IN THIS ISSUE

* Ron Jeffries on estimation
* Steven K. Roberts on Gonzo Engineering
* Mark Chu-Carroll on P-adic Math
* Johanna Rothman on interviewing
* Paul Callaghan on a new paradigm
* and John Shade on fear and loafing
Contents

FEATURES

Estimation ................................................................................................................................. 6
by Ron Jeffries
Two months ago in these pages, Ron Jeffries told us estimation is evil. Now he’s back to tell us it’s a necessary evil—and that, done right, it isn’t even evil.

Launching a Gonzo Engineering Project ............................................................ 13
by Steven K. Roberts
The legendary gonzo engineer shares his secrets for pursuing crazy dreams and succeeding in this series.

P-adic Math ......................................................................................................................... 22
by Mark Chu-Carroll
P-adic numbers are an alternative to the reals. They have real applications, but playing with p-adics is also math geek fun.

Questions Never to Ask a Candidate ................................................................. 29
by Johanna Rothman
If you’re asking candidates any of these questions, you should stop now. Johanna tells you why.

Dependent Types ............................................................................................................. 33
by Paul Callaghan
Paul digs deeper into dependent types, which can often replace tests in creating confidence in code.
DEPARTMENTS

Up Front ........................................................................................................................... 1
by Michael Swaine
This issue's about strange math, gonzo engineering, and some ideas that may change your thinking.

Coming Up ................................................................................................................ 2
Changes are coming. Here's the scoop.

Choice Bits ................................................................................................................ 3
We follow Twitter so you don't have to.

Calendar ..................................................................................................................... 43
Want to meet one of the Pragmatic Bookshelf authors face-to-face? Here's where they'll be in the coming months.

Bookshelf ................................................................................................................... 47
by Andy Hunt
What's new and what's hot from the Pragmatic Bookshelf.

Shady Illuminations .............................................................................................. 48
by John Shade
John faces his fear and hands it his car keys and wallet.
Real and complex numbers extend the rational number system, providing solutions to equations like $x^2 = 2$ and $x^2 = -2$. They’re pretty handy things to have around. But real and complex numbers are not the only way to extend the rational numbers. P-adic numbers also do the job. P-adic math has some really peculiar properties, but because it encodes congruence information very naturally, it turns out to be easier to do certain things in P-adic math than in real/complex math.

Like proving that no three positive integers $a$, $b$, and $c$ can satisfy the equation $a^n + b^n = c^n$ for any integer value of $n$ greater than two.

But P-adic math is worthy of your attention even if you’re not trying to come up with your own proof of Fermat’s Last Theorem. Because it will stretch your mind. Mark Chu-Carroll’s introduction to the bizarre world of P-adic math, where numbers run backward and closeness doesn’t mean what you think it means, is a mind-stretcher.

The other articles in this issue should give your mind some exercise, too. Like our lead article by Ron Jeffries on estimation. Two months ago in these pages Ron convinced you that estimation is evil. He’s back this month to tell you that estimation is necessary. And he’s right both times.

Paul Callaghan will stretch your mental muscles with his article on dependent types. Paul’s done a lot of work on dependent types and thinks the idea is so powerful and important that it could just be a whole new programming paradigm.

Johanna Rothman warns about the questions you should never ask when interviewing prospective team members. And Steven K. Roberts continues sharing his secrets for pursuing your crazy dream without going broke in the process. John Shade puts in a good word for fear. And our new “Coming Up” page describes one of the new departments we have in the works.

One other thing: We’ve been polling you on whether you’d be willing to pay something for PragPub. We’d love to keep giving the magazine away, but it’s just not sustainable. The answer seems to be, “Yes—something.” So we’re soon going to be asking you to pay a little something for PragPub. We’re still nailing down the details, and the system won’t be everything we want immediately. We’ll continue to tweak it based on your feedback. Because you’re the boss. Particularly if you’re paying us.
Coming Up

Your Career Is Your Job

Starting next month, Johanna Rothman and Andy Lester will be starting a new Department discussing job hunting, interviews, hiring, and other work life issues. I’ll let them describe what they’re planning. –Mike

We’ve long wanted to work together on a project, and when PragPub editor Michael Swaine asked if we’d like to explore the topic of cultural fit in hiring, we immediately agreed. The result was a pair of articles in the March issue of PragPub, one from the hiring manager’s point of view and one from the candidate’s.

As we talked, we realized that we both constantly push the message that all the interesting questions have an “It depends” answer. It’s a really important message to get out there. Read any jobs subreddit on reddit.com, such as /r/jobs or /r/cscareerquestions, and you’ll see a common expectation that there’s a single right answer for everything. There isn’t.

When Johanna blogged about favorite interview questions recently, she was pleasantly surprised at the variety of questions readers shared. One of her readers asks the question, “Tell me about a difficult problem you have debugged.” The way he asks this question, the answer could be just about anything. An excellent question. But there is no One Best Interview Question.

Andy is often asked, “Should I put X on my resume?” where X could be hobbies, or a job that lasted only six months, or fraternity associations. His answer is always “It depends,” because what you really need is to understand the issues to consider in answering this question for yourself.

We’re both wary and weary of cookie-cutter answers to career questions. Your career is yours, and yours alone. We can suggest alternatives and things to consider, but you are the best decision maker for you.

So we’re not going to offer you cookie-cutter answers. We’ll present as much perspective on a topic as we can, to help you make your own decisions that best fit your situation. We won’t always agree. There’s room for disagreement, even between us.

Finally, we’re really interested in what you want to know about. What are your job, career, and work-life concerns or questions? Let us know. –JR & AL

Andy is the author of Land the Tech Job You Love.

Choice Bits

What the Cool Kids Are Tweeting

We follow Twitter so you don’t have to.

Looking for our top-eleven list? We didn’t eliminate it; we just moved it to a new page toward the back of the magazine where we talk about what’s new on the Bookshelf.

Words of Advice

- We’re gonna have to accept that our programming communities aren’t as private as they once were. The in-laws are coming over. Hide the bong. — @garannm
- Postel’s law: Be conservative in what you do, be liberal in what you accept from others. — @estellevw
- There’s always room for something better than Jello. — @RonJeffries
- “Here’s a mantra to remember: TED Talks — interesting if true.” — @hnatko
- Whenever you see a smiling child, remember: she’s never read a comment in her life, and she’s doing just fine. — @AvoidComments
- “A good way to gain credibility is not to have lied to the same audience last month” from http://bit.ly/148rHdC with @EdwardTufte — @KentBeck
- 1. Take pictures of every cat in your neighborhood. 2. Make missing cat posters with the pictures. 3. Get all the cats. — @lazerdoov
- git undo! git undo! — @anna_debenham

Humble Reflections

- I should audit my tweets some time to see exactly how many typos I have in them. — @tswicegood
- Working from home has taught me I need to come up with better excuses to my daughter about where all the cookies went during the day. — @antiheroine
- So this is what inbox zero feels like. Less “zen” than “where did my life go, while I was getting to inbox zero?” — @anamariecox
- I’m always thrilled when I hear a smart programmer say, “it turns out…” I know I’m about to learn something interesting. — @KentBeck
- I’m in my HTML5 book, throwing out anything related to IE6 and 7. It’s an awesome feeling. — @bphogan
- I love when I find sticky notes to myself that detail the next steps in a project. My present knows how absentminded my future self is. — @jessebeach

Trenchant Observations

- Sure, Steve Jobs made the “black turtleneck/blue jeans/sneakers” look famous… but let’s not forget who committed to it first: Jonny Quest. — @BradBirdA113
- I believe this is the first time Georgetown has fielded a team entirely selected from its political science department. — @anamariecox
• It’s not that Clojure/Lisp has a lot of parentheses. It’s just that we removed everything else. — @puredanger

• I wish to note Android’s voice-to-text feature does not handle well the distressed calf noise dawn and I often use to communicate feelings. — @marick

• I’m not sure the word Master should be in the name of anything that can be learned in 2 days. I propose 5 years of being called a Scrumster. — @richardadalton

• “When the spontaneous symmetry is broken in an evolving environment, here comes modularity.” reads like poetry *and* science, though. — @terriko

• Just bought a TV for less than the unsubsidized price of an iPhone. It will likely last 10+ years. Can’t see Apple entering this business. — @marcoarment

• Just learnt that “pot plant” does not translate accurately from British English to American English. — @Suw

• Hey, remember DLLs? More like DLols amirite? Seriously though, DLLs can burn in hell. — @ryan

• Sorry to hear about the fire at Chez Panisse. On the plus side, the fire was sourced locally. — @SFNick

• It appears that people are using the prefix ”artisanal” freely and totally without irony. — @edd

• OH: “Red is the new green.” — @emmaguy

### Profound Musings

• I’m wondering where all of the 2 cents go that I keep reading about. Someone must be getting rich. — @rbates

• I’m all for buying DRM content that can be switched off in the server. I plan to pay for it with DRM money that has the same property. — @raganwald

• It’s a fine line between “out of date” and “progressively retro.” — @KentBeck

• My big question in big data is this: what happens to a domain when most of what is knowable is known? (markets, privacy, crime, etc.) — @edd

• What’s Apple doing gloating about JD Powers’ awards? I thought that was for 2nd tier car makers and insurance agents. http://yfrog.com/es7dxphaj — @dhh

### What the Rich and Famous Are Doing

• Getting my personalized @Makerbot 3D shrunken head with @EverettCase (@MakerBot Store). — @SteveCase

• One of the #lemurs had twins! Will keep you posted on all the lemurs’ progress. http://vtng.in/mlp — @richardbranson

• First “hottest…” list I have been on and wouldn’t you just know it? It’s a parody. The 41 hottest guys in tech. — @jayrosen_nyu

• 8 things to make with leftover socks. http://is.gd/RuGL2A — @GuyKawasaki

### Movies Refactored

• Are we still doing the whole #agilemovies thing? I have plenty more up my sleeve… — @MarkDalgamo

• Return to the Planning of the Apes #agilemovies — @tastapod

• I know what you shipped last summer #agilemovies — @lassekoskela

• Any Which Way You Kanban. #agilemovies — @jasonlittle

• The why don’t we just outsource to China Syndrome #agileMovies — @sparrk
Who Are Those Guys?

First, they’re not all guys. Second, we have to confess that we cleaned up their punctuation and stuff a little. OK, who we followed this month: Marco Arment, Ryan Bates, J. Renée Beach, Kent Beck, Brad Bird, Ryan Block, Reg Braithwaite, Richard Branson, Steve Case, Ana Marie Cox, Mark Dalgarno, Richard Dalton, Anna Debenham, Don’t Read Comments, Edd Dumbill, Dan Duvall, Jeff Gothelf, Emma Guy, David Heinemeier Hansson, Brian P. Hogan, Andy Ihnatko, Ron Jeffries, Guy Kawasaki, Lasse Koskela, Kason Little, Brian Marick, Garann Means, Alex Miller, Jen Myers, Dan North, Terri O, James O’Brien, J. B. Rainsberger, Jason Reid, Jay Rosen, Suw, Travis Swicegood, Nick Turner, and Estelle Weyl.

Fair’s fair. You can follow us at [www.twitter.com/pragpub](http://www.twitter.com/pragpub).
Estimation

The Best We Can Do

by Ron Jeffries

Two months ago in these pages, Ron Jeffries told us estimation is evil. Now he’s back to tell us it’s a necessary evil—and that, done right, it isn’t even evil.

Yes, there are many abuses that organizations perpetrate around the notion of estimation. Most developers have been injured by these abuses more than once in their career.

Unfortunately, this leads to a common naive view among would-be Agile practitioners whose learning is at an early stage: they want to abolish estimation entirely. This is probably not possible and it is certainly not ideal. We do have the ability to estimate how rapidly we can do some things, and the business can make better decisions if we share what we know.

Development isn’t just a smooth-running machine that cranks out features rapidly. I know it’s comfortable to think we have no responsibility for the business’s concerns of money, time, and dates. It’s just not true. What we build is not solely the purview of some Product Overlord who decides what we’ll do and takes all the risk. The business people may make final calls, but many of the decisions are best made by the team—not just how to do things, but what to do. We do have a responsibility to help guide the project, using our creativity and our special knowledge.

The Agile Manifesto says, “The best architectures, requirements, and designs emerge from self-organizing teams,” and that’s what we meant (emphasis mine). Agile developers often have a good sense of how long things will take, and this is a valuable component of selecting and refining requirements. Developers need to step up and talk about the probable cost of what they are asked to do.

Concerns with Estimation

There really are a number of serious problems relating to the estimation of software work. In a previous article “Estimation is Evil”[U1], I described some of them. Let’s review:

- Predefined backlogs of work reduce creativity and inhibit steering the project for success.
- Demanding delivery of “everything” by some fixed date is a trick that never works.
- Treating estimates as promises leads to conservative teams and disappointing results.
- Trying to improve the “quality” of estimates is attention that could be better paid to improving the quality of the product.
- And so on.

In that article, I argue that the result of this focus is “weak Agile.” In essence, teams focused on estimation are generally working the short end of the lever.
of cost vs. value. Better teams focus more on value and less on cost. They still consider cost, because cost is one important component of delivering value. But value is their primary interest.

