheep on our team who work on Windows can build for iOS with Unreal.

Unreal also has much better built-in tools for most common game development tasks, including great tools for creating physics assets (ragdolls), designing character AI, and doing dynamic level loading and unloading. Lightmap baking is considerably faster and the interface is far more intuitive than Unity’s.

The Downsides

It’s not all grins and giggles, though. UE4 is a relatively new toolset with a lot of brand new, completely rewritten functionality, and there have been bumps along the way. There are a few parts of UE4 that simply feel unfinished and we’re still climbing a learning curve as we adjust to a UE4-based workflow. We’ve opened many bug reports over the last couple of months. Fortunately, Epic has been responsive to our bug reports and seems eager to make UE4 meet our needs. The UE4 development community has been helpful and supportive as well, and we’ve seen regular feature releases and bug fix releases since we started using it.

Since UE4 is the first release of the Unreal Engine to really go after small- and medium-sized game shops, there’s currently no comparison between the new UE4 Marketplace and Unity’s Asset Store. The Unity Asset Store is a thriving marketplace with a lot of great tools and content. It will be quite some time before the UE4 Marketplace has a comparable amount of content. This hasn’t been a problem for us because we have our own content creation team for Republic Sniper and most of the Asset Store items we bought for Unity were to accomplish tasks that UE4 handles natively.

The single biggest issue for us with UE4 has been the performance of the Mac editor, which lags considerably behind the Windows version when running on the same hardware. This is a little frustrating, since the introduction of the Mac editor was one of the draws that lead us towards UE4. Mac performance is bad enough that it's painful to run the UE4 Editor on a laptop or an older desktop. I find the performance quite acceptable on my Late 2013 Mac Pro, but when I work on my 2012 Retina MacBook Pro, or at home on my Mid-2010 Mac Pro, I resort to bootcamp and the Windows version of the Unreal Editor to get work done.

The Final Word on UE4

There's no such thing as a flawless or perfect set of development tools — UE4 is no exception. However, despite a few rough edges, I'm incredibly happy with the switch. UE4 is a professional toolkit in all respects, honed over more than sixteen years and dozens of AAA game titles. Despite that, it's also an extraordinarily approachable toolset. UE4 allows us to create a game that lives up to our expectations and we're really excited once again to be making this game.

Don't read this as a condemnation of Unity. We're still using Unity for another game project and have no intention of switching it to UE4. Unity is a great fit for that project, just like UE4 is a great fit for Republic Sniper. A year ago, Unity was the undisputed leader among an anemic set of game engines for indies and small game studios. Now, we have two strong options with their own strengths and weakness. For some projects, either tool will work well. For other projects, one may be a substantially better fit.

As a company that does both in-house and contract game development, having more options is a great thing. We believe strongly in using the best tool for any given job. The more good tools we have available, the more likely we are to find the right tool.

Epic's decision to go after Unity's market also means increased competition. That competition will push both companies to make better tools. That's a good thing for game developers and ultimately for people who play games. ■

Jeff is an author, speaker, software developer and teacher. His first book, Beginning iPhone Development, stands as the all-time best-selling book on learning to develop for the iPhone and iPad.

About MartianCraft: MartianCraft is building the future today. Working with our clients we have shaped elections, redefined the newspaper, and delivered joy to millions. We aren't just mobile natives, we helped spark the revolution.

Reprinted with permission of the original author.
First appeared in *hn.my/unreal* (martiancraft.com)

Letter to a Young Haskell Enthusiast

By GERSHOM BAZERMAN

The following letter is not about what "old hands" know and newcomers do not. Instead, it is about lessons that we all need to learn more than once, and remind ourselves of. It is about tendencies that are common, understandable, and come with the flush of excitement of learning any new thing that we understand is important. It's about the difficulty, always, in trying to decide how best to convey that excitement and sense of importance to others in a way that they will listen. It is written more specifically, but only because I have found that if we don't talk specifics as well as generalities, the generalities make no sense. This holds for algebraic structures, and it holds for other, vaguer concepts no less. It is a letter full of things I want to remember, as well as advice I want to share. I expect I will want to remind myself of it when I encounter somebody who is wrong on the internet, which, I understand, may occur on rare occasion.

