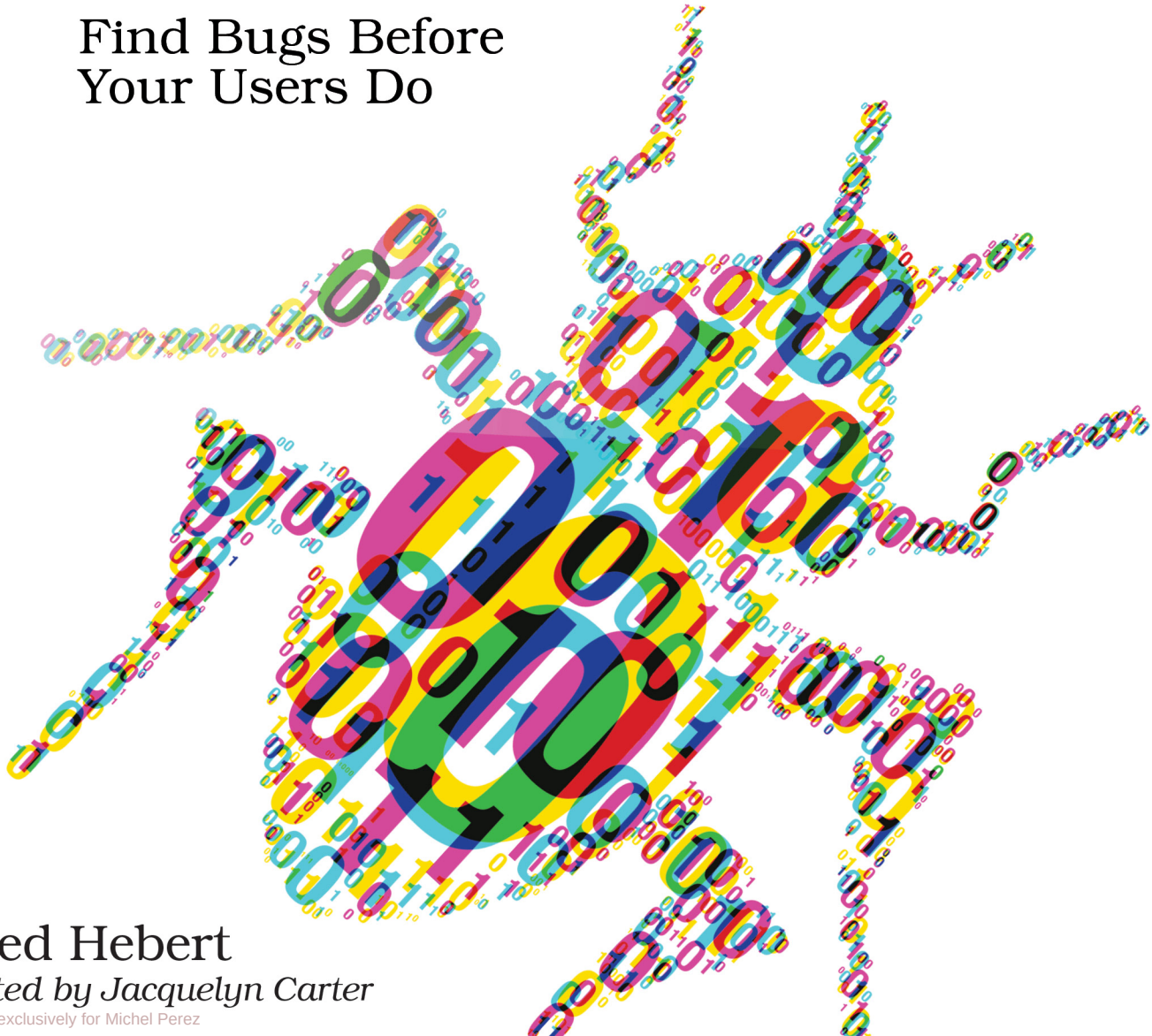
The Pragmatic Programmers

Property-Based Testing with PropEr, Erlang, and Elixir

Find Bugs Before Your Users Do



Fred Hebert

edited by Jacquelyn Carter

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/fhproper/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

Andy

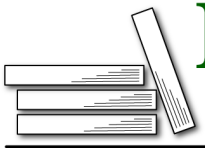
Property-Based Testing with PropEr, Erlang, and Elixir

Find Bugs Before Your Users Do

Fred Hebert

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-621-1
Book version: B1.0—August 8, 2018

Contents

Part II — Stateless Properties in Practice

Part III — Stateful Properties

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

Beta 1—August 8, 2018

- Initial release
- Please tell us what you think! Currently, most of the Elixir code is in an appendix, but we will be changing that. By the time the book is published, we plan to include the Elixir code along with the Erlang code in the chapters. We have some ideas, but we're still looking for ways to improve that. If you have any suggestions, please send them to us at fhproper@pragprog.com.

Introduction

When I finished my first book, *Learn You Some Erlang*, I told myself "never again". There's something distressing about spending months and years of work writing a book, spending all bits of free times you can find on it, putting aside other projects and hobbies, and rewriting your own texts close to a dozen times. You reach the point where before you're even done, you're tired of writing about the topic you chose to write about.

I knew all of that was waiting for me if I ever wanted to write another book. I decided to do it anyway because I truly believe property-based testing is something amazing, worth learning and using. In fact, part of the reason why I wanted to write a book was that I wanted to use property-based testing in projects at work and online, and it's generally a bad idea to introduce a technology when only one person in the team knows how it works.

It is a better compromise to spend all that time and effort writing a book than never using Property-Based testing in a team when I know what it can do and bring to a project. Hopefully, you'll feel that learning about it in here is worth your time as well.

Who is This Book For

If you know enough of Erlang or Elixir to feel comfortable writing a small project, you're fit for this book. There's a few things that might be a bit confusing, but you should be able to pull through.

If you've got experience with unit testing and TDD, you will feel comfortable with most of this book's testing concepts. While the text does not advocate using TDD or not (we avoid this whole debate), techniques that use properties to help design your programs are still shown, and constitute a valuable option to explore a new problem space.

If you are, like me, one of these grumpy people who are annoyed with the quality of software, and feel that you cannot trust yourself to deliver high quality code every time—you know your code will come back to haunt you

sooner or later—then you will probably consider property-based testing as a godsend.

Why Both Elixir and Erlang

The Erlang and Elixir communities possibly suffer from a kind of *narcissism of minor differences*; a kind of hostility exists between both communities, based on small differences between the two languages and how we do things, despite being so much closer to each other than any other language or platform out there.

This book represents a conscious effort to bridge the gap between both communities, and see both groups join strengths rather than compete with each other; it is one small part, attempting to use one property-based testing tool, with one resource, to improve the code and tests of one community.

What's in This Book

This book covers pretty much everything you need to get started, up to the point where you feel confident enough to use the most advanced features of PropEr. We'll start smoothly, with the basic and foundational principles of property-based testing, see what the framework offers us to get started, make our way through thinking in properties, write our own custom data generators, and then really start going wild. You'll see how property-based testing can be used in a realistic project (and where it should not be used), various techniques to make the best use of it possible and get the most value out of it. We'll also cover properties to test more complex stateful systems, a practice that is useful for integration and system tests.

Those are the topics covered, but more than anything, you may get a set of strategies to think about new approaches to test your software. Rather than just writing repetitive examples for tests, or just generating random data to throw at the code, you'll learn new techniques to find new bugs you never thought could be hiding in your code. You will also gain tools to reason about how to build software, how to explore the problem space, and evaluate the fitness of the solutions we choose to fix them.

How to Read This Book

You should feel comfortable just reading all chapters in order. The first part of the book contains truly essential material if you wish to get your fundamentals right and get started properly. The second part applies properties in more realistic scenarios to gain comfort, and the last part covers stateful tests.

But really, it is likely easiest to just read things in order. Some chapters have questions and exercises at the end. You can skip these if you want, but going through them will be a good way to reinforce your understanding of the material covered. The exercises are added particularly when a lot of theory was introduced in the chapter and when the material will come back again and again through following chapters. And because "exercises left for the reader" with no guidance are annoying as hell, all solutions are provided.

If you are mainly an Elixir reader, you should be prepared to flip pages quite a bit. To ease the flow of the text, translations are all in an appendix, in order. If you are using a paper book, you might want to keep two bookmarks: one for the main text, and one for the translations. If you are using an ebook version, your life will be made much easier by using an ebook reader that supports going back to the previous page or link encountered.

Online Resources

The apps and examples shown in this book can be found at the Pragmatic Programmers website for this book.[1]

The book relies on the PropEr[2] library. Its online documentation[3] will invariably prove useful.

Elixir users will use the PropCheck[4] wrapper library, which also has its own online documentation[5].

1. https://pragprog.com/book/fhproper/property-based-testing-with-proper-erlang-and-elixir
2. https://github.com/proper-testing/proper
3. https://proper-testing.github.io/
4. https://github.com/alfert/propcheck
5. https://hexdocs.pm/propcheck

Part I

The Basics

First things first. This part contains all the material we need to give you a solid foundation on which to build, and includes important material explaining what Property-Based Testing is, how to get set up, and information to let you get familiar with thinking and properties and the tools the framework provides for you.

CHAPTER 1

Foundations of Property-Based Testing

Testing can be a pretty boring affair. It's a necessity you can't avoid: the one thing more annoying than having to write tests is not having tests for your code. Tests are critical to safe programs, especially those that change over time. They can also prove useful to help properly design programs, helping us write them as a user as well as implementers. But mostly, tests are repetitive burdensome work.

Take a look at this example test that will check that an Erlang function can take a list of pre-sorted lists, and will always return them merged as one single sorted list:

```
merge_test() ->
    [] = merge([]),
    [] = merge([[]]),
    [] = merge([[],[]]),
    [] = merge([[],[],[]]),
    [1] = merge([[1]]),
    [1,1,2,2] = merge([[1,2],[1,2]]),
    [1] = merge([[1],[],[]]),
    [1] = merge([[],[1],[]]),
    [1] = merge([[],[],[1]]),
    [1,2] = merge([[1],[2],[]]),
    [1,2] = merge([[1],[],[2]]),
    [1,2] = merge([[],[1],[2]]),
    [1,2,3,4,5,6] = merge([[1,2],[],[5,6],[],[3,4],[]]),
    [1,2,3,4] = merge([[4],[3],[2],[1]]),
    [1,2,3,4,5] = merge([[1],[2],[3],[4],[5]]),
    [1,2,3,4,5,6] = merge([[1],[2],[3],[4],[5],[6]]),
    [1,2,3,4,5,6,7,8,9] = merge([[1],[2],[3],[4],[5],[6],[7],[8],[9]]),
    Seq = seq(1,100),
    true = Seq == merge(map(fun(E) -> [E] end, Seq)),
    ok.
```

This is slightly modified code taken from the Erlang/OTP test suites for the lists module, one of the most central libraries in the entire language. It is boring, repetitive. It is the developer trying to think of all the possible ways the code can be used, and to make sure that the result is predictable. You could probably think of another 10 or 30 lines that could be added and it could still be significant and explore the same code in somewhat different ways. Nevertheless, it's perfectly reasonable, usable, readable, and effective test code. The problem is that it's just so repetitive that a machine could do it. In fact, that's exactly the reason why traditional tests are boring. They're carefully laid out instructions where we tell the machine which test to run every time, with no variation, as a safety check.

It's not just that a machine *could* do it, it's that a machine *should* do it. We're spending our efforts the wrong way, and we could do better than this with the little time we have.

This is why property-based testing is one of the software development practices that generated the most excitement in the last few years. It promises better, more solid tests than nearly any other tool out there, with very little code. This means, similarly, that the software we develop with it should also get better—which is good, since overall software quality is fairly dreadful (including my own code). Property-based testing offers a lot of automation to keep the boring stuff away, but at the cost of a steeper learning curve and a lot of thinking to get it right, particularly when getting started. Here's what an equivalent property-based test could look like:

```
sorted_list(N) -> ?LET(L, list(N), sort(L)).

prop_merge() ->
    ?FORALL(List, list(sorted_list(pos_integer())),
        merge(List) == sort(append(List))).
```

Not only is this test shorter with just 4 lines of code, it covers more cases. In fact, it can cover hundreds of thousands of them. Right now the property-based test probably looks like a bunch of gibberish that can't be executed (at least not without the PropEr framework), but in due time, this will be pretty easy for you to read, and will possibly take less time to understand than even the long form traditional test would.

Of course, not all test scenarios will be looking so favorable to property-based testing, but that is why you've got this book. Let's get started right away: in this chapter, we'll see what are the results we should expect from property-based testing, and see what principles are behind the practice and how they influence the way to write tests. We'll also pick the tools we need to get going,

since, as you'll see, property-based testing does require a framework to be useful.

Promises of Property-Based Testing

Property-based tests are different from traditional ones, and require more thinking. A lot more. Good property-based testing is a learned and practiced skill, much like playing a musical instrument or getting good at drawing: you can get started and benefit from it in plenty of friendly ways, but the skill ceiling is very high. You could write property tests for years and still find ways to improve and get more out of them. Experts at property-based testing can do some pretty amazing stuff.

The good news is that even as a beginner, you can get good results out of property-based testing. You will be able to write simple, short, and concise tests that automatically comb through your code the way only the most obsessive tester could. Your code coverage will go high and stay there even as you modify the program without changing the tests. You'll even be able to use these tests to find new edge cases without even needing to modify anything.

With a bit more experience, you'll be able to write straightforward integration tests of stateful systems that find complex and convoluted bugs nobody even thought could be hiding in there. In fact, if you currently feel like a codebase without unit tests is not trustworthy, you'll probably discover the same phenomenon with property testing real soon. You'll have seen enough of what properties can do to see the shadows of bugs that haven't been discovered yet, and just feel something is missing until you've given them a proper shakedown.

Overall, you'll see that property-based testing is not just using a bunch of tools to automate boring tasks, but a whole different way to approach testing, and to some extent, software design itself. Now that's a bold claim, but we only have to look at what experts can do to see why that might be so. An exmaple can be found in Thomas Arts' slide set[1] and presentation[2] from the Erlang Factory 2016. In that talk, he mentioned using QuickCheck (the canonical property-based testing tool) to run tests on Project FIFO,[3] an open source cloud project. With a mere 460 lines of property tests, they covered 60,000 lines of production code, and uncovered 25 important bugs, including:

1. http://www.erlang-factory.com/static/upload/media/1461230674757746pbterlangfactorypptx.pdf
2. https://youtu.be/iW2J7Of8jsE
3. https://project-fifo.net

- timing errors
- race conditions
- type errors
- incorrect use of library APIs
- errors in documentation
- errors in the program logic
- system limits errors
- errors in fault handling
- one hardware error.

Considering that some studies estimate that developers average 6 software faults per 1000 lines of code, then finding 25 important bugs with 460 lines of tests is quite a feat. That's finding over 50 bugs per 1000 lines of test, with each of these lines covering 140 lines of production code. If that does not make you want to become a pro, I don't know what will.

Let's take a look at some more expert work. Joseph Wayne Norton ran a QuickCheck suite[4] of under 600 lines over Google's levelDB to find sequences of 17 and 31 specific calls that could corrupt databases with ghost keys. No matter how dedicated to the task someone is, nobody would have found it easy to come up with the proper sequence of 31 calls required to corrupt a database.

Again, those are amazing results; that's a surprisingly low amount of code to find a high number of non-trivial errors on software that was otherwise already tested and running in production. Property-Based Testing is so impressive that it has wedged itself in multiple industries, including mission-critical telecommunication components,[5] databases,[6] components of cloud providers' routing and certificate-management layers, IoT platforms, and even in cars.[7]

We won't start right at that level, but if you follow along (and do the exercises and practice enough), we might get pretty close. Hopefully, the early lessons will be enough for you to start applying property-based testing in your own projects. As we'll go, you'll probably feel like writing fewer and fewer traditional tests, and replace them with property tests instead. This is a really good thing, since you'll be able to delete code that you won't have to ever maintain again, at no loss in the quality of your software.

4. http://htmlpreview.github.io/?https://raw.github.com/strangeloop/lambdajam2013/master/slides/Norton-QuickCheck.html
5. http://www.erlang-factory.com/conference/ErlangUserConference2010/speakers/TorbenHoffman
6. http://www.erlang-factory.com/upload/presentations/255/RiakInside.pdf
7. http://www2015.taicpart.org/slides/Arts-TAICPART2015-Presentation.pdf

But as in playing music or drawing, things will sometimes be hard. When a we hit a wall, it can be tempting (and easy) to tell ourselves that we just don't have that innate ability that experts must have. It's important to remember property-based testing is *not* a thing reserved for wizard programmers. The effort required to improve is continuous, but the progress is gradual and stepwise. Each wall you hit reveals an opportunity for improvement. We'll get there, one step at a time.

Properties First

Before jumping to our tools and spitting out lines of code, the first thing to do is to get our fundamentals right and stop thinking property-based testing is about tests. It's about *properties*. Let's take a look at what the difference is, and what thinking in properties looks like.

Conceptually, properties are not that complex. Traditional tests are often example-based: you make a list of a bunch of inputs to a given program, and then list the expected output. You may put in a couple of comments about what the code should do, but that's about it. Your test will be good if you can have examples that can exercise all the possible program states you have.

By comparison, property-based tests have nothing to do with listing examples by hand. Instead, you'll want to write some kind of meta test: you find a rule that dictates the behavior that should always be the same no matter what sample input you give to your program, and encode that into some executable code—a *property*. A special test framework will then generate examples for you and run them against your property to check that the rule you came up with is respected by your program.

In short, traditional tests have you come up with examples that indirectly imply rules dictating the behavior of your code (if at all), and property-based testing asks you to come up with the rules first, and then tests them for you. It's a similar distinction as the one you'd get between imperative and declarative programming styles, but pushed to the next level.

The Example-Based Way

Here's an example. Let's say we have a function to represent a cash register. The function should take in a series of bills and coins representing what's already in the register, an amount of money due to be paid by the customer, and then the money handed by the customer to the cashier. It should return the bills and coins to cover the change due to the customer.

An approach based on unit tests may look like the following:

```
Line 1  %% Money in the cash register
    2  Register = [{20.00, 1}, {10.00, 2}, {5.00, 4},
    3              {1.00, 10}, {0.25, 10}, {0.01, 100}],
    4  %% Change               = cash(Register, PriceToPay, MoneyPaid),
    5  [{10.00, 1}]            = cash(Register, 10.00, 20.00),
    6  [{10.00, 1}, {0.25, 1}] = cash(Register,  9.75, 20.00),
    7  [{0.01, 18}]            = cash(Register,  0.82,  1.00),
    8  [{10.00, 1}, {5.00, 1}, {1.00, 3}, {0.25, 2}, {0.01, 13}]
    9                          = cash(Register,  1.37, 20.00).
```

On line 5, the test says that a customer paying a $10 item with $20 should expect a single $10 bill back. Line 6 says that for a $9.75 purchase paid with $20, a $10 bill with a quarter should be returned, for a total of $10.25. Finally, the test on line 8 shows a $1.37 item paid with a $20 bill yields $18.63 in change, with the specific cuts shown.

That's a fairly familiar approach. Come up with a bunch of arguments with which to call the function, do some thinking, and then write down the expected result. By listing many examples, we try to cover the full set of rules and edge cases that describe what the code should do. In property-based testing, we have to flip that around and come up with the rules first.

The Properties-Based Way

The difficult part is figuring out how to go from our abstract ideas about the program behavior to specific rules expressed as code. Continuing with our cash register example, two rules to encode as properties could be:

- The amount of change is always going to add up to the amount paid minus the price charged.
- The bills and coins handed back for change are going to start from the biggest bill possible first, down to the smallest coin possible. This could alternatively be defined as trying to hand the customer as few individual pieces of money as possible.

Let's assume we magically encode them into functioning Erlang code (doing this for real is what this book is about—we can't show it all in the first chapter). Our test, as a property, could look something like this:

```
for_all(RegisterMoney, PriceToPay, MoneyPaid) ->
    Change = cash(RegisterMoney, PriceToPay, MoneyPaid),
    sum(Change) == MoneyPaid - PriceToPay
    and
    fewest_pieces_possible(RegisterMoney, Change).
```

Given some amount of money in the register, a price to pay, and an amount of money given by the customer, call the `cash/3` function, and then check the

change given. You can see the two rules encoded there: the first one checking that the change balances, and the second one checking that the smallest amount of bills and coins possible is returned (the implementation of which is not shown).

This property alone is useless. What is needed to make it functional is a property-based testing framework. The framework should figure out how to generate all the inputs required (`RegisterMoney`, `PriceToPay`, and `MoneyPaid`), and then it should run the property against all the inputs it has generated. If the property always remains true, the test is considered successful. If one of the test cases fails, a good property-based testing framework will modify the generated input until it can come up with one that can still provoke the failure, but that is as small as possible—a process called *shrinking*. Maybe it would find a mistake with a cash register that has a billion coins and bills in it, and an order price in the hundreds of thousands of dollars, but could then replicate the same failure with only $5 and a cheap item. If so, our debugging job is made way easier by the framework, because a simple input means it's way easier to walk the code to figure out what happened.

For example, such a framework could generate inputs giving a call like `cash([{1.00, 2}], 1.00, 2.00)`, Whatever the denomination, we might expect the `cash/3` function would return a $1 bill and pass. Sooner or later, it would generate an input such as `cash([{5.00, 1}], 20.00, 30.00)`, and then the program would crash and fail the property because there's not enough change in the register. Paying a $20 purchase with $30, even if the register holds only $5 is entirely possible: take $10 from the $30 and give it back to the customer. Is that specific amount possible to give back though? In real life, yes. We do it all the time. But in our program, since the money taken from the customer does not come in as bills, coins, or any specific denomination, there is no way to use part of the input money to form the output. The interface chosen for this function is wrong, and so are our tests.

Comparing Approaches

Let's take a step back and compare example-based tests with property tests for this specific problem. With the former, even if all the examples we had come up with looked reasonable, we easily found ourselves working within the confines of the code and interface we had established. We were not really testing the code, we were describing its design, making sure it conforms to expectations and demands while making sure we don't slip in the future. This is valuable for sure, but properties gave us something more: they highlighted a failure of imagination.

Example-based unit tests made it easy to lock down bugs we could see coming, but those we couldn't see coming at all were left in place, and would probably made it to production. With properties (and a framework to execute them), we can instead explore the problem space more in depth, and find bugs and design issues much earlier. The math is simple: bugs that make it to production are by definition bugs that we couldn't see coming ahead of time. If a tool lets us find them earlier, then production will see fewer bugs.

To put it another way, if example-based testing helps ensure that code does what we expect, property-based testing forces the exploration of the program's behavior to see what it can or cannot do, helping find whether our expectations were even right to begin with. In fact, when we test with properties, the design and growth of tests requires an equal part of growth and design of the program itself. A common pattern when a property fails will be to figure out if it's the system that is wrong, or if it's our idea of what it should do that needs to change. We'll fix bugs, but we'll also fix our understanding of the problem space. We'll be surprised by how things we thought we knew were far more complex and tricky than we thought, and how often it happens.

That's thinking in properties.

Property-based Testing In Your Project

What we need now is a framework. As opposed to many testing practices that require a tiny bit of scaffolding and a lot of care, property-based testing is a practice that requires heavy tool assistance. Without a framework, we have no way to generate data and all we're left with a encoded rules that don't get validated. If you use a framework that does not generate great data or does not let you express the ideas you need, it will be very hard to get as high of a quality in your tests as you would with a good one.

As the book title implies, we will use PropEr. It can be used by both Erlang and Elixir projects, and integrate with the usual build tools used in both languages. There are other frameworks available, namely QuickCheck framework by Quviq[8], and Triq. These two frameworks and PropEr are similar enough to each other that if your team is using any of them, you'll be able to follow along with the text without a problem. This remains true if you're using Elixir, and you might also have heard about StreamData,[9] which is a property-based testing framework exclusive to Elixir. PropEr has fancier features than

8. http://quviq.com/
9. https://github.com/whatyouhide/stream_data

StreamData, particularly with regards to stateful testing. The concepts should not be hard to carry over from framework to framework in any case.