It’s true that many top teams do not estimate in any real sense of the term. They work on small stories, small enough that to get a sense of progress you can just count them. They work on the most valuable things first—using whatever notion of value they have. They adjust their outlook frequently, and adjust what they work on next in accord with what is happening. They work to produce a continuous flow of valuable software, and they deploy it as frequently as possible, often daily.

Since estimation is often associated with dysfunctional or weak teams, and since great teams seem to work without estimation, many Agile practitioners strongly resist estimation of any kind. And it’s good to build up a team—and an organization—that is strong enough to work without estimation. But this is an advanced way to work, not the way to start. It’s not good to refuse to estimate, or even to resent it, in an organization where it’s needed. On the contrary, we need to learn to do it well, in a way that influences the organization positively.

**Budgeting a New Project**

Now, let’s look at just one legitimate need, helping the organization decide whether to go ahead with a project.

When something goes wrong with our house or our car, and we don’t have all the money in the world, we probably get estimates on fixing things. We probably consider alternatives: if our trunk lid is scratched, should we repaint the whole car for a perfect color match, just spray the trunk lid, try to polish out the scratches, or just live with a scratched trunk? We pick an option that seems to be the best use of our money.

Business people budgeting projects have the same problem of allocating funds. They need to consider how to deliver as much value as much possible, within budget. In a small organization trying to create some amazing new product, the concern is even more pressing. We need to start generating positive cash flow before we run out of money!

In situations like these, we really need some estimation. We need to know whether it’s wise to undertake this new project. Business people have some estimates in mind already: How many people will be interested? What percentage of prospects will turn into buyers? How much will buyers pay for the product?

But they need to decide: Can we begin to bring in enough money to stay alive before our cash runs out?

The decision maker knows that all these estimates are estimates, and uses his best judgment to put value to them. But there’s a key area he doesn’t know much about: How much of this product can we get done, by what date, for how much money? This is a technical question and it needs to be answered by technical people.
Yes, of course we don’t know either. We really don’t know how long it will take. But let’s face it, we have a better idea of how long things take than our poor entrepreneur has, and he needs our help.

I believe that we can make a good guess at whether it’s a week, a month, a quarter, or a year before we start getting a good sense of how fast we can progress. And that guess, wild though it may be, is a better guess than the business person can make.

Can’t we just say “Try it?” It seems we should be able to say something like this:

“Our team costs ten thousand dollars a week. Let’s start working on the most important, most risky, most informative parts of your product, and build up a sense of how hard it is and how long things take. You’ll see us building what you ask for. You’ll see it working. You’ll be able to decide whether to keep investing or to stop.”

It seems to make sense to say that—unless it’s your personal Ten Large rolling out every week. Then you have more questions. “Am I going to know within a week? Will I have to wait a month to know? Will I know by June? Holy cow, guys, there’s over a hundred grand between now and June. That’s big money! Cut me a break here, will you? How much is this thing going to cost?”

We need to step up to this kind of challenge. We need to give as solid an answer as we can, even when we don’t have an absolutely concrete answer.

We’re standing in sand. How do we get concrete?

The question is, “How long will it take, and how much money, to get to a product people might buy?”

We’d like to say, “Well, start spending money with us, and if you get bored, stop,” but that’s not helpful. Can we turn this idea around a bit? Let’s begin with some very broad estimation just to get a sense of where we are.

We probably have some sense of whether it’ll take a week, a month, or six months to make something people can use. If so, let’s talk about it. Our investor has some amount in mind that he can spend. Let’s find out how much that is, or get a sense of it in conversation.

Suppose we kind of think that maybe we could get a rudimentary but usable product in six months, and the investor is willing to invest up to about a million dollars to get to positive cash flow. A million dollars is about a year of our time, and if it took us nine months rather than six, we’d need to bring in a million dollars in three months to get positive. Let’s talk about that. How rapidly does he expect revenue to grow with a first release? Suppose he thinks revenue will grow that rapidly, so that if we do take nine months, we’ll be OK. That sounds risky to me, but it sounds like this project is at least worth exploring.

Remember how Agile works. We want to get our business people fully engaged in making live decisions about spending money based on what they see. So let’s get to thinking about feature selection, not just dates.
Estimation You Can Live With

“OK, intuitively, we think we could get an initial release in six months, and you think if it even took nine months, the project would be viable. That looks good, but it still seems risky. We’d hate to see you spend that much and get nothing, and these are still wild guesses.

“So let’s do this: Let’s take two weeks to produce a more concrete answer. Two weeks will cost you only $20,000, which is a lot less than a million. In that two weeks, you’ll work actively with us on the most important aspects of the product. Some will be things like look and feel, and some will be driven by other business or technical risks that we can foresee now. At the end of the two weeks, we decide whether to go forward. Our job is to show you, concretely, what we can build that convinces us all whether we should go ahead. Your job is to guide us by describing what you want, and to work with us to identify risks and concerns.

“If after two weeks, things look bad, we’ll know it and we’ll recommend that you stop. If things look good so far, we’ll decide together what the next major decision point will be. It might be a month out, or three months out. Frankly, we’re perfectly happy to do this every two weeks.

“Yes, that’s right. Every two weeks we’ll talk with you about what to do next. We’ll do it and show it to you. If it’s good enough, we’ll continue. If not, we’ll stop.

“That way, your risk is never more than another $20,000. You can decide whether to use that money to get more information, to redirect our efforts, or that it would better be spent in some other way.

“Can we work like that?”

If he says yes, let’s get started. If he says no, we need to figure out why, and whether this project is for us.

This is definitely estimation, and it’s a kind of estimation that we can do in concert with our business-side people rather than in conflict with them.

We estimate, with some certainty, that we can produce useful information in two weeks. If we think we can’t do it, we’d better suggest four or six or eight weeks.

Furthermore, we estimate that a viable product is probably possible in six to nine months. If we have no idea, or worse yet, we sincerely doubt it, then we cannot legitimately invite him to find out—at least not in the terms above. We owe it to him to say, “Frankly, this sounds like a two- or three-year project to us. We could be wrong: here’s what we’d have to learn to find out. Here’s what it would cost to find out, and our best guess right now is that the answer would be bad news.”

Sometimes we have a good idea what we can do. Sometimes we know less. Either way by estimating what we know, we can frame experiments—investments—that improve our ability to decide what to do.

When it comes to spending other people’s money, I think we owe it to them to do that.
This is Estimation, Not Negotiation!

An estimate doesn’t have to be “This project will be done on Tuesday, May 14th, at 2:35PM.” An estimate is supposed to express a range of possibilities. It could be perfectly OK to say something like, “We see no way to get this done in April. If everything went perfectly, we might be ready in mid-May. Based on our experience, we’re thinking June, maybe July.”

Whether we say that or not, odds are we know it, or suspect it. And if that’s what we think, the project needs that information. However, if we put it that way, we’re going to get push back. Someone is going to challenge us to prove beyond a shadow of a doubt that we can’t get done by April 30th, and if somehow we do that, they’ll say, “What about May 5th?” Bah, we hate when they do that. So how can we give the project the information it needs without playing this game?

Move from Date Estimation to Velocity Estimation

Let’s use the Five Card method from the previous article, not to estimate a date, but to break down the whole project into sub-stories small enough that our team can do, say, five of them in two weeks. Of course we might only get three done. We might get six but we doubt it. So we come back with an estimate in terms of velocity:

“We have broken down this project into small stories. We believe that on stories of this size, we’ll be able to do between three and five every two weeks. And at the end of each two week period, everything we’ve done will be visible, tested, integrated, and working. So whether it’s five or six or two per week, we’ll know, and you’ll know.

“If you can remove or defer some of these stories and still have a viable product, you will bring the product in sooner. If you can simplify them, you can bring the product in sooner. If you add stories, or make things more difficult, you’ll push the product out.

“It’s up to you. We can split things down to this size, we can guess our speed now at about three items per two weeks, and you’ll know every two weeks what’s really happening. You can use that information to decide what to do, and what to defer, so that you can get the best possible product for any date you pick.”

Help Management Learn to Use Velocity

Now management may still want to negotiate this with us. If they add up the figures to get dates, and then start trying to adjust the dates, don’t negotiate: turn the conversation back to the rate of progress. “The date is up to you. If you choose more to do, it will be later. If you choose less, it will be sooner. We think we’ll do between three and five items every two weeks. We could be wrong: if we are, you’ll know as soon as we will.”

Someone might try to push the velocity. “Can’t you do seven every two weeks? That way you could get done on time.”

Remember how Agile works: developers work at an optimal sustainable pace; developers aren’t in charge of dates; the business side owns the dates, and owns the responsibility to meet the dates.
The developers’ job is to deliver at the best possible pace: “We think we can do three to five. If we happen to go faster, you’ll know. If we go slower, you’ll know that too. When we get done depends on that, but it depends even more on how much, or how little, you give us to do. What’s put in by the date is up to you. You should plan on three to five items per two weeks, and keep an eye on what actually happens.”

They may push more than once for faster. But we have the facts at our disposal, such as they are. “Based on our experience, we’ll get between three and five done. It would be unwise to assume anything higher than that, because it’s not likely to happen. If something happens that lets us go faster, we’ll all know it and you can select more items into the release.”

All this, of course, is more about communication, not much about estimation, and emphatically not about negotiation. The estimation was trivial: we did it way up there when we split the cards. Now we’re communicating what we know, what we guess, what we believe.

We give management our best estimates of our production rate, and we stick to them, improving them only as we learn. We continue to make it clear that the business people control the date by choosing how much work they ask for.

Optimizing the Product

Whether you’re all for estimates or not, you should know that Agile is about building the product incrementally, in good condition from day one to the very end, and getting value from it as soon as possible. This requires great technical skill, but technical skill is not enough. The people paying for our work need to allocate their resources. They need to manage their cash, and they need to coordinate our work with activities and stakeholders outside our team.

To do this effectively, they need information on how long things take, which translates into how much things cost, and into how long it takes before we start getting our money back. They need information that we have to do this job well. The fact that our information is vague, flawed, and uncertain is not a reason to hold it back. We need to get the information into their hands, and to do it in a way that gives them the best chance of using it well.

Refusing to estimate is not the right way to do that. Estimation is necessary, and effective communication of estimation is necessary as well.

The best way to do things is to develop small features at the best possible sustainable pace, and to use the information generated to help the business people decide what to ask for, and what to defer. To do this, we must do estimation. We slice stories to an estimated small size, and we estimate our rate of delivering them. This kind of estimation isn’t bad at all: it’s incredibly useful and it’s what the best agile teams do.

About the Author

Ron Jeffries has been developing software longer than most people have been alive. He holds advanced degrees in mathematics and computer science, both earned before negative integers had been invented. His teams have built operating systems, compilers, relational database systems, and a large range of applications. Ron’s software products have produced revenue of over half a billion dollars, and he wonders why he didn’t get any of it.

Send the author your feedback or discuss the article in the magazine forum.
External resources referenced in this article:

2.mailto:michael@pragprog.com?subject=estimating
Launching a Gonzo Engineering Project

IV: The Media Dance

by Steven K. Roberts

The infamous 580-pound, 105-speed BEHEMOTH, with Mac, SPARC, and DOS environments as well as satellite datacomm, HF/VHF/UHF ham radio, heads-up display, head mouse, handlebar keyboard, 6-level security system, speech synthesis, 72 watt solar array, and deployable landing gear to keep the monster upright on killer hills. The bike now resides in The Computer History Museum.

This is the fourth installment in a series of articles unlike anything we’ve ever published. Steven K. Roberts has figured out how to live passionately, pursuing crazy dreams and building fantastic machines (like BEHEMOTH and Microship, both of which are pictured and briefly described here) and going on amazing adventures. He calls what he does Gonzo Engineering, and in this series he tells you everything you need to know in order to pursue your own crazy gonzo engineering dream.

The Media Dance

I don’t think my projects would have survived without all the media coverage over the years. It has opened more doors than I probably even realize, led directly to writing and speaking gigs, smoothed the way to countless sponsorships, sparked friendly waves and warm hospitality offers, and probably even saved my life on narrow roads (“Goddamn, slow down a second, Bubba, there’s that computer dude I heard about on the TV!”) Though one could certainly make cynical observations about all this, it is true that in our media-centric culture a little fame goes a long way—although it does not, alas, automatically imply fortune. Trust me on this.

Like sponsorship, media presence is an essential component of a gonzo engineering project; it has a way of making everything else easier. But there is an art to it... don’t assume that just because you’re doing something original, reporters will flock to your lab and write gushing stories. Let’s explore this a bit.

The Digestive Tract of a Horse

One of the most important things to remember is that you have to somehow manage to be newsworthy without drifting too far into the domain of self-promotion: there is nothing more tedious than someone who is always tooting his own horn and viewing the world through the filter of his own personal obsession. Media professionals have a finely tuned radar for this sort of thing—as well as blatant product promotion. They know when they’re being manipulated, and it can backfire badly since they want to be in charge of the story and do not welcome attempts on the part of the subject to run the show.