YOU'VE RECENTLY ENTERED the world of strongly typed functional programming, and you've decided it is great. You've written a program or two or a library or two, and you're getting the hang of it. You hop on IRC and hear new words and ideas every day. There are always new concepts to learn, new libraries to explore, new ways to refactor your code, new typeclasses to make instances of.

Now, you're a social person, and you want to go forth and share all the great things you've learned. And you have learned enough to distinguish some true statements from some false statements, and you want to go and slay all the false statements in the world.

Is this really what you want to do? Do you want to help people? Do you want to teach people new wonderful things? Do you want to share the things that excite you? Or do you want to feel better about yourself, confirm that you are programming better, confirm that you are smarter and know more, or reassure yourself that your adherence to a niche language is ok by striking out against the mainstream? Of course, you want to do the former. But a part of you probably secretly wants to do the latter, because in my experience that part is in all of us. It is our ego: it drives us to great things, but it also can hold us back, make us act like jerks, and, worst of all, stand in the way of communicating with others about what we truly care about.

Haskell wasn't built on great ideas, although it has those. It was built on a culture of how ideas are treated. It was not built on slaying others' dragons, but on finding our own way; not tearing down rotten ideas (no matter how rotten) but showing by example how we didn't need those ideas after all.

In functional programming, our proofs are not by contradiction, but by construction. If you want to teach functional programming, or preach functional programming, or even just have productive discussions as we all build libraries and projects together, it will serve you well to learn that ethic.

You know better than the next developer, or so you think. This is because of something you have learned. So how do you help them want to learn it, too? You do not tell them this is a language for smart people. You do not tell them you are smart because you use this language. You tell them that types are for fallible people, like we all are. They help us reason and catch our mistakes, because while software has grown more complex, we're still stuck with the same old brains. If they tell you they don't need types to catch errors, tell them that they must be much smarter than you, because you sure do. But even more, tell them that all the brainpower they use to not need types could turn into even greater, bigger, and more creative ideas if they let the compiler help them.

This is not a language for clever people, although there are clever things that can be done in this language. It is a language for simple things and clever things alike; sometimes we want to be simple, and sometimes we want to be clever. But we don't give bonus points for being clever. Sometimes, it's just fun, like solving a crossword puzzle or playing a tricky Bach prelude, or learning a tango. We want to keep simple things simple so that tricky things are possible.

It is not a language that is "more mathematical" or "for math" or "about math." Yes, in a deep formal sense, programming is math. But when someone objects to this, this is not because they are a dumb person, a bad person, or a malicious person. They object because they have had a bad notion of math foisted on them. "Math" is the thing that people wield over them to tell them they are not good enough, that they cannot learn things, that they don't have the mindset for it. That's a dirty lie. Math is not calculation — that's what computers are for. Nor is math just abstract symbols. Nor is math a prerequisite for Haskell. If anything, Haskell might be what makes somebody find math interesting at all. Our equation should not be that math is hard, and so programming is hard. Rather, it should be that programming can be fun, and this means that math can be fun, too. Some may object that programming is not only math, because it is engineering as well, and creativity, and practical tradeoffs. But, surprisingly, these are also elements of the practice of math, if not the textbooks we are given.

I have known great Haskell programmers, and even great computer scientists who know only a little linear algebra maybe, or never bothered to pick up category theory. You don't need that stuff to be a great Haskell programmer. It might be one way. The only thing you need category theory for is to take great categorical and mathematical concepts from the world and import them back to programming, and translate them along the way so that others don't need to make the same journey you did. And you don't even need to do that, if you have patience, because somebody else will come along and do it for you, eventually.