PropEr is usable on its own through either manual calls or command line utils. This is fine for some isolated tests and quick demos, but it's usually nicer to be able to get the framework to fit your everyday development workflow instead.

Erlang Workflow

We will first get things going in Erlang. You should have the language installed along with the `rebar3`[10] build tool. If you have used the Common Test or EUnit frameworks before, you may know that you can use `rebar3 ct` or `rebar3 eunit` to run the tests for your project. To keep things simple, we'll use a PropEr plugin that will let us call `rebar3 proper` and get similar results. That way, everyone feels at home even with brand new tools.

If you're one of the folks who enjoys putting your unit tests along with the source code, PropEr requires us to take a different approach, more in line with Common Test's vision: tests are in a separate directory and do not get to be part of the artifacts that could ever end up in production.

Fear Not the GPLv3 License

PropEr is licensed under the GPLv3. People are often worried about having to attach and ship GPL-licensed code with their projects. Placing the tests and dependencies that are related to PropEr in their own directory means that they are only used as development tools in a testing context, and never in production. This is true with `rebar3` and for `mix` as well in Elixir.

As such, whenever you will build a release with code to ship and deploy, none of the test code nor PropEr itself will be bundled or linked to your program. This tends to put most corporate lawyers at ease, without preventing the authors from getting contributions back in case the tool is modified or used at the center of a commercial product.

Let's make a fake project out of a template:

```
$ rebar3 new lib pbt
===> Writing pbt/src/pbt.erl
===> Writing pbt/src/pbt.app.src
===> Writing pbt/rebar.config
===> Writing pbt/.gitignore
===> Writing pbt/LICENSE
```

10. http://www.rebar3.org/

```
===> Writing pbt/README.md
$ cd pbt
```

That's a regular library project named pbt. The PropEr plugin and dependency must be added to the configuration of each individual project you want to use them with. You can do this by editing the rebar.config file so it contains the following:

Foundations/erlang/pbt/rebar.config

```
%% the plugin itself
{plugins, [rebar3_proper]}.

%% The PropEr dependency is still required to compile the test cases,
%% but only as a test dependency
{profiles,
    [{test, [
        {deps, [proper, recon]}
    ]}
]}.
```

This sets up the PropEr plugin for Rebar3 so that we can invoke it from the command line, and sets up the framework as a dependency for our test builds only, and only within that project. Let's first check that the plugin works by invoking any command:

```
$ rebar3 help proper
«automatically fetching and building plugin»
Run PropEr test suites
Usage: rebar3 proper [-d <dir>] [-m <module>] [-p <properties>]
                     [-n <numtests>] [-v <verbose>] [-c [<cover>]]
                     [--retry [<retry>]] [--regressions [<regressions>]]
«a lot more help output»
```

The build tool fetches the things it needs, and everything is now available within that directory. We're now ready to write a property, which we can do in the next section, after going over the equivalent setup for Elixir.

Elixir

If you're more familiar with Elixir, you can still use PropEr with mix[11], Elixir's build tool. By adding the propcheck[12] package to your project configuration, mix will be able to find and execute PropEr properties within the same files as those that contain standard Elixir test cases written in ExUnit, the language's default test framework.

Once again we'll start by setting up a standard project, this time in Elixir.

11. http://elixir-lang.github.io/getting-started/mix-otp/introduction-to-mix.html
12. https://hex.pm/packages/propcheck

```
$ mix new pbt
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/pbt.ex
* creating test
* creating test/test_helper.exs
* creating test/pbt_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd pbt
    mix test

Run "mix help" for more commands.
$ cd pbt
```

Within that project, modify the mix.exs file to include propcheck dependency for tests:

Foundations/elixir/pbt/mix.exs

```
# Run "mix help deps" to learn about dependencies.
defp deps do
  [
    {:propcheck, "~> 1.0", only: :test}
  ]
end
```

This makes PropEr available to the tool, and makes it so properties are perceived like regular ExUnit unit tests. As such, there's no need to use special commands, just the regular ones you may be used to. To get going we just have to remember to fetch the dependencies:

```
$ mix deps.get
«fetching package»
$  mix test
==> proper
«build information»
==> propcheck
«build information»
==> pbt
«build information»
```

Alright, things seem to be in place there as well. We can now run a property.

Running A Property

The properties we've seen so far were pseudocode-like and kind of just there. Now that we have a stand-in directory structure with a structure similar to what we'd have a in a real project, we're going tie everything up together. We'll add a property to the project and execute it. The property is going to be very basic and test nothing of significance, but it will give you a brief idea of what things look like, and how they should all fit together.

Erlang

As mentioned earlier, PropEr's properties are not located within the same module as your production source code. Instead, we'll enforce a strict separation of production code and test code, by forcing all properties to be placed in standalone modules under the test/ directory, right next to your Common Test and EUnit test suites, if you have any.

Properties must be added to modules that have a name starting with prop_ (so that the rebar3 plugin picks them up), and are otherwise regular Erlang code. You can add a module named prop_foundations under test/prop_foundations.erl, or you can also use templates to do the same thing for you by calling the following within the pbt/ directory:

```
$ rebar3 new proper foundations
===> Writing test/prop_foundations.erl
```

The file should contain Erlang code like the following:

Foundations/erlang/pbt/test/prop_foundations.erl

```
-module(prop_foundations).
-include_lib("proper/include/proper.hrl").

prop_test() ->
    ?FORALL(Type, term(),
        begin
            boolean(Type)
        end).

boolean(_) -> true.
```

That's what a property looks like. It's regular Erlang code–although a bit macro heavy. Calling the prop_test() function from the shell or any other bit of Erlang code wouldn't do anything useful, but when PropEr gets its hands on it, we have a test. That property always returns true and will always pass. We'll go through how they work in Chapter 2, *Writing Properties*, on page 17, but for now, let's just ask our build tool to run it:

```
$ rebar3 proper
```

```
«build information»
===> Testing prop_foundations:prop_test()
.....................................................................
....................
OK: Passed 100 test(s).
===>
1/1 properties passed
$ rebar3 proper -n 10000
«build information»
===> Testing prop_foundations:prop_test()
.....................................................................
......«a lot more dots»
OK: Passed 10000 test(s).
===>
1/1 properties passed
```

And alright, that's a test. Well, many of them. Each period is a specific test sample, and there's a hundred of them. As you can see in the second command, we can run as many as we want by using the `-n` argument. We've got 10,000 test cases in 10 lines of code! Everything is definitely in its right place.

Elixir

For Elixir, a similar mechanism is required. Properties go in the `test/` directory under regular test modules. Open up the file at `test/pbt_test.exs` that mix created for us, and replace its code with the following:

Foundations/elixir/pbt/test/pbt_test.exs

```
defmodule PbtTest do
  use ExUnit.Case
  use PropCheck

  property "always works" do
    forall type <- term() do
      boolean(type)
    end
  end

  def boolean(_) do
    true
  end
end
```

The `use` instructions load the macro definitions required, and instead of the `test "description" do ... end` format required by ExUnit, we use `property "description" do ... end` to write properties. Within that property, you can probably see the parts that are in common with Erlang.

You can ask `mix` to run the property for you through the `mix test` command:

```
$ mix test
```

```
  1 defmodule PbtTest do
.

Finished in 0.05 seconds
1 property, 0 failures

Randomized with seed 189382
```

While Erlang's build tool happily writes out one period per property sample, mix will group all of them under a single one if they pass. Aside from that, things are roughly the same. There's been a hundred executions, but all bundled up together.

Wrapping Up

We've now gone through a little bit of an overview of what property-based testing asks of us and offers in return, and have gotten things working in a typical project structure for both Erlang and Elxir. We've also gotten our toolbox in order and even though we've successfully run a property (and over 10,000 tests), chances are that we're not quite at a point where we feel like we really understand what goes on in the framework and how we could write arbitrary properties.

That's fine though because next chapter will get us there. We'll look into the specific format of properties, how we can tell the framework what data to generate for them, how to interpret test results, and so on.

CHAPTER 2

Writing Properties

Property-based testing requires us to approach testing differently from what we're used to. As we've seen earlier, the core of properties is coming up with rules about a program that should always remain true. That's the hard part. But just having that will not be enough: we'll need to find a way to turn these rules into executable code, so that a specific framework (PropEr in our case) can exercise them. We will also need to tell the framework about what kind of inputs it should generate so that it can truly challenge the rules, some thing we call a *generator*. Once we take the rules encoded as code, the generators, and then put them together through the framework, we have a property.

Properties come into multiple flavors, but the two main categories are stateless properties, and stateful properties. In traditional example-based testing, it's usually a good idea to get started with simple unit test. In property-based testing, stateless properties are their equivalent. They are a great fit to validate components that are isolated, stateless, and without major side-effects. Stateless properties are still usable for more complex stateful integration and system tests, but for those use cases, stateful properties (seen in Chapter 9, *Stateful Properties*, on page 147) are more appropriate.

The properties we've seen and discussed in the previous chapter were all stateless, albeit a bit abstract. In this chapter, we'll make everything concrete and see how stateless properties are structured so that we can read and understand them. We will also see what data generators are offered out of the box by PropEr, with some ways of composing them together. Finally, we'll run some more properties. This will give us the opportunity to figure out how to read the result of failing test cases, and to see how to fix them.

Structure of Properties

Even though this chapter will focus on stateless properties, do know that *all* properties will share a common basic structure. Fancier stateful properties will only add content and special calls, but the core will remain the same. All properties go into files that contain an Erlang (or Elixir) test module, and must respect a specific format. What we'll be looking for here is the structure we'll use within these modules to let PropEr know what the rules to test are, and to let it know how to generate the data it should use to test them.

File Structure

Let's start by taking another look at the code generated within an existing project when using the templates provided by rebar3's PropEr plugin, so that you get a feel for the general way properties should be laid out. In the root of any standard Erlang project, call the following command:

```
$ rebar3 new proper base
===> Writing test/prop_base.erl
```

The generated file contains the `prop_base` module, a test suite that is divided in three sections: one section for the many properties we will want to test, one for helper functions, and one for custom data generators:

```
Line 1  -module(prop_base).
        -include_lib("proper/include/proper.hrl").
        -
        -%%%%%%%%%%%%%%%%%%
      5 %%% Properties %%% %
        -%%%%%%%%%%%%%%%%%%
        -prop_test() ->
        -    ?FORALL(Type, mytype(),
        -        begin
     10 -            boolean(Type)
        -        end).
        -
        -%%%%%%%%%%%%%%%
        -%%% Helpers %%%
     15 -%%%%%%%%%%%%%%%
        -boolean(_) -> true.
        -
        -%%%%%%%%%%%%%%%%%%
        -%%% Generators %%%
     20 -%%%%%%%%%%%%%%%%%%
        -mytype() -> term().
```

The section starting at line 5 is for properties. You can have many properties per module, each within a dedicated function starting with the `prop_` prefix.

It's usually a good idea to put all the properties at the top of the file, so that whoever maintains your code will have an easy time seeing what's being tested exactly. Here we only have the prop_test() property, but before digging into that one, let's see the other sections of the file.

The two other sections are mostly just there to help us organize our code whenever we'll have multiple properties and jamming everything into each of them would be too repetitive or unreadable. Here the second section contains helper functions. We'll typically use this part of the file to extract functions that are common to multiple properties, or code that is simply long enough that it hurts readability of the properties themselves. When you have few properties or they're all short, you may want to omit this section.

The last part of the module contains generators. Each generator is a function that returns metadata representing a given type. This is a high level recipe for data generation, which PropEr knows how to interpret. By default, PropEr provides a bunch of them, but eventually we'll write custom generators. This section's purpose is therefore similar to the helpers sections, but it's dedicated to custom data generators. Whenever the data we need the framework to generate for us becomes fairly complex or shared across properties, moving these generators there will be helpful. We won't need this section for now, but as soon as we get to Chapter 4, *Custom Generators*, on page 45 we'll make use of it.

When a module's main purpose is to expose a bunch of tests or properties, we tend to call it a *test suite*. Not all property test suites have that same exact structure as the one here, and while some people will prefer a different code organization, this is the default one recommended by the rebar3 templates.

For Elixir users, files can have a similar structure:

```
defmodule PbtTest do
  use ExUnit.Case
  use PropCheck

  # Properties
  property "description of what the property does" do
    forall type <- my_type() do
      boolean(type)
    end
  end

  # helpers
  defp boolean(_) do
    true
  end

  # generators
```

```
  def my_type() do
    term()
  end
end
```

But since Elixir properties get to run within the existing ExUnit files your project might have, you will probably want to make the properties work with your existing test module structure instead.

Property Structure

Let's take a look at the prop_test() property. The thing standing out is the ?FORALL Erlang macro, which has been imported through the module attribute on line 2. The macro has the form:

```
?FORALL(InstanceOfType, TypeGenerator,
        PropertyExpression).
```

And in Elixir it looks like:

```
property "some description" do
  forall instance_of_type <- type_generator do
    property_expression
  end
end
```

The data for the test case is generated by the functions we will enter in the TypeGenerator position, called the *generators*. The framework will take these generators, execute them, and turn them into actual data, which will then be bound to the InstanceOfType variable. This variable is then made available within PropertyExpression, a piece of arbitrary code that must return true if the test is to pass, or false if it is to fail. Since PropertyExpression is a single expression, you'll frequently see begin ... end blocks used in that area to wrap more complex sequences of expressions into a single one.

The ?FORALL macro can be translated to:

```
proper:forall(TypeGenerator, fun(InstanceOfType) -> PropertyExpression end).
```

This format may feel less magical, and removes the need for the begin ... end expression. However, since all property-based testing frameworks in Erlang implement the ?FORALL macro interface, we'll use that format to make sure everyone feels at home.

Execution Model

To figure out what the macro ends up doing at run time, let's see the prop_test() property being tested by the framework:

```
$ rebar3 proper
«build output»
===> Testing prop_base:prop_test()
.....................................................................
```

In this output, PropEr finds our test suite (the prop_base module), finds all the properties it contains—only prop_test() for now—and executes them. Every period represents an instance of PropEr taking our property and testing it against some input. So we have one property run a hundred times, and succeeding for every one of them. For each run, PropEr did the following:

1. it expanded the generator (which was defined with the term() function provided by PropEr) to a given piece of data
2. it passed the expanded data to the property, binding it to the InstanceofType variable
3. it validated that the actual PropertyExpression returned true

That term() generator is something that is not standard in Erlang. That's where property-based testing frameworks do a bit of magic. The include file directives at the top of the file is causing all of it:

```
-include_lib("proper/include/proper.hrl").
```

The included proper.hrl file contains macro code and inlines and imports multiple functions provided by PropEr. The term() function is actually shorthand for the proper_types:term() function call. Try playing with it in the shell:

```
$ rebar3 as test shell
«build output and shell prelude»
1> proper_types:term().
{'$type',[{env,[{'$type',[{env,{inf,inf}},
                          {generator,{typed,#Fun<proper_types.7.91186128>}},
                          {is_instance,{typed,#Fun<proper_types.8.911818>}},
                          {shrinkers,[#Fun<proper_types.9.91186128>]},
                          [...]
```

This is a high level recipe for data generation, which PropEr knows how to interpret. To debug generators in the terminal, you can ask PropEr to materialize an instance of the data type through the proper_gen:pick(Generator) function call. Play with it a bit in the shell, to get a feel for how things work:

```
2> proper_gen:pick(proper_types:number()).
{ok,-6}
3> proper_gen:pick(proper_types:term()).
{ok,{30,'iPMÒR\237M\203',[],0,-2.3689578345518227,{},
     [[],4,{}],
     -0.04598152960751201}}
```

```
4> proper_gen:pick(proper_types:term()).
{ok,['^ l&\212iå×\210\n',1.5133511315056003,
     -8.964623044991622,-49.99970474088979,
     {<<1:1>>,1,5.666317902012531,{}},
     3.799206557402714,-0.5871676980812601,3]}
```

To experiment with Elixir, you should instead call `MIX_ENV="test" iex -S mix` to get a shell, and then call `:proper_types.term()` and `:proper_gen.pick()`.

Here, each sample is returned in a tuple of the form `{ok, Data}`. In the first case, `number()` returns -6. For the two other cases, since `term()` represents almost *any* Erlang data type (pids, for example, are not generated), what we get as a result is utter garbage, which can nevertheless be useful in some cases. The ability to get some sampling of the data generated will prove useful when debugging generators, especially when they grow in complexity later on.

We've just seen `number()` and `term()`, but those are only two of a large generator zoo. We should get familiar with more of them, since they end up being pretty critical in properly exercising properties.

Default Generators

Generators are a huge part of where a property-based testing framework's magic comes from. While we do the hard work of coming up with properties, the efficiency with which they will be exercised entirely depends on what kind of inputs will be passed to them. A framework with bad generators is not an interesting framework, and great generators will directly impact how much trust you can put in your tests.

Generators are functions that contain a bunch of internal parameters that direct their randomness, how the data they create gets more or less complex, and information about other generators that can be used as parameters. We really don't need to know how their internals work to use them; from our point of view, they're just functions that can be combined together. Being familiar with what data they can generate is however important, since it lets us really create all the kinds of Erlang data we might need. There's even times where you'll feel like you'd like to be able to use generators outside of a testing context because they are just that nifty.

PropEr comes with quite a few basic data generators out of the box. They tend to represent all the relevant basic datatypes of the language (aside from opaque data like pids, references, or sockets), with some special ones that allow you to do things such as making sure integers are in a given range, or make one generator that actually represents multiple other generators.

Here's a few of the important ones. A full list is available in Appendix 3, *Generators Reference*, on page 231—it would be a good idea to get familiar with them.

Generator	Data Generated	Sample
any()	Any Erlang term PropEr can produce. Same as term()	any of the samples below
atom()	Erlang atoms	'ós43Úrcá\200'
binary()	Binary data aligned on bytes	<<2,203,162,42,84,141>>
boolean()	Atoms true or false. Also bool()	true, false
choose(Min, Max)	Any integer between Min and Max inclusively. Same as range(Min, Max)	choose(1, 1000) => 596
float()	Floating point number. Also real()	4.982972307245969
integer()	An integer	89234
list(Type)	A list of terms of type Type	list(boolean()) => [true, true, false]
non_empty(Gen)	Constrains a binary or a list generator into not being empty	non_empty(list()) => [abc]
number()	A float or integer	123
range(Min, Max)	Any integer between Min and Max inclusively	range(1, 1000) => 596
string()	Equivalent to list(char()), where 'char()' returns character codepoints between 0 and 1114111 inclusively	"^DQ^W^R/D" (may generate weird escape sequences!)
{T1, T2, ...}	A literal tuple with types in it	{boolean(), char()} => {true, 1}

You don't need to remember all of these by heart. Just put a bookmark in the appendix part of the book so you have a quick reference handy, or go look at the online documentation for PropEr[1] or Quickcheck[2] when you need to find generators. However, it may be useful to play with the generators in the Erlang shell to get familiar with them. Try to see the results of typing the previous generators in the shell, and try some of those in the appendix as well:

```
$ rebar3 as test shell
1> proper_gen:pick(proper_types:range(-500,500)).
{ok,-64}
```

1. https://proper-testing.github.io/
2. http://quviq.com/documentation/eqc/index.html

```
2> proper_gen:pick({proper_types:bool(), proper_types:float()}).
{ok,{true,0.8565227550368821}}
3> {ok,F} = proper_gen:pick(proper_types:function(3, proper_types:list())).
{ok,#Fun<proper_gen.31.64457461>}
4> F(1,2,three).
[[-7,'(\226\212_J',-3,[<<0:3>>,{},[[]],53,'',{[]},-5],á,-1]]
5> proper_gen:pick(proper_types:non_empty(
5>    proper_types:list(proper_types:number()))).
{ok,[-38.585674396308065,17,9,-0.17548322369895938,3.7420633140371597]}
```

Even though the proper_types: prefix is required in the shell, it will not be needed in our actual test modules, since the framework will auto-import default generators there.

Ignore the Warnings

When experimenting with generators, you may find yourself faced with a warning like this:

WARNING: Some garbage has been left in the process dictionary and the code server to allow for the returned function(s) to run normally. Please run proper:global_state_erase() when done.

Don't worry, this is a thing PropEr displays in the shell when experimenting. Nothing will stay past the current session in your terminal, and you can usually safely ignore these.

We have the basic syntax of properties with the ?FORALL macro, a bit of an idea about what rules should be like from the previous chapter, and now we're completing the trio with a lexicon of generators that can create arbitrary data. What we need to do now is tie it all together.

Putting it All Together

To get started with writing our own properties, it may be useful to compare them with the type of tests we'd already have in place in an existing project. We'll start with a simple function, one that finds the largest element in a list. Coming up with properties is not all easy, so beginning with regular tests can sometimes be a good idea since it can at least help us flesh out what we think the code should be doing.

Standard unit tests would probably look a bit like this, using the EUnit syntax:

```
biggest_test() ->
    ?assert(5  =:= biggest([1, 2, 3, 4, 5])),
    ?assert(8  =:= biggest([3, 8, 7, -1])),
    ?assert(-5 =:= biggest([-10, -5, -901])),
    ?assert(0  =:= biggest([0])).
```

With regular unit tests, we put together a list of assertions, and hope that it will cover all cases. Here we check for lists that are pre-sorted, lists without any specific order, single-element lists, and lists containing any of positive or negative numbers. Are there any other edge cases that could exist? That's hard to say if we don't already know what they could be from experience.

With properties, we should be able to get a far better coverage of edge cases, including those we wouldn't think of by ourselves. For a property, we need to find a rule to describe the behavior of the biggest/1 function according to these examples. The obvious one is that the function should return the biggest number of the list passed in. The problem is figuring out how to encode that rule into a property. The function is so simple and such a direct implementation of the rule that it's hard to make an encoding of the rule that is not the function itself!

Instead we'll use a second implementation of the function, and use it as a sanity check with the existing one. If both versions agree, they're both possibly right (or equally wrong, so make sure one implementation is obviously correct). Here we'll use lists:last(lists:sort(L)) as a simple way to validate the function. It's not obvious why we're using that call specifically, but we'll get to the rationale behind that in the next chapter. For now, let's focus on getting the thing working:

```
prop_biggest() ->
    ?FORALL(List, list(integer()),
        begin
            biggest(List) =:= lists:last(lists:sort(List))
        end).
```

All properties must have a name of the form prop_Name() since the harness for PropEr in rebar3 looks at the prop_ prefix, both in the file name and the function, to know which pieces of code to run. For Elixir, propcheck just looks for the property keyword introducing the test in a regular ExUnit file.

The type generator list(integer()) is a composition of types, generating a list of integers for each iteration. We can parameterize type generators whenever it makes sense: a list can take a type as an argument, and so can a tuple, but an integer would not make sense to parametrize. The generated data is then bound to the List variable, which will be used in the property's body. The expression biggest(List) =:= lists:last(lists:sort(List)) will validate the biggest/1 function by comparing that the biggest element of the list is equivalent to the last element of a sorted list. Any expression that may evaluate to true or false can be used in the property body.

The implementation for biggest/1 is left entirely to us, but we should avoid making it use lists:sort/1 and lists:last/1 since that would nullify our test by making the code the same as the test, and so if we make a mistake in one place, it's going to be in the other place as well. Here's a possible implementation:

```
%%%%%%%%%%%%%%%
%%% Helpers %%%
%%%%%%%%%%%%%%%
biggest([Head | _Tail]) ->
    Head.
```

This is obviously wrong, since the biggest element is not always going to be the first of the list. Running the tests will only confirm that:

```
Line 1  $ rebar3 proper
    -   ===> Verifying dependencies...
    -   ===> Compiling pbt
    -   ===> Testing prop_base:prop_biggest()
    5   !
    -   Failed: After 1 test(s).
    -   An exception was raised: error:function_clause.
    -   Stacktrace: [{prop_base,biggest,
    -                    [[]],
   10                    [{file,"/Users/ferd/code/self/pbt/pbt/test/prop_base.erl"},
    -                     {line,16}]},
    -                {prop_base,'-prop_biggest/0-fun-0-',1,
    -                    [{file,"/Users/ferd/code/self/pbt/pbt/test/prop_base.erl"},
    -                     {line,10}]}].
   15   []
    -
    -   Shrinking (0 time(s))
    -   []
```

We get an unsurprising failure, after a single run. The PropEr output tells us about which function failed on output line 8, with its arguments (as a list) on line 9. We can see the initially failing data set as generated by the framework on line 15, and one simplified by the framework—a mechanism called *shrinking*—one on line 18. In both cases, the empty list ([]) is displayed since it cannot be simpified any further. In short, our biggest/1 function failed when called with an empty list.