But with a little care, you can use this to your advantage... just make sure that you’re providing a clear and consistent image rather than amorphous raw material that can be digested into a different story entirely. This happened to us one year at the Hackers Conference; a crew from CBS News showed up, took a few hours of footage, and produced a completely unrelated piece on computer crime illustrated with B-roll of some of the industry’s leading luminaries in the woods above Silicon Valley.
The media were never invited back, of course, but damage bordering on libel had already been done.

Admittedly, a whole conference is a bit hard to control, and the wide range of colorful personalities gave them enough material to support just about any fiction; they could have done a piece on the favorite hobbies of extra-terrestrials had they so desired, armed as they were with both a serious shortage of knowledge and extensive video of übergeeks at play.

Fortunately, as the brains behind a gonzo engineering project, you have the opportunity to keep a much tighter rein on the media beast... it can even be your friend.

I was lucky... my first lesson along these lines was delivered in a helpful and non-destructive way. I was doing an interview in 1984 with CBS Morning News, my first national television appearance. Unlike their evening-news counterparts, this team was professional, their production standards were high, and we got along beautifully. But still I was in for a shock: with the cameras rolling, the interviewer asked how the computer networks figured into my bicycle-touring life. I launched into a long and precise explanation, filled with rhapsodic asides and pithy anecdotes.

"Cut!" cried the producer. "I can’t use any of that. You’re referring forward to things you haven’t said yet and back to things you said five minutes ago—how am I supposed to edit? I want 20-second speech bites, ending on a downward inflection with your mouth closed.” This was delivered with a grin, so I knew he was yanking my chain a bit, but there was truth in his words. As painful as it is, you have to be able to package even the most abstract concepts in clear, memorable packets... pausing in between to suggest natural edit points. And you need to do it in a way that still sounds like natural speech, not like a rehearsed stand-up delivered by a news reporter. It’s a fine line, and takes practice.

The metaphor to remember is that the media is a big horse: if you stuff straw into one end, something remarkably unlike straw comes out the other. But if you insert bricks, you get bricks out the other end. You need to develop concise comments and colorful sayings that collectively define your work, then casually speak in those during interviews the way a Zen master speaks in koans.

We can extend the media-as-horse metaphor a bit further: it is always hungry, and this works to your advantage. Every minute of air time, every column-inch in the newspaper, and every page of editorial space in magazines has to be filled with content... day after day, month after month. Coupled with fierce competition at every level, this translates into a lot of media channels just aching to tell your story if it’s sufficiently interesting. The trick is giving them a heads-up without making it obvious that you want media coverage; as in the dating scene, there’s nothing like an air of desperation to send people scurrying in the opposite direction. I’ve had hundreds of articles and interviews, but have never called a station or publication to offer an interview—not once.

So what’s the trick?

Well, think about it from the reporter’s perspective. Deadline is approaching, and there’s a hole to fill. What’s hot, new, visual, funny, poignant, disturbing, or likely to yield a Pulitzer Prize? You develop a keen eye for fresh material:
trolling the web, listening for scuttlebut, and scanning other publications in non-overlapping markets. Do you see the implications of this? Sometimes all it takes is one or two stories to trigger a whole series—I’ve had otherwise unremarkable local news items explode within days to a spot on CNN, calls from feature-writers for glossy trade magazines, requests for photos and interviews from international business publications, and an hour on the Donahue show. Starting the process can be as simple as having someone else call in a tip, or visibly doing what you do and letting them notice you. Occasionally, the symbiosis with sponsors can come into play here, as companies who actually sell things are less shy about initiating contact with the press and may welcome the news hook that you represent.

If you do need to draw attention to yourself, do it in the form of a low-key press release with an absolute minimum of hype and exaggeration. They’ve seen it all, believe me, and anything that looks blatantly self-promotional or less than professional goes straight to the recycling bin.

Basically, the only trick is getting started... and then keeping the story interesting. Let’s look at this a bit more closely.

The Microship, the result of an 8-year development project involving extensive sponsorship, students, and volunteer teams. This is an amphibian pedal/solar/sail micro-trimaran with retractable wheels, hydraulic systems, 480 watts of peak-power-tracked solar panels, and zippy performance under sail. BEHEMOTH, the Microship, and the later Nomadness project are all documented at microship.com.

The Project Moniker

Much of the art of being interesting to the media is simply having an image that can be grasped and communicated in a short news segment. Think about a spot on the evening news, or a 300-word “tight and bright” piece in the inside pages of USA Today: if it takes an hour to explain to a fellow geek what you’re doing, it might be impossible to get it across effectively to a bright-eyed reporter whose last assignment involved converting a consumer product press release into a photo/caption puff piece. In other words, those sound bites we talked about earlier aren’t just for use during the artificial reality of being on camera; you have to be able to convey the essence of your project in few words
with startling clarity, tweak it in various compelling ways to appeal to different markets, then keep it dynamic as the years pass so it doesn’t fade into the static.

The entertainment industry has a name for this: the elevator pitch. If you can’t get the story idea across in the time it takes to ride the elevator, then it is too complicated.

An important component of this, as trivial as it may sound, is the project name. You need a memorable handle, and one unique enough that even the most watered-down article or happy-news blurb will leave people with a way to find out more by typing something into Google. Otherwise, you lose one of the most important side-effects of media coverage—the ability to attract volunteers, sponsors, and growing image recognition. Consider these two hypothetical spots about the same fictional project, delivered in the voice of an evening-news anchor:

“And finally this evening, there’s a man in the U district who’s been working for three years on a way to help people interact with computers using a whole variety of gestures, not just typing on a keyboard and manipulating a mouse. He uses cameras and sensors to keep track of the movements of a user’s hands, allowing the creation of a customized visual language to increase the efficiency of man-machine communication."

“And finally this evening, meet the Gesturizer. This remarkable device was invented by a man in the U district, and it lets you interact with your computer the way the deaf communicate with sign language. He says it takes the average user only two days to double the efficiency of computer use."

By coming up with a catchy name (even a silly one like “Gesturizer,” which does not exist at this writing) our subject has created a clear label that ensures every news story will unerringly point to his website, or at least be easily searchable. When you hear another story about this project six months later, it will sound familiar; with any luck, you’ve even bookmarked gesturizer.com in your browser or signed up for his occasional email updates. And something with a quirky and memorable name certainly makes for a more lively news story than the abstract explanation in the first example.

Notice also, in the second snippet, that there is something like a sound bite nestled in there. It’s a fair bet that the reporter scribbled “interact w/ comp like deaf w/ sign language” during the interview, knowing that a visual image would make a good story lead. There is even a startling statistic to further capture the imagination: wow, only a couple of days’ practice to double my efficiency? Where do I sign?

By maintaining steady media coverage for the Gesturizer over a few years, our fictional developer enjoys the luxury of growing image recognition, thus creating a context for new stories and updates:

“You may recall the amazing Gesturizer that we reported on last Spring. Well, the inventor now reports that his system can actually recognize sign language from across the room, no longer requiring a computer user to remain tethered to a keyboard and mouse—or even a desk! Our science reporter, Bob Archibald, has the story…”

Without a memorable name and a tasty sound bite, this follow-up story would have had a much less compelling introduction.
This sort of thing may be anathema to geeks who want to remain focused on system development, not activities that seem better suited to a marketing department. Clever names, websites, image-management, media spin, sound bites... what does all this have to do with gonzo engineering? Believe me, it pays off, even if you haven't the slightest intention of ever selling anything—you can't run a massive project in a vacuum, and the media is your best ticket to meme-propagation.

**The Art of the Demo**

If you thought that was bad, now I'm really going to make you gag. When you're dealing with mainstream media instead of highly targeted journals in your native field, you have to allow for the possibility that they might not have the slightest clue what you're talking about. I've actually been on the other side of this; the first time I covered an Artificial Intelligence conference for *Byte*, it took a while for me to see natural language systems as anything more than fancy incarnations of Eliza. (That particular subculture had a real PR problem—most of their publishing activities were incestuously contained within the AI community itself, and the general media was getting a distorted, hype-filled view of what was going on... creating inflated expectations that were destined to lead to disappointment.)

The problem here is that when the topic is a sufficiently interesting system, there is usually a rich shared context with the audience that you can normally just assume. This is why publishing technical articles is easy; you already know that your geek readers understand the prior art, and will thus get your clever hack or new paradigm if you explain it in that context. But now sit a writer for the *Podunk Weekly News* down at the console and perform your usual demo. Blank stares? Tentative mouse clicks on irrelevant objects? If you're hearing questions like, "So, um, is this thing connected up to the Internet? Does it get TV? Who would actually use it?" then you need to create another class of demo.

This was easy and fun with *Winnebiko* and *BEHEMOTH*—showing the act of writing-while-riding was not particularly dramatic, though the more clever reporters identified with it immediately. (I am, after all, in their business... yet not enslaved to a desk.) But I needed a few cute things that would play well on camera while conveying the essence of the system—the operational equivalent of a sound bite. I coded up some simple scripts that could be remotely invoked with keypad commands via a hand-held ham-radio transceiver, and used those to orchestrate scenarios that worked particularly well with the TV news.

In North Carolina one afternoon, a reporter stepped in front of my bike and began his summarizing "stand-up." As his shadow fell across the solar array, I surreptitiously sent a Touch-Tone command and the bike's synthesizer sternly spoke: "Excuse me. You are blocking my sunlight." It was priceless... still on camera, he quickly stepped away, saying, "Oh, I'm sorry."

I had a canned collection of remotely triggerable speech strings, and in almost every interview they came in handy. There were also some large-font files (themselves serving as "text bites") that I would pop up and edit when demonstrating the handlebar keyboard, eliminating the *quick brown fox* or
worse, the *fsdfsdfs* syndrome. A couple of favorite CDs were always at hand to show off the music system; I became adept at popping the top off the Qualcomm OmniTRACS satellite terminal and demonstrating azimuthal tracking upon loss and reacquisition of signal; I would hand the reporter my sweaty helmet and let them see their own contact information in the heads-up display. Coupled with ongoing patter about the uses of the underlying technology while on a bicycle tour and its implications for future widespread nomadic connectivity, this sort of thing added visual appeal and contributed to my ability to drive the story with consistent content, rather than just answer the usual questions (What did that cost to build? Has anyone tried to steal it? Is it hard to pedal up hill? Have you had any accidents? Where do you sleep? What do you do in the rain?)

**Leveraging the Media Portfolio**

Over time, with a little care and luck, a sufficiently interesting project will begin to accumulate clips—newspaper articles, tear sheets from magazines, and a collection of video dubs. This stuff is gold, and can serve you well for years if you use it wisely.

First, keep good records, both in the form of a mailable document and a list on your website. Media coverage is one of those things, like credit, that exhibits positive feedback: the more you have, the easier it is to get more. Any doubts a reporter may have about your credibility will evaporate in the face of a good list of previous coverage, although there is a limiting factor in that some publications are sensitive about being "scooped." When I first passed through Seattle on the bike in 1986, I interviewed with both of the daily papers... and the *Post-Intelligencer* dropped the story when they found out I had already been on the front page of the *Times*. (That was annoying, but not as bad as the New Orleans *Times-Picayune* that killed the story because I didn’t have a “handicap” that would make a cross-country bicycle trip newsworthy!) But in the larger world of magazines, this has rarely been a problem; there is always a different angle available, and the value of a story becomes amplified by the celebrity effect.

Another reason to keep these lists, of course, is to facilitate sponsorship proposals. Even if someone hasn’t already read about you somewhere, they are sure to be impressed by a long list of publication credits backed up with a few carefully selected photocopies. The collection becomes immensely valuable and will be important to you for most of your life, so put those articles in protective sleeves and keep them in binders! The best ones should even have backups.

Note how this keeps looking like a 3-way symbiosis: sponsors get media coverage, you get goodies, the media gets stories about interesting technology. All three components have to be carefully balanced, ultimately driven by the energy of the project itself. If you plan it well and keep it evolving in interesting directions, it can even become self-sustaining.
Three-way symbiosis of project, sponsors, and media. Everybody wins as long as the process keeps moving forward, and it is up to you to drive that. New toys are inspiring; build something amazing with them, and the press will tell your story. If you craft interviews to honestly underscore the role of your sponsor(s) without blatant flag-flying, then you can keep the outer loop going indefinitely. The hard part is the leap of faith on all sides necessary to get it started. Inner, direct connections are just as important, and the most productive relationships contain all the elements shown here.

A Public Presence

We have been talking about a public presence already, of course, but media coverage is a unique phenomenon: although you can increase the odds of accuracy and a positive spin with a good name and some clever sound bites, the published stories are out of your control. If some reporter thinks you’re a colorful nutcase and gives the story a snide “takes all kinds!” slant, there’s not really much you can do about it... and it certainly isn’t going to help build a positive project image. (Don’t believe the old nonsense about any publicity being good as long as they spell your name right!)