The most important thing (though not hardest part) about teaching and spreading knowledge is to emphasize that this is for everyone. Nobody is too young, too inexperienced, too old, too set in their ways, too excitable, insufficiently mathematical, etc. Believe in everyone, attack nobody, even the trolliest.* Attacking somebody builds a culture of sniping and argumentativeness. It spreads to the second trolliest, and so forth, and then eventually to an innocent bystander who just says the wrong thing to spark bad memories of the last big argument.

The hardest thing, and the second most important, is to put aside your pride. If you want to teach people, you have to empathize with how they think and feel. If your primary goal is to spread knowledge, then you must be relentlessly self-critical of anything you do or say that gets in the way of that. And you don't get to judge that: others do. And you must just believe them. I told you this was hard. So if somebody finds you off-putting, that's your fault. If you say something and somebody is hurt or takes offense, it is not their fault for being upset or feeling bad. This is not about what is abstractly hurtful in a cosmic sense; it is about the fact that you have failed, concretely, to communicate as you desired. So accept the criticism, apologize for giving offense (not just for having upset someone but also for what you did to hurt them), and attempt to learn why they feel how they feel.

Note that if you have made somebody feel crummy, they may not be in a mood to explain why or how, because their opinion of you has already plummeted. So don't declare that they must or should explain themselves to you, although you may politely ask. Remember that knowledge does not stand above human behavior. Often, you don't need to know exactly why a person feels the way they do, only that they do, so you can respect that. If you find yourself demanding explanations, ask yourself, if you knew this thing, would that change your behavior? How? If not, then learn to let it go.

Remember also that they were put off by your actions, not by your existence. It is easy to miss this distinction and react defensively. "Fight-or-flight" stands in the way of clear thinking and your ability to empathize; try taking a breath and maybe a walk until the adrenaline isn't derailing your true intentions.

Will this leave you satisfied? That depends. If your goal is to understand everything and have everybody agree with regards to everything that is in some sense objectively true, it will not. If your goal is to have the widest, nicest, most diverse, and most fun Haskell community possible, and to interact in an atmosphere of mutual respect and consideration, then it is the only thing that will leave you satisfied.

If you make even the most modest (to your mind) mistake, be it in social interaction or technical detail, be quick to apologize and retract, and do so freely. What is there to lose? Only your pride. Who keeps track? Only you. What is there to gain? Integrity, and ultimately that integrity will feel far more fulfilling than the cheap passing thrills of cutting somebody else down or deflecting their concerns.

Sometimes it may be, for whatever reason, that somebody doesn't want to talk to you, because at some point your conversation turned into an argument. Maybe they did it, maybe you did it, and maybe you did it together. It doesn't matter. Learn to walk away. Learn from the experience how to communicate better, how to avoid that pattern, how to always be more positive, friendly, and forward-looking. Take satisfaction in the effort in that. Don't talk about them behind their back, because that will only fuel your own bad impulses. Instead, think about how you can change.

Your self-esteem doesn't need your help. You may feel you need to prove yourself, but you don't. Other people, in general, have better things to do with their time than judge you, even when you may sometimes feel otherwise. You know you're talented, that you have learned things, and built things, and that this will be recognized in time. Nobody else wants to hear it from you. The more they hear it, the less they will believe it; the more it will distract from what you really want, which is not to feed your ego, not to be great, but to accomplish something great, or even just to find others to share something great with. In fact, if anyone's self-esteem should be cared for, it is that of the people you are talking to. The more confident they are in their capacity and their worth, the more willing they will be to learn new things, and to acknowledge that their knowledge, like all of ours, is limited and partial. You must believe in yourself to be willing to learn new things, and if you want to cultivate more learners, you must cultivate that self-belief in others.