Getting the biggest entry out of an empty list is nonsensical and should reasonably fail. The use case can be ignored by forcing the property to run on non-empty lists:

WritingProperties/erlang/pbt/test/prop_base.erl

```
Line 1  -module(prop_base).
    -   -include_lib("proper/include/proper.hrl").
```

```
-
-
5 %%%%%%%%%%%%%%%%%%
  %%% Properties %%%
-
-
-
10
-
```

```
%%%%%%%%%%%%%%%%%%
%%% Properties %%%
%%%%%%%%%%%%%%%%%%
prop_biggest() ->
    ?FORALL(List, non_empty(list(integer())),
        begin
            biggest(List) =:= lists:last(lists:sort(List))
        end).
```

Elixir translation on page 187

The non_empty/1 generator on line 8 wraps the generator for lists of integers so that no empty list comes out. Running the test again, we get:

```
$ rebar3 proper
===> Verifying dependencies...
===> Compiling pbt
===> Testing prop_base:prop_biggest()
.........!
Failed: After 10 test(s).
[-1,6,3,1]

Shrinking ...(3 time(s))
[0,1]
===>
0/1 properties passed, 1 failed
===> Failed test cases:
prop_base:prop_biggest() -> false
```

9 tests passed. The tenth one failed. Initially, it did so with the list of numbers [-1,6,3,1]. PropEr then shrunk the list to a simpler expression that triggers the failing case: a list with two numbers, where the second one is bigger than the first one. This is legitimate input causing a legitimate failure. The code needs to be patched:

```
%%%%%%%%%%%%%%%%
%%% Helpers %%%
%%%%%%%%%%%%%%%%
biggest([Head | Tail]) ->
    biggest(Tail, Head).

biggest([], Biggest) ->
    Biggest;
biggest([Head|Tail], Biggest) when Head > Biggest ->
    biggest(Tail, Head);
biggest([Head|Tail], Biggest) when Head < Biggest ->
    biggest(Tail, Biggest).
```

This code iterates over the entire list while keeping note of the biggest element seen at any given point. Whenever an element is bigger than the noted element,

it replaces it. At the end of the iteration, the biggest element seen is returned. With the new implementation, the previous case should be resolved:

```
$ rebar3 proper
===> Verifying dependencies...
===> Compiling pbt
===> Testing prop_base:prop_biggest()
.................!
Failed: After 18 test(s).
An exception was raised: error:function_clause.
Stacktrace: [{prop_base,biggest,
                        [[0,6,2,17],0],
                        [{file,"/Users/ferd/code/self/pbt/pbt/test/prop_base.erl"},
                         {line,19}]},
             {prop_base,'-prop_biggest/0-fun-0-',1,
                        [{file,"/Users/ferd/code/self/pbt/pbt/test/prop_base.erl"},
                         {line,10}]}].
[0,-4,0,6,2,17]

Shrinking ..(2 time(s))
[0,0]
===>
0/1 properties passed, 1 failed
===> Failed test cases:
  prop_base:prop_biggest() -> false
```

After 18 runs, PropEr found a bug. The initial failing case was [0,-4,0,6,2,17]. The actual problem with that result is not necessarily obvious. Shrinking however reduces the *counterexample* (the failing case) to [0,0], and the interpretation is simpler: if a list is being analyzed and the currently largest item is equal to the one being looked at (double-check the list of arguments in the stacktrace if this is not clear!), the comparison fails.

A quick patch can address this:

WritingProperties/erlang/pbt/test/prop_base.erl

```
Line 1 biggest([Head | Tail]) ->
   2     biggest(Tail, Head).
   3
   4 biggest([], Biggest) ->
   5     Biggest;
   6 biggest([Head|Tail], Biggest) when Head >= Biggest ->
   7     biggest(Tail, Head);
   8 biggest([Head|Tail], Biggest) when Head < Biggest ->
   9     biggest(Tail, Biggest).
```

Elixir translation on page 187

the > operator is changed for >= on line 6 in order to handle equality. The property finally holds:

```
$ rebar3 proper
===> Verifying dependencies...
===> Compiling pbt
===> Testing prop_base:prop_biggest()
.....................................................................
.....................
OK: Passed 100 test(s).
===>
1/1 properties passed
```

We're all good! We can trust our function to be quite alright.

Elixir and Test Repeats

Elixir users may have to remove the _build/propcheck.ctex file before rerunning tests to ensure fresh runs every time.

Most of the development that takes place for stateless properties follows that iterative pattern: write a property you feel makes sense, then throw it against your code. Whenever there's a failure, figure out if the property itself is wrong (as when we added the non_empty() generator), or if the program itself is wrong. Modify either accordingly to your needs.

Wrapping Up

You should now be good when it comes to knowing the basic syntax and operation of stateless properties. We've seen how the modules that contain properties tend to be structured, the syntax of a basic property, a bunch of generators to describe the data our properties will need, how to run them, and then how to debug our tests when they fail.

Writing more properties on your own should not technically be a problem, although it may prove challenging to figure out exactly what is it that makes a good property, and how to come up with them. The next chapter will help with that by showing multiple strategies that can prove useful when trying to translate a rule into a property.

Exercises

Question 1

What function can you call to get a sample of a generator's data?

Question 2

Explain what the following property could be testing or demonstrating:

WritingProperties/erlang/pbt/test/prop_exercises.erl

```
prop_a_sample() ->
    ?FORALL({Start,Count}, {integer(), non_neg_integer()},
        begin
            List = lists:seq(Start, Start+Count),
            Count+1 =:= length(List)
            andalso
            increments(List)
        end).

increments([Head | Tail]) -> increments(Head, Tail).

increments(_, []) -> true;
increments(N, [Head|Tail]) when Head == N+1 -> increments(Head, Tail);
increments(_, _) -> false.
```

Elixir translation on page 188

Feel free to add output and then run it and to see it execute. Do note that adding output may generate a lot of noise, so you may want to limit the number of tests for each call to make it easier to check things one bit at a time.

CHAPTER 3

Thinking In Properties

In the last chapter, we went over basic properties, including their syntax and generators that are available out of the box. We played with the biggest(List) function, ensuring it behaved properly. You may now have a good idea of what a property looks like and how to read them, but chances are you don't feel too comfortable writing your own; it can take people a long time to feel like they know how to write *effective* properties. All things considered, a big part of being good at coming up with properties is just a question of experience: do it over and over again and try your hand at it until it feels natural. That would usually take a long time, and we're going to try to speed things up.

Writing good properties is challenging and requires more effort than standard tests, but this chapter should provide some help. We're going to go through techniques that make the transitioning from standard tests to "thinking in properties" feel natural. You will become more efficient at figuring out how to go from a vague feeling of what your program should do to having clear and well-defined properties.

We'll go over a few tips and tricks to help us figure out how to write decent enough properties in tricky situations. First, we'll try *modeling* our code, so we can skip over a lot of the challenging thinking that would otherwise be required. When that does not work, we'll try generalizing properties out of traditional example-based cases, which will help us figure out the rules underpinning our expectations. Another approach we'll see is about finding invariants so that we can ratchet up from trivial properties into a solid test suite. Finally we'll implement symmetric properties as a kind of very easy cheat code for some specific problems.

Modeling

Modeling essentially requires you to write an indirect and very simple implementation of your code—often an algorithmically inefficient one—and pit it against the real implementation. The model should be so simple it is obviously correct. You can then optimize the real system as much as you want: as long as both implementations behave the same way, there is a good chance that the complex one is as good as the obviously correct one, but faster. So for code that does a conceptually simple thing, modeling is useful.

Let's revisit the biggest/1 function from last chapter and put it in its own module:

ThinkingInProperties/erlang/pbt/src/thinking.erl

```
-module(thinking).

-export([biggest/1]).

biggest([Head | Tail]) ->
    biggest(Tail, Head).

biggest([], Biggest) ->
    Biggest;
biggest([Head|Tail], Biggest) when Head >= Biggest ->
    biggest(Tail, Head);
biggest([Head|Tail], Biggest) when Head < Biggest ->
    biggest(Tail, Biggest).
```

The function iterates over the list in a single pass: the largest value seen is held in memory, and replaced any time a larger one is spotted. Once the list has been fully scanned, the largest value seen so far is also the largest value of the list. That value is returned, and everything is fine.

The challenge is coming up with a good property for it. The obvious rule to encode is that the function should return the biggest number of the list passed in. The problem with this obvious rule is that it's hard to encode: the biggest/1 function is so simple and such a direct implementation of the rule that it's hard to make a property that is not going to be a copy of the function itself. Doing so would not be valuable, because we're likely to repeat the same mistakes in both places and we might as well just not test it.

In these cases, modeling is a good idea. So for this function, we need to come up with an alternative implementation that we can trust to be correct to make the property work. Here's the property for this case:

ThinkingInProperties/erlang/pbt/test/prop_thinking.erl

```
-module(prop_thinking).
-include_lib("proper/include/proper.hrl").

%%%%%%%%%%%%%%%%%%
```

```
%%% Properties %%%
%%%%%%%%%%%%%%%%%%
prop_biggest() ->
    ?FORALL(List, non_empty(list(integer())),
        begin
            thinking:biggest(List) =:= model_biggest(List)
        end).
```

Most modeling approaches will look like that one. The crucial part is the model, represented by model_biggest/1. To implement the model, what we can do is pick standard library functions to give us our alternative, slower, but so-simple-it-must-be-correct implementation:

```
model_biggest(List) ->
    lists:last(lists:sort(List)).
```

Elixir translation on page 188

Since sorting a list orders all its elements from the smallest one to the largest one possible, picking the last element of the list should logically give us the biggest list entry. Running the property shows that it is good enough:

```
$ rebar3 proper -p prop_biggest
===> Verifying dependencies...
===> Compiling pbt
===> Testing prop_thinking:prop_biggest()
.....................................................................
.............................
OK: Passed 100 test(s).
===>
1/1 properties passed
```

Chances are pretty much null that all the functions involved are buggy enough that we can't trust our test to be useful and it only passes by accident. In fact, as a general rule when modeling, we can assume that our code implementation is going to be as reliable as the model to which we compare it. That's why you should always aim to have models so simple they are obviously correct. In the case of biggest/1, it's now as trustworthy as lists:sort/1 and lists:last/1 are.

Modeling is also useful for integration tests of stateful systems with lots of side-effects or dependencies, where "how the system does something" is complex, but "what the user perceives" is simple. Proper libraries or systems often hide such complexities from the user in order to appear useful at all. Databases, for example, can do a lot of fancy operations to maintain transactional semantics, avoid loss of data, and keep good performance, but a lot of

these operations can be modeled with simple in-memory data structures accessed sequentially.

Finally, there's a rare but great type of modeling that may be available, called the *oracle*. An oracle is a reference implementation, possibly coming from a different language or software package, and can therefore act as a pre-written model. The only thing required to test the system is to compare your implementation with the oracle and ensure they behave the same way.

If you can find a way to model your program, you can get pretty reliable tests that are easy to understand. You have to be careful about performance—if your "so simple it is correct" implementation is dead slow, so will your tests be—but models are often a good way to get started.

Generalizing Example Tests

Modeling tends to work well, as long as it is possible to write the same program multiple times, and that one of the implementations so simple it is obviously correct. This is not always practical, and sometimes not possible. We need to find better properties. That's significantly harder than finding *any* property, which can already prove difficult and requires a solid understanding of the problem space. A good trick to find a property is to just start by writing a regular unit test and then abstract it away. We can take the steps that are common to coming up with all individual examples and replace them with generators.

In the previous section, we said that `biggest/1` is as reliable as `lists:sort/1` and `lists:last/1`, the two functions we used to model it in its property. Our model's correctness entirely depends on these two functions doing the right thing. To make sure they're well-behaved, we'll write some tests demonstrating they work as expected. Let's see how we could write a property for `lists:last/1`. This function is so simple that we could just consider it to be axiomatic—just assume it's correct—and a fundamental block of the system. For this kind of function, traditional unit tests are usually a good fit since it's really easy to come up with examples that should be significant. Let's see how we could transform examples into a property. After all, if we can get a property to do the work for us, we'll have thousands of examples instead of the few we'd come up with, and that's objectively better coverage.

Let's take a look at what example tests could look like for `lists:last/1`, so that we can generalize them into a property:

```
last_test() ->
    ?assert(-23 =:= lists:last([-23])),
```

```
    ?assert(5 =:= lists:last([1,2,3,4,5])),
    ?assert(3 =:= lists:last([5,4,3])).
```

If we were to write this test by hand, we would:

1. construct a list by picking a bunch of numbers
    - pick a first number
    - pick a second number
    - ...
    - pick a last number
2. take note of the last number in the list as the expected one
3. check that the value expected is the one obtained

Since the last sub-step of *1.* ("pick a last number") is the one we really want to focus on, we can break it from the other sub-steps by using some clever generator usage. If we group all of the initial sub-steps in a list and isolate the last one, we get something like {list(number()), number()}. Here it is used in a property:

ThinkingInProperties/erlang/pbt/test/prop_thinking.erl

```
prop_last() ->
    %% pick a list and a last number
    ?FORALL({List, KnownLast}, {list(number()), number()},
        begin
            KnownList = List ++ [KnownLast], % known number appended to list
            KnownLast =:= lists:last(KnownList) % known last number is found
        end).
```

Elixir translation on page 189

And just like that, we'll get hundreds or even millions of example cases instead of a few unit tests all done by hand. Of course, we now have to believe the ++ operator will correctly append items to a list if we want to trust this new property. We're getting pulled in deeper: is it possible to make a model out of it? Can it be turned into a simpler property, or just tested with traditional unit tests? This is ultimately a question about which parts of the system you just trust to be correct, and that is left for you to decide. It's challenging to write a lot of significant tests for very simple cases, but when this is what you need, the next technique can help.

Invariants

Some programs and functions are just complex to describe and reason about. They could be needing a ton of small parts to all work right for the whole to be correct, or we may not be able to assert their quality because it is just

hard to define. For example, it's hard to say why a meal is good, but it might include a bunch of criteria like: the ingredients are cooked adequately, the food is hot enough, it is not too salty, not too sweet, not too bitter, it is well-presented, the portion size is reasonable, and so on. Those factors are all easier to measure objectively and can be a good proxy for "the customer will enjoy the food." In a software system, we can identify similar conditions or facts that should always remain true. We call them *invariants*, and testing for them is a great way to get around the fact things may just be ambiguous otherwise.

If an invariant were to be false at any time, you would know something is messed up. Seriously messed up. Here are some examples:

- A store cannot sell more items than it has in stock
- In a binary search tree, the left child is smaller and the right child is greater than their parent's value.
- Once you insert a record in a database, you should be able to read it back and not see it as missing.

A single invariant on its own is usually not enough to show a piece of code is working as expected. However, if we can come up with many invariants and small things to validate, and if they *all* always remain true, we can gain a lot more confidence in the ability of our code base to work well. Strong ropes are built from smaller threads put together. If you've ever read papers or proofs about why a given data structure works, you'll find that almost all aspects of them come from ensuring a few invariants are respected.

For property-based testing, we can write a lot of simple properties, each representing one invariant. As we add more and more of them, we can build a strong test suite that overall demonstrates that our code is rock solid.

The `lists:sort/1` function is a good example of a piece of code that could be checked with invariants. How can we identify the invariants though? We could pick the first one by saying "a sorted list has all the numbers in it ordered from smallest to largest". The problem is that this is such a complete and accurate description of the whole function that if we used it as an invariant, we'd need a complete sorting function to test it. This is a bit circular as it boils down to saying "a proper sort function is a function that sorts properly." A test that is written the same way as the code it tests is not useful.

Instead we should try to break it down into smaller parts. Something like "each number in a sorted list is smaller than (or equal to) the one that follows". The difference is small, but important. In the first case, we declare the final state of the entire list; the intended outcome. In the latter case, we mention

an invariant that should be true of any pair of elements, and not the whole output. We can do an entirely local verification without having the whole picture. Then, when we apply the property to every pair, we indirectly test for a fully ordered output:

ThinkingInProperties/erlang/pbt/test/prop_thinking.erl

```
prop_sort() ->
    ?FORALL(List, list(term()),
            is_ordered(lists:sort(List))).

is_ordered([A,B|T]) ->
    A =< B andalso is_ordered([B|T]);
is_ordered(_) -> % lists with fewer than 2 elements
    true.
```

Elixir translation on page 189

Not bad. A good side-effect of this approach is that the implementation is almost guaranteed to be different from the test: we only validated that some property held, and did not transform the input at all. No modeling is involved here. As mentioned earlier though, a single invariant is not very solid. If we had written a sort function as follows, it would always pass:

```
sort(_) -> [].
```

We need more invariants to ensure the implementation is right. We can look for other properties that should always be true and easy to check, such as:

- the sorted and unsorted lists should both have the same size
- any element in the sorted list has to have its equivalent in the unsorted list (no element added)
- any element in the unsorted list has to have its equivalent in the sorted list (no element dropped)

Let's see how these could be implemented:

ThinkingInProperties/erlang/pbt/test/prop_thinking.erl

```
%% @doc the sorted and unsorted list should both remain of the same size
prop_same_size() ->
    ?FORALL(L, list(number()),
            length(L) =:= length(lists:sort(L))).

%% @doc any element in the sorted list has to have its equivalent in
%% the unsorted list
prop_no_added() ->
    ?FORALL(L, list(number()),
        begin
            Sorted = lists:sort(L),
            lists:all(fun(Element) -> lists:member(Element, L) end, Sorted)
        end).
```

```
%% @doc any element in the unsorted list has to have its equivalent in
%% the sorted list
prop_no_removed() ->
    ?FORALL(L, list(number()),
        begin
            Sorted = lists:sort(L),
            lists:all(fun(Element) -> lists:member(Element, Sorted) end, L)
        end).
```

Elixir translation on page 189

That's better. Now it's harder to cheat your way through the properties, and we can trust our tests:

```
$ rebar3 proper
«build output»
===> Testing prop_sort:prop_sort()
.....................................................................
.........................
OK: Passed 100 test(s).
===> Testing prop_sort:prop_same_size()
.....................................................................
.........................
OK: Passed 100 test(s).
===> Testing prop_sort:prop_no_added()
.....................................................................
.........................
OK: Passed 100 test(s).
===> Testing prop_sort:prop_no_removed()
.....................................................................
.........................
OK: Passed 100 test(s).
===>
4/4 properties passed
```

Each of these properties is pretty simple on its own, but they turn out to really make a solid suite against almost any sorting function. The other great part of it is that some invariants are easy to think about, usually fast to validate, and they are almost always going to be useful as a sanity check no matter what. They will combine well with every other testing approach you could think of.

A small gotcha here is that our tests now depend on other functions from the lists module. This brings us back to the discussion on "when do we stop", since we need to trust these other functions if we want our own tests to be trustworthy. We could just call the shots and say we trust them, especially since they are given to us by the language designers. It's a calculated risk. But there's another interesting approach we could use by testing them all at once.

Symmetric Properties

From time to time, you may find it a bit difficult to figure out what component depends on which other one to succeed. Two bits of code may perform opposite actions, such as an encoder and a decoder. You need the encoder to test the decoder, and the decoder to test the encoder. In other cases, you may have a chain of operations that could be made reversible: editing text and undoing changes, translating text from French to English to Spanish and back to French, passing a message across multiple servers until it's back to its original one, or having a character in a game walking in all directions until it makes its way back to its origin.

Whenever you have such a reversible sequence of actions that you can assemble one together, you can write one of the most concise types of properties: *symmetric properties*. Symmetric properties' trick is that you test all of these moving parts at once; if one action is the opposite of the other, then applying both operations should yield the initial input as its final output. You pass in some data, apply the reversible sequence of operations, and check that you get the initial data back. If it's the same, then all the parts must fit well together.

Let's say we have a piece of code that does encoding and decoding. We could write the following property for it:

```
prop_symmetric() ->
    ?FORALL(Data, list({atom(), any()}),
        Data =:= decode(encode(Data))).
```

This property demonstrates that a lists of keys and value pairs can go a round of encoding and decoding without changing, showing that our encoding and decoding mechanisms are stable and lossless. If you're a proponent of Test-Driven Development's approach of "make the test pass really simply, and then refactor", then you know you will be able to defeat this test by writing an implementation like:

```
encode(T) -> T.
decode(T) -> T.
```

The property will pass all the time. The problem is that while the chosen property is useful, it is not sufficient on its own to be a good test of encoding and decoding. It checks that the encoding and decoding together don't lose any information, but we don't have anything to check that data actually gets encoded at all. An additional property of encoding could be added to the same test:

ThinkingInProperties/erlang/pbt/test/prop_thinking.erl
```
prop_symmetric() ->
    ?FORALL(Data, list({atom(), any()}),
        begin
            Encoded = encode(Data), is_binary(Encoded) andalso
            Data =:= decode(Encoded)
        end).
```

Elixir translation on page 190

The is_binary(Encoded) call forces the encoder to at least transform something. The encoder decoder may now need to look like:

```
%% Take a shortcut by using Erlang primitives
encode(T) -> term_to_binary(T).
decode(T) -> binary_to_term(T).
```

Other properties could include ideas like “only ASCII characters are used”, or “the returned binary value has proper byte alignment”. Those would be invariants, which *anchor* broad generic properties with the specifics of a given implementation. Traditional tests are also a good way to anchor broad tests, and they may be simpler to come up with as well.

The invariants will show that each individual part of the chain does some things right, and are not flat-out broken. The symmetric properties will show that all the distinct parts must compose and play well together, and that overall a large part of our implementation has to be reasonable. With both types of properties, we have some very minimalistic tests that show that a lot of stuff must be going right in our system; we get large coverage with almost no effort.

Putting it All Together

We’ve written a good lot of properties in this chapter covering multiple techniques, and they’ve been a bit all over the place. Let’s put them back into the context of the full test suite so we can give them a once-over:

ThinkingInProperties/erlang/pbt/test/prop_thinking.erl
```
-module(prop_thinking).
-include_lib("proper/include/proper.hrl").

%%%%%%%%%%%%%%%%%%
%%% Properties %%%
%%%%%%%%%%%%%%%%%%

prop_biggest() ->
    ?FORALL(List, non_empty(list(integer())),
        begin
            thinking:biggest(List) =:= model_biggest(List)
```

```
        end).

prop_last() ->
    %% pick a list and a last number
    ?FORALL({List, KnownLast}, {list(number()), number()},
        begin
            KnownList = List ++ [KnownLast], % known number appended to list
            KnownLast =:= lists:last(KnownList) % known last number is found
        end).

prop_sort() ->
    ?FORALL(List, list(term()),
            is_ordered(lists:sort(List))).

%% @doc the sorted and unsorted list should both remain of the same size
prop_same_size() ->
    ?FORALL(L, list(number()),
            length(L) =:= length(lists:sort(L))).

%% @doc any element in the sorted list has to have its equivalent in
%% the unsorted list
prop_no_added() ->
    ?FORALL(L, list(number()),
        begin
            Sorted = lists:sort(L),
            lists:all(fun(Element) -> lists:member(Element, L) end, Sorted)
        end).

%% @doc any element in the unsorted list has to have its equivalent in
%% the sorted list
prop_no_removed() ->
    ?FORALL(L, list(number()),
        begin
            Sorted = lists:sort(L),
            lists:all(fun(Element) -> lists:member(Element, Sorted) end, L)
        end).

prop_symmetric() ->
    ?FORALL(Data, list({atom(), any()}),
        begin
            Encoded = encode(Data), is_binary(Encoded) andalso
            Data =:= decode(Encoded)
        end).


%%%%%%%%%%%%%%%
%%% Helpers %%%
%%%%%%%%%%%%%%%
model_biggest(List) ->
    lists:last(lists:sort(List)).

is_ordered([A,B|T]) ->
    A =< B andalso is_ordered([B|T]);
is_ordered(_) -> % lists with fewer than 2 elements
    true.
```

```
%% Take a shortcut by using Erlang primitives
encode(T) -> term_to_binary(T).
decode(T) -> binary_to_term(T).

%%%%%%%%%%%%%%%%%%
%%% Generators %%%
%%%%%%%%%%%%%%%%%%

%% nothing!
```

Elixir translation on page 190

As you can see, the structure shown here is similar to what was introduced in *Structure of Properties*, on page 18. Because there's many properties, it's a bit simpler to scan them all if they're all in one section. Helpers are put together, even if there's no reuse across properties for now; with most editors or IDEs, it will be easy to jump from a property to its helpers anyway, and a familiar structure for most suites will help readability.