There is another aspect to the public face of the project, however, and over this you have complete control. Thanks to the Internet and common design tools, anybody can be a publisher these days. If you were born after 1980 or so, you may find it hard to believe that there was a time when every geek didn’t have a website with content-management software, mailing list, blog, and a portfolio of domain names reserving his or her pet neologisms. In the Olden Days, we actually typed things directly onto paper, character-by-character (sometimes even paying professional “typesetters” if something needed large fonts or right-justification), then produced expensive print runs of hardcopy publications. Distribution, in the pre-Net era, involved affixing colorful gummed receipts for the transportation charges, then handing over these
“stamped” hardcopies, sorted by zip code, to uniformed workers who would transport them to their destinations in a sort of physical analogue of packet-switching. <creak> Am I dating myself? (No, but I am sitting by the phone waiting for me to call.)

It almost goes without saying that any gonzo engineering project should have a website, updated frequently with news, photos, downloadables, YouTube clips, bloggage, white papers, personnel sketches, public appearances, media coverage, sponsor lists with links, and related resources. I don’t really need to tell you how or why to do this. Just do it.

Beyond that, I highly recommend setting up a few email lists. Some members of “your public” will in fact be proactive enough to check your website regularly (or use RSS) to see what’s new, but there’s a lot of cool stuff out there competing for mind share and, at least in my own overloaded life, I find that I rarely have time to “web surf” anymore. If something is not immediately relevant or referred by a trusted source, I am not likely to stumble across it in the act of poking around… the thousands of bookmarks accumulated since the “Mosaic” era do me little good. If I have a question, I Google my way to the answer; I read a few regular news-aggregators and blogs of current interest; I scan the dozen or so forums that manage to hold my attention. This is probably typical.

But a mailing list is the original “push” technology. Once someone expresses interest in your project, you get to drive the update process. As long as your content is interesting and you don’t start spamming or posting too frequently, that person will probably stay on your list throughout the life of the project. Admin and bouncie-processing is a pain, but it’s worth it; my nomadness list, now over 15 years old, is one of my most important assets. Among the thousands of names are some amazing people: almost without fail, I can mention a conundrum in one of my updates and someone will provide the answer within hours. The list includes sponsors, writers, CEOs, friends, volunteers, engineers, marine architects, sailors, advisors, potential hospitality sites, the curious, the skeptical, and a few non-English speakers who probably thought they were signing up for information about Microchip Technologies PIC development tools and wonder why they occasionally get rambling email about some barco ridicule.

This list is transmit-only, meaning that I am the only person who can post. With thousands of names, the usual discussion and noise could easily get out of hand (“how do I get the *%$! off this list?” followed by a wave of flames), so I restrict this “channel” to pure publishing. But there are other needs as well...

I maintain active discussion lists for technomads in general, participants in the planned flotilla of boatlets, and developers working on the project. Multiple devlists come and go over time to segment the latter into more focused subgroups. (I should point out to the same youngsters who had to be told about paper mail that there was a time when, for practical reasons, volunteers for a project had to be in the same town—not scattered across the planet and unlikely to ever meet face-to-face.)

In addition to websites and mailing lists, it is advisable to have at least one reasonably well-produced print publication that describes the project. Over
the years, I’ve tried to put out quarterly journals, but that’s too much like work (though I once told impatient readers that I actually meant “every quarter-decade”); more successful has been a small series of technical monographs whose flagship is a 110-page self-published book (From BEHEMOTH to Microship (ISBN 1-929470-00-2) that gives a light overview of the whole series of bikes and boats. It’s not highly technical, but it does generate a few nickels while doubling as a presentation piece for those who wander by the lab to help. It’s even a useful resource for reporters, letting them dig around for quotes at their leisure instead of recycling my comfy old sound bites again and again.

For a while, I ambitiously planned to expand the monograph series to a whole product line covering all aspects of the Microship project, but doing so is time-consuming and expensive (and it’s way too easy to end up with either a pile of obsolete inventory or a lot of wasted time designing “publishing on demand” documents that were never, um, demanded). The alternative is easy, fun, and has a “cost of goods sold” that is arbitrarily close to zero: a library of PDF documents tucked into a shopping cart that takes PayPal. These can be sold or given away, further spreading the memes of the project and maybe even generating another nickel or two.

Coupled with appearances in more traditional media, these various activities generate a rich flow of outgoing information that gives a project the appearance of healthy activity... an essential image to maintain. Next time out, we’ll look at one last component in the complex blend that is necessary to propel a dream past inertial effects and potential barriers, maximizing its chances of reaching escape velocity.

About the Author
Steven Roberts was the original “technomad,” covering 17,000 miles around the US on a computerized recumbent bicycle from 1983-1991 while publishing tales via Compuserve and GEnie, then extending the same design objectives to water with an amphibian pedal/solar/sail micro-trimaran that consumed all available resources until 2002. As is typical of homebuilt boat projects, however, by the time it was finished he didn’t really want to do that anymore... so he has since made the transition to a full-time life aboard a 44-foot steel pilothouse sailboat, and is now based in the San Juan Islands north of Seattle.

The ship is extensively networked with embedded data collection and control systems, streaming telemetry, and a user-interface layer reminiscent of the Enterprise... with a wrap-around console that includes communications, R&D lab, audio production studio, and a piano. Roberts has published six books ranging from travel and adventure to microprocessor design, and prior to becoming a technomad spent a decade developing custom industrial control systems, early home computers, and other paleo-geekery. More on his technomadic projects can be found at microship.com [U2] (with the new boat at nomadness.com [U3]). He is publishing the ongoing technical narrative of the new project as a monthly PDF “Nomadness Report,” as well as a series of Boat Hacking design packages detailing the subsystems.

Send the author your feedback [U4] or discuss the article in the magazine forum [U5].
P-adic Math

Playing with Unreal Numbers

by Mark Chu-Carroll

P-adic numbers are an alternative to the reals. They have real applications, but playing with p-adics is also math geek fun.

I’m a guy who spends a lot of time in abstract math-land. I love math, and one of my passions is trying to take the beautiful abstractions of math, and explain them in ways that make them at least a little bit comprehensible to people who don’t spend as much time with their heads in the clouds as I do.

When someone says “math,” what you probably think of is numbers. Math is actually a lot more than numbers—but even when we’re dealing with something as common and everyday as numbers, there are different ways of looking at things that are both fascinating and strange.

Everyone is familiar with the idea of the real numbers. The real numbers are part of a progression of types of numbers. In math, we generally start with the natural numbers—that’s 0, 1, 2, and so on. Then we expand the range of values we can describe by adding negative numbers, which gives us integers. Then we allow ratios of two different numbers, which gives us fractions. Finally, we add irrational numbers, and we get the reals.

P-adics are an alternative (in fact, the only real alternative!) to the real numbers. Instead of going from the rationals to the reals, we can go from the rationals to the p-adic numbers.

A Real Alternative

There’s a reason why we couldn’t stop at the rational numbers, why we needed to keep going and add another class of numbers. When you look at the class all of the possible rational numbers, you find that there are gaps—places where we know that there must be a number, and yet, if we limit ourselves to fractions, there isn’t anything to fit.

For example, we know that there must be some number such that if we multiply it by itself, the result is 2. That number is almost 1 4/10ths, but that’s off by about 1/100th. We can try 141/100, but that’s not quite right either: it’s off by about 4/1000. That’s still not quite right—our refined estimate is now off by about 2/10,000. No matter how far we go, no fraction is ever quite right. There are fractions that are just a tiny, miniscule bit too small, and there are fractions that are a tiny miniscule bit too big, but there’s no fraction that’s exactly the right number.

That number, which we call the square root of two, fits into a gap in the rational numbers. It’s not a rational number, but we know that there’s a number there! If we look at those gaps carefully, we’ll find that most numbers are actually in those gaps! The real numbers and the p-adics are both ways of creating number systems that allow us to define the numbers that fit in to those gaps.
The easiest way to understand p-adic numbers is to think about how we represent real numbers in base-p. In real numbers, we represent them using the powers of the base. So, for example, in base 10, when we write 123, what we mean is 1 * 10^2 + 2*10^1 + 3*10^0. We can do the same thing with other number bases—so, for example, in base 5, if we wrote 24.31, that would mean 2*5^1 + 4*5^0 + 3*5^-1 + 1*5^-2—or, in the familiar base-10, 14.64.

Using our normal notation for real numbers, we fill in the gaps between the rational numbers by writing numbers as an integer part, followed by a decimal point, followed by a fractional part. For real numbers that aren’t rationals, we say that the fractional part goes on forever. So the square root of two starts 1.4142135624, and continues on and on, forever, without repeating. That gives us the real numbers. In that system, every number can be written using a finite number of digits to the left of the decimal point, and an infinite number of digits to the right.

P-adic numbers are exactly the opposite: every p-adic number has a finite number of digits to the right of the decimal, but it can have an infinite number to the left!

**Defining a P-adic Number System**

Defining a system of p-adic numbers starts off being pretty similar to how we compute the representation of numbers in a standard numeric base. We need to pick a prime number, p as the base of our system—that’s the p in p-adic. So base-7 in p-adic is called 7-adic numbers.

To express a number n in base 7, take n modulo 7. That’s the right-most digit of your number. Divide by 7, dropping the fractional part, and take the result modulo 7, and that’s the second-rightmost digit. Continue this until there’s nothing left.

For example, take the number 222 in base-10. If we wanted to represent that in base-7, we’d do:

1. If we divide 222 by 7, we get 31 with a remainder of 5. So the rightmost digit is 5.
2. We take 31, and divide it by 7, giving us 4, with a remainder of 3. So the second digit is 3.
3. We’re left with 4, so the last digit is 4.

So—222 in base-7 is 435. Well, it’s the same in 7-adic. For a particular base B, all positive integers are written the same in both base-B and B-adic. Integer arithmetic that doesn’t involve negative numbers is also the same.

There’s one really big catch that’s a bit of a mind-blower for most people. If you’re used to the real numbers, you know that 35 in base 10 and 43 in base-8 are the same number—they’re just different representations. But in p-adic numbers, each possible p creates a whole different system of numbers! 35 in 10-adic is not the same number as 43 in 8-adic. As we’ll see when we get to metrics, they’re quite different. Each p-adic base produces a distinct system of p-adic numbers, and you can’t convert between them as freely as you can in the conventional reals. Decimal notation and hexadecimal notation are just
different ways of writing numbers in the same number system; 2-adic and 3-adic are different number systems!

**Doing Arithmetic in P-adic**

The first essential difference between p-adic numbers and the reals comes when you try to do arithmetic.

As I said earlier, for integers, if you don’t do anything with negative numbers, p-adic arithmetic is the same as real number integer arithmetic. In fact, if you’re working with p-adic numbers, there are no negative numbers at all! In a p-adic number system, subtracting 1 from 0 doesn’t give you -1. It “rolls over” like an infinite odometer. So for 7-adic, 0-1 = ...666666666! That means that arithmetic gets a bit different. Actually, it’s really pretty easy: you just do arithmetic from the right to the left, exactly the way that you’re used to.

For addition and subtraction, p-adic works almost like normal real-number arithmetic using decimals. Addition is basically what you know from decimal arithmetic. Just go right to left, adding digits, and carrying to the left.

So, for example, in 5-adic, if you have a number ...33333, and 24, to add them, you’d go through the following steps.

1. 3 + 4 is 7, which is 12 in base-5. So the first digit of the sum is 2, and we carry 1.
2. 3 + 2 is 5, plus the carried 1 is 6—so again, 12 in base-5. So the second digit is also 2, and we carry 1.
3. 3 + 0 is 3, plus the carried 1 is 4, so the third digit is 4.
4. For all the rest of the infinite digits streaming off to the left, it’s 3 + 0 = 3.

So the sum is ...3333422.

To do subtraction, it’s still basically the same as what you’re used to from reals. There’s just one simple change: infinite borrowing. In normal subtraction, you can borrow from the position to your left if there’s anything to your left to borrow from. For example, in decimal, if you wanted to subtract 9 from 23, you’d borrow 10 from the 2, then subtract 9 from 13, giving you a result of 14. But if you wanted to subtract 3-9, you couldn’t borrow, because there’s nothing to the left to borrow from. In p-adic, you can always borrow. If you’re subtracting 3-9 in 10-adic, then you can borrow from the 0 to the left of the 3. Of course, there’s nothing there—so it needs to borrow from its left. And so on—giving you an infinite string of 9s. So 3-9 in 10-adic gives you ....999999994.

Let’s do a full example: ...33333 - 42 in 5-adic.

1. As always, we start from the right. 3 - 2 = 1, so the first digit is 1.
2. Since 3 is smaller than 4, we need to borrow 1—so we have 13 base 5, which is 8. 8 - 4 = 4. So the second digit is 4.
3. For the third digit, we just subtract the borrowed 1, so it’s 2.

So the result is ...3333241.
It Gets Stranger

Multiplication and division get even stranger in p-adic. Because we can’t have an infinite number of digits to the right of the decimal, p-adic ends up representing fractions using infinite numbers of digits on the left of the decimal. And that means that we get collisions between fractions and negative numbers. (This should start to give you a clue why each p-adic base is really a different number system: the correspondance between roll-over negatives and infinitely long fractions is different in each base.) It’s easiest to see how this works by looking at a concrete example.