Knowledge is not imposing. Knowledge is fun. Anyone, given time and inclination, can acquire it. Don't only lecture, but continue to learn, because there is always much more than you know. (And if there wasn't, wow, that would be depressing, because what would there be to learn next?) Learn to value all opinions, because they all come from experiences, and all those experiences have something to teach us. Dynamic typing advocates have brought us great leaps in JIT techniques. If you're interested in certain numerical optimizations, you need to turn to work pioneered in C++ or Fortran. Like you, I would rather write in Haskell. But it is not just the tools that matter but the ideas, and you will find they come from everywhere.

In fact, we have so much to learn that we direct our learning by setting up barriers — declaring certain tools, fields, languages, or communities not worth our time. This isn't because they have nothing to offer, but it is a crutch for us to shortcut evaluating too many options at once. It is fine, and in fact necessary, to narrow the scope of your knowledge to increase its depth. But be glad that others are charting other paths! Who knows what they will bring back from those explorations.

If somebody is chatting about programming on the internet, they're already ahead of the pack, already interested in craft and knowledge. You may not share their opinions, but you have things to learn from one another, always. Maybe the time and place aren't right to share ideas and go over disputes. That's ok. There will be another time and place, or maybe there won't be. There is a big internet full of people, and you don't need to be everybody's friend or everybody's mentor. You should just avoid being anybody's enemy, because your time and theirs is too precious to waste it on hard feelings instead of learning new cool stuff.

This advice is not a one-time proposition. Every time we learn something new and want to share it, we face these issues all over again — the desire to proclaim, to overturn received wisdom all at once — and the worse the received wisdom, the more vehemently we want to strike out. But if we are generous listeners and attentive teachers, we not only teach better and spread more knowledge, but also learn more, and enjoy ourselves more in the process. To paraphrase Rilke's "Letter to a Young Poet": Knowledge is good if it has sprung from necessity. In this nature of its origin lies the judgment of it: there is no other. ■

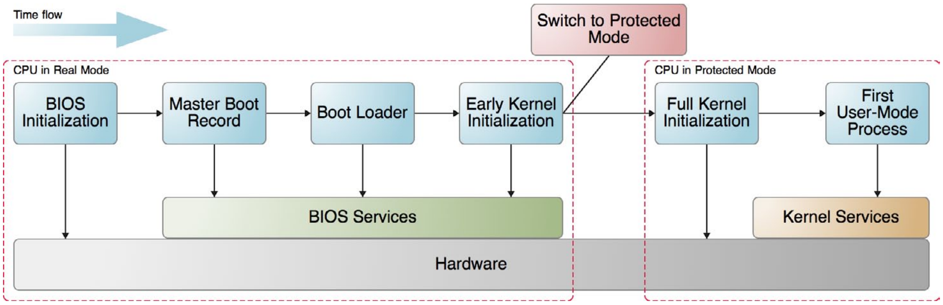
Gershom Bazerman is is an organizer of the NY Haskell Users Group, the NY Homotopy Type Theory Reading group, and a member of the Haskell.org committee. He has written a number of widely used Haskell packages, most notably the JMacro library for programmatic generation of JavaScript.

Reprinted with permission of the original author.
First appeared in *hn.my/younghaskell* (comonad.com)

How Computers Boot Up

By GUSTAVO DUARTE

Booting is an involved, hacky, multi-stage affair — fun stuff. Here's an outline of the process:

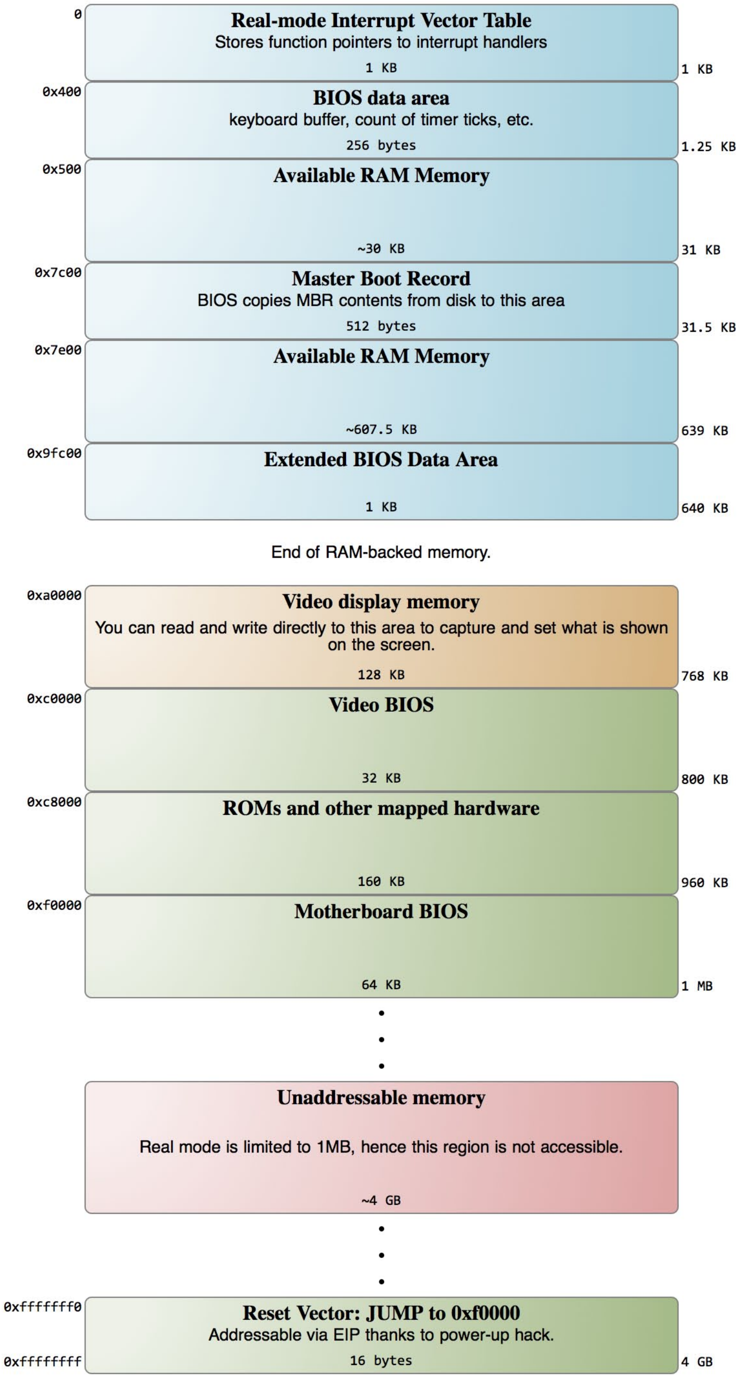


Things start rolling when you press the power button on the computer (no! do tell!). Once the motherboard is powered up, it initializes its own firmware (the chipset and other tidbits) and tries to get the CPU running. If things fail at this point (e.g., the CPU is busted or missing) then you will likely have a system that looks completely dead except for rotating fans. A few motherboards manage to emit beeps for an absent or faulty CPU, but the zombie-with-fans state is the most common scenario based on my experience. Sometimes USB or other devices can cause this to happen: unplugging all non-essential devices is a possible cure for a system that was working and suddenly appears dead like this. You can then single out the culprit device by elimination.

If all is well, the CPU starts running. In a multi-processor or multi-core system one CPU is dynamically chosen to be the bootstrap processor (BSP) that runs all of the BIOS and kernel initialization code. The remaining processors, called application processors (AP) at this point, remain halted until later on when they are explicitly activated by the kernel. Intel CPUs have been evolving over the years, but they're fully backwards compatible. That means that modern CPUs can behave like the original 1978 Intel 8086, which is exactly what they do after power up. In this primitive power up state, the processor is in real mode with memory paging disabled. This is like ancient MS-DOS where only 1 MB of memory can be addressed and any code can write to any place in memory; there's no notion of protection or privilege.