An interesting thing in the suite is about prop_biggest(), which was our initial model-based property. We've then added almost half a dozen other related properties extending the model we used, to gain more confidence that it is reliable. In the examples here, things were simple enough, but in a larger system the same approach is entirely desirable. It's hard to make one good solid property that covers everything, the same way it's hard to write one big function that does everything.

This is the last tip that could prove helpful here. Testing is like writing regular code: when the problem appears to be complex, do not try to have one big property that validates all the things at once. It is often simpler to break things up and attack one chunk at a time. Have multiple property tests for each of the properties you can identify in your code. Trying to have two or three shorter and concise properties can make the overall test suite much clearer (and faster) than a large convoluted one, especially when it comes to debugging. It also lets you build a suite progressively, rather than all at once and hoping it's good enough.

Wrapping Up

In this chapter, you've seen multiple ways to help come up with properties. We've been over modeling our system with a simpler or alternative implementation and checking that they both work the same, and you saw how example tests could be generalized to the point where their creation is automated. We also looked at how multiple small invariants can be put together to make a test more solid, and finished up with some symmetric properties. For most

properties, any of these four approaches will represent a decent starting point if you're not too sure how to get going.

We'll have plenty of chances to put these ideas in practice. However, you'll soon see that some properties are hard to test with just the default generators: if you want to focus on a behavior for a specific class of input, then very broad generators are not the best way to get that done. In the next chapter, we'll see how to efficiently drill down into the data generation phase of properties, improving our tests with custom generators.

Exercises

Question 1

What are 3 types of strategies that can be used to help design properties?

Question 2

Using prop_sort() as a source of inspiration, write some properties to show that lists:keysort/2 works properly

Question 3

The following property uses a model to validate that it works fine. However, the property is failing. Can you figure out if the model or the system under test is to blame? Can you fix it?

ThinkingInProperties/erlang/pbt/test/prop_exercises.erl

```
prop_set_union() ->
    ?FORALL({ListA, ListB}, {list(number()), list(number())},
        begin
            SetA = sets:from_list(ListA),
            SetB = sets:from_list(ListB),
            ModelUnion = lists:sort(ListA ++ ListB),
            lists:sort(sets:to_list(sets:union(SetA, SetB))) =:= ModelUnion
        end).
```

Elixir translation on page 191

Question 4

The following property verifies that two dictionaries being merged results in each of the entries being in there only once, without conflict, by comparing each of the keys is present:

ThinkingInProperties/erlang/pbt/test/prop_exercises.erl

```
prop_dict_merge() ->
    ?FORALL({ListA, ListB}, {list({term(), term()}), list({term(), term()})},
        begin
            Merged = dict:merge(fun(_Key, V1, _V2) -> V1 end,
```

```
                                dict:from_list(ListA),
                                dict:from_list(ListB)),
            extract_keys(lists:sort(dict:to_list(Merged)))
            ==
            lists:usort(extract_keys(ListA ++ ListB))
        end).

extract_keys(List) -> [K || {K,_} <- List].
```

Elixir translation on page 192

Our code reviewer however assert that the test is not solid enough because it only superficially tests the result of merging both dictionaries. What parts of its inputs and outputs are not being validated properly, and how would you improve it?

Question 5

Write a function that counts the number of words in a given string of text, and write a property that validates its implementation. Only consider spaces (‘ ’) to be word separators.

CHAPTER 4

Custom Generators

By now you've seen how to write properties, use default generators, and have gotten tips to help come up with new properties. If you only were to use these tools, chances are you'd already be able to improve things quite a bit in the tests you've got in your existing projects, but properties can get much more useful.

For our properties to be even more useful, we may need to get fancier. In the previous chapter, we've seen how to test an encoder and a decoder with symmetric properties, but say we suspected there was a bug whenever there are more than 255 elements in a map (255 is a number that could hit the upper limits of what a *byte* can store). Would you know how to validate that your property generates any of these elements? Would you know how to generate that data at all if it weren't there already?

To put it another way, the properties we've written so far have relied on fairly random generated data to act as inputs. The theory is that with enough random walks, we can eventually get into every nook and cranny of the code and find all the edge cases. There's however no guarantee that this will ever happen. In fact the opposite could be true: since the runs are limited, and given that edge cases are rare, random data could effectively just be very average at finding bugs. For example, let's imagine we have a system that has a subtle bug that only gets triggered when a key gets overwritten too many times in a database. If PropEr generates sequences of random keys for our properties, chances are that few of them will be duplicates. If the keys generated are almost never duplicates, our property is unlikely to exercise overwrites at all, and therefore unlikely to ever uncover the bug. There's a difference to be made between variety of data inputs and variety of operations.

In this chapter, we'll see what makes a generator tick. We'll get to play with all kinds of ways to generate the best data available for our tests: managing

their size, applying arbitrary transformations on the data, preventing some values from being possible, controlling their distribution, handling recursive generators, and representing generators as symbolic calls. But first, let's take a look at evaluating the quality and fitness of our generators, to know which of these techniques we should apply.

The Limitations of Default Generators

Default generators as we have used them are critical building blocks for our properties. They cover a large potential data space, which is absolutely useful in some types of tests. The properties that could be interesting to us may however require a very narrow focus on some limited edge cases. Generators covering a large data space can be useful to discover unexpected issues and problems, but possibly not great at exploring tricky areas known to be likely to contain bugs in our programs.

Let's say that we're writing a list of key and value pairs to insert in a map, and we want to make sure that as long as we enter keys and values in there, all the keys will be found in the map. We may get a property that looks a bit like this:

```
prop_dupes() ->
    ?FORALL(KV, list({key(), val()}),
        begin
            M = maps:from_list(KV),
            [maps:get(K, M) || {K, _V} <- KV], % crash if K's not in map
            true
        end).

key() -> integer().
val() -> term().
```

Elixir translation on page 192

First of all, you may have noticed that this property uses custom functions for generators; that's entirely legal. Any function can be used to wrap generators up into a higher-level, more descriptive construct, and you are encouraged to do so when you can. This property works fine, but if you were the one to implement maps:from_list/1, you might ask yourself questions such as what happens when the list is large, or when keys in there are duplicated. How would you go about measuring that? And if the numbers are wrong, how would you go about fixing it?

To know if there's a problem, you could jump into the shell and do some sampling:

```
$ rebar3 as test shell
```

```
«build info»
1> proper_gen:sample(proper_types:list(
1>    {prop_generators:key(), prop_generators:val()}
1> )).
[{-5,-7}]
[{2,<<40,97,2,213,1:1>>},
 {5,[{{},11,0,-12,{},'\034\202g\204jÿ?»4·à'},<<104:7>>]},
 {70,<<1:1>>},
 {1,[{'h\231U',{}},{},[]]},
 {7,
  {-4.2121702104995915,
   <<225,143,31,32,72:7>>,
   -2.9483965331393125,'\216´_\205£4¢mwr\210',-1.0863023018036206,
   'ÉÏ»<\235½'}}]
«a lot more output»
```

The problem with that is that it's not necessarily very obvious whether things are repeated or not, and how that would compare to running the property 10,000 times, and it requires some programmer discipline to keep checking and analyzing these things over time as people modify and refactor code and tests. Instead, the gathering of statistics is something that should be built directly into our tests.

Gathering Statistics

While there's nothing wrong with using in-shell samples to get a quick feel for how things work, a more pragmatic approach will be to get some values displayed to our faces every time we run the tests, so that we can know at a glance whether anything looks odd or out of place. There are two functions that can be used for this: collect/2 and aggregate/2.

We'll focus on collect/2 first since it's a bit more specific and straightforward, and then expand to aggregate/2.

Collecting

The collect(Value, PropertyResult) function allows you to gather the values of one specific metric per test, and build stats out of all the runs that happened for a property. It is a bit special in that you need to use it to wrap the actual property result and add context to it:

CustomGenerators/erlang/pbt/test/prop_generators.erl

```
prop_collect1() ->
    ?FORALL(Bin, binary(), collect(byte_size(Bin), is_binary(Bin))).
```

Elixir translation on page 192

The first argument is the metric from which you want to build statistics—here it's the binary's length—and the second argument is the result of the property. Under the hood, collect/2 takes both values and wraps them up in a way that lets PropEr both gather the metrics and validate the properties.

If we run the property, we see:

```
$ rebar3 proper
«build output»
===> Testing prop_generators:prop_collect1()
«build output»
OK: Passed 100 test(s).

10% 2
7% 0
7% 3
6% 1
6% 6
6% 9
6% 11
5% 7
«more statistics»
```

It works! Except that in this case, the statistics are not really convincing since we just get individual numbers with limited repetitions. If we instead group the values by a given range (by groups of 10), we get better results:

CustomGenerators/erlang/pbt/test/prop_generators.erl

```
prop_collect2() ->
    ?FORALL(Bin, binary(),
            collect(to_range(10, byte_size(Bin)), is_binary(Bin))).

to_range(M, N) ->
    Base = N div M,
    {Base*M, (Base+1)*M}.
```

Elixir translation on page 192

The to_range/2 function places a value M into a given bucket of size N. If you run that property, you'll get a much clearer result:

```
===> Testing prop_generators:prop_collect2()
«bunch of output»
OK: Passed 100 test(s).

56% {0,10}
27% {10,20}
13% {20,30}
3% {30,40}
1% {40,50}
===>
1/1 properties passed
```

In this case, more than 80% of generated binaries had a length between 0 and 20 bytes. If we know that our code has special handling only happening when binaries hit 1 megabyte in size, we know the current generator is not good enough.

If we wanted to try it with our map encoding property to find how often duplicate keys are used, we might want to do:

CustomGenerators/erlang/pbt/test/prop_generators.erl

```
prop_dupes() ->
    ?FORALL(KV, list({key(), val()}),
      begin
        M = maps:from_list(KV),
        [maps:get(K, M) || {K, _V} <- KV], % crash if K's not in map
        collect(
            {dupes, to_range(5, length(KV) - length(lists:ukeysort(1,KV)))},
            true
        )
      end).
```

Elixir translation on page 193

In this one, we collect the number of times keys were duplicated by taking the full list length (length(KV)) and subtracting the number of unique keys from it (lists:ukeysort(1, K)); the difference is going to be the number of duplcated keys.

Run this one and you'll get:

```
$ rebar3 proper -p prop_dupes
«build info»
===> Testing prop_generators:prop_dupes()
.....................................................................
.....................
OK: Passed 100 test(s).

95% {dupes,{0,5}}
5% {dupes,{5,10}}
```

Those are not very good results. Almost no keys are duplicated. To remedy this, we may need to go look at our default generators and see if we could do anything to help improve things.

For example, let's change the definition of key() so that we can probabilistically get better results. We'll use the oneof/1 generator for this. oneof([ListOfGenerators]) will randomly pick one of the generators within the list passed to it. It has two aliases in PropEr, named union(Types) and elements(Types), and they can all be used interchangeably.

QuickCheck differences

In QuickCheck, oneof() will try to shrink a failing case by finding a failing element whereas elements() will try to shrink counterexamples by focusing towards the first elements of the list; PropEr does not make that distinction.

Using that one, key() might be rewritten as:

```
key() -> oneof([range(1,10), integer()]).
val() -> term().
```

Elixir translation on page 193

With this generator, we still get the ability to generate the full range of all integers, but drastically increase the chance that some of them will be between 1 and 10. Over multiple runs, we're almost guaranteed to get repeated keys. But don't just believe it, go try it:

```
$ rebar3 proper -p prop_dupes
«build info»
===> Testing prop_generators:prop_dupes()
.....................................................................
.........................
OK: Passed 100 test(s).

80% {dupes,{0,5}}
12% {dupes,{5,10}}
5% {dupes,{10,15}}
3% {dupes,{15,20}}
```

The amount of repetition is now much higher. From the same property, you get different scenarios and more interesting use cases. And any time someone runs the property or modifies it, they'll get to see the statistics to know if something looks odd.

The downside though is that collect/2 can focus on only a single value per property, not more. Instead, for cases where you need something a bit fancier, aggregate/2 is there for you.

Aggregating

Another function to gather statistics is aggregate(). aggregate() is similar to collect(), with the exception that it can take a list of categories to store. To see it in action, here's a bit of code that gathers the distribution of cards being handed to a player:

CustomGenerators/erlang/pbt/test/prop_generators.erl

```
prop_aggregate() ->
    Suits = [club, diamond, heart, spade],
```

```
    ?FORALL(Hand, vector(5, {oneof(Suits), choose(1,13)}),
            aggregate(Hand, true)). % `true' makes it always pass
```

Elixir translation on page 193

This property's generator creates a hand of five cards in a list. There may be duplicate cards, since nothing keeps the generator from returning something like {1, spade} 5 times. The list of cards is passed directly to aggregate/2, with no transformations. The function can use these multiple values and generate statistics for the overall set of possible cards:

```
$ rebar3 proper
«bunch of output»
OK: Passed 100 test(s).

3% {spade,11}
2% {club,1}
2% {club,8}
2% {heart,4}
2% {heart,8}
2% {heart,9}
2% {diamond,3}
«bunch of output»
```

In terms of distribution, this seems to tell us all cards can be obtained rather uniformly. This can be done even though multiple cards are drawn at every round.

Adding Code Coverage

Running the command as rebar3 do proper -c, cover -v will give coverage analysis to your code. For Elixir users, mix test --cover will generate a static report in the cover/ directory.

Another interesting case for aggregation is the one where we might be interested in gather metrics on various data categories. Let's say we're analyzing generated text to see if it contains tricky characters. We may want to take the original string, and see each class of characters we get, according to some categories we define:

CustomGenerators/erlang/pbt/test/prop_generators.erl

```
prop_escape() ->
    ?FORALL(Str, string(),
            aggregate(classes(Str), escape(Str))).

escape(_) -> true. % we don't care about this for this example

classes(Str) ->
    L = letters(Str),
    N = numbers(Str),
    P = punctuation(Str),
```

```
    O = length(Str) - (L+N+P),
    [{letters, to_range(5,L)}, {numbers, to_range(5,N)},
     {punctuation, to_range(5,P)}, {others, to_range(5,O)}].

letters(Str) ->
    length([1 || Char <- Str,
                 (Char >= $A andalso Char =< $Z) orelse
                 (Char >= $a andalso Char =< $z)]).

numbers(Str) ->
    length([1 || Char <- Str, Char >= $0, Char =< $9]).

punctuation(Str) ->
    length([1 || Char <- Str, lists:member(Char, ".,;:'\"-")]).
```

Elixir translation on page 193

Run this code, and you'll see something like this:

```
$ rebar3 proper -p prop_escape
«build info»
===> Testing prop_generators:prop_escape()
.....................................................................
.........................
OK: Passed 100 test(s).

25% {numbers,{0,5}}
25% {punctuation,{0,5}}
24% {letters,{0,5}}
10% {others,{0,5}}
5% {others,{5,10}}
4% {others,{10,15}}
2% {others,{15,20}}
2% {others,{20,25}}
1% {others,{25,30}}
0% {letters,{5,10}}
```

If our code testing for escaping was dealing with SQL, then the amount of punctuation we're throwing at it is worryingly low: most characters fit the other category and are things we may not be interested in. That's a clear sign that we'd need to use something fancier for string generation.

The aggregate/2 function, along with collect/2, will be critical to most of the validation for our generators, along with possibly code coverage metrics. Now that you know how to identify properties needing better generators, we can fully dive into the mechanisms that are available to help us improve them.

Basic Custom Generators

Once you discover the need for your data generation to be retargeted to explore a problem space better, there's plenty of constructs available. We've seen how

to use oneof(Generators) to impact our ability to get a more specific result set, but other more flexible tools exist to get even more control. In this section, we'll see the basic building blocks of writing custom generators.

Those building blocks are controlling the size and amount of generated data, applying transformation to the generators themselves, restricting and filtering out some data from your generators, and finally, playing with probabilities when none of the previous approaches work.

Resizing Generators

The first and simplest thing to do with generators is to make them grow bigger: make an integer larger, a list longer, or a string have larger codepoints. In fact, PropEr makes that happen for us on its own as the tests run. The way generator growth works is that PropEr internally uses a size parameter. Its value is usually small at first, but as tests run the value is increased, and the data generated gains in complexity along with it. This allows the framework to start testing our systems with initially small data, and to then progressively make it more and more complex.

This is part of a strategy to find easy edge conditions such as 0, empty containers or strings, handling negative or positive numbers (which turn out to cover a large set of bugs) early on. Then, as each early test passes against these expected edge conditions, the framework grows the data it generates to find trickier bugs. The more tests pass in a run, the larger the data generated becomes.

It's possible that your code base is pretty solid, or that you suspect it may just fail on edge cases that only can be detected through very complex cases. It may not be worth your time to wait hundreds or thousands of tests before the data generated is getting intricate enough to be interesting, and it would be good to be able speed this up.

You can use the resize(Size, Generator) function to force a given size (a positive integer) onto a generator. In prop_collect2(), introduced in *Collecting*, on page 47, we counted the size of binaries, with the following results:

```
56% {0,10}
27% {10,20}
13% {20,30}
3% {30,40}
1% {40,50}
```

By resizing the generator to some arbitrary value, the data size can be increased:

CustomGenerators/erlang/pbt/test/prop_generators.erl
```
prop_resize() ->
    ?FORALL(Bin, resize(150, binary()),  % <= resized here
            collect(to_range(10, byte_size(Bin)), is_binary(Bin))).
```

Elixir translation on page 194

The value 150 was chosen arbitrarily, through trial and error. Running the test again reports new statistics:

```
15% {90,100}
10% {110,120}
9% {80,90}
8% {130,140}
7% {40,50}
7% {50,60}
7% {120,130}
«more statistics»
```

The data sizes are still varied; they are not fixed to exactly 150, but where 80% of the results were 20 bytes of smaller before, the vast majority of them now sits well above 40 bytes in size.

One caveat of using the resize function with static factors is that some of the variability of result sets is possibly lost. It is possible that both smaller or larger sizes would prompt interesting results, and having all of them in a single property would be useful. Another caveat is that relative sizing of various elements becomes cumbersome.

For example, the following property test's generator, in order to be realistic, should create shorter names than biographies, but still biographies larger than what would usually be one sentence:

CustomGenerators/erlang/pbt/test/prop_generators.erl
```
prop_profile1() ->
    ?FORALL(Profile, [{name, resize(10, string())},
                      {age, pos_integer()},
                      {bio, resize(350, string())}],
        begin
            NameLen = to_range(10,length(proplists:get_value(name, Profile))),
            BioLen = to_range(300,length(proplists:get_value(bio, Profile))),
            aggregate([{name, NameLen}, {bio, BioLen}], true)
        end).
```

Elixir translation on page 194

Because size requirements vary between elements of the generator—part of it should be smaller than another one—multiple resize/2 function calls must be made and kept synchronized so that the data ratio is respected. But

because we used static sizes, we've lost PropEr's helpful tendency to try very large or very small values. The only way we could get it back would be by manually modifying all those resize calls to increase or decrease their values in unison. That's annoying.

To avoid having to micromanage all these calls to make sure everything is preserved, you can instead use the ?SIZED(VarName, Expression) macro, which introduces the variable VarName into the scope of Expression, bound to the internal *size* value for the current execution. This *size* value changes with every test, so what we do with the macro is change its scale, rather than replacing it wholesale. Here's the same property using it:

CustomGenerators/erlang/pbt/test/prop_generators.erl

```
prop_profile2() ->
    ?FORALL(Profile, [{name, string()},
                      {age, pos_integer()},
                      {bio, ?SIZED(Size, resize(Size*35, string()))}],
        begin
            NameLen = to_range(10,length(proplists:get_value(name, Profile))),
            BioLen = to_range(300,length(proplists:get_value(bio, Profile))),
            aggregate([{name, NameLen}, {bio, BioLen}], true)
        end).
```

Elixir translation on page 194

In this property, the bio string is specified to be 35 times larger than the current size, which is implicitly the size value for name and age values. Doing this allows us to scale both the name and the biography relative to each other without imposing an anchor in terms of absolute size to the framework. This, in turn, provides some additional flexibility for the framework to do what it wants:

```
$ rebar3 proper -m prop_generators -p prop_profile1,prop_profile2
===> Testing prop_generators:prop_profile1()
«test output»

45% {name,{0,10}}
40% {bio,{0,300}}
10% {bio,{300,600}}
5% {name,{10,20}}

===> Testing prop_generators:prop_profile2()
«test output»

32% {name,{0,10}}
28% {bio,{0,300}}
12% {bio,{300,600}}
10% {name,{10,20}}
7% {name,{20,30}}
4% {bio,{600,900}}
```

```
3% {bio,{900,1200}}
1% {bio,{1200,1500}}
1% {name,{30,40}}
```

As you can see, with the value obtained through the ?SIZE/2 macro applied relatively to a single generator with resize/2, PropEr is able to generate a much wider range of sizes, while we get to preserve the difference in ratio between names and biographies' lengths.

We could even combine approaches to get more variability, but over larger basic sizes if we wanted to. We could for example use an expression like ?SIZED(Size, resize(min(100,Size)*35, string())) to ensure a minimal size without imposing a ceiling on the value. We can see the beginning of our ability to tune each generator individually to our needs.

Transforming Generators

In the course of writing generators, one of the patterns you'll encounter is the need to generate data types that cannot simply be described with just the basic Erlang data structures that default generators support. For example, you may want to generate a first-in-first-out queue of key/value pairs, using the queue[1] module. Just using tuples and lists won't be enough to enforce the internal constraints of the data structure. In some cases, the data structure may be *opaque* (meaning you cannot or should not peek at how it's built, just stick to the interface the module exposes), and then you're just plain out of luck. To solve this problem, PropEr exposes macros that lets you apply arbitrary transformations to data while generating it.

To illustrate the issue, let's take a look at the queue module. If we wanted to have a queue of keys and values stored as tuples, we couldn't safely create it out of default generators without digging inside the implementation and understanding how all the data is handled internally. Instead what we'd need to do is write a property similar to this:

CustomGenerators/erlang/pbt/test/prop_generators.erl

```
prop_queue_naive() ->
    ?FORALL(List, list({term(), term()}),
        begin
            Queue = queue:from_list(List),
            queue:is_queue(Queue)
        end).
```

Elixir translation on page 195

1. http://erlang.org/doc/man/queue.html

Because we can only use queue:from_list/1 on an actual list—and therefore not from within the generator, we call it within the property. The problem with this approach is that even though we want a queue, we generate a list of tuples. Once in the property, we must then convert the list to an actual queue data structure. This splits up the generation of the data type and the conversion between the generator and the property. Any property requiring a queue is stuck using the generator, and then manually adding bits that should belong in it to the property. Even worse, if you wanted to generate a list of queues, the majority of the generators would have to live inside the property rather than within the generators themselves. That's bad abstraction.

A better approach would be to use the ?LET(InstanceOfType, TypeGenerator, Transform) macro apply a transformation to the generated data itself. The macro takes the TypeGenerator and binds it to the InstanceOfType variable. That variable can then be used in the Transform expression as if it were fully evaluated. The evaluation of the final generator is still deferred until later though, which is important because it lets us transform it without preventing it from being composable with others generators. In other words, ?LET lets you accumulate function calls to run on the generator whenever it will be evaluated.