The fraction 1/3 can’t be written as finite-length string in base-5. In 5-adic, that means we can’t write it using digits to the right of the decimal point—we would need an infinite number of digits, and we’re not allowed to do that. Instead, we need to write it with an infinite number of digits to the left! 1/3 in 5-adic is: ...1313132.

Looks crazy, but it does work: if you do a tabular multiplication, right to left, multiplying ...1313132 by 3 gives you one! Let’s work it through:

• Start from the right: the rightmost digit is 2. 3*2 is 6, which is 11 in base 5; so the rightmost digit is 1, and you carry a one.
• The next digit is 3: 3 times 3 is—9, plus the carried 1, gives 10—which is 20 in base-5, so the next digit is a 0, and you carry 2.
• The next digit is 1: 3*1 is 3 plus the carried 2 = 5; so 0, carry 1.

And so on—the rest are zeroes, so ....131313132 * 3 = 1, so ....131313132 == 1/3 in 5-adic.

OK, we’ve shown that ....131313132 is 1/3 because we get 1 when we multiply it by 3, but how can we actually compute this value of 1/3? Just like we do with decimal real numbers: by division.

Division in p-adics is actually easy. The trick is that like all of the other arithmetic, it goes from right to left. Suppose we want to divide N by M. To make it clear, we’ll talk about the digits of M and N using subscripts, so the rightmost digit of a number X is X₁; the second-rightmost is X₂, etc. The multiplication algorithm is:

1. Start at the rightmost digit of both numbers.
2. Find the smallest number d which, multiplied by M, has Nᵢ as its rightmost digit.
3. Subtract d*Mᵢ from N.
4. Drop the trailing last digits from N, giving N’.
5. Now divide N’ by M, and put d on the right.

Let’s walk through 1/3 in 5-adic:

• The rightmost digit of 1 is 1.
• What, multiplied by 3 will have a trailing digit of 1 in base-5? 2*3=6, which is 11 in base-5. So d = 2.
• Now we subtract the “11” from 1—which is really ...00001. So it becomes ...44440.
• We drop the trailing 0, and \( N' \) is \( ...4444 \).

• So now we divide \( ...4444 \) by 3. What’s the smallest number which, multiplied by 3, ends in a 4 in base-5? \( 3 \times 3 = 9 \), which is 14 in base-5. So the next digit of the result is 3.

• Now, we subtract 14 from \( ...4444 \). That gives us \( ...4440 \). Drop the zero, and it’s \( ...4443 \).

• Next digit is a 1, leaving \( ...444 \).

Crazy, huh? There is one catch about division: it only really works if the \( p \)-base in your \( p \)-adic system is a prime number. Otherwise, you get into trouble, because your \( p \)-adic system of numbers isn’t a field if \( p \) is non-prime.

Although it takes getting used to, arithmetic with the \( p \)-adics is actually simpler than it is with conventional real numbers. Everything goes right to left. It’s all more uniform. In fact, \( p \)-adic has been seriously proposed as a number representation for computer hardware, because the hardware is much easier to build when everything can be done uniformly right to left!

**The Power of \( p \)-adic Metrics**

So that’s the basics of arithmetic. But what makes \( p \)-adic numbers really interesting and valuable is metrics.

Metrics are one of those ideas that are simultaneously simple and astonishingly complicated. The basic concept of a metric is straightforward: I’ve got two numbers, and I want to know how far apart they are. But it turns out that there are many different ways of defining “how far apart” things are. Our common notion comes from our geometric intuition. In math, though, you can’t ever just rely on intuition: you need to be able to define things precisely. And precisely defining a metric is difficult. It’s also fascinating: you can create the real numbers from the integers and rationals by defining a metric, and the metric will reveal the gaps between the rationals. Completing the metric—filling in those gaps—gives you the real numbers. Or, if you fill them in differently, the \( p \)-adic numbers.

To define just what a metric is, we need to start with *fields* and *norms*. A field is an abstract algebraic structure that describes the behavior of numbers. It’s an abstract way of talking about the basic structure of numbers with addition and multiplication operations.

A norm is a generalization of the concept of absolute value. If you’ve got a field \( F \), then a norm of \( F \) is a function \( | x | \) from values in \( F \) to non-negative numbers, with the following properties:

1. \( | x | = 0 \) if and only if \( x = 0 \).
2. \( | x y | = | x | | y | \)
3. \( | x + y | \leq | x | + | y | \)

A norm on \( F \) can be used to define a distance metric \( d(x, y) \) between \( x \) and \( y \) in \( F \) as \( | x - y | \).

For example, the absolute value is clearly a norm over the real numbers, and it defines the Euclidean distance between them.
Using the norm, we can see where the gaps between the rational numbers come from.

You can define a sequence $a$ of values in $F$ as $a = \{ a_i \}$ for some set of values $i$. There’s a special kind of sequence called a Cauchy sequence, which is a sequence where $\lim_{i,j \to \infty} |a_n - a_m| = 0$.

You can show that any Cauchy sequence converges to a real number. But even if every element of a Cauchy sequence is a rational number, it’s pretty easy to show that many (in fact, most!) Cauchy sequences do not converge to rational numbers. There’s something in between the rational numbers which Cauchy sequences of rational numbers can converge to, but it’s not a rational number. When we talk about the gaps in the rational numbers, that’s what we mean. (Yes, I’m hand-waving a bit, but getting into the details would be a distraction, and this is the basic idea!)

When you’re playing with number fields, the fundamental choice that you get is just how to fill in those gaps. If you fill them in using a metric based on a Euclidean norm, you get the real numbers. What makes the p-adic numbers is just a different norm, which defines a different metric.

The idea of the p-adic metric is that there’s another way of describing the distance between numbers. We’re used to thinking about distance measured like a ruler on a numberline, which is what gives us the reals. For the p-adics, we’re going to define distance in a different way, based on the structure of numbers. The way that the p-adic metric works is based on how a number is built relative to the prime-number base.

We define the p-adic metric in terms of the p-adic norm exactly the way that we defined Euclidean distance in terms of the absolute value norm. For the p-adic number, we start off with a norm on the integers, and then generalize it. In the p-adic integers, the norm of a number is based around the largest power of the base that’s a factor of that number: for an integer $x$, if $p^n$ is the largest power of $p$ that’s a factor of $x$, then the the p-adic norm of $x$ (written $|x|_p$) is $p^{-n}$. So the more times you multiply a number by the p-adic base, the smaller the p-adic norm of that number is.

The way we apply that to the rationals is to extend the definition of p-factoring: if $p$ is our p-adic base, then we can define the p-adic norm of a rational number as:

- $|0|_p = 0$
- For other rational numbers $x$: $|x|_p = p$ to the $-\text{ord}_p(x)$ power where:
  - If $x$ is a natural number, then $\text{ord}_p(x)$ is the exponent of the largest power of $p$ that divides $x$.
  - If $x$ is a rational number $a/b$, then $\text{ord}_p(a/b) = \text{ord}_p(a) - \text{ord}_p(b)$.

Another way of saying that is based on a property of rational numbers and primes. For any prime number $p$, you can take any rational number $x$, and represent it as a $p$-based ratio $p^{n(a/b)}$, where neither $a$ nor $b$ is divisible by $p$. That representation is unique—there’s only one possible set of values for $a$, $b$, and $n$ where that’s true. In that case, the p-adic norm of $x$, $|x|_p = p^{-n}$.

Ok, that’s a nice definition, but what on earth does it mean?
Two p-adic numbers \( x \) and \( y \) are close together if \( x - y \) is divisible by a large power of \( p \).

In effect, this is the exact opposite of what we’re used to. In the real numbers written out in decimal for as a series of digits, the metric says that the more digits numbers have in common moving from left to right, the closer together they are. So 9999 and 9998 are closer than 9999 and 9988.

But with p-adic numbers, it doesn’t work that way. The p-adic numbers are closer together if, moving right to left, they have a common prefix. The distance ends up looking very strange. In 7-adic, the distance between 56666 and 66666 is smaller than the distance between 66665 and 66666!

As strange as it looks, it does make a peculiar kind of sense. The p-adic distance is measuring a valuable and meaningful kind of distance between numbers—their distance in terms of their relationship to the base prime number \( p \). That leads to a lot of interesting stuff, much of which is, to be honest, well beyond my comprehension. For example, the Wiles proof of Fermat’s last theorem uses properties of the p-adic metric!

About the Author
Mark Chu-Carroll is a PhD computer scientist and professional software engineer. He works as a server engineer at foursquare. His professional interests include collaborative software development, programming languages and tools, and how to improve the daily lives of software developers. Mark blogs about math related topics on Scientopia. Aside from general geekery and blogging, he plays classical music on the clarinet, traditional Irish music on the wooden flute, and folds elaborate structures out of paper.

Send the author your feedback or discuss the article in the magazine forum.

External resources referenced in this article:

[3] mailto:michael@pragprog.com?subject=p-adic numbers
Questions Never to Ask a Candidate

Ask What You Want to Know

by Johanna Rothman

If you’re asking candidates any of these questions, you should stop now. Johanna tells you why.

Do you have questions that you have honed over years of interviewing, questions you are certain tell you everything you need to know about a candidate?

You might be right. And you might be asking an irrelevant question that doesn’t buy you any information at all, except reinforcing your preconceptions about a candidate. Those are strong words. I can back them up.

Here are some of the worst questions to ask a candidate. If you are asking any of these questions after I try to dissuade you, let’s go have a beverage together over on the discussion forum and discuss why.

How Would You Move Mt. Fuji?

There are several problems with this question. The first is that it’s hypothetical. A hypothetical question, “How would you,” allows a candidate to be hypothetical in answering the question.

Hypothetically, I’m perfect. My husband and children can attest to the fact that I am not. Other people can too, but we’ll stop there. When you ask a hypothetical question, you beg the candidate to answer the question in an arm-waving fashion. Don’t do it. You want answers based on a candidate’s real experience, answers that point to how a candidate will work for you, in your context.

The other problems with this question are: Who moves Mt. Fuji at work? Why would you want to? This question doesn’t reflect the reality of your workplace, unless you are in construction, or some form of major earth-moving. If you are hiring technical people, how can they answer this question?

Oh, there are plenty of answers online. It’s a fun question. But this question is in the category of puzzles and riddles that are irrelevant to how most of us work at work.

If you cannot draw a direct correlation from the question to your work, don’t ask it. I am serious. That means no math puzzles unless the candidate has to solve math problems. No manhole cover problems unless the candidate has to determine manhole cover answers. And no Towers of Hanoi problems, unless the candidate has to solve those for work, either. Each of these is an irrelevant question, because they don’t assess how the candidate works.

I can hear you asking me now, “What do I ask instead?” Ask great behavior-description questions, such as, “Tell me about a problem you solved recently.” Now, you sit back and let the candidate talk. I have more guidance about this in Hiring Geeks That Fit [U1].
Tell Me About Yourself

Some interviewers like to start an interview with this question to put a candidate at ease and build rapport. Their instincts are good. You do want to build rapport and put a candidate at ease. But this particular question? Not so hot.

This question is too vague for anyone with more than a few years of experience. The question wastes interview time because it doesn’t help the interviewer get to either technical experience or cultural fit. If you replace this question with a question like this one, “Tell me about your most recent project: what was your role?” followed by, “What kinds of challenges did you have?” then you start to have a conversation and you learn what the candidate thinks a challenge is. That puts a candidate at ease, because the candidate has a concrete question to answer. It’s also a great cultural fit question, because it tells you what this candidate finds challenging.

Tell Me About Your Weaknesses or Strengths

If you ask me this question, I can make any weakness sound like a strength. Any, at all. “I tend to be intense at the end of a release, when we’re focused on getting the product out the door. I want to make sure we release a great product.” Who could argue with that? Well, what if the release criteria are date, date, and just the date, and I want to focus on something else? That might be a problem, right? But not the way I’ve phrased my weakness. You can’t tell that from the way I answered. By the way, I wouldn’t do that. No, sirree. I would say, “Let’s stick with the release criteria. We do have release criteria, right?”

Candidates who’ve had any interview coaching know how to present their weaknesses in the best possible light. Candidates who haven’t had coaching are thinking like mad, trying to turn their weaknesses into strengths.

And, how can you tell if a strength really is one? I once interviewed a developer who told me—unasked—that his major strength was his ability to see the architecture fully formed before anyone wrote a line of code. I was astonished. “How can you do that, before you see any features?” “I’m just that good,” he pronounced. Okay. Maybe he was, but his arrogance arose in other places, too. I didn’t think he was that good and neither did the rest of the team.

Many of our weaknesses and strengths are culturally-dependent. Okay, maybe not when I was improving my writing, and my colleagues were saying, “A verb, JR, a verb!” But that was easy to describe. I could easily say, “I used to have my memos go through six rounds of review. Now they go through three rounds of review.”

I describe myself as blunt and direct, feedback I received from more than one manager. When I give talks, I often say this is a feature in a consultant, but more often a defect in an employee. It’s a cultural issue. What people can talk about is a part of the culture.