Most registers in the CPU have well-defined values after power up, including the instruction pointer (EIP) which holds the memory address for the instruction being executed by the CPU. Intel CPUs use a hack whereby even though only 1MB of memory can be addressed at power up, a hidden base address (an offset, essentially) is applied to EIP so that the first instruction executed is at address 0xFFFFFFF0 (16 bytes short of the end of 4 gigs of memory and well above one megabyte). This magical address is called the reset vector and is standard for modern Intel CPUs.

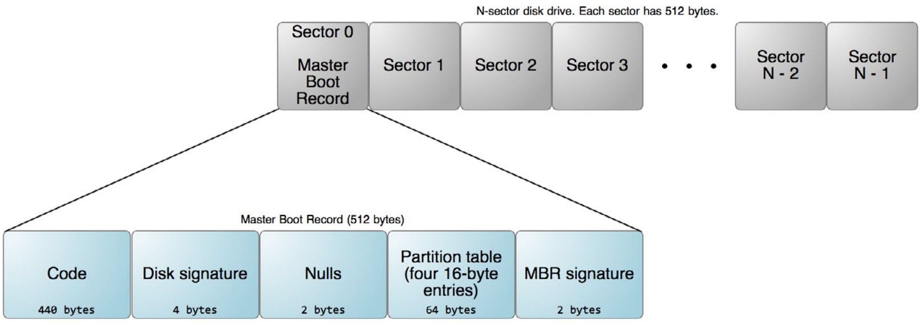
The motherboard ensures that the instruction at the reset vector is a jump to the memory location mapped to the BIOS entry point. This jump implicitly clears the hidden base address present at power up. All of these memory locations have the right contents needed by the CPU thanks to the memory map kept by the chipset. They are all mapped to flash memory containing the BIOS since at this point the RAM modules have random crap in them. An example of the relevant memory regions is shown below:


0
Real-mode Interrupt Vector Table
Stores function pointers to interrupt handlers
1 KB
1 KB
0x400
BIOS data area
keyboard buffer, count of timer ticks, etc.
256 bytes
1.25 KB
0x500
Available RAM Memory
~30 KB
31 KB
0x7c00
Master Boot Record
BIOS copies MBR contents from disk to this area
512 bytes
31.5 KB
0x7e00
Available RAM Memory
~607.5 KB
639 KB
0x9fc00
Extended BIOS Data Area
1 KB
640 KB
End of RAM-backed memory.
0xa0000
Video display memory
You can read and write directly to this area to capture and set what is shown on the screen.
128 KB
768 KB
0xc0000
Video BIOS
32 KB
800 KB
0xc8000
ROMs and other mapped hardware
160 KB
960 KB
0xf0000
Motherboard BIOS
64 KB
1 MB
Unaddressable memory
Real mode is limited to 1MB, hence this region is not accessible.
~4 GB
0xfffffff0
Reset Vector: JUMP to 0xf0000
Addressable via EIP thanks to power-up hack.
0xffffffff
16 bytes
4 GB

The CPU then starts executing BIOS code, which initializes some of the hardware in the machine. Afterwards the BIOS kicks off the Power-on Self Test (POST) which tests various components in the computer. Lack of a working video card fails the POST and causes the BIOS to halt and emit beeps to let you know what's wrong, since messages on the screen aren't an option. A working video card takes us to a stage where the computer looks alive: manufacturer logos are printed, memory starts to be tested, and angels blare their horns. Other POST failures, like a missing keyboard, lead to halts with an error message on the screen. The POST involves a mixture of testing and initialization, including sorting out all the resources — interrupts, memory ranges, I/O ports — for PCI devices. Modern BIOSes that follow the Advanced Configuration and Power Interface build a number of data tables that describe the devices in the computer; these tables are later used by the kernel.

After the POST the BIOS wants to boot up an operating system, which must be found somewhere: hard drives, CD-ROM drives, floppy disks, etc. The actual order in which the BIOS seeks a boot device is user configurable. If there is no suitable boot device the BIOS halts with a complaint like "Non-System Disk or Disk Error." A dead hard drive might present with this symptom. Hopefully this doesn't happen and the BIOS finds a working disk allowing the boot to proceed.