Here's the same queue generator, but with the ?LET/3 macro:

CustomGenerators/erlang/pbt/test/prop_generators.erl

```
prop_queue_nicer() ->
    ?FORALL(Q, queue(),
            queue:is_queue(Q)).

queue() ->
    ?LET(List, list({term(), term()}),
         queue:from_list(List)).
```

Elixir translation on page 195

With this form, there is no more bleeding of parts of generators into properties themselves. The composable aspect also means that a list of generators would be as simple as calling list(queue()). A custom generator of this kind has all the same capabilities as any built-in one.

This covers the need to *transform* a piece of data into another one. Another related piece of functionality you may find yourself needing is preventing some data from being generated altogether, rather than a transformation. How can we prevent a generator from generating data?

Imposing Restrictions

A common trait to all default generators is that they're pretty broad in the data they generate, and from time to time, we'll want to exclude specific counterexamples (the technical word for "input causing a property to fail"). In fact, we already needed to do that when we used the non_empty() generator to remove empty lists or binaries from the generated data set. Such a filter generator can be implemented with the ?SUCHTHAT(InstanceOfType, TypeGenerator, BooleanExp) macro.

The macro works in a similar manner as ?LET/3: the TypeGenerator is bound to InstanceOfType, which can then be used in BooleanExp. One distinction is that BooleanExp needs to be a boolean expression, returning true or false. If the value is true, the data generated is kept and allowed to go through. If the value is false, the data is prevented from being passed to the test; instead, PropEr will try to generate a new piece of data that hopefully satisfies the filter. The non_empty() filter can in fact be implemented as:

```
non_empty(ListOrBinGenerator) ->
    ?SUCHTHAT(L, ListOrBinGenerator, L =/= [] andalso L =/= <<>>).
```

Elixir translation on page 195

Quite simply, if the data generated is an empty list or an empty binary, the generator must try again. Maps or queues are not impacted by this filter, but you could write your own constraints as well:

```
non_empty_map(Gen) ->
    ?SUCHTHAT(G, Gen, G =/= #{}).
```

Elixir translation on page 195

Filtering too hard

There is a limit to the number of times PropEr will retry building a generator; after too many failed attempts, it will give up with an error message such as Error: Couldn't produce an instance that satisfies all strict constraints after 50 tries.

Similarly, generating a list of even or uneven numbers could be done by using ?SUCHTHAT() macros:

```
even() -> ?SUCHTHAT(N, integer(), N rem 2 =:= 0).
uneven() -> ?SUCHTHAT(N, integer(), N rem 2 =/= 0).
```

Elixir translation on page 195

However, in this specific case it may be faster to use a transform to get there by using the ?LET() macro instead of filtering with ?SUCHTHAT():

```
even() -> ?LET(N, integer(), N*2).
uneven() -> ?LET(N, integer(), N*2 + 1).
```

Elixir translation on page 195

Since these transforms can generate correct data on the first try every time, they will be more efficient than using ?SUCHTHAT. Whenever you use a filter, try to see if you could be reworking a restriction into a transformation. Think probabilistically: will you need to filter out a tiny portion of the generator's possible space? A filter's perfect. If you're going to filter out a significant chunk of the potential data, then a transformation may pay for itself in speed.

Another thing to note is that not just guard expressions are accepted in ?SUCHTHAT macros. For example, the following generator looks for ISO Latin1[2] strings by using the io_lib:printable_latin1_list/1 function[3], which will let us restrict down the range of string():

```
latin1_string() ->
    ?SUCHTHAT(S, string(), io_lib:printable_latin1_list(S)).
```

A similar one for unicode—ensuring nothing goes out of range—would be:

```
unicode_string() ->
    ?SUCHTHAT(S, string(), io_lib:printable_unicode_list(S)).
```

With transforms, filters, and resizes, we can get pretty far in terms of retargeting our generators to do what we want. The latin1 generator shows something interesting though. The default string() generator has a large search space, and therefore filtering out the unwanted data can be expensive. On the other hand, most unicode characters cannot be represented within latin1, and transforming the generated strings themselves would also be expensive: how would you map emojis to latin1 characters?

Unicode Generation

If you need valid Unicode strings, it is simpler to use utf8() as a base generator. It will return a properly encoded UTF-8 binary. If you need strings, you can then use a ?LET macro to turn it back with unicode:characters_to_list/1

We can't get what we want (and have it done efficiently) with transforms and filters alone, and for this specific issue, resizing wouldn't be of much help

2. https://en.wikipedia.org/wiki/ISO/IEC_8859-1
3. http://erlang.org/doc/man/io_lib.html#printable_latin1_list-1

either. Instead, we will have to build our own generators while controlling probabilities to make them do what we want.

Changing Probabilities

The last fundamental building block that really gives us control over data generation is having the ability to tweak the probabilities of how data is generated. By default, the generators introduced in *Default Generators*, on page 22 are either going to generate data in a large potential space, like string(), number(), or binary(), or in a rather narrow scope, such as boolean() or range(X,Y). Using ?LET() lets us transform *all* of that data, and ?SUCHTHAT() lets us remove *some* of it, but it's hard to get something in between. When you truly need a custom solution, probabilistic generators can help.

We've seen oneof(ListOfGenerators) already, which helped us gain more repeatable keys in the following generator:

```
key() -> oneof([range(1,10), integer()]).
```

This shows how two distinct generators can be used together to help build and steer things in the direction we want. The oneof(Types) generator is simple and useful, but the most interesting one is frequency(), which will allow you to control and choose the probability of each generator it contains.

Let's take strings as an example, since they were already causing us problems. Just using string() tended to yield a lot of control characters, codepoints that were pretty much anything, and very little in terms of the latin1 or ASCII characters we English readers would be used to. And let's not even think about words or sentences. This can be remediated with frequency/1:

CustomGenerators/erlang/pbt/test/prop_generators.erl

```
text_like() ->
    list(frequency([{80, range($a, $z)},              % letters
                    {10, $\s},                        % whitespace
                    {1,  $\n},                        % linebreak
                    {1, oneof([$., $-, $!, $?, $,])}, % punctuation
                    {1, range($0, $9)}                % numbers
                   ])).
```

Elixir translation on page 195

Using it gives us:

```
1> proper_gen:pick(proper_types:string()).
{ok,[35,15,0,12,2,3,3,1,10,25]}
2> proper_gen:pick(prop_generators:text_like()).
{ok,"rdnpw hxwd"}
3> proper_gen:pick(proper_types:resize(79, prop_generators:text_like())).
```

```
{ok,"vyb hhceqai  m f ejibfiracplkcn gqfvmmbspbt\nn.qbbzwmd"}
```

This generator produces something a lot closer to realistic text than what is obtained through string(). It's no Shakespeare yet and may look more like a Scrabble rack, but with more monkeys hitting the keys randomly, we may get there. In any case, by tweaking the frequency values, specific characters can see their probability raised or lowered as required to properly exercise a property.

This is what would let us ensure that a parser for Comma-Separated Values (CSV)[4] could get a higher frequency of quotation marks, commas, and line breaks to find more parsing-related corner cases, whereas a generator for XML would want to focus on < and > characters, for example.

Using frequency() or oneof() with other custom generators therefore lets you choose and mix and match all kinds of techniques to get as flexible as you need. If you need a list of "mostly sorted data", then you could make a generator like:

```
mostly_sorted() ->
    ?LET(Lists,
         list(frequency([
            {5, sorted_list()},
            {1, list()}
         ])),
         lists:append(Lists)).

sorted_list() ->
    ?LET(L, list(), lists:sort(L)).
```

Elixir translation on page 196

This would give you a bunch of sublists that may or may not be sorted, all concatenated into a big list. This would have been much harder to do with just ?LET(), ?SUCHTHAT(), and ?SIZED().

But even though probabilistic generators are nicer than just default ones, for cases like CSV or XML, more structure would make sense; just random tokens thrown around won't necessarily lead to much, and so we'll need more advanced techniques.

Fancy Custom Generators

You can get pretty far with just the basic techniques, but whenever they can't bring you up to where you want your generators want to be, you'll have to

4. https://en.wikipedia.org/wiki/Comma-separated_values

look into more advanced techniques. But even then, the advanced techniques will still make use of the basic ones, so don't worry, they'll remain useful.

In this section, we'll focus on how we can make use of them and put them in a fancier context through writing our own recursive generators, with some need for laziness. We'll then finish it up by introducing symbolic calls, which help make complex generators more understandable once properties fail.

Recursive Generators

Whenever a piece of data can be represented by a repetitive or well-ordered structure, or when a step-by-step approach can be used to create the data, recursion is our friend. We will see how that works, and even how that can compose with probabilistic generation, although there are some pitfalls there.

Let's say we have a robot, and want to give it directions. We might want to test things such as whether it can go through a room without hitting obstacles, whether it can cover all areas, and so on, but first we need to be able to create and plan an itinerary for it. Let's say our robot works on grid with coordinates, and can go left, right, up, or down. A simple generator might look like:

```
path() -> list(oneof([left, right, up, down])).
```

Elixir translation on page 196

And it would be fine. If we wanted to eliminate things like "going left then right" or "going up then down" where moves cancel each other, a `?LET()` would probably be fine as well: just scan the list, and every time two opposite directions are taken, drop them from the list. But what if we wanted to generate a path such that the robot never crosses a part of its path it has already covered? Doing so with a `?SUCHTHAT` might not be super efficient, and it might also be hard to do with a `?LET()`. However, it's kind of easy to do with recursion.

Tracking if we've been somewhere before would require us to internally track coordinates with `{X,Y}` values:

- our robot always starts at `{0,0}`
- going left means subtracting `1` from the `X` value: `{-1,0}`
- going right means adding `1` to the `X` value: `{+1,0}`
- going up means adding `1` to the `Y` value: `{0,+1}`
- going down means subtracting `1` from the `Y` value: `{0,-1}`

All we need to do then is track coordinates in a map; if a value exists in the map, we finish. Also, in order to put an upper bound on the path length, we'll use a low-probability event of terminating right away.

Our generator for this might look like:

```
path() ->
    %  Current, Acc, VisitedMap     ToIgnore
    path({0,0}, [], #{{0,0} => seen}, []).

path(_Current, Acc, _Seen, [_,_,_,_]) -> %% all directions tried
    Acc; % we give up
path(Current, Acc, Seen, Ignore) ->
    frequency([
        {1, Acc}, % probabilistic stop
        {15, increase_path(Current, Acc, Seen, Ignore)}
    ]).
```

Elixir translation on page 196

So this is an interesting bit of code. The first function is just a wrapper, where path() calls out to a recursive path/4 function. That function takes the current coordinate ({X,Y}), the current path it has built (a list of the form [up, down, left, ...]), the map of visited coordinates, and a list of directions to ignore. These directions are those that have been attempted recently and that resulted in a conflict. If the list contains all four of them, then we know there's no way to go that won't result in a repeated run, and so we give up.

The second function clause uses frequency/1 to ensure that once in a while, we stop. This will prevent a case where we'd just go forwards forever and never ever finish generating data. In the vast majority of cases (with a 15-to-1 probability), we'll try and lengthen the path. The function that does this is defined as follows:

```
increase_path(Current, Acc, Seen, Ignore) ->
    DirectionGen = oneof([left, right, up, down] -- Ignore),
    ?LET(Direction, DirectionGen,
      begin
        NewPos = move(Direction, Current),
        case Seen of
            #{NewPos := _} -> % exists
                path(Current, Acc, Seen, [Direction|Ignore]); % retry
            _ ->
                path(NewPos, [Direction|Acc], Seen#{NewPos => seen}, [])
        end
      end).

move(left, {X,Y}) -> {X-1,Y};
move(right, {X,Y}) -> {X+1,Y};
move(up, {X,Y}) -> {X,Y+1};
move(down, {X,Y}) -> {X,Y-1}.
```

Elixir translation on page 196

Let's decompose this one. First, we have a call to oneof([left, right, up, down] -- Ignore). This generator basically tries to pick a random direction that has not been chosen yet. By default, this Ignore list is [], which means all directions are attempted. The tricky bit is that this call to oneof() returns a generator, not an actual value; PropEr will take care of changing the generator into a value.

If we want to use the value of that generator within the current generator, the ?LET() macro lets us do that. Remember that ?LET() chains up operations to be run once PropEr kind of *actualizes* generators into data, so writing the function as above lets us use the result (Direction) from the generator (DirectionGen) within the macro's transfomed expression (begin ... end).

Within that expression, we apply the position by calling move/2 on the current one, and look it up in the Seen map. If it is found there, it means this direction crosses a path we've taken before, so we retry while ignoring it. If the position has not been visited before, we call path/4 again with the new data.

If you try to run this generator in the shell, it will possibly take a very long time to return, and with some other generators that only probabilistically stop, it may never return. The more branches, the costlier. That's because of the order of evaluation in Erlang. To evaluate frequency(), which is a regular function, its arguments need to be expanded. Since the arguments to frequency() include the generator itself, the generator must be called first, and the process loops deeper and deeper, until memory runs out.

For those specific cases, PropEr provides a ?LAZY() macro, which allows you to defer the evaluation of an expression until it is actually required by the generator. This fixes things:

```
path(_Current, Acc, _Seen, [_,_,_,_]) -> %% all directions tried
    Acc; % we give up
path(Current, Acc, Seen, Ignore) ->
    frequency([
        {1, Acc}, % probabilistic stop
        {15, ?LAZY(increase_path(Current, Acc, Seen, Ignore))}
    ]).
```

Elixir translation on page 197

This macro lets us defer the evaluation of its contents until they are needed, which means that we can now safely use it within alternative clauses. Now the generator should run:

```
$ rebar3 as test shell
«build info»
1> proper_gen:pick(prop_generators:path()).
{ok,[down,right,down,left,left]}
```

```
2> proper_gen:sample(prop_generators:path()).
[right,down,right,up,up,right,up,left]
[]
[down,right,down,left,left,left,left,down,down,left,up,up,left,
 up,left,up,left,up,left,left,up,right,right,right,down,right,
 right,up,up,right,down,down,right,down,down,right,up,up,right,
 right,right,right,down,right,up,right]
[right,right,right,up,right,right,down,left,down]
[left,up,right,up,left,up,up,up]
[down,right,down,left,left,down,down,down,right,down,left,left,
 down,left,up,up,up,right,up,left,left,up,right,up,left,left,up,
 right,up,right,right,down,down,right,up,up,right]
[down,left,down,down,right]
[left,left,down]
[up,left,up,right,right,down,right,right,up,right,right,up,left,
 up,up,right,down,right,down]
[]
[left,left,up,left,down,down,right,down]
ok
```

That's good, but there's a small problem. The caveat with this approach is that probabilities for recursion can mean one of two things:

1. Because the probabilities are fixed, the size of the created data structure will generally be unchanging and always average to a ratio in line with the defined probabilities (here, 15-to-1 probabilities means we may have paths roughly 15 steps long on average)
2. Because there are probabilities at all, there is a lingering chance some data structures will be enormous, possibly large enough to crash our program.

Both of them are undesirable on their own, but we may end up with tests that get both. In some cases, the potential variation for very large or very small structures will be desirable, but not always. We can improve the generator by forcing it to be a bit more deterministic in its size by writing recursive functions that are much more similar to what we usually write in regular Erlang or Elixir code:

```
path(0, _Current, Acc, _Seen, _Ignore) -> % directions limit
    Acc; % max depth reached
path(_Max, _Current, Acc, _Seen, [_,_,_,_]) -> % all directions tried
    Acc; % we give up
path(Max, Current, Acc, Seen, Ignore) ->
    increase_path(Max, Current, Acc, Seen, Ignore).

increase_path(Max, Current, Acc, Seen, Ignore) ->
    DirectionGen = oneof([left, right, up, down] -- Ignore),
    ?LET(Direction, DirectionGen,
```

```
    begin
      NewPos = move(Direction, Current),
      case Seen of
          #{NewPos := _} -> % exists
              path(Max, Current, Acc, Seen, [Direction|Ignore]); % retry
          _ ->
              path(Max-1, NewPos, [Direction|Acc],
                   Seen#{NewPos => seen}, [])
      end
    end).
```

Elixir translation on page 197

The only thing done here is adding a counter (`Max`) to each function clause, which halts the execution when it reaches `0`. You'll see that this is pretty much how you'd write any recursive function to generate a list of a known length in Erlang or Elixir. In fact, because we have only distinct function clauses with no `frequency()` usage anymore, the `?LAZY()` macro is no longer necessary.

All we need to add is a wrapper to seed the size parameter. That's where the `?SIZED` macro can be of use:

```
path() ->
    ?SIZED(Size,
           %   Max, Current, Acc,  VisitedMap     ToIgnore
           path(Size, {0,0}, [], #{{0,0} => seen}, [])).
```

Elixir translation on page 197

And just like that, we got pretty good recursion with a great way to dynamically resize recursive data structures. The `Size` parameter will grow along with test execution, and if you want it to be growing faster than what PropEr gives, just multiply it. This will give us much more interesting path variations than probabilistic generators.

```
$ rebar3 as test shell
«build info»
1> proper_gen:sample(prop_generators:path()).
[down,down,down,down,down,left,left,down,down,down]
[down,left,left,down,left,down,left,down,left,up,left]
[down,down,down,right,right,up,left,up,up,up,right,down]
[down,left,down,left,down,down,left,down,left,up,up,right,up]
[down,right,right,right,down,left,left,down,left,up,left,up,left,up]
[down,right,up,right,up,up,left,left,left,up,left,left,up,left,down]
[right,right,up,up,left,left,left,up,up,right,up,up,up,up,right,down]
[down,left,down,right,right,up,right,right,down,right,right,right,down,
 down,left,left,left]
[left,left,down,left,left,up,left,up,left,down,left,left,down,right,down,
 right,up,right]
```

```
[down,left,left,up,right,up,up,up,up,up,left,left,left,up,right,right,up,
 right,right]
[down,right,right,up,left,up,right,up,up,right,up,left,up,left,down,down,
 down,left,left,left]
ok
```

In comparison to the previous version's samples, this one has a path length capped and guided by size, meaning the framework can do a better job at scaling the list size up and down and the generator will never end up in an infinite loop. This also prevents us from getting a fixed ratio of empty lists (if there is a 5% chance to stop at any iteration, you should likely expect around 5% of empty lists as well). If you're interested in testing the diversity of paths, the latest approach is probably both faster and likelier to play well with PropEr.

The same approach can be used to generate any recursive structure: trees of a depth proportional to the size (or with as many elements as the size), grammar rules encoding more complex variations, or even sequences of transitions in a state machine, which would require many nested and mutually recursive generators.

The technique may also sound interesting to build side-effectful generators, such as those populating ETS tables or even writing files to disk, but those would be a bit more problematic to debug since we can't easily see how they got to be in their end state.

Symbolic Calls

Some of the data structures or state that we need to generate for tests can be quite opaque and difficult to decipher. Think of debugging a binary protocol by looking at the individual bytes once the whole thing is encoded, or creating a process and sending it a bunch of messages to prime its state. The output of a failing property will be a bunch of hard-to-read bits and bytes, or a term like `<0.213.0>`, which frankly don't help a whole lot no matter how much shrinking you may apply to them. The solution to this problem is a special category of generators built from *symbolic calls*.

A symbolic call is just a special notation for function calls so that they can be represented as data in generators. Rather than executing the operation straight away, the calls are built up as a data structure. Once the generator materializes, they get executed at once. The notation for them is `{call, Module, Function, ArgumentList}`—a format supported by QuickCheck and Triq as well as PropEr. There is also `{$call, Module, Function, ArgumentList}` as a format (named *automated symbolic call*), which is similar, but a bit friendlier in practice:

Function Call	Symbolic Call	Automated Symbolic Call (PropEr only)
sets:new()	{call, sets, new, []}	{'$call', sets, new, []}
queue:join(Q1, Q2)	{call, queue, join, [Q1, Q2]}	{'$call', queue, join, [Q1, Q2]}
lists:sort([1,2,3])	{call, lists, sort, [[1,2,3]]}	{'$call', lists, sort, [[1,2,3]]}
local(Arg) (if exported)	{call, ?MODULE, local, [Arg]}	{'$call', ?MODULE, local, [Arg]}

Using either formats of symbolic calls can make things simpler when looking at shrunken results. Let's try with the Erlang dict data structure, which can be quite opaque if you don't know how it's implemented. This example will also show the difference between symbolic calls and automated symbolic calls.

CustomGenerators/erlang/pbt/test/prop_generators.erl

```
dict_gen() ->
    ?LET(X, list({integer(),integer()}), dict:from_list(X)).

dict_symb() ->
    ?SIZED(Size, dict_symb(Size, {call, dict, new, []})).

dict_symb(0, Dict) ->
    Dict;
dict_symb(N, Dict) ->
    dict_symb(N-1, {call, dict, store, [integer(), integer(), Dict]}).

dict_autosymb() ->
    ?SIZED(Size, dict_autosymb(Size, {'$call', dict, new, []})).

dict_autosymb(0, Dict) ->
    Dict;
dict_autosymb(N, Dict) ->
    dict_autosymb(N-1, {'$call', dict, store, [integer(), integer(), Dict]}).
```

Elixir translation on page 198

This specifies three generators: one with the normal function calls, one with symbolic calls, and one with automated symbolic calls. Here are three matching properties that will pretty much always fail:

CustomGenerators/erlang/pbt/test/prop_generators.erl

```
prop_dict_gen() ->
    ?FORALL(D, dict_gen(), dict:size(D) < 5).

prop_dict_symb() ->
    ?FORALL(DSymb, dict_symb(), dict:size(eval(DSymb)) < 5).

prop_dict_autosymb() ->
    ?FORALL(D, dict_autosymb(), dict:size(D) < 5).
```

Elixir translation on page 198

See how the symbolic call form has to call eval(SymbolicGenerator) in order to use the code within the property. This is a downside where the implementation details of the generator leak into the property. The automated symbolic call by comparison is automatically evaluated by PropEr before being passed to the property. Let's see how they fail:

```
===> Testing prop_generators:prop_dict_gen()
..............!
Failed: After 16 test(s).
{dict,5,16,16,8,80,48,{[],[],[],[],[],[],[],[],[],[],[],
 [],[],[],[],[]},{{[],[[3|-6]],[],[],[],[[-1|-5]],
[[-14|10],[2|3]],[[5|1]],[],[],[],[],[],[],[],[]}}}
«more output»
===> Testing prop_generators:prop_dict_symb()
.............!
Failed: After 15 test(s).
{call,dict,store,[-6,8,
  {call,dict,store,[46,-13,
    {call,dict,store,[2,-2,
      {call,dict,store,[22,-2,
        {call,dict,store,[-12,-2,
          {call,dict,new,[]}]}]}]}]}]}]}
«more output»
===> Testing prop_generators:prop_dict_autosymb()
............!
Failed: After 13 test(s).
{'$call',dict,store,[7,2,
  {'$call',dict,store,[-1,2,
    {'$call',dict,store,[1,18,
      {'$call',dict,store,[-9,0,
        {'$call',dict,store,[-10,3,
          {'$call',dict,new,[]}]}]}]}]}]}
«more output»
```

As you can see, the first property (prop_dict_gen()) returns a weird data structure. That one is a bit hard to debug since it's hard to see what inputs we can give a dictionary to yield that result. Both forms of symbolic calls let us see the keys and values inserted to yield a failing dictionary.

With the help of symbolic calls, generated terms that are too obscure (whether through their complex size or because side-effects hide the way the state was obtained) can be made simpler to debug. Even if we were to use a generator to configure a remote server into a given state, symbolic calls would let us know which sequence of calls may have led up to a failure.

Wrapping Up

You now have all the tools you need to make the fanciest of all generators. You're ready to make some that have a very wide-spectrum approach to fuzz your systems, down to very accurate ones to exercise very specific invariants your code should respect. All of these methods can be mixed together if you desire so. Nothing keeps you from using symbolic calls created through complex recursive generators that filter with `?SUCHTHAT` and transform with `?LET`, with some terms created probabilistically with a dynamic size.

Going overboard is not the best of ideas though. It's not because you have a fancy set of hammers that you have to crush all the nails with them. We've got strategies to come up with properties and to build generators, but what you've got to go through next is finding strategies to make proper use of properties within a project. The next chapter will cover responsible testing, how to know when enough is enough, or where a little property-testing magic can go a long way.