Instead of asking about strengths or weaknesses, consider a question such as, “Tell me about the problem you solved that you have the most pride in. When was it and what happened?” Now, let the candidate talk and tell you the story of his or her success. How long ago was it? If it was more than a year or so ago,
ask for a more recent example in this way, “That was great. Do you have a recent example of something you did that you have a similar pride of ownership about?”

You can also ask the candidate what he or she is learning. “Tell me what you are learning about these days.” That’s a leading question—it presupposes that the candidate is learning something. If you want a non-leading version of that question, you can use this question, “Do you do anything specific to increase your knowledge of your field,” which is a closed question. If the candidate answers yes, you can ask, “Please tell me what it is,” and wait for the answer.

Where Do You Want to Be In Five Years?

During an interview, we are curious about the candidate. We do want to know where people want to be in five years. And, if our organizations could guarantee employment that long, this seems like it might be a good question.

But we have so many unknowns that this question, as it stands, is not a great question. With a little refinement, we can make it into a great question. The real question is this: “What do you want to discover?”

Do you want to know if a candidate has ambition? In that case, ask this question, “Tell me about a time you wanted a particular role. What did you do?”

Do you want to know if a candidate wants to remain technical? In that case ask, “Are you looking to remain technical or do you see yourself moving into some other role?” It’s always best to ask directly.

It depends on what you want to know. When one manager asked me this question long ago, I sat there and debated with myself. I answered truthfully, “I don’t want to be sitting in your chair. I want to be your boss, or your boss’s boss.” See what I mean about the blunt and direct? I did receive an offer from those folks. But I decided to go somewhere else where the pace was faster.

Make Your Questions Sell You and Your Company

When you transition from irrelevant questions, such as the “How do you move Mt. Fuji?” question, you have to think much harder about your questions. You have to craft your questions. Use your job analysis (see my previous article, “Finding the Geek Who Fits,”[U2]) to identify what’s essential to the job and ask about that.

The problem with irrelevant questions is that you have One Right Answer in mind. When someone supplies a different answer, you think, “Oh no, that can’t be right,” and consider rejecting the candidate. Rejecting candidates who don’t fit because of their technical skills or because they don’t fit the culture is perfect. Rejecting them because they flunked an arbitrary irrelevant question? Not perfect. And it’s not a way to sell the candidate on your organization.

Before you use the same old questions, ask yourself: How well is this question working for me? Should I keep it or retire it?

For more guidance on interview questions, see Hiring Geeks That Fit[U3].
About the Author

Johanna Rothman helps leaders solve problems and seize opportunities. She consults, speaks, and writes on managing high-technology product development. She enables managers, teams, and organizations to become more effective by applying her pragmatic approaches to the issues of project management, risk management, and people management. She writes the Pragmatic Manager email newsletter and two blogs on www.jrothman.com. Please do go there, wander around and subscribe!

Send the author your feedback or discuss the article in the magazine forum.

External resources referenced in this article:

5. mailto:michael@pragprog.com?subject=hiring
This month, we dig deeper into dependent types. Dependent types were introduced in an earlier article in this series as a richer language of types, allowing us to explain more about what a program was doing, by encoding structural information and proofs about properties. I closed the previous article promising that we could replace some (maybe all) of the measures required to gain confidence in the word wrap code by using such a richer language, specifically replacing tests and hand-waving assertions with firm statements of what properties and details we expect to hold and back them up with proofs.

Now, the formal methods people have been able to do something like this for a while, but it's not exactly been convenient to work with—and certainly not as immediately useful as testing.

What dependent types brings to the table is more convenience and elegance: we're able to say more about our intentions inside the program itself, not as some added extra like paper proofs or tests, and the mechanisms work naturally with the language.

This change is quite radical: it offers new ways to think about programming, and the extra richness of language enables new tools to help with programming. For these reasons, I think it could represent a new paradigm in programming.

Of course, it's not a silver bullet. There will be new things to learn, and will require more effort in places. But what you stand to gain in return is more confidence in your code.

There are two things I want to cover this month, with more to follow in later months.

- introduce some of the new tools for growing programs with dependent types, and
- cover some of the basic ideas in dependent types which we will need for the word-wrap code.

You may have seen syntax-directed editors, ranging from inserting templates into code up to some kinds of refactoring. The tools we’ll cover go a bit further, building syntactically valid code that is able to use types to propagate some of the semantic information with the ability to fill in some details automatically. As we switch to a more powerful type system, such tools become even more useful.

Finally, here's a reminder of some of the aspects of the word-wrap problem where dependent types can help:

- Does long-word splitting always produce the right size and order of fragments?
• Will line packing never overflow, assuming that the input is valid?
• Can we guarantee that the output doesn’t add or delete or permute non-space characters?

Language and implementation

The “implementation” I’m going to use is Idris. Idris is being developed by Edwin Brady (of WhiteSpace fame) as a tool for practically-oriented dependently-typed programming. Edwin is very actively doing research on applying such advanced ideas to resource-bound embedded systems and protocols, so it’s far from being a toy. It even has a FFI! The code is on github and has a growing list of contributors.

Idris is a mix of editor, interpreter, and compiler. The Idris language is based on Haskell, but with full dependent types taking center-stage rather than being an add-on. Simple Idris programs can be read as vanilla Haskell (so it has a reasonable intersection with Haskell, albeit with some syntax changes), and the more powerful language is there when it’s needed. Part of the research is to investigate the interplay between the two languages.

Idris can generate stand-alone executables compiled via C (very compact, compared to Haskell’s GHC compiler), and more recently can compile into Java and Javascript. The interpreter provides a REPL, allowing arbitrary code to be run from the prompt. The editor part is actually a kind of proof assistant to help with the finer detail of growing dependently typed programs. This is the component we will look at first, using basic Haskell programs.

Similar systems based on type theory also exist, such as Agda, Coq, Epigram, Lego, or Twelf, with many overlaps of features though differing in several technical details too. These ideas are starting to filter into more popular languages, for example, GHC has recently added some weak support for “holes.”

Installing Idris

Idris is currently distributed only as source, so you’ll need to compile it yourself.

First, install the Haskell Platform. Packages are available for standard *nix (including Macs). Typically these pull in recommended stable packages of standard tools like GHC (the compiler), Cabal (the package manager, like 'gem'/‘bundler’ or 'npm'), and various useful libraries. There’s a Windows installer for the corresponding bundle of binaries.

The most recent stable (source) release of Idris is on the standard Haskell package server, known to the world as Hackage; the entry for Idris is here. To install this version, run the following:

```bash
cabal update
cabal install idris
```

This will download the source, configure and build it with the tools in the Haskell Platform, and install it as ~/.cabal/bin/idris (with libraries and resources being stored elsewhere under ~/.cabal).

You can also grab the repository from github; there’s a README but just typing make will compile and install the current branch into ~/.cabal/bin/idris.
Running the executable should show some ASCII art and a prompt.

**A very simple program**

From the prompt, enter `:one.idr`. It should complain about a missing file, since we've not created it yet. Enter the editor with `:e`. Which editor gets run depends on the current settings of environment variables `EDITOR` or `VISUAL`, and chances are it will run a relative of `vi`. (If you've not used `vi` or `vim` before, now's your chance. And check out Drew Neil's excellent guide.) Now enter this text:

```idr
module Main
main : IO ()
main = putStrLn $ "usual string"
```

Idris will type-check the file and load the definitions into the interpreter. You can evaluate `main` but this shows the corresponding call to the underlying library function via the FFI. Compare to the result of `print (1 + 2)`—notice that it computes the string "3" to be shown. (Supposedly you can use `:x` to properly execute the (monadic) IO action, though not at present, but it is coming soon.) Since this module is called `Main` and contains a definition of `main`, which is the standard entry point for Haskell and Idris executables, then we can compile it and execute the resulting binary. Use the REPL command `:exec` to trigger this.

If you want to save the resulting binary, use command `:c foo`, where `foo` is the desired output file. You can then run `./foo` outside of Idris. Alternatively, use `js foo` to generate JavaScript code, which can be run via NodeJS with `node ./foo`.

(Generation to Java code is being worked on, and should be ready soon.)

You can see other useful REPL commands by typing `:`?, or view the latest Idris tutorial for more information. (There's also a copy in the repository.) Idris also accepts some command-line options at start-up, so use `--help` to see these.

**Leaving bits out**

Next, let's see how we can grow our program using Idris’ editing tools. Start a new file `two.idr` and enter this text.

```idr
module Main
foo : List Int -> List Int
foo xs = if List.length xs < 4
  then ?case_one
  else ?case_two
main : IO ()
main = putStrLn $ show $ foo [1,2,3]
```

The key point is that we haven’t finished writing `foo` yet: the `?name` expressions represent “holes” which need to be filled in, and the names serve to distinguish the various holes. We can also refer to holes as “meta-variables,” in the sense of them being placeholders for something else, and as existing **outside** of the actual programming language.

Idris will check this code and report success. But can we compile it and run the whole program? Of course not, because bits are missing. However, we can
check parts of it via Idris' interpreter. Try \texttt{foo [1,2,3]}. It doesn't crash, and instead does as much as it can before being blocked by the missing piece, hence shows you \texttt{Main.case\_one [1,2,3] : List Int}. (Actually, it de-sugars \texttt{[1,2,3]} into a combination of cons and empty list, but I'll spare you those details.) In other words, it was able to run the if-test and decide which branch to use. It would show \texttt{Main.case\_two [1,2,3,4,5]} if you supplied a longer list. This is highly useful for checking bits of code without writing the whole lot.

Behind the scenes, Idris has created two new top-level values to represent the holes, and used type information from the context to determine what type the hole should have. You can check such types with (e.g.) `:t case\_one` which will show \texttt{List Int -> List Int}, hence it needs to be a function which takes and returns a list of ints. There weren't any arguments when `?case\_one` first appeared, so why does it need to be a function? Well, `?case\_one` appeared in a context where variable \texttt{xs} was bound, so plausibly the code for \texttt{case\_one} might need to refer to \texttt{xs}. A simple way to handle this potential dependency is to treat holes as functions and pass in the values of the bound variables. This is why you see \texttt{Main.case\_one [1,2,3]} etc when you evaluate \texttt{foo [1,2,3]}.

Before we fill in something for \texttt{case\_one}, we can start to lay down some assertions or tests to try later. Suppose we want \texttt{foo} to reverse short lists. Then we can define a simple Boolean assertion, which we expect to return \texttt{True} when the code is complete. It won't do so now, and instead shows how far it got (basically, it's blocked by a missing case).

\begin{verbatim}
assert_1 : Bool
assert_1 = foo [1,2,3] == [3,2,1]
\end{verbatim}

Such assertions rely on us running them explicitly in order to check whether they are \texttt{True} or not, just like other tests. We can do better: use logic. This will be covered in more detail later, but for now we will just state the required condition as a \texttt{theorem} (or as a \texttt{proposition} if you prefer) and leave its corresponding proof until later (via the same mechanism as is used for holes). Just note that the binary operator is `=`, signifying propositional equality at the type level, rather than the usual `==` for equality tests computed by the code.

\begin{verbatim}
proof_1 : foo [1,2,3] = [3,2,1]
\end{verbatim}

**Filling in the first case**

Type `:p case\_one` to start the "proof assistant" mode on the hole \texttt{case\_one}. Idris will show the following, which indicates that you need to supply or 'prove' a value of type \texttt{List Int -> List Int}.

\begin{verbatim}
------------ Goal: ------------
{ hole 0 }:
    List Int -> List Int
\end{verbatim}

We can fill in a full value right away with the command \texttt{exact reverse}, i.e., indicating that the current goal is exactly solved by the (function) value \texttt{reverse}, but let's explore other ways to grow functions.

The main tactic to grow a function is to assume its argument and then try to grow the function's body. Run the command `intro xs` and observe the result.
Other goals:  

\{ \text{hole 0} \}  

Assumptions:  

\text{xs : List Int}  

Goal:  

\{ \text{hole 1} \}:  
\text{List Int}  

It’s now asking us to prove a new goal (denoted by \text{hole 1}), which needs to be a value of type \text{List Int}. Notice an assumption \text{xs : List Int} has appeared, representing the input coming into the function we’re working on. The original goal \text{hole 0} is still there too, for reference.

We know we want this first case to reverse its input, so we need to use \text{reverse} somewhere. One option is to use \text{exact reverse xs} (or \text{exact (reverse xs)} if you prefer). If you try this, you can use \text{undo} to go back a step. A more powerful tactic is \text{refine}, which fills in the head (or outer) function and then infers any arguments it can. So, being specific that we want the version of \text{reverse} for lists rather than the slightly magical version for strings, try \text{refine List.reverse} and observe the effects:

The other goals and assumptions are unchanged, but \text{hole 1} has disappeared and we’re now being asked for another list. Now, it’s not obvious at present where this list fits in, but you might realise that it corresponds to the main argument for \text{reverse}. You can get confirmation of sorts from the \text{term} command, which shows the current state of the function you are growing in the proof assistant mode. You should see something like \text{reverse \{__pi_arg 1000\}} in the output, thus confirming it is the argument for \text{reverse}.

In fact, what the \text{refine FOO} tactic does is apply expression \text{FOO} to new metavariables until its type matches the general pattern of the goal’s type—hence it is filling in the hole with \text{List.reverse ?arg} and then creating a new subgoal to find out what \text{?arg} is. The matching is done via a form of \text{unification algorithm} \text{[U11]} - using the same kind of inference behind logic programming and behind Haskell’s type inference. (If you don’t know much about unification, maybe start with this \text{list of examples}\text{[U12]; it’s also a good coding exercise to help you gain a good understanding of tree manipulation and of unification itself}.)

Anyway, this argument should be \text{xs}, so fill that in with \text{exact xs}. Idris will announce the end of the proof process with “\text{No more goals}”. You can check the result with \text{term} and should see in full the function we constructed with the tactics:\text{\xs : Prelude.List.List Int \Rightarrow Prelude.List.reverse Int \xs}. If you don’t like the result, you can issue \text{undo} a few times and try a different approach.

Exit the proof mode with \text{qed}. Idris will display the sequence of tactics used to construct the result, and double-check the resulting term against the original, expected type. Assuming there’s no problem, you can append the resulting proof to your file with \text{a}. The proof is shown below, and it is just a list of the steps we took.
Main.case_one = proof {
    intro xs;
    refine List.reverse;
    exact xs;
}

You can also look at the corresponding code created for case_one by evaluating case_one, though beware that Idris will try to execute as much as it can and so shows a version which has expanded reverse into its lower-level definition—but it can’t compute any further until bound variable xs is replaced by some data on which it can pattern match. If you want to get it to compute more, give it something to work on! For example, you can now evaluate case_one [1,2,3] and examine the result.

GOTCHA: You will need to move the five lines of proof for case_one from the end of the file to the point after the definition of foo. It’s quite ok to mix code and proof blocks like this. The move is required because Idris’ type checker will need the proof term to be defined before it checks other code or proofs which rely on it. (It’s not an oversight in design—I’m doing things in a slightly unconventional order here.)

A quick detour into Curry-Howard

Notice that we went through a sequence of “proof steps” and ended up with some code. This is not accidental, and it’s one aspect of the Curry-Howard correspondence.

Recall that this correspondence identifies the similarity between (logical) propositions and types, and between proofs and programs. We’ll be using this similarity soon, as we start to prove properties of the code inside the language.

It’s a big topic, of course, and too much for this short article. The wikipedia page seems a fair place to start if you want to know more.

Also notice that Idris is saving the proof steps as part of our program, and will use these to reconstruct the code when it loads the file from text. (Idris also saves the results in compiled form, but that’s another story.)

Checking the results

Remember the assertion we added earlier? We can run it now—just type assert_1 and observe it returns True. But, with the extra machinery of logic at our disposal, we can do better. Add this code to the file.

proof_of_assert_1 : foo [1,2,3] = List.reverse [1,2,3]

This declares name proof_of_assert_1 to have a type of foo [1,2,3] = List.reverse [1,2,3], which represents the logical proposition that the expression foo [1,2,3] produces the same result as expression List.reverse [1,2,3].

You could also call it a conjecture—with the current definitions in force—that the two expressions amount to the same thing.

Now, it isn’t the same as the code we wrote for assert_1. That code will return the constant True if the two operands compute to the same list (and this has to run the code in full to determine whether it is True or not).
It will also depend on how \( == \) has been defined: it might do something strange like treat all odd numbers as equal, e.g., \( 2 == 4 \) could be \( \mathsf{True} \). So, it’s something we have to compute inside the language.

In contrast, propositions like \( X = Y \) are something we can reason about and don’t always have to run the code. For example, we know \( \mathsf{foo} [1,2,3] \) will expand to \( \mathsf{case\_one} [1,2,3] \) and this in turn expands to \( \mathsf{List.reverse} [1,2,3] \). At this point, the inference mechanism will accept the two sides as being the same. Also, there’s no sense of \( \mathsf{True} \) or \( \mathsf{False} \) here: the system either has a proof or it does not, and if it does have one, it can use it to prove something else, otherwise it is stuck.

This distinction may seem strange at first, but please persevere with a few more examples, and it might get a bit clearer!

Anyway, what do we do with conjectures? Prove them, that’s what.

Start the proof mode with \( \texttt{p proof\_of\_assert\_1} \), and you will be asked for a proof. There’s a useful tactic for situations like this: \( \texttt{trivial} \). This tries a few standard proof techniques on the current goal, and in this case it is able to confirm the proposition by computing both sides to the same thing.

After \( \texttt{qed} \), you’ll see a very short proof.

It’s useful to know what’s happening under the hood, so evaluate \( \texttt{proof\_of\_assert\_1} \) and you will see the term \( \texttt{refl} \). For now, you can understand \( \texttt{refl\_x} \) as the proof that \( x = x \), or the proof that equality is reflexive.

**Tests into proofs**

Now, we’ve proved our assertion and because of type checking, the assertion will be checked whenever the file is reloaded, so it guarantees the assertion still holds after any code changes. This is what conventional test frameworks give us.

So here’s something that tests can’t do. Start with this type signature:

\[
\text{case\_one\_proof} : (l : \text{List\ Int}) \\
\rightarrow (\text{length } l < 4) = \mathsf{True} \\
\rightarrow \mathsf{foo} l = \mathsf{reverse} l 
\]

The type represents the proposition: “for all lists \( l \), where the length of \( l \) is less than 4, then \( \mathsf{foo} \) on \( l \) is the same as reversing \( l \).” This says precisely how we want \( \mathsf{foo} \) to behave on short lists, for all possible inputs. Compare this to a set of tests which are aimed at capturing the same requirement. The tests are a collection of examples, and additional reasoning will be required if we want confidence that they are representative of the wider picture—whereas the statement above covers all possibilities. All of them. It’s the difference between “there exists” and “for all.”

Let’s do the proof. Basically, we’re going to use the hypothesis or assumption about the length of input to push some computation through the definitions, until we get to a point where the sameness is obvious. The first two steps are \( \texttt{intro l} \) and \( \texttt{intro H} \). This moves the function’s two inputs into the proof context as assumptions, leaving a proof state like this:
What \( H \) represents deserves some explanation. It's an assumption of a proof of \( \text{length} \ l < 4 = \text{True} \), not an actual proof of it. We often speak of terms like this being hypotheses—which we will use to prove the required conclusion. Likewise, \( l \) represents an arbitrary list. Notice that \( H \) gives us some extra info about what \( l \) is. It's a bit like how mocks are used in several test frameworks, where you have some minimal object equipped with predictable behaviour, and the associated tests examine the consequences. For example, \( l \) could be some anonymous thing such that \( \text{length} \ l < 4 \).

Next, we need to use \( H \) to simplify the goal \( \text{foo} \ l = \text{reverse} \ l \), aiming for a point where the goal is trivially satisfied with \( \text{refl} \). So we're going to use \( H \) to rewrite part of the goal and see what comes out. In this case, we want the term \( \text{if} \ \text{length} \ l < 4 \ \text{then} ... \) to change to \( \text{if} \ \text{True} \ \text{then} ... \).

One useful property of equality is substitution, which says if \( A = B \) then if we have \( P(A) \) then we have \( P(B) \) for all properties \( P \). Put another way, it we know \( A = B \) and that we have some \( P \) that holds for \( A \), then we can infer that \( P \) holds for \( B \) too, and can't be used to distinguish them. (A related idea is that if \( A \) and \( B \) are really equal, then all properties of \( A \) hold for \( B \) too, if that helps.)

It's a common inference step, so Idris has a special tactic for it: \( \text{rewrite} \). One slight niggle though—we need the operands of \( H \) in the other order, but this is ok since equality is symmetric (another requirement of equality).

(I could have cheated and used \( \text{True} = \text{length} \ l < 4 \) as the hypothesis, but it's useful to know about symmetry.) This means the next proof step is \( \text{rewrite} \langle \text{sym} \ H \rangle \). The result might not be too comprehensible but you can use the tactic \( \text{compute} \) to force a bit more of the computation, and so get this—which even if we don't know what it means, we can see it's the same for both sides.

So, we can finish the proof with \( \text{trivial} \). Notice that we don't have to compute any further, either. It doesn't matter what \( l \) turns out to be, because the logic ensures the result has the required properties. The completed proof sequence is shown below. I've omitted the \( \text{compute} \) steps because they are really just for cosmetic purposes, so we can see what's going on. They make no difference to the proof.

```
{ hole 3 }:
Prelude.List.reverse.0#reverse' Int (Prelude.ListNil) l =
Prelude.List.reverse.0#reverse' Int (Prelude.ListNil) l
```

```
Main.case_one_proof = proof {
  intro l;
  intro H;
  rewrite (sym H);
  trivial;
}
Filling in the rest

We’ve covered just one case, but for this we have a rock-solid statement of what it should do and a proof that it does what it should. If subsequent changes to the code alter its behavior then the proofs will not type check. We won’t be able to ignore it.

I’ll let you think up some functionality and properties for the second case. Have a go at proving things yourself! If you get stuck, post your code to the magazine’s forum and I’ll try to help you out.

As a final point, what does case_one_proof [1,2,3] refl produce? Yes, our proof is a function (with a dependent type) and we can apply it to things. What does it mean to use the proof like this? And what about case_one_proof [1,2,3,4,5]?

The wider picture

You’ve seen now how to do a very simple development in a type theory setting. The code and proof were quite basic, but I deliberately chose a straightforward example so we could cover the key ideas without too much distraction.

We’ve barely scratched the surface of what this paradigm offers, so tune in next month. I’ll cover topics like dependently typed data structures, including vectors (sized lists) and dependent tuples. We’ll use vectors to code up size constraints on words that are being packed into lines, and to provide guarantees about the splitting of long words. Dependent tuples allow data and proofs to be packaged up and manipulated in a convenient way. And as the types become more interesting, then the proof tools will be able to do more interesting and useful things for us.

There’s much more to explore on the tool support side too. Several proof assistants have more powerful tactics which can automate straightforward proofs like the ones we saw here.

It’s not magic though: all they do is a bit of search through various combinations of the basic tactics we’ve just seen, maybe guided by some heuristics on the form of available assumptions and hypotheses, plus any relevant theorems that we’ve already proved. Idris will probably support this kind of automation in the near future. So, don’t conclude that working in this way means a lot of drudge work!

Nor are these tools limited to simple examples either. In the past two decades, such tools have been used to provide mechanical verification of complex protocols and many important results from maths. The four color theorem for map coloring is a good example.

But notice where the human fits into this process. The proof assistant manages the fine detail and the checking, and provides basic tools for standard reasoning steps, possibly with some limited automation for the relatively obvious things. The human does the creative part, of guiding the high-level operation of the proof assistant, the bit that is currently too hard for mechanization. (That’s why we call them proof assistants.)

A similar idea applies to programming! The human sketches out the high-level details and lets the machine fill in the obvious things. We should try to program like this. Why not?
About the Author

Dr Paul Callaghan first learnt about type theory in 1996, and was immediately fascinated. It filled in many of the unsatisfactory gaps in conventional functional programming, and helped him understand why some programs in Haskell felt “wrong” and “incomplete.” Paul was fortunate to do research work with some very talented people from 1997 to 2007, during which time he implemented a niche proof assistant called Plastic and experimented with various ideas like DSLs, Mathematical Vernacular, and Coercive Subtyping. Other bits of his bio can be seen on earlier articles. Paul also flies big traction kites and can often be seen being dragged around inelegantly on the beaches of North-east England, much to the amusement of his kids. He blogs at free-variable.org[14] and tweets as @paulcc_two[15].

Send the author your feedback[16] or discuss the article in the magazine forum[17].

External resources referenced in this article:

15. http://twitter.com/paulcc_two
16. mailto:michael@pragprog.com?subject=functional programming
Calendar

Want to meet one of the Pragmatic Bookshelf authors face-to-face? Here’s where they’ll be in the coming months.

2013-03-28
Wrapping RubyMotion - Discussing techniques for making Objective-C APIs more pleasant in Ruby
Clay Allsopp (author of RubyMotion [U1])
#inspect, Brussels Belgium [U2]

2013-04-02
Functional Programming Patterns
Michael Bevilacqua-Linn (author of Functional Programming Patterns in Scala and Clojure [U3])
Philly ETE [U4]

2013-04-04
A 1-day conference dedicated to Cucumber, Specification by Example and BDD.
Aslak Hellesøy (author of The RSpec Book [U5], The Cucumber Book [U6], and Cucumber Recipes [U7])
CukeUp!, London [U8]

2013-04-04
This Is Your Brain On Software
Paolo Perrotta (author of Metaprogramming Ruby [U9])
Ancient City Ruby, St. Augustine, Florida, USA [U10]

2013-04-05
JavaScript related topic
Venkat Subramaniam (author of Practices of an Agile Developer [U11], Programming Groovy [U12], Programming Scala [U13], Programming Concurrency on the JVM [U14], Programming Groovy (2nd edition) [U15], and Functional Programming in Java [U16])
DenverJS - Denver, CO [U17]

2013-04-05
Talks on HTML 5 and JVM languages.
Venkat Subramaniam
NFJS, NYC [U18]

2013-04-05
Zen and the Art of iOS Gesture Recognizers
Jonathan Penn (author of Test iOS Apps with UI Automation [U19])
CocoaConf Dallas [U20]

2013-04-06
Advanced UI Automation Techniques
Jonathan Penn
CocoaConf Dallas [U21]

2013-04-08
A two-day conference for technical writers, documentarians, and all those who write the docs.
Jim R. Wilson (author of Seven Databases in Seven Weeks [U22])
Write The Docs, Portland OR [U23]

2013-04-12
Workshop "Gumption Traps Reloaded" with Ivan Moore and talk "Is eXtreme Programming still alive and kicking?"
Rachel Davies (author of Agile Coaching [U24])
ACCU, Bristol, UK [U25]

2013-04-16
Vim Masterclass
Drew Neil (author of Practical Vim [U26])
Online [U27]

2013-04-18
In this talk, Marcus will walk you through a design pattern that he has been using on iOS for applications that require and use a large amount of data that is frequently requested from the internet.
Marcus S. Zarra (author of Core Data (2nd edition) [U28])
CocoaConf, San Jose, CA [U29]
2013-04-18  Mobile Movies with HTTP Live Streaming  
Chris Adamson (author of iOS SDK Development [U30])  
CocoaConf San Jose [U31]

2013-04-18  Core Audio in iOS 6  
Chris Adamson  
CocoaConf San Jose [U32]

2013-04-18  Core Audio Workshop (All-Day Tutorial)  
Chris Adamson  
CocoaConf San Jose [U33]

2013-04-19  Advanced UI Automation Techniques  
Jonathan Penn  
CocoaConf San Jose [U34]

2013-04-19  Zen and the Art of iOS Gesture Recognizers  
Jonathan Penn  
CocoaConf San Jose [U35]

2013-04-21  5.5 days of leadership training for people who want to learn to be problem solving leaders.  
Johanna Rothman (author of Behind Closed Doors [U36], Manage It! [U37], Manage Your Project Portfolio [U38], and Hiring Geeks That Fit [U39])  
Problem Solving Leadership Workshop, Albuquerque, NM [U40]

2013-04-21  Talks on HTML 5 and JVM languages.  
Venkat Subramaniam  
NFJS, Reston, VA [U41]

2013-04-22  A talk on something other than Rails. This is a Rails conference about everything except for Rails.  
Chad Fowler (author of The Passionate Programmer (2nd edition) [U42] and Rails Recipes [U43])  
Railsberry - Krakow, Poland [U44]

2013-04-22  “How to Design Indexes, Really” and “Extensible Data Modeling with MySQL“  
Bill Karwin (author of SQL Antipatterns [U45])  
Percona Live MySQL Conference and Expo 2013 [U46]

2013-04-27  JRuby - Imagine mixing the easy syntax, powerful language features, and rapid development of Ruby with the speed, power, and flexibility of the Java Virtual Machine? Ruby and the JVM go together like chocolate and peanut butter, and you'll see some examp  
Brian P. Hogan (author of HTML5 and CSS3 [U47], Programming Concurrency on the JVM [U48], The Rails View [U49], Crafting Rails Applications [U50], Deploying Rails [U51], iOS SDK Development [U52], Web Development Recipes [U53], The Developer’s Code [U54], Deploying with JRuby [U55], tmux [U56], Programming Groovy (2nd edition) [U57], Test IOS Apps with UI Automation [U58], and The Healthy Programmer [U59])  
Twin Cities Code Camp, Minneapolis, MN [U60]

2013-05-03  Talks on JVM related languages  
Venkat Subramaniam  
NFJS, Charlotte, NC [U61]

2013-05-03  HTTP: Get to Know the Foundations of Your Career  
Jonathan Penn  
Rustbelt Refresh [U62]

2013-05-04  iOS Media APIs  
Chris Adamson  
MobiDevDay Detroit 2013 [U63]

2013-05-07  Workshops and talks on various topics  
Venkat Subramaniam  
Great Indian Developer Summit, Bangalore [U64]

2013-05-14  Pragmatic Agility  
Andrew Hunt (author of Pragmatic Thinking and Learning [U65] and Practices of an Agile Developer [U66])  
Triangle.rb, Raleigh NC
2013-05-14  Scala for the Intrigued
Venkat Subramaniam
JUG, San Francisco, CA

2013-05-17  Various topics related to JVM and HTML5
Venkat Subramaniam
NFJS, Dallas, TX

2013-05-20  Tutorial: Hiring for Your Team: Culture Trumps Skills
Johanna Rothman
Let's Test Conference, Sweden

2013-05-21  Keynote: Becoming a Kick-Ass Test Manager
Johanna Rothman
Let's Test Conference, Sweden

2013-05-22  Talks on various topics related to Groovy and the JVM.
Venkat Subramaniam
GR8Conf, Copenhagen

2013-05-24  Keynote
Andrew Hunt
Path to Agility, Columbus OH

2013-05-28  Flow Control with Promises: Learn to control async tasks in JavaScript with Promise-based interfaces.
Trevor Burnham (author of CoffeeScript and Async JavaScript)  
Fluent 2013

2013-05-29  Change -- How long does it take?
Staffan Nöteberg (author of Pomodoro Technique Illustrated and Pomodoro Technique Illustrated (audio book))
DevSum 2013, Stockholm

2013-06-05  Various topics related to Java and architecture
Venkat Subramaniam
Software Architecture Summit, Berlin

2013-06-06  Exploding Management Myths: Johanna will explain the management myths and what to do instead. She’ll tackle favorites such as 100% utilization, no time for training, promoting the best technical person and more.
Johanna Rothman
Agile Development Conference/Better Software, Las Vegas

2013-06-10  Using Akka and Scala in a custom Paxos implementation.
Michael Bevilacqua-Linn
Scaladays NYC

2013-06-10  Three talks
Andrew Hunt
Norwegian Developer Conference, Oslo, Norway

2013-06-12  "Well Behaved Web Apps in the Haute Societe of Native Apps"
Adrian Kosmaczewski
QCon New York

2013-06-12  Talks on various topics
Venkat Subramaniam
NDC Oslo

2013-06-23  Case: study: Is eXtreme Programming still alive and kicking?
Rachel Davies
SPA2013 London, UK

2013-07-16  Workshops and talks on various topics.
Venkat Subramaniam
UberConf

2013-07-24  10 Reasons You’ll Love Dart
Chris Strom (author of The SPDY Book, Dart for Hipsters, and 3D Game Programming for Kids)
OSCON, Portland
2013-07-25  Getting Started with 3D Programming in Three.js (tutorial)
            Chris Strom
            OSCON, Portland

2013-09-11  Get a flying start with BDD, the collaborative process that’s changing the face of
            software development.
            Matt Wynne (author of The Cucumber Book and Cucumber Recipes)
            BDD Kickstart, Barcelona

2013-09-14  TBD
            David Chelimsky (author of The RSpec Book)
            Baruco

2013-09-14  Hexagonal rails
            Matt Wynne
            Barcelona Ruby Conference

2013-10-25  Talk (and possibly a tutorial)
            Dave Thomas (author of Programming Ruby (2nd edition),
            Agile Web Development with Rails (3rd edition), The Ruby Object Model and Metaprogramming,
            Agile Web Development with Rails (4th edition), Programming Ruby 1.9 & 2.0 (4th edition), and
            Agile Web Development with Rails 3.2 (4th edition))
            Rakuten Technology Conference, Tokyo
Here’s what’s new and what’s hot from the Pragmatic Bookshelf.

**What’s New**

Test Your Software, or Your Users Will. We said that back in *The Pragmatic Programmer* book and it still rings true today—even more so, with tons of users now testing your iOS app, and reviewing it accordingly! If you write iOS apps, you need *Test iOS Apps with UI Automation: Bug Hunting Made Easy* now in beta at pragprog.com/book/jptios.

**What’s Hot**

Top-Ten lists are passé—ours goes to 11. These are the top titles that folks are interested in currently, along with their rank from last month. This is based solely on direct sales from our online store.

<table>
<thead>
<tr>
<th>Rank</th>
<th>New/Not New</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NEW</td>
<td>Practical Programming</td>
</tr>
<tr>
<td>2</td>
<td>NEW</td>
<td>Functional Programming Patterns in Scala and Clojure</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Programming Ruby 1.9 &amp; 2.0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Agile Web Development with Rails</td>
</tr>
<tr>
<td>5</td>
<td>NEW</td>
<td>Modern C++ Programming with Test-Driven Development</td>
</tr>
<tr>
<td>6</td>
<td>NEW</td>
<td>Functional Programming in Java</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>Practical Vim</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>The Definitive ANTLR 4 Reference</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>Good Math</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>Core Data</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>The Cucumber Book</td>
</tr>
</tbody>
</table>

**What’s Happening**

But to really be in the know, you need to subscribe to our weekly newsletter. It’ll keep you in the loop, it’s a fun read, and it’s free. All you need to do is create an account on pragprog.com (email address and password is all it takes) and select the checkbox to receive newsletters.
A startup may be driven by inspiration or ambition or ego or hunger. But in any mature business, all the really critical decisions are based on fear. And this is as it should be.

Because we're all doomed. It's only a matter of time before the buzzards are picking at your bones. What we call success is just that period of freefall between jumping out of the plane and hitting the ground. The only sane reaction to all this, of course, is terror, but the question is: which kind of terror?

You have to Know Your Fears. There are many species of fear, from the little fear nits that get under your collar and itch all the way up to the big scary elephant in the room fear. To help you identify your fears, you need a knowledgable Fear Guide. Only with expert guidance can you be sure that you are responding to the appropriate fear.

The ABCs of Fear

Bertrand Meyer, the creator of the language Eiffel, has given a lot of thought to this, and has developed a system for categorizing software projects based on how scary they are. He doesn't put it that way, for some reason, so I'll describe his system, highlighting the fear factors.

The lowest fear level is C, for Casual. Software from projects in this category can be unreliable, incomprehensible, not produce the right results, use too much memory, and run as slow as an old dog. An old French dog. You can use any programming language to write them, even Python, because nobody will ever read your code. Your primary fear with this kind of project is that the client's check will bounce. Most software projects, Meyer explains, fall in this category. (And like the tree falling in the woods, nobody hears them when they do.)

The next level of fear is B for Business. In projects at the B level, somebody's business depends on the proper functioning of the software you write. Not your business, thank God. Somebody else's. The business will assign somebody to the project whose job it is to keep pushing you to internalize the business's fears. Don't let them. Write B-level software in C++ and the suits will think it's serious stuff.

The highest fear level is A for Acute. If you screw up at this level, people will die. Evil will triumph. The killer asteroid will get through. You should be scared even to be assigned to an A project. If you are, the only chance you have of getting through it in one piece is to do all your programming in Eiffel.

I am just paraphrasing, but that seems to be the basic idea.
Alistair Cockburn is One Scary Guy

Alistair Cockburn is another expert on managing fear. He has learned to manage it so well that he is constantly jumping out of airplanes. Actual airplanes, not the metaphorical planes I mentioned earlier. This might give you some trepidation about taking his advice on your software project. And that would be a Good Thing, provided that you know how to leverage your trepidation.

Cockburn has developed a more elaborate, two-dimensional scale, but it all comes down to how much it hurts, and whom.

In Cockburn’s scale, the lowest fear level is fear of being made uncomfortable, followed by fear of losing the milk money, fear of losing the beer money, and fear of zombies eating your face (again, I’m paraphrasing).

The second dimension is size of the project, which maps to size of milk or beer budget or the number of people in danger of having their faces eaten or being made uncomfortable.

Basically, once you assign your project to a cell in this 4-by-n Fear Grid, you will be in a position to be able to use your fear wisely, knowing what strategy to apply, whether to circle the wagons, cover your posterior, or get out while the getting’s good. With agility.

Each of these scales measures the fear inherent in a project. But did you notice? They were both created by supremely confident and skilled individuals who could probably knock off your scary project alone in an afternoon.

That can’t be good. I suggest that you really need the perspective of someone who comes to the process without a clue. Someone who is pretty sure he would fail spectacularly. Then you’ll get some useful perspective.

I’m thinking Corey Levitan.

You know about Corey? For years he wrote a column in the Las Vegas Review-Journal called “Fear and Loafing” for which, every week, he took a job that he was ridiculously unqualified for. Skydiving Elvis. Ballerina. Midwife. And then he wrote about his experience. Which, predictably, was never all that successful, but was generally funny and frequently scary. Like the time he was a nude model.

So that’s the ticket. We need Corey Levitan to develop a Software Project Fear and Loafing Scale.

I know you can remember projects where it would have been appropriate.

About the Author

John Shade was born under a cloud in Montreux, Switzerland, in 1962. Subsequent internment in a series of obscure institutions of ostensibly higher learning did nothing to brighten his outlook. The only thing he’s not afraid of is fear itself. Follow John on Twitter, send him your feedback, or discuss the article in the magazine forum.