The BIOS now reads the first 512-byte sector (sector zero) of the hard disk. This is called the Master Boot Record, and it normally contains two vital components: a tiny OS-specific bootstrapping program at the start of the MBR followed by a partition table for the disk. The BIOS, however, does not care about any of this. It simply loads the contents of the MBR into memory location 0x7c00 and jumps to that location to start executing whatever code is in the MBR.



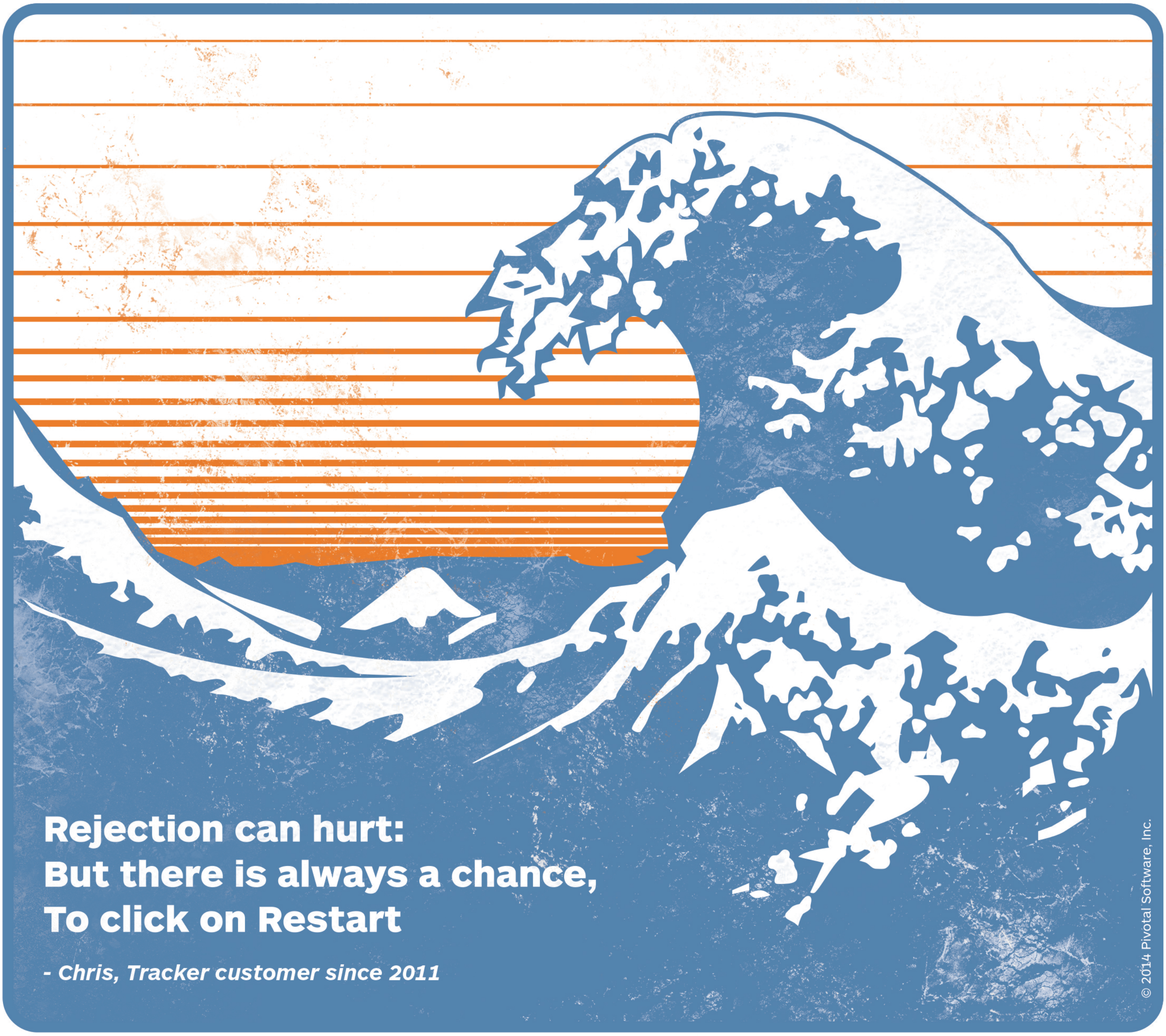
The specific code in the MBR could be a Windows MBR loader, code from Linux loaders such as LILO or GRUB, or even a virus. In contrast the partition table is standardized: it is a 64-byte area with four 16-byte entries describing how the disk has been divided up Traditionally Microsoft MBR code takes a look at the partition table, finds the (only) partition marked as active, loads the boot sector for that partition, and runs that code. The boot sector is the first sector of a partition, as opposed to the first sector for the whole disk. If something is wrong with the partition table you would get messages like "Invalid Partition Table" or "Missing Operating System." This message does not come from the BIOS but rather from the MBR code loaded from disk. Thus the specific message depends on the MBR flavor.