Exercises

Question 1

Which functions can be used to check the distribution of generated entries in a test run?

Question 2

Which macro can be used to apply regular Erlang function to a generator to modify it?

Question 3

When and why would you use the `?LAZY` macro?

Question 4

The following probabilistic generator creates trees that are not necessarily balanced:

CustomGenerators/erlang/pbt/test/prop_exercises.erl

```
%% The tree generates a data type that represents the following types:
-type tree() :: tree(term()).
-type tree(T) :: {node,
                  Value :: T,
                  Left  :: tree(T) | undefined,
                  Right :: tree(T) | undefined}.

tree() ->
```

```
    tree(term()).

tree(Type) ->
    frequency([
        {1, {node, Type, tree(Type), undefined}},
        {1, {node, Type, undefined, tree(Type)}},
        {5, {node, Type, tree(Type), tree(Type)}}
    ]).
```

Elixir translation on page 198

Make sure it can consistently terminate and generate trees of an interesting size. Using the ?SIZED macro may be an advantage for better complexity scaling.

Question 5

The following generators allow you to generate stamps of the format {Hour, Minute, Second}:

CustomGenerators/erlang/pbt/test/prop_exercises.erl

```
stamp() -> {hour(), min(), sec()}.
hour() -> choose(0,23).
min()  -> choose(0,59).
sec()  -> choose(0,59).
```

The following are pairs of modified or restricted timestamps. Compare their implementations, and explain which of each pair is the most appropriate.

CustomGenerators/erlang/pbt/test/prop_exercises.erl

```
%% Return hours in the morning
am_stamp1() ->
    ?SUCHTHAT({H,_,_}, stamp(), H < 12).
am_stamp2() ->
    ?LET({H,M,S}, stamp(), {H rem 12, M, S}).
```

CustomGenerators/erlang/pbt/test/prop_exercises.erl

```
%% Return two ordered timestamps
stamps1() ->
    ?SUCHTHAT({S1, S2}, {stamp(), stamp()}, S1 =< S2).
stamps2() ->
    ?LET({S1, S2}, {stamp(), stamp()}, {min(S1,S2), max(S1,S2)}).
```

CustomGenerators/erlang/pbt/test/prop_exercises.erl

```
%% Return any time that does not overlap standup meetings
no_standup1() ->
    ?SUCHTHAT({H,M,_}, stamp(), H =/= 9 orelse M > 10).
no_standup2() ->
    ?LET({H,M,S}, stamp(),
         case H of
            9 when M =< 10 -> {8, M, S};
            _ -> {H,M,S}
         end).
```

Elixir translation on page 199

Question 6

Write a symbolic generator that would create a file with various bytes in it, and return the active file handler (or file descriptor) for it. When a property fails, the user should be able to see the failing output without having to peek inside files.

You may need wrappers for file:open/2 and file:write/2 to ensure easier composability of these functions. Here's an example of those:

CustomGenerators/erlang/pbt/test/prop_exercises.erl

```
file_open(Name, Opts) ->
    {ok, Fd} = file:open(Name, Opts),
    %% ensure the file is refreshed on each test run
    file:truncate(Fd),
    Fd.

file_write(Fd, Data) ->
    ok = file:write(Fd, Data),
    Fd.
```

Elixir translation on page 199

Part II

Stateless Properties in Practice

Let's get practical, and see how properties can be used in actual projects. We'll also cover some more advanced topics in here.

CHAPTER 5

Responsible Testing

A common sight in a team that just started learning property-based testing is that almost all the new tests written are properties, even when it doesn't really make sense to do so. The pendulum swings too far and too fast, and the project suffers for it. In this chapter, we will see multiple example situations where property tests may or may not be appropriate, how they could be enhanced through careful addition of traditional tests in a regular project or other external tools like Dialyzer, or where they may just be plain inappropriate. We'll try to get from knowing *how* properties work—you can write generators, measure their efficiency, and come up with properties through various strategies—to having good feel of *when* properties work best.

We'll take a practical project based on the birthday greeting kata[1]. This exercise asks us to organize unit tests for a little application in a manner such that as few tests as possible will need to be modified or trashed when implementation details or requirements change. It's an exercise usually meant for object-oriented languages, but since we're using Elixir and Erlang here, we'll also see how to approach the design in a functional manner.

The exercise will contain parts having to do with data storage, text parsing, data manipulation, and templating. We will *only focus on unit tests*, and we'll leave more side-effectful integration and system tests to later chapters such as Chapter 9, *Stateful Properties*, on page 147 and *the (as yet) unwritten Chapter 10, Case Study: Bookstore,* . For now, we'll see how to choose between traditional example-based unit tests and property tests, how they could be used together at once to complement each other, and we'll get a bit of a feel to figure out how to add properties to our workflow in a project started from scratch.

1. http://matteo.vaccari.name/blog/archives/154

The Specification

We're going to write a program that will first load a set of employee records from a flat file, and then send a greeting email to each employee whose birthday is today. The flat file containing employee records is given to us and looks a bit like a *comma-separated values* (CSV)[2] file. We're given a sample that looks like:

```
last_name, first_name, date_of_birth, email
Doe, John, 1982/10/08, john.doe@foobar.com
Ann, Mary, 1975/09/11, mary.ann@foobar.com
```

The e-mail sent on an employee's birthday should contain text like:

> Subject: Happy birthday!
>
> Happy birthday, dear John!

On its own, this is straightforward. The challenge comes from the additional constraints given:

- The tests written should be *unit tests*, meaning none of the tests should talk to a database, touch the filesystem, interact with the network, or toy with the environment (config). Tests that do any of these are qualified as integration tests and are out of scope. This actually takes restraint to do!
- The CSV format won't be kept forever. Eventually, a database or web service should be used to fetch the employee records, and similarly for the e-mail sending. The tests should be written to require as few modifications as possible whenever these implementation details change.

Let's see how we can approach the design of this system, one step at a time.

Thinking About Program Structure

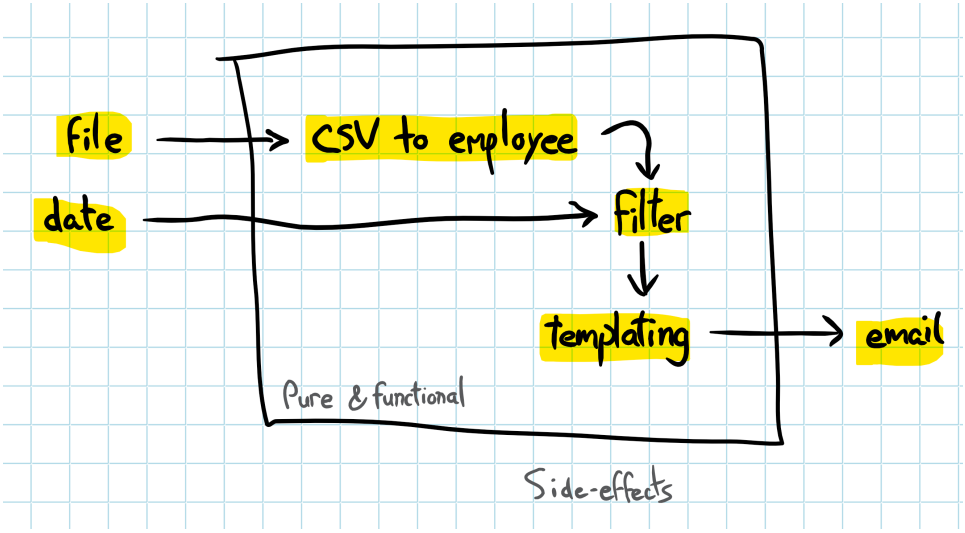
The single most helpful thing we can do to make our testing experience easier is to design a system that is inherently testable. To do this well, thinking in terms of what are the observable effects we want to test against is crucial. In object-oriented code, the traditional approach asks us to limit ourselves to specific objects, and to observe their behavior and side-effects carefully. To prevent the test scope from being too large, mocking[3] and dependency injection[4] represent a good way to wall things off.

2. https://en.wikipedia.org/wiki/Comma-separated_values
3. https://en.wikipedia.org/wiki/Mock_object
4. https://en.wikipedia.org/wiki/Dependency_injection

In the case of Erlang and Elixir, these practices can and will make sense when testing specific actors, although this may bring us closer to integration testing. For most cases though, since we're doing functional programming, the lesson taught by pure languages like Haskell is that side-effects can be grouped together at one end of the system, and we can keep the rest of the code as pure as possible. Haskell enforces that mechanism, but nothing prevents us from doing it by hand here. Let's see how we could classify the different actions in our program:

Side-Effectful	Functional
Reading a flat file	Converting CSV data to employee records
Finding today's date	Searching for employees based on a date
Sending an e-mail	Formatting an e-mail as a string

Things are a bit clearer now. Everything on the left column will go in integration tests (not covered in this chapter), and everything on the right column can go in unit tests and is in scope. In fact, if we use this classification to design the entire system, we can have something fairly easy to test, as shown in the following figure:



Everything in the box should be pure and functional, everything outside of it will provide side-effects. Some main() function will have the responsibility of tying both universes together. With this approach, the program becomes a sequence of transformations carried over known (but configurable) bits of data. This structure is not cast in stone, but should be a fairly good guideline that ensures strict decoupling of components, easy testing, and simple refactoring.

With this top-level view in place, we can start working on specific components that can then be integrated together after the fact. The building blocks to be written are:

1. CSV parsing of terms into maps
2. Filtering of objects based on date or time (if the program used an SQL database or a service to handle search, we wouldn't have to write this)
3. Putting the filtering and CSV parsing together into an employee module providing a well-isolated interface
4. Templating of the email and subject to be sent, based on the employee data

Although sending e-mails and reading from disk are side-effectful items that are out of scope, we'll add a fifth step where we tie up all the parts together with a top-level module (exposing the main() function mentioned earlier). For this, we can use a simple escript project format, which lets us run simple programs easily:

```
$ rebar3 new escript bday
===> Writing bday/src/bday.erl
===> Writing bday/src/bday.app.src
===> Writing bday/rebar.config
===> Writing bday/.gitignore
===> Writing bday/LICENSE
===> Writing bday/README.md
```

And then edit the configuration to add PropEr to the project:

ResponsibleTesting/erlang/bday/rebar.config

```
%% the plugin itself
{plugins, [rebar3_proper]}.

{escript_incl_apps, [bday]}.
{escript_main_app, bday}.
{escript_name, bday}.

%% The PropEr dependency is still required to compile the test cases,
%% but only as a test dependency
{profiles,
    [{test, [
        {erl_opts, [nowarn_export_all]},
        {deps, [proper, recon]}
    ]}
]}.
```

Given Elixir has mix run -e Mod.function as a command that can run an arbitrary function, a regular project can be used there instead. Just make sure to reconfigure the mix file in it the same way we've done in *Elixir*, on page 12:

```
$ mix new bday
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
«more build output»
```

Including the proper mix file modifications:

```
# Run "mix help deps" to learn about dependencies.
defp deps do
  [
    {:propcheck, "~> 1.0", only: :test}
  ]
end
```

With this in place, we can start working on individual functional parts.

CSV Parsing

The first part of the program we'll work on is the one handling CSV conversion. There's no specific order which is better than another one in this case, and starting with CSV instead of filtering or e-mail rendering is entirely arbitrary. In this section, we'll explore how to come up with the right type of properties for encoding and decoding CSV, and how to get decent generators for that task. We'll also see how regular example-based unit tests can be used to strengthen our properties, and see how each of them fares compared to the other.

CSV is a loose format that nobody really implements the same way. It's really a big mess, even though RFC 4180[5] tries to provide a simple specification:

- Each record is on a separate line, separated by *CRLF* (a \r followed by a \n)
- The last record of the file may or may not have a CRLF after it (it is optional)
- The first line of the file may be a header line, ended with a CRLF. In this case, the problem description includes a header, which will be assumed to always be there
- Commas go between fields of a records

5. https://tools.ietf.org/html/rfc4180

- Any spaces are considered to be part of the record (the example in the problem description does not respect that, as it adds a space after each comma even though it's clearly not part of the record)
- Double-quotes (") can be used to wrap a given record
- fields that contain line-breaks (CRLF), double-quotes, or commas must be wrapped in double quotes
- All records in a document contain the same number of fields
- A double-quote within a double-quoted field can be escaped by preceding it with another double-quote ("a""b" means a"b)
- field values or header names can be empty
- Valid characters for records include only:

```
! #$%&'()*+-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`
abcdefghijklmnopqrstuvwxyz{|}~
```

Which means the official CSV specs won't let us have employees whose names don't fit that pattern, but if you feel like doing so, you can always extend the tests later and improve things. For now though, we will implement this specification, and as far as our program is concerned, whatever we'll find in the CSV file will be treated as correct.

For example, if a row contains `a, b, c`, we will consider the three values to be `"a"`, `" b"`, and `" c"` with the leading spaces, and patch them up in our program, rather than modifying the CSV parser we'll write. This is because, in the long run, it will be simpler to reason about our system if all independent components are well-defined reusable units, and we instead only need to reason about adapters to glue them together. Having business-specific code and work-arounds injected through all layers of the codebase is usually a good way to write unmaintainable systems.

Out of the approaches we've seen in Chapter 3, *Thinking In Properties*, on page 31, we could try the following:

- Modeling: make a simpler less efficient version of CSV parsing and compare it to the real one
- Generalizing example tests: a standard unit test would be dumping data, then reading it, and making sure it matches expectations; we need to generalize this so one property can be equivalent to all examples
- Invariants: finding a set of rules that put together to represent CSV operations well enough
- Symmetric properties: serialize and unserialize the data, ensuring results are the same

The latter technique is the most interesting one for parsers and serializers, since we need encoded data to validate decoding, and that decoding is required to make sure encoding works well. Both sides will need to agree and be tested together no matter what. Plugging both into a single property tends to be ideal. All we need after that is to *anchor* the property with either a few traditional unit tests or simpler properties to make sure expectations are met.

Let's start by writing tests first, so we can think of properties before writing the code. Since we'll do an encoding/decoding sequence, generating Erlang terms that are encodable in CSV should be the first step. CSV contains rows of text records separated by commas. We'll start by writing generators for the text records themselves, and assemble them together later. We'll currently stay with the simplest CSV encoding possible: everything is a string. How to handle integers, dates, and so on, tends to be application-specific.

Because CSV is a text-based format, it contains some escapable sequences, which turn out to always be problematic no matter what format you're handling. In CSV, as we've seen in the specification, escape sequences are done through wrapping strings in double-quotes ("), with some special cases for escaping double-quotes themselves. For now, let's not worry about it besides making sure the case is well-represented in our data generators:

ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
field() -> oneof([unquoted_text(), quotable_text()]).

unquoted_text() -> list(elements(textdata())).

quotable_text() -> list(elements([$\r, $\n, $", $,] ++ textdata())).

textdata() ->
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"
    ":;<=>?@ !#$%&'()*+-./[\\]^_`{|}~".
```

Elixir translation on page 200

The field() generator depends on two other generators: unquoted_text() and quotable_text(). The former will be used to generate Erlang data that will require no known escape sequence in it once converted, whereas the latter will be used to generate sequences that may possibly require escaping (the four escapable characters are only present in this one). Both generators rely on textdata(), which contains all the valid characters allowed by the specification.

You'll note that we've put an Erlang string for textdata() with alphanumeric characters coming first, and that we pass it to list(elements()). This approach will randomly pick characters from textdata() to build a string, but what's interesting is what will happen when one of our tests fail. Because elements() shrinks towards the first elements of the list we pass to it, PropEr will try to

generate counter-examples that are more easily human-readable when possible by limiting the number of special characters in there. Rather than generating `{#$%a~`, it will maybe try to generate `ABFe#c` once a test fails.

We can now put these records together. A CSV file will have two types of rows: a header on the first line, and then data entries in the following ones. In any CSV document, we expect the number of columns to be the same on all rows:

ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
header(Size) -> vector(Size, name()).

record(Size) -> vector(Size, field()).

name() -> field().
```

Elixir translation on page 200

Those generators basically generate the same types of strings for both headers and rows, with a known fixed length as an argument. `name()` is defined as `field()` because they have the same requirements specification-wise, but it's useful to give each generator a name according to its purpose: if we end up modifying or changing requirements on one of them, we can do so with minimal changes. We can then assemble all of that jolly stuff together into one list of maps that contain all the data we need:

```
csv_source() ->
    ?LET(Size, pos_integer(),
         ?LET(Keys, header(Size),
              list(entry(Size, Keys)))).

entry(Size, Keys) ->
    ?LET(Vals, record(Size),
         maps:from_list(lists:zip(Keys, Vals))).
```

Elixir translation on page 200

The `csv_source()` generator picks up a `Size` value that represents how many entries will be in each row. By putting it in a `?LET` macro, we make sure that whatever the expression that uses `Size` is, it uses a discrete value, and not the generator itself. This will allow us to use `Size` multiple times safely with always the same value in the second `?LET` macro. That second macro generates one set of headers (the keys of every map), and then uses them to create a list of entries.

The entries themselves are specified by the `entry/2` generator, which creates a list of record values, and pairs them up with the keys from `csv_source()` into a map. This generates values such as:

```
$ rebar3 as test shell
```

```
«build output»
1> proper_gen:pick(prop_csv:csv_source()).
{ok,[#{[] => "z","&_f" => "t,:S","cH^*M" => "{6Z#"},
     #{[] => "kS3>","&_f" => "/","cH^*M" => "eK"},
     #{[] => "~","&_f" => [],"cH^*M" => "Bk#?X7h"}]}
2> proper_gen:pick(prop_csv:csv_source()).
{ok,[#{"D" => "\nQNUO","D4D" => "!E$0; )KL",
       "R\r~P{qC-" => "4L(Q4-N9","T6FAGuhf" => "wSP4jONE3Q"},
     #{"D" => "!Y7H\rQ?I7\r","D4D" => [],
       "R\r~P{qC-" => "}66W2I9+?R","T6FAGuhf" => "pF8/C"},
     #{"D" => [],"D4D" => "?'_6","R\r~P{qC-" => "j|Q",
       "T6FAGuhf" => "f$s7=sFx2>"},
     #{"D" => "e;ho1\njn!2","D4D" => ".8B{k|+|}",
       "R\r~P{qC-" => "V","T6FAGuhf" => "a\"/J\rfE#$"},
«more maps»
```

As you can see, all the maps for a given batch share the same keys, but have varying values. Those are ready to be encoded and passed to our property:

ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
prop_roundtrip() ->
    ?FORALL(Maps, csv_source(),
            Maps =:= bday_csv:decode(bday_csv:encode(Maps))).
```

Elixir translation on page 201

Running it at this point would be an instant failure since we haven't written the code to go with it. Since this chapter is about tests far more than how to implement a CSV parser, we'll go over the latter rather quickly. Here's an implementation that takes about a hundred lines:

ResponsibleTesting/erlang/bday/src/bday_csv.erl

```
-module(bday_csv).
-export([encode/1, decode/1]).

%% @doc Take a list of maps with the same keys and transform them
%% into a string that is valid CSV, with a header.
-spec encode([map()]) -> string().
encode([]) -> "";
encode(Maps) ->
    Keys = lists:join(",", [escape(Name) || Name <- maps:keys(hd(Maps))]),
    Vals = [lists:join(",", [escape(Field) || Field <- maps:values(Map)])
            || Map <- Maps],
    lists:flatten([Keys, "\r\n", lists:join("\r\n", Vals)]).

%% @doc Take a string that represents a valid CSV data dump
%% and turn it into a list of maps with the header entries as keys
-spec decode(string()) -> list(map()).
decode("") -> [];
decode(CSV) ->
    {Headers, Rest} = decode_header(CSV, []),
    Rows = decode_rows(Rest),
```

```
    [maps:from_list(lists:zip(Headers, Row)) || Row <- Rows].
```

First, there's the public interface with two functions: encode/1 and decode/1. The functions are fairly straightforward, delegating the more complex operations to private helper functions. Let's start by looking at those helping with encoding:

```
%%%%%%%%%%%%%%%
%%% PRIVATE %%%
%%%%%%%%%%%%%%%

%% @private return a possibly escaped (if necessary) field or name
-spec escape(string()) -> string().
escape(Field) ->
    case escapable(Field) of
        true -> "\"" ++ do_escape(Field) ++ "\"";
        false -> Field
    end.

%% @private checks whether a string for a field or name needs escaping
-spec escapable(string()) -> boolean().
escapable(String) ->
    lists:any(fun(Char) -> lists:member(Char, [$",$,,$\r,$\n]) end, String).

%% @private replace escapable characters (only `"') in CSV.
%% The surrounding double-quotes are not added; caller must add them.
-spec do_escape(string()) -> string().
do_escape([]) -> [];
do_escape([$"|Str]) -> [$", $" | do_escape(Str)];
do_escape([Char|Rest]) -> [Char | do_escape(Rest)].
```

If a string is judged to need escaping (according to escapable/1), then the string is wrapped in double-quotes (") and all double-quotes inside of it are escaped with another double-quote. With this, encoding is covered. Next there's decoding's private functions:

```
%% @private Decode the entire header line, returning all names in order
-spec decode_header(string(), [string()]) -> {[string()], string()}.
decode_header(String, Acc) ->
    case decode_name(String) of
        {ok, Name, Rest} -> decode_header(Rest, [Name | Acc]);
        {done, Name, Rest} -> {[Name | Acc], Rest}
    end.

%% @private Decode all rows into a list.
-spec decode_rows(string()) -> [[string()]].
decode_rows(String) ->
    case decode_row(String, []) of
        {Row, ""} -> [Row];
        {Row, Rest} -> [Row | decode_rows(Rest)]
    end.
```

```
%% @private Decode an entire row, with all values in order
-spec decode_row(string(), [string()]) -> {[string()], string()}.
decode_row(String, Acc) ->
    case decode_field(String) of
        {ok, Field, Rest} -> decode_row(Rest, [Field | Acc]);
        {done, Field, Rest} -> {[Field | Acc], Rest}
    end.

%% @private Decode a name; redirects to decoding quoted or unquoted text
-spec decode_name(string()) -> {ok|done, string(), string()}.
decode_name([$" | Rest]) -> decode_quoted(Rest);
decode_name(String) -> decode_unquoted(String).

%% @private Decode a field; redirects to decoding quoted or unquoted text
-spec decode_field(string()) -> {ok|done, string(), string()}.
decode_field([$" | Rest]) -> decode_quoted(Rest);
decode_field(String) -> decode_unquoted(String).
```

Decoding is done by fetching headers, then fetching all rows. A header line is parsed by reading each column name one at a time, and a row is parsed by reading each field one at a time. At the end you can see that both fields and names are actually implemented as quoted or unquoted strings:

```
%% @private Decode a quoted string
-spec decode_quoted(string()) -> {ok|done, string(), string()}.
decode_quoted(String) -> decode_quoted(String, []).

%% @private Decode a quoted string
-spec decode_quoted(string(), [char()]) -> {ok|done, string(), string()}.
decode_quoted([$"], Acc) -> {done, lists:reverse(Acc), ""};
decode_quoted([$",$\r,$\n | Rest], Acc) -> {done, lists:reverse(Acc), Rest};
decode_quoted([$",$, | Rest], Acc) -> {ok, lists:reverse(Acc), Rest};
decode_quoted([$",$" | Rest], Acc) -> decode_quoted(Rest, [$" | Acc]);
decode_quoted([Char | Rest], Acc) -> decode_quoted(Rest, [Char | Acc]).

%% @private Decode an unquoted string
-spec decode_unquoted(string()) -> {ok|done, string(), string()}.
decode_unquoted(String) -> decode_unquoted(String, []).

%% @private Decode an unquoted string
-spec decode_unquoted(string(), [char()]) -> {ok|done, string(), string()}.
decode_unquoted([], Acc) -> {done, lists:reverse(Acc), ""};
decode_unquoted([$\r,$\n | Rest], Acc) -> {done, lists:reverse(Acc), Rest};
decode_unquoted([$, | Rest], Acc) -> {ok, lists:reverse(Acc), Rest};
decode_unquoted([Char | Rest], Acc) -> decode_unquoted(Rest, [Char | Acc]).
```

Elixir translation on page 201

Both functions to read quoted or unquoted strings mostly work the same, except quoted ones have specific rules about unescaping content baked in. And with this, our CSV handling is complete.

The code was developed against the properties by running the tests multiple times and refining the implementation iteratively. For brevity, we'll skip all the failed attempts that did some dirty odd parsing, except for one failing implementation that's particularly interesting since it had a failure against the following input:

```
\r\na
```

This is technically a valid CSV file with a single column, for which the empty name "" is chosen (commas only split values, so a single \r\n means a 0-length string as a value on that line), and with a single value "a". The expected output from decoding this is [#{"" ⇒ "a"}]. The first version of the parser had no way to cope with such cases, since I couldn't imagine them either. The parser shown previously is handling such cases, but the digging and rewriting has been skipped for brevity.

If you run the property over the previous (correct) implementation, you'll find it still fails on this tricky test:

```
bday_csv:encode([#{""=>""},#{""=>""}]) => "\r\n\r\n"
bday_csv:decode("\r\n\r\n")            => [#{"" => ""}]
```

This is an ambiguity embedded directly in the CSV specification. Because a trailing \r\n is acceptable, it is impossible to know whether there is an empty trailing line or not in the case of 1-column data sets. Above one column, at least one comma (,) is going to be on the line. At one column, there is no way to know.

Under 50 lines of tests were enough to discover inconsistencies in RFC 4180 itself, inconsistencies that cannot be reconciled or fixed in our program. Instead, we'll have to relax the property, making sure we don't cover that case by changing csv_source() and adding +1 to every Size value we generate. That way, we shift the range for columns from 1..N to 2..(N+1), ensuring we always have 2 or more columns in generated data.

ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
csv_source() ->
    ?LET(Size, pos_integer(),
         ?LET(Keys, header(Size+1),
              list(entry(Size+1, Keys)))).
```

Elixir translation on page 202

After this change, the property works fine. For good measure, we should add a unit test representing the known unavoidable bug to the same test suite, documenting known behavior:

```
-module(prop_csv).
-include_lib("proper/include/proper.hrl").
-include_lib("eunit/include/eunit.hrl").
-compile(export_all).

«existing code»

%%%%%%%%%%%%%
%%% EUnit %%%
%%%%%%%%%%%%%

%% @doc One-column CSV files are inherently ambiguous due to
%% trailing CRLF in RFC 4180. This bug is expected
one_column_bug_test() ->
    ?assertEqual("\r\n\r\n", bday_csv:encode([#{""=>""},#{""=>""}])),
    ?assertEqual([#{"" => ""}], bday_csv:decode("\r\n\r\n")).
```

Elixir translation on page 203

The suite can be run with rebar3 eunit as well as rebar3 proper. Using prop_ as a prefix to both the module and properties lets the proper plugin detect what it needs. For Eunit, the _tests suffix for functions lets it do the proper detection. If you also wanted to use the common test framework in Erlang, the _SUITE suffix should be added to the module.

There is a last gotcha implicit to the implementation of our CSV parser: since it uses maps, duplicate column names are not tolerated. Since our CSV files have to be used to represent a database, it is probably a fine assumption to make about the data set that column names are all unique. All in all, we're probably good ignoring duplicate columns and single-columns CSV files since it's unlikely database tables would be that way either, but it's not fully CSV-compliant. This gotcha was discovered by adding good old samples from the RFC into the EUnit test suite:

ResponsibleTesting/erlang/bday/test/prop_csv.erl

```
rfc_record_per_line_test() ->
    ?assertEqual([#{"aaa" => "zzz", "bbb" => "yyy", "ccc" => "xxx"}],
                 bday_csv:decode("aaa,bbb,ccc\r\nzzz,yyy,xxx\r\n")).

rfc_optional_trailing_crlf_test() ->
    ?assertEqual([#{"aaa" => "zzz", "bbb" => "yyy", "ccc" => "xxx"}],
                 bday_csv:decode("aaa,bbb,ccc\r\nzzz,yyy,xxx")).

rfc_double_quote_test() ->
    ?assertEqual([#{"aaa" => "zzz", "bbb" => "yyy", "ccc" => "xxx"}],
                 bday_csv:decode("\"aaa\",\"bbb\",\"ccc\"\r\nzzz,yyy,xxx")).

rfc_crlf_escape_test() ->
    ?assertEqual([#{"aaa" => "zzz", "b\r\nbb" => "yyy", "ccc" => "xxx"}],
                 bday_csv:decode("\"aaa\",\"b\r\nbb\",\"ccc\"\r\nzzz,yyy,xxx")).

rfc_double_quote_escape_test() ->
```

```
    %% Since we decided headers are mandatory, this test adds a line
    %% with empty values (CLRF,,) to the example from the RFC.
    ?assertEqual([#{"aaa" => "", "b\"bb" => "", "ccc" => ""}],
                 bday_csv:decode("\"aaa\",\"b\"\"bb\",\"ccc\"\r\n,,")).

%% @doc this counterexample is taken literally from the RFC and cannot
%% work with the current implementation because maps have no dupe keys
dupe_keys_unsupported_test() ->
    CSV = "field_name,field_name,field_name\r\n"
          "aaa,bbb,ccc\r\n"
          "zzz,yyy,xxx\r\n",
    [Map1,Map2] = bday_csv:decode(CSV),
    ?assertEqual(1, length(maps:keys(Map1))),
    ?assertEqual(1, length(maps:keys(Map2))),
    ?assertMatch(#{"field_name" := _}, Map1),
    ?assertMatch(#{"field_name" := _}, Map2).
```

Elixir translation on page 203

The last test was impossible to cover with the current property implementation, so doing it by hand in an example case still proved worthwhile. In the end, ignoring comments and blank lines, there's 27 lines of example tests that let us find one "gotcha" about our code and validates specific cases against the RFC, and 19 lines of property-based tests which let us exercise our code to the point we found inconsistencies in the RFC itself (which is not too representative of the real world)[6]. That's impressive.

All in all this combination of example-based unit tests and properties is a good match. The properties can find very obtuse problems that require complex searches into the problem space, both in breadth and in depth. On the other hand, they can be written in a way that they're general enough that some basic details could be overlooked. In this case, the property exercised encoding and decoding exceedingly well, but did not do it infallibly—we programmers are good at making mistakes no matter the situation, and example tests could also catch some things. They're great when acting as *anchors*, an additional safety net making sure our properties are not drifting away on their own.

Another similar good use of unit tests are to store *regressions*, specific tricky bugs that need to be explicitly called out or validated frequently. PropEr with Erlang and Elixir both contain options to store and track regressions automatically if you want them to. Otherwise, example tests are as good of a place as any to store that information.

With the CSV handling in place, we can now focus on filtering employee records.

6. http://tburette.github.io/blog/2014/05/25/so-you-want-to-write-your-own-CSV-code/

Filtering Records

We've got a module to convert CSV to maps, and we know that we'll need employee records and a way to filter them to move the project forwards. We could start by defining the records' specific fields, but since we know they'll be implemented using maps, and that maps are a fairly dynamic data structure, then nothing prevents us from jumping directly to the filtering step.

In this section, we'll see a case where even though the problem space is large, we can explore it *better* with example tests than with properties. The reason for this is that the type of data we can get is very regular and easy to enumerate, such that a brute force strategy pretty much guarantees a more exhaustive and reliable testing approach than a probabilistic one with properties. Properties are not *always* the best way to go about tests, and this is one case where this true. By filtering employee maps based on dates, we'll see that even when they're not the best tool, properties can still be a useful source of inspiration to come up with examples as well.

In most implementations that would rely on an external component to filter and sort records, the functionality would be provided at the interface level: in a SQL query or as arguments to an API for example. It would therefore not require tests at the unit level at all, and maybe just integration ones, if any. Filtering itself is straightforward: just use a standard library function like lists:filter/2 and pass in the date with which we want to filter. What's trickier is ensuring that the predicate passed to the function is correct.

Since the birthday search is based on 366 possible dates to verify, it could be reasonable to just run through all of them through an exhaustive search. In practice there's a few more interleavings to consider: leap years, whether there's more than one matching employee, and so on. For example, to run an exhaustive search, we would need a list of 366 employees (or 732, or even 1098 to ensure more than one employee per day), each with their own birthday. We would then need to run the program for every day of every year starting in 2018 (current year of this writing) until 2118, and making sure that each employee is greeted once per year on the same day as their birthday.

That gives slightly more than one million runs to cover the whole foreseeable future. We can try a sample run to let us estimate how long that would take, to see if it's worth doing. Let's get an approximation by running a filter function as many times as we'd need to cover around a hundred years with 1098 employees:

```
1> L = lists:duplicate(366*3, #{name => "a", bday => {1,2,3}}),
1> timer:tc(fun() ->
```

```
1>     [lists:filter(fun(X) -> false end, L) || _ <- lists:seq(1,100*366)], ok
1> end).
{17801551,ok}
```

Around 18 seconds. Not super fast, but not insufferable. This could be made faster by checking only 20 years, which takes under 3 seconds. That may be good enough to warrant bypassing unit tests and property-based testing altogether. Brute force it is! We'll inspire ourselves from the principles we use when coming up with properties, but do so outside of any property-based testing framework. The general approach to write our tests will be similar, except that we'll replace generators with fully deterministic data sets that cover all cases.

Oh yes, and then there are leap years to handle. There's a well-known formula to know which year will be a leap year, with Erlang implementing it in the calendar:is_leap_year/1 function (working on years greater than 0). It looks a bit like this:

```
-spec is_leap_year(non_neg_integer()) -> boolean().
is_leap_year(Year) when Year rem 4 =:= 0, Year rem 100 > 0 -> true;
is_leap_year(Year) when Year rem 400 =:= 0 -> true;
is_leap_year(_) -> false.
```

By using such a function, we can hand-roll a full exhaustive property test. Rather than using a ?FORALL macro, we'll write some code to generate exhaustive data (as opposed to PropEr generators, which are unpredictable) and then manually validate every call in a lists:foreach/2 call:

ResponsibleTesting/erlang/bday/test/bday_filter_tests.erl

```
-module(bday_filter_tests).
-include_lib("eunit/include/eunit.hrl").

%% Property
bday_filter_test() ->
    Years = generate_years_data(2018,2038),
    People = generate_people_for_year(3),
    lists:foreach(fun(YearData) ->
        Birthdays = find_birthdays_for_year(People, YearData),
        every_birthday_once(People, Birthdays),
        on_right_date(People, Birthdays)
    end, Years).

find_birthdays_for_year(_, []) -> [];
find_birthdays_for_year(People, [Day|Year]) ->
    Found = bday_filter:birthday(People, Day), % <= function being tested
    [{Day, Found} | find_birthdays_for_year(People, Year)].
```

Elixir translation on page 204

The data being generated is stored in Years, for the ability to generate all dates in a year, and People, which contains all the employees in a company. the search itself is run for each date of a year by the function find_birthdays_for_year/2, which just calls our actual implementation under test, bday_filter:birthday(People, Day). The result is then passed to two assertions, every_birthday_once/2 and on_right_date/2. Let's start by digging into the generators, so that we can see what data the rest of the tests will use.

The first generator is for years, which really just iterates through all possible days and assembles a list of all dates:

```
%% Generators
generate_years_data(End, End) ->
    [];
generate_years_data(Start, End) ->
    [generate_year_data(Start) | generate_years_data(Start+1, End)].

generate_year_data(Year) ->
    DaysInFeb = case calendar:is_leap_year(Year) of
        true -> 29;
        false -> 28
    end,
    month(Year,1,31) ++ month(Year,2,DaysInFeb) ++ month(Year,3,31) ++
    month(Year,4,30) ++ month(Year,5,31) ++ month(Year,6,30) ++
    month(Year,7,31) ++ month(Year,8,31) ++ month(Year,9,30) ++
    month(Year,10,31) ++ month(Year,11,30) ++ month(Year,12,31).

month(Y,M,1) -> [{Y,M,1}];
month(Y,M,N) -> [{Y,M,N} | month(Y,M,N-1)].
```

Elixir translation on page 204

There's the little special case for leap years adding a 29th day to February, but that's about it.

Now for generating people, things are a bit trickier. The first thing we'll use is a *seed year*, which just contains all possible days and months—a leap year, such as 2016, is required—and then use it to create employee records. By using the dates as a seed for employee creation, we simply ensure that we'll get one employee per date. If we need more than one employee per date, we just need to run the function more than once. This is precisely what the generate_people_for_year(N) function does:

```
generate_people_for_year(N) ->
    YearSeed = generate_year_data(2016), % leap year so all days are covered
    lists:append([people_for_year(YearSeed) || _ <- lists:seq(1,N)]).

people_for_year(Year) ->
    [person_for_date(Date) || Date <- Year].
```

```
person_for_date({_, M, D}) ->
    #{"name" => make_ref(),
      "date_of_birth" => {rand:uniform(100)+1900,M,D}}.
```

Elixir translation on page 205

You can see that people_for_year is just an iterator that will create one person for each date, and person_for_date/1 will generate a unique name (an Erlang *reference*), and pick a random birth year from 1901 to 2000. It doesn't really matter if the fake data we generate tells us someone is born on February 29 of a non-leap year since we're searching by the month and day of the *current year*, and the birth year is ignored.

So that's for the generation. Let's get back to our assertions, our manual replacement for rules for properties. The two rules we're testing are:

1. every birthday is found exactly once (nobody is left behind nor found too often)
2. every birthday is found on the right date (don't write a system that cheats by telling everyone "happy birthday" on January 1st)

Those can be implemented using regular EUnit assertion macros, which cause a failure if the condition is false, and will just succeed by not crashing otherwise. The function every_birthday_once(People, Birthdays) takes in a list of all employees (People) and all birthdays found through filtering (Birthdays), builds a list of how many birthdays were not found or found more than once, and asserts that both lists are empty—because they've been found exactly once:

```
%% Assertions
every_birthday_once(People, Birthdays) ->
    Found = lists:sort(lists:append([Found || {_, Found} <- Birthdays])),
    NotFound = People -- Found,
    FoundManyTimes = Found -- lists:usort(Found), % usort drops dupes
    ?assertEqual([], NotFound),
    ?assertEqual([], FoundManyTimes).

on_right_date(_People, Birthdays) ->
    [?assertEqual({M,D}, {PM,PD})
     || {{Y,M,D}, Found} <- Birthdays,
        #{"date_of_birth" := {_,PM,PD}} <- Found].
```

Elixir translation on page 205

The assertion in on_right_date/2 just checks that an employee's birthday (PM and PD for the month and day respectively) land on the right month and day used for the search.

If you run the test, it will unsurprisingly fail since bday_filter:birthday/2 is currently undefined. Let's write a first implementation:

ResponsibleTesting/erlang/bday/src/bday_filter.erl
```
-module(bday_filter).
-export([birthday/2]).

birthday(People, {_Year, Month, Day}) ->
    lists:filter(
      fun(#{"date_of_birth" := {_,M,D}}) -> {Month,Day} == {M,D} end,
      People
    ).
```

The implementation is straightforward, using lists:filter/2 to check for the right month and date. Running it fails in a fun way:

```
$ rebar3 eunit
«build and test output»
Failures:

  1) bday_filter_tests:bday_filter_test/0: module 'bday_filter_tests'
     Failure/Error: ?assertEqual([], NotFound)
       expected: []
            got: [#{"date_of_birth" => {1940,2,29},
                    "name" => #Ref<0.1413896841.873201667.109143>},
                  #{"date_of_birth" => {1917,2,29},
                    "name" => #Ref<0.1413896841.873201667.109509>},
                  #{"date_of_birth" => {1972,2,29},
                    "name" => #Ref<0.1413896841.873201667.109875>}]
     %% lists.erl:1338:in `lists:foreach/2`
«more output»
```

So the ?assertEqual([], NotFound) assertion fails, which tells us that we have birthdays that have not been found while they should have been. We don't have shrinking since we're writing our own properties without a framework, but by looking at the dates, we can figure out that something failed when looking up birthdays related to February 29. It's pretty certain that the problem with our filter function is that folks with their birthdays on leap days will be ignored on non-leap years.

We can patch this up by making sure that if we hit February 28 on a non-leap year, we should also look people whose birthday would be on the 29th:

ResponsibleTesting/erlang/bday/src/bday_filter.erl
```
-module(bday_filter).
-export([birthday/2]).

birthday(People, {Year, 2, 28}) ->
    case calendar:is_leap_year(Year) of
        true -> filter_dob(People, 2, 28);
        false -> filter_dob(People, 2, 28) ++ filter_dob(People, 2, 29)
```

```
    end;
birthday(People, {_Year, Month, Day}) ->
    filter_dob(People, Month, Day).

filter_dob(People, Month, Day) ->
    lists:filter(
      fun(#{"date_of_birth" := {_,M,D}}) -> {Month,Day} == {M,D} end,
      People
    ).
```

Elixir translation on page 206

Run the EUnit tests again and you should see another failing case:

```
$ rebar3 eunit
«build and test output»
Failures:

  1) bday_filter_tests:bday_filter_test/0: module 'bday_filter_tests'
     Failure/Error: ?assertEqual({2,28}, { PM , PD })
       expected: {2,28}
            got: {2,29}
     %% lists.erl:1338:in `lists:foreach/2`
«more output»
```

The test is now off. Since people whose birthday is on February 29 are greeted on the 28th in non-leap years, the on_right_date/2 assertion gets tripped: the 29th is clearly not the 28th. The test is wrong here, and it needs patching—the same kind of deal we get with regular property testing—by relaxing the assertion. We'll make it so the test accepts people being greeted on the wrong day when their birthday falls on an invalid date for the given search year:

ResponsibleTesting/erlang/bday/test/bday_filter_tests.erl

```
on_right_date(_People, Birthdays) ->
    [calendar:valid_date({Y,PM,PD}) andalso ?assertEqual({M,D}, {PM,PD})
     || {{Y,M,D}, Found} <- Birthdays,
        #{"date_of_birth" := {_,PM,PD}} <- Found].
```

Elixir translation on page 205

The reason for picking calendar:valid_date/1 over checking leap years explicitly is that it lets us avoid reusing the same implementation between the function being tested and the test itself. It has a better chance of catching us making mistakes than duplicating the logic around would.

Now running the EUnit suite yields:

```
$ rebar3 eunit
===> Verifying dependencies...
===> Compiling bday
===> Performing EUnit tests...
```

```
........

Top 8 slowest tests (1.403 seconds, 88.3% of total time):
  bday_filter_tests:bday_filter_test/0: module 'bday_filter_tests'
    1.403 seconds
  prop_csv:rfc_double_quote_test/0
    0.000 seconds
  prop_csv:rfc_record_per_line_test/0
    0.000 seconds
«more output»

Finished in 1.589 seconds
8 tests, 0 failures
```

Not too bad! Under 1.5 seconds for an exhaustive property-like test of the next 20 years of operation. While it's technically not property-based testing, the results are more trustworthy in this case since we're going over all possibilities rather than some random selected ones. We just get to debug things ourselves since there's no shrinking.

The lesson here is that even though property tests can be very exciting, it's always good to look for alternative ways to test things that might be more effective. In this case, when the various states or possible inputs are limited, exhaustive testing is a very interesting way to do better than what a property would.

Now that we have CSV handling and employee filtering, what we're missing is the actual employee module to tie both parts together.

Employee Module

The employee module is the part where we'll bridge the parsing of records with the ability of filtering and searching through employees. Proper isolation of concerns should make it possible for both types of users, those who create and those who consume employee data, to do everything they need without knowing about the requirements of the other.

In this section we'll first tackle the requirements, then work on the CSV adapter, and finally tie up the internal usage of employee records. This will let us go through a critial-but-annoying part of the system—the plumbing, full of annoying business rules—and see how we can use can properties to validate that.

Setting Requirements

Our challenge here will be to come up with an internal data representation that can easily be converted to CSV, while also allowing us to use employee

entries the way we would handle any other Erlang data structure. Of course, we'll want to test this with properties. Let's start with the transformations from what the CSV parser hands to us. The document format was:

```
last_name, first_name, date_of_birth, email
Doe, John, 1982/10/08, john.doe@foobar.com
Ann, Mary, 1975/09/11, mary.ann@foobar.com
```

What we can notice here is that:

- the fields are messy and have extra leading spaces
- the dates are in "YYYY/MM/DD" format whereas Erlang works on {Year,Month,Day} tuples and Elixir uses a *Date struct*

The transformation requires additional processing after the conversion from CSV. This could usually be done or handled by a framework or adapter. For example, most PostgreSQL connection libraries will convert the internal data type for dates and time to Erlang's {{Year,Month,Day}, {H,Min,Sec}} tuple format without much of a problem. In the case of CSV, the specification is really lax and as such it is our responsibility to convert from a string to the appropriate type, along with some additional validation.

We'll define the following functionality:

- a from_csv/1 function that takes a CSV string and returns a cleaned-up set of maps representing individual employees. Erlang's opaque types[7] will let us use Dialyzer to ensure that nobody looks at the internal employee data set other than as a thoroughly abstract piece of data. This will allow us to change data representations later, moving from CSV to a SQL-based iterator transparently, for example. Remember that we're aiming for long term flexibility, not the simplest thing that can work.
- an accessor for each field (last_name/1, first_name/1, date_of_birth/1, email/1)
- a function to search employees by birthday

This should encapsulate all of our requirements without a problem.

Adaptating CSV Data

Since we have already tested CSV conversion itself, what we need to do is take the output from the parser and hammer it into shape. For our tests, this means that we won't have to work with generating CSV data, but really work with generating the data that needs cleaning up. Let's start with the leading spaces. We know all the fields required, and we know that all of them but the

7. http://erlang.org/doc/reference_manual/typespec.html#id80458

first will be messy, so a property about that only needs to ensure that once handled, no fields start with whitespace:

ResponsibleTesting/erlang/bday/test/prop_bday_employee.erl

```
-module(prop_bday_employee).
-include_lib("proper/include/proper.hrl").

%%%%%%%%%%%%%%%%%%
%%% Properties %%%
%%%%%%%%%%%%%%%%%%
prop_fix_csv_leading_space() ->
    ?FORALL(Map, raw_employee_map(),
        begin
            Emp = bday_employee:adapt_csv_result(Map),
            Strs = [X || X <- maps:keys(Emp) ++ maps:values(Emp), is_list(X)],
            lists:all(fun(String) -> hd(String) =/= $\s end, Strs)
        end).
```

Testing Private Functions in Elixir

Unfortunately, Elixir does not allow private functions to be tested. The properties related to data conversion using that function are therefore not translated, even if the private function will be.

As you can see, we rely on the yet undefined `raw_employee_map()` generator, call the adapting function (which will be private), and then check that in none of the keys nor values, the first character is a space. Let's see how to implement the generator. The first trick there is that instead of generating a map with the `map()` generator provided by PropEr, we'll build one from a proplist. The issue with the default generator as supported at the time of this writing is that it takes types for keys and values, and does not let us set specific values easily. A proplist and a `?LET` macro will do just fine:

ResponsibleTesting/erlang/bday/test/prop_bday_employee.erl

```
%%%%%%%%%%%%%%%%%%
%%% Generators %%%
%%%%%%%%%%%%%%%%%%
raw_employee_map() ->
    %% PropEr's native map type does not allow pre-defined static
    %% keys (just `map(T1, T2)') so we convert from a proplist
    ?LET(PropList,
         [{"last_name", prop_csv:field()}, % 1st col has no leading space
          {" first_name", whitespaced_text()},
          {" date_of_birth", text_date()},
          {" email", whitespaced_text()}],
         maps:from_list(PropList)).

whitespaced_text() ->
    ?LET(Txt, prop_csv:field(), " " ++ Txt).

text_date() ->
```

```
    %% leading space and leading 0s for months and days; since we're
    %% checking string formats, it doesn't matter if dates are invalid
    ?LET({Y,M,D}, {choose(1900,2020), choose(1,12), choose(1,31)},
         lists:flatten(io_lib:format(" ~w/~2..0w/~2..0w", [Y,M,D]))).
```

Elixir translation on page 206

You can also see that we reuse the previously defined prop_csv:field() generator we defined when writing the CSV parser; no reason to reinvent that one. All we do is add an unconditional leading whitespace in front of it. We also write a generator for the arbitrary date format provided for us in the file sample, which will need to be properly cleaned up as well.

Running this property will fail because the bday_employee:adapt_csv_result/1 function does not exist. Since it's not going to be in our interface, though, we shouldn't export it in usual circumstances. With traditional example tests, people would colocate their EUnit test within the bday_employee module. A nicer trick to keep our tests separate from the production code is to use a conditional macro based on the TEST profile to only export the function while writing tests:

ResponsibleTesting/erlang/bday/src/bday_employee.erl

```
-module(bday_employee).

%% CSV exports
-export([from_csv/1]).

-ifdef(TEST).
-export([adapt_csv_result/1]).
-endif.
```

Then we can start focusing on implementing the CSV conversion itself. We'll use a special approach by wrapping all our conversions into a {raw, ...} tuple, hidden behind an opaque type. We'll see why real soon but for now, just trust that this will buy us a lot of flexibility for future changes.

```
-opaque employee() :: #{string() := term()}.
-opaque handle() :: {raw, [employee()]}.
-export_type([handle/0, employee/0]).

-spec from_csv(string()) -> handle().
from_csv(String) ->
    {raw, [adapt_csv_result(Map) || Map <- bday_csv:decode(String)]}.
```

Elixir translation on page 207

Then we can add the function we're actually testing:

```
-spec adapt_csv_result(map()) -> employee().
adapt_csv_result(Map) ->
    maps:fold(fun(K,V,NewMap) -> NewMap#{trim(K) => trim(V)} end,
              #{}, Map).
```

```
trim(Str) -> string:trim(Str, leading, " ").
```

Let's run the tests to see how this goes:

```
$ rebar3 proper -m prop_bday_employee
===> Verifying dependencies...
===> Compiling bday
===> Testing prop_bday_employee:prop_fix_csv_leading_space()
!
Failed: After 1 test(s).
An exception was raised: error:badarg.
Stacktrace: [{erlang,hd,[[]],[]},
             {prop_bday_employee,'-prop_fix_csv_leading_space/0-fun-1-',1,
                 [{file,
                      "«absolute path»/bday/test/prop_bday_employee.erl"},
                  {line,13}]},
             {lists,all,2,[{file,"lists.erl"},{line,1213}]}].
«more output»
```

So the error here is when the function hd([]) is called; of course we can't check what the first character of a string is if the string is empty. In fact, we could argue that empty strings should be replaced by the atom undefined, or :nil in Elixir.

Making that change will implicitly fix the test at the same time, since it already filters out non-string results with is_list(X) in the property (we could add a test to check for undefined values, but we'll skip it for brevity):

```
-spec adapt_csv_result(map()) -> employee().
adapt_csv_result(Map) ->
    maps:fold(fun(K,V,NewMap) -> NewMap#{trim(K) => maybe_null(trim(V))} end,
              #{}, Map).

trim(Str) -> string:trim(Str, leading, " ").

maybe_null("") -> undefined;
maybe_null(Str) -> Str.
```

Run the property again and you'll see that it passes. The next requirement is to convert known date columns (such as "date_of_birth") into the internal Erlang date format. We'll check this one with a new property, rather than adding complexity to the previous one:

ResponsibleTesting/erlang/bday/test/prop_bday_employee.erl

```
prop_fix_csv_date_of_birth() ->
    ?FORALL(Map, raw_employee_map(),
        case bday_employee:adapt_csv_result(Map) of
            #{"date_of_birth" := {Y,M,D}} ->
                is_integer(Y) and is_integer(M) and is_integer(D);
            _ ->
                false
```

```
        end).
```

The generator sequence is identical to prop_fix_csv_leading_whitespace(), but rather than looking at strings, we make sure that the date format is valid. To make it pass, we must implement the matching code in the bday_employee module by patching the adapt_csv_result/1 function:

ResponsibleTesting/erlang/bday/src/bday_employee.erl

```
-spec adapt_csv_result(map()) -> map().
adapt_csv_result(Map) ->
    NewMap = maps:fold(
        fun(K,V,NewMap) -> NewMap#{trim(K) => maybe_null(trim(V))} end,
        #{},
        Map
    ),
    DoB = maps:get("date_of_birth", NewMap), % crash if key missing
    NewMap#{"date_of_birth" => parse_date(DoB)}.

trim(Str) -> string:trim(Str, leading, " ").

maybe_null("") -> undefined;
maybe_null(Str) -> Str.

parse_date(Str) ->
    [Y,M,D] = [list_to_integer(X) || X <- string:lexemes(Str, "/")],
    {Y,M,D}.
```

Elixir translation on page 207

Run the properties and everything should pass:

```
$ rebar3 proper -m prop_bday_employee
===> Verifying dependencies...
===> Compiling bday
===> Testing prop_bday_employee:prop_fix_csv_leading_space()
.....................................................................
.....................
OK: Passed 100 test(s).
===> Testing prop_bday_employee:prop_fix_csv_date_of_birth()
.....................................................................
.....................
OK: Passed 100 test(s).
===>
2/2 properties passed
```

Good! We've covered the entire CSV correction and integration. We now just have to put the accessor functions in place.

Using Employees

Because the employee module ties together the data conversion functionality with the search itself, it turns out to be the perfect place for us to hide

implementation details about the data storage layer. If we do things right, the bits that rely on the data to do something useful get isolated from how the data is obtained. This is not necessarily that big of a deal for tests, but it's a critical part of future-proofing our system design.

We'll start with data accessors, which should be trivial and simple enough they don't require testing. We just have to add them to bday_employee and export them:

```
-export([from_csv/1, last_name/1, first_name/1, date_of_birth/1, email/1]).

«existing code»

-spec last_name(employee()) -> string() | undefined.
last_name(#{"last_name" := Name}) -> Name.

-spec first_name(employee()) -> string() | undefined.
first_name(#{"first_name" := Name}) -> Name.

-spec date_of_birth(employee()) -> calendar:date().
date_of_birth(#{"date_of_birth" := DoB}) -> DoB.

-spec email(employee()) -> string().
email(#{"email" := Email}) -> Email.
```

Elixir translation on page 207

Now our users can truly ignore the underlying map implementation.

If you're careful, you'll have noticed that all these accessors just take a direct employee() type value as an argument (a map), but from_csv(String) returned a value of type handle() ({raw, [employee()]}). The question is how are we going to transition from one to the other? The answer is that this opaque handle() type is a way to represent an abstract resource that can be substituted at a later point.

For example, if at a later date we were to replace the CSV-backed employee storage with a database, we may change the definition of handle() from {raw, [employee()]} to become {db, Config, Connection, SQLQuery}. This data structure could then be built and modified lazily, and only be materialized with real data by executing a final query. A similar functionality backed by a microservice could be using a handle of the form {service, URI, ExtraData}, and so on. All that's needed is a function to *actualize* the result set into a discrete list of maps users are free to poke into:

```
-spec fetch(handle()) -> [employee()].
fetch({raw, Maps}) -> Maps.
```

Elixir translation on page 207

Export this function by adding -export([fetch/1]). with other export attributes. In the current implementation, the {raw, ...} format just lets us do everything locally with the same interface. With this in mind, we can now represent every call that could be a remote query or action through this format. For example, we can implement the filter_birthday/3 call as:

```
-spec filter_birthday(handle(), calendar:date()) -> handle().
filter_birthday({raw, Employees}, Date) ->
    {raw, bday_filter:birthday(Employees, Date)}.
```

Elixir translation on page 207

That whole mechanism a bit complex, so let's write a property that shows how this should all be used together:

ResponsibleTesting/erlang/bday/test/prop_bday_employee.erl

```
prop_handle_access() ->
    ?FORALL(Maps, non_empty(list(raw_employee_map())),
        begin
            CSV = bday_csv:encode(Maps),
            Handle = bday_employee:from_csv(CSV),
            Partial = bday_employee:filter_birthday(Handle, date()),
            ListFull = bday_employee:fetch(Handle),
            true = is_list(bday_employee:fetch(Partial)),
            %% Check for no crash
            _ = [{bday_employee:first_name(X),
                  bday_employee:last_name(X),
                  bday_employee:email(X),
                  bday_employee:date_of_birth(X)} || X <- ListFull],
            true
        end).
```

Elixir translation on page 208

You can see that we first generate all the employee maps, encode them with our CSV encoder, and then extract them with bday_employee:from_csv/1, which creates an opaque handle. This opaque handle is then passed to bday_employee:filter_birthday/2 along with the result of date() (an Erlang built-in function returning today's date). If you're a purist for unit testing, feel free to replace date() with any hard-coded date, possibly one from the existing employee set. That filter returns a new handle, and we can only get a list of employees once bday_employee:fetch/1 is called. Then we just look to make sure everything works without crashing, and that's the whole property.

What's interesting with this program structure is that if you were to use, say, Ecto from Elixir, raw SQL queries, or any other ORM, the only thing you'd need to change *as a caller* is how the initial fetch for data is done. Say something like bday_employee:from_sql(...) or bday_employee:from_cache(...) instead of

bday_employee:from_csv(...) and then the rest of the changes can be hidden within the module itself. This is generally a good sign that we've prevented leaky abstractions: swapping the implementation of the backing structure altogether does not really break the interface we have chosen for consumers.

We can run the full suite and get some coverage statistics to see how it goes:

```
$ rebar3 do eunit -c, proper -c, cover -v
«build information»
===> Performing EUnit tests...
........
«timing output»
8 tests, 0 failures
«PropEr output»
===>
4/4 properties passed
«build information»
===> Performing cover analysis...
  |------------------------|------------|
  |                 module |  coverage  |
  |------------------------|------------|
  |              bday_csv  |      100%  |
  |         bday_employee  |      100%  |
  |           bday_filter  |      100%  |
  |------------------------|------------|
  |                  total |      100%  |
  |------------------------|------------|
```

That kind of vanity metrics feels good! Perfect is not a sign that there are no bugs, but it's probably still looking better than if we had found out all our properties only exercised 20% of the code.

We have most of the functional components of our system in place now. All we need is to take the employees we are looking for and create the e-mails we'll want to send.

Templating

The last bit left to do before tying it all up is templating. This is going to be a simple section with limited tests, but still manageable with properties. The requirement is straightforward. Send an email whose content is just `Happy birthday, dear $first_name!`. The function should take one employee term and that's it. Since the focus is on unit tests and we won't send actual emails, only templating needs coverage for now. Let's start by writing a property in a standalone suite:

ResponsibleTesting/erlang/bday/test/prop_bday_mail_tpl.erl

```
-module(prop_bday_mail_tpl).
```

```
-include_lib("proper/include/proper.hrl").

prop_template_email() ->
    ?FORALL(Employee, employee_map(),
            nomatch =/= string:find(bday_mail_tpl:body(Employee),
                                    maps:get("first_name", Employee))
    ).

employee_map() ->
    %% Convert from a proplist to have specific keys in a map
    ?LET(PropList,
         [{"last_name", non_empty(prop_csv:field())},
          {"first_name", non_empty(prop_csv:field())},
          {"date_of_birth", {choose(1900,2020),choose(1,12),choose(1,31)}},
          {"email", non_empty(prop_csv:field())}],
         maps:from_list(PropList)).
```

Elixir translation on page 208

The string:find/2 function looks for a given string within another one and returns it if found, or nomatch if missing. One gotcha is that some fields are defined as nullable in the employee module (they may return undefined). The initial specification did not mention if it were possible or not for them to be missing, but since the sample of two entries all had fields, we will assume that so will our production data, and our generators reflect that fact.

With the following implementation, the test should pass every time:

ResponsibleTesting/erlang/bday/src/bday_mail_tpl.erl

```
-module(bday_mail_tpl).
-export([body/1]).

-spec body(bday_employee:employee()) -> string().
body(Employee) ->
    lists:flatten(io_lib:format("Happy birthday, dear ~s!",
                                [bday_employee:first_name(Employee)])).
```

A trivially correct convenience function that extracts all that is needed for an email to be sent (address, subject, body) can be added to provide further decoupling:

```
-export([full/1]). % add this export near the top of the file

-spec full(bday_employee:employee()) -> {[string()], string(), string()}.
full(Employee) ->
    {[bday_employee:email(Employee)],
     "Happy birthday!",
     body(Employee)}.
```

Elixir translation on page 209

The email address is put in a list since e-mail clients typically allow more than one entry in the To: field.

And with this in place, the last individual component of the system is done. All we've got to do is assemble everything together.

Plumbing it all Together

It's now time to take all of these well-tested bits, and integrate them with each other. Fortunately for us, integration is not a concern for this chapter (we care about *unit* tests) so we can just throw everything together quickly and consider our job done:

ResponsibleTesting/erlang/bday/src/bday.erl

```
-module(bday).
-export([main/1]).

main([Path]) ->
    {ok, Data} = file:read_file(Path),
    Handle = bday_employee:from_csv(binary_to_list(Data)),
    Query = bday_employee:filter_birthday(Handle, date()), % date = local time
    BdaySet = bday_employee:fetch(Query),
    Mails = [bday_mail_tpl:full(Employee) || Employee <- BdaySet],
    [send_email(To, Topic, Body) || {To, Topic, Body} <- Mails].

send_email(To, _, _) ->
    io:format("sent birthday email to ~p~n", [To]).
```

Elixir translation on page 209

The e-mail client is not implemented since it is also out of scope for now (and is left as an exercise to the reader, to steal a frustrating quote from academia). For now we'll stick with a simple io:format/2 call to stand in for the actual e-mail client handling and call it a day. We just need to package the program to be run. Erlang lets you do this by calling rebar3 escriptize, generating a standalone script. If you're using elixir, call mix run -e Bday.run("db.csv") instead:

```
$ rebar3 escriptize
===> Verifying dependencies...
===> Compiling bday
===> Building escript...
```

The script is generated in the _build/default/bin directory. The program can now be run, as long as you ensure the sample CSV file has full CRLF line terminations:

ResponsibleTesting/erlang/bday/priv/db.csv

```
last_name, first_name, date_of_birth, email
Doe, John, 1982/10/08, john.doe@foobar.com
Ann, Mary, 1975/09/11, mary.ann@foobar.com
```

```
Robert, Joe, 2002/03/18, born.today@example.com
```

Don't forget to ensure one of the employees in the sample database has their birthday *today* for the program to output anything:

```
$ _build/default/bin/bday priv/db.csv
sent birthday email to ["born.today@example.com"]
```

And it works! All in all, we can now say that coverage is good, critical units are tested, and changing implementations for our data layer will have limited effects in the testing code base.

We can pat ourselves on the back for a job well done.

Wrapping Up

You are now in a pretty good place when it comes to writing properties. You should feel better about balancing them with regular example-based unit tests, as you've seen it done for regression cases. Using example tests as an anchoring mechanism to backstop your properties can help make them a lot more trustworthy. In some cases, just exhaustively enumerating the whole data set is a possibility, letting you cover more than what properties would in the first place.

The experience gained writing properties against a well-specified program is sure to prove useful, but there's another entire face to property-based testing. We can use properties to explore the design of our program itself, rather than just testing a well-specified one. In the next chapter, we'll do just that, by using them in properties-driven development.

CHAPTER 6

Properties-Driven Development

If you're reading this book, it's because programming is kind of difficult. Difficult enough that we need fancy tools to make sure our programs are doing the right thing. But while programming is hard, an even harder thing is probably just being able to figure out *what our programs should be doing* in the first place, especially when we find all kinds of fun corner cases. There are plenty of tools available to help here, and for software developers, Test-Driven Development (TDD) is one of the most frequently used approaches. TDD forces you to position yourself as a user of the program rather than the implementer first. Before writing any new feature, a failing test exercising the feature must first be written, and then the code needs to be written to make it pass. All program improvements are a series of small iterations built on well-understood foundations.

This kind of approach can be interesting in the context of property-based testing: before writing a program, we'll want to think about what it should do. Unsurprisingly, we'll want to encode these assumptions as rules for properties. Then, we'll run our program against them as we go. What happens next is a series of increasing failures, where we have to figure out if it's the program that is wrong, or our understanding of what it should do in the first place that needs to change.

In this chapter, we'll go through a properties-driven approach, and explore techniques related to positive tests (validate what the program does) and negative ones (test what the program cannot handle). We'll also push generators further than we've done before, which will be a good exercise on its own. We'll do all of that by developing a short program inspired by the *Back to the Checkout* code kata,[1] where we implement a pricing system for a supermarket.

1. http://codekata.com/kata/kata09-back-to-the-checkout/

The Specification

Although it'd be great to have a well-defined specification, we will work from a more realistic starting point by keeping things vague. We'll write a system that is given a bunch of items that have been scanned. We have to look up the price of these items, and calculate the total amount to charge. A little fun bit to add is that some items grouping give an instant rebate: you may get two articles for the price of one, or specific amounts like 5 bags of chips for $7.

In fact, the code kata gives the following table:

Item	Unit Price	Special Price
A	50	3 for 130
B	30	2 for 45
C	20	
D	15	

The checkout function should accept items in any order, so that the sequence `["B","A","B"]` would give the 2 for 45 rebate for the Bs being bought. The only call that needs to be exposed is `checkout:total(ListOfItemsBought, UnitPrices, SpecialPrices) → Price`.

That's it. The requirements are kind of vague, but we can extract the following list of things to tackle out of it:

- items are identified by unique names (non-empty strings)
- the list of items has no set length (though it appears non-empty), and the items it contains are in no specific order
- the unit prices appear to be integers (not a bad idea, since floating point numbers lose precision; we can assume values are written in cents, for example)
- the special prices are triggered only once the right number of items of the proper type can match a given special
- all items have a unit price
- it is not mandatory for a given item to have a special price.

The rest is seemingly undefined or underspecified, so there's a good chance the assumptions we bake into the program about the rest could be tricky and bug-prone.

Aside from these requirements, it's a good idea to try and see if you can come up with a few properties. Here are two possible ones:

- without any special prices defined, the total is just the sum of all of the items' prices
- with a known count for each item, the total comes from two parts added together:
  1. the special price multipled by the number of times it can be applied to a given set of items
  2. the remainder (not evenly divisible by the special's required number of items), which is added on top of the special prices

Since we're interested in discovering and improving the design of the program through properties, we can get started by implementing the two first properties we've come up with, and then writing the code to make them pass. This will bring us to a point where the basic program does what we want. Then, we'll see how properties can be used in *negative tests*, checking that not only the program does what we want, but it also deals with things we don't want to happen.

Just before we get going, though, don't forget to set up a new project as we did in prior chapters such as Foundations on page 10—call `rebar3 new lib checkout` for Erlang, `mix new checkout` for Elixir, and then add PropEr to dependencies.

Writing the First Test

The first property does not have to be very fancy. It's in fact even better if it is simple and down to earth. Start with something trivial-looking that represents how we want to use the program first and foremost. Then it's going to be our job as a developers to make sure we can write code that matches our expectations, or to change them. Of the two properties we had, the simplest one concerns counting sums without caring about specials.

We'll want to avoid a property definition such as `sum(ItemList, PriceList) =:= checkout:total(ItemList, PriceList, [])` since that would really risk making the test similar to the implementation. A good approach to try here is generalizing regular example-based tests. Let's imagine a few cases:

```
20 = checkout:total(["A","B","A"], [{"A",5},{"B",10}], []),
20 = checkout:total(["A","B","A"], [{"A",5},{"B",10},{"C",100}], []),
115 = checkout:total(["F","B","C"], [{"F",5},{"B",10},{"C",100}], []),
«and so on»
```

That's actually a bit tricky to generalize. It's possible the way you come up with examples is really to just make a list of items, assign them prices, pick

items from the list and then sum them up yourself. Even if it's not really straightforward, we can build on that. The base step to respect here is really something looking like:

```
ExpectedPrice = checkout:total(ChosenItems, PriceList, [])
```

If we write a generator that gives us all known values for these variables, we *can* use the steps the same way we'd do it with examples. The generator will literally generate test cases so we don't have to do so. So let's get a property that looks like this:

PropertiesDrivenDevelopment/erlang/checkout/test/prop_checkout.erl

```
-module(prop_checkout).
-include_lib("proper/include/proper.hrl").
-compile(export_all).

%%%%%%%%%%%%%%%%%%
%%% Properties %%%
%%%%%%%%%%%%%%%%%%
prop_no_special1() ->
    ?FORALL({ItemList, ExpectedPrice, PriceList}, item_price_list(),
            ExpectedPrice =:= checkout:total(ItemList, PriceList, [])).
```

Elixir translation on page 210

The generator for the property will need to generate the three expected arguments: a list of items bought by the customer (ItemList), the expected price of those items (ExpectedPrice), and then the list of items with their prices as expected by the register itself (PriceList).

Since the price list is required to generate the item list and expected prices, the generator will need to come in layers with ?LET macros:

```
%%%%%%%%%%%%%%%%%%
%%% Generators %%%
%%%%%%%%%%%%%%%%%%
item_price_list() ->
    ?LET(PriceList, price_list(),
         ?LET({ItemList, ExpectedPrice}, item_list(PriceList),
              {ItemList, ExpectedPrice, PriceList})).
```

Elixir translation on page 210

The price list itself is a list of tuples of the form [{ItemName, Price}]. The ?LET macro actualizes the list into one value that won't change for the rest of the generator. This means that the item_list/1 generator can then use PriceList as the actual Erlang data structure rather than the abstract intermediary format PropEr uses. But first, let's implement the price_list/0 generator:

```
%% generate a list of {ItemName, Price} to configure the checkout
```

```
price_list() ->
    ?LET(PriceList, non_empty(list({non_empty(string()), integer()})),
         lists:ukeysort(1, PriceList)). % remove duplicates
```

Elixir translation on page 210

Here, price_list/0 generates all the tuples as mentioned earlier, each with an integer for the price. To avoid duplicate item options, such has having the same hot-dogs at two distinct prices within the same list, we use lists:ukeysort(KeyPos, List). That function will remove all list items that share the same key as any item that has already been seen, ensuring we have only unique entries.

Now, for the little sleight of hand, we have to use the PriceList as a seed for item_list/1, which should return a complete selection of items along with their expected price:

```
%% set up recursive generator for the purchased item list along with
%% its expected price, based off the price list.
item_list(PriceList) ->
    ?SIZED(Size, item_list(Size, PriceList, {[], 0})).

item_list(0, _, Acc) -> Acc;
item_list(N, PriceList, {ItemAcc, PriceAcc}) ->
    ?LET({Item, Price}, elements(PriceList),
         item_list(N-1, PriceList, {[Item|ItemAcc], Price+PriceAcc})).
```

Elixir translation on page 210

For the tests to be able to pass, we'll have to write the implementation code itself. Let's start with a real minimal case that should just work:

PropertiesDrivenDevelopment/erlang/checkout/src/checkout.erl

```
-module(checkout).

-export([total/3]).

-type item() :: string().
-type price() :: integer().

-spec total([item()], [{item(), price()}], any()) -> price().
total(ItemList, PriceList, _Specials) ->
    lists:sum([proplists:get_value(Item, PriceList) || Item <- ItemList]).
```

Elixir translation on page 210

Run the property and you'll see that it does work.

Now that we have a working property, the next step we should take is to validate how effective it is. This is a bit tedious, but it lets us make sure that

the 100 or so tests we run for the property are actually different. We can do this in a rather straightforward manner using the collect/2 function:

PropertiesDrivenDevelopment/erlang/checkout/test/prop_checkout.erl

```
prop_no_special2() ->
    ?FORALL({ItemList, ExpectedPrice, PriceList}, item_price_list(),
            collect(
              bucket(length(ItemList), 10),
              ExpectedPrice =:= checkout:total(ItemList, PriceList, [])
            )).

%%%%%%%%%%%%%%%
%%% Helpers %%%
%%%%%%%%%%%%%%%
bucket(N, Unit) ->
    (N div Unit) * Unit.
```

Elixir translation on page 211

Running this property will then reveal metrics such as:

```
27% 0
27% 10
20% 20
20% 30
6% 40
```

This appears reasonable. Lists of 0 to 40 items being bought probably covers most cases. Alternatively running the call with collect(