Boot loading has gotten more sophisticated and flexible over time. The Linux boot loaders Lilo and GRUB can handle a wide variety of operating systems, file systems, and boot configurations. Their MBR code does not necessarily follow the "boot the active partition" approach described above. But functionally the process goes like this:

1. The MBR itself contains the first stage of the boot loader. GRUB calls this stage 1.
2. Due to its tiny size, the code in the MBR does just enough to load another sector from disk that contains additional bootstrap code. This sector might be the boot sector for a partition, but could also be a sector that was hard-coded into the MBR code when the MBR was installed.
3. The MBR code plus code loaded in step 2 then read a file containing the second stage of the boot loader. In GRUB this is GRUB Stage 2, and in Windows Server this is c:\NTLDR. If step 2 fails in Windows you'd get a message like "NTLDR is missing." The stage 2 code then reads a boot configuration file (e.g., grub.conf in GRUB, boot.ini in Windows). It then presents boot choices to the user or simply goes ahead in a single-boot system.
4. At this point the boot loader code needs to fire up a kernel. It must know enough about file systems to read the kernel from the boot partition. In Linux this means reading a file like "vmlinuz-2.6.22-14-server" containing the kernel, loading the file into memory, and jumping to the kernel bootstrap code. In Windows Server 2003 some of the kernel start-up code is separate from the kernel image itself and is actually embedded into NTLDR. After performing several initializations, NTDLR loads the kernel image from file c:\Windows\System32\ntoskrnl.exe and, just as GRUB does, jumps to the kernel entry point.
5. There's a complication worth mentioning. The image for a current Linux kernel, even compressed, does not fit into the 640K of RAM available in real mode. My vanilla Ubuntu kernel is 1.7 MB compressed. Yet the boot loader must run in real mode in order to call the BIOS routines for reading from the disk, since the kernel is clearly not available at that point. The solution is the venerable unreal mode. This is not a true processor mode, but rather a technique where a program switches back and forth between real mode and protected mode in order to access memory above 1MB while still using the BIOS. If you read GRUB source code, you'll see these transitions all over the place (look under stage2/ for calls to real_to_prot and prot_to_real). At the end of this sticky process the loader has stuffed the kernel in memory, by hook or by crook, but it leaves the processor in real mode when it's done. ■

Gustavo Duarte founded his first start up as a freshman in high school, building a web-based stock market analysis tool in Brazil. He sold that company at 18 and emigrated to the US, and now divides his time between the two countries developing software, authoring technical material, and riding snow and waves. He can be reached at *gustavo@duartes.org*

Reprinted with permission of the original author.
First appeared in *hn.my/bootup* (duartes.org)


Rejection can hurt:
But there is always a chance,
To click on Restart
- Chris, Tracker customer since 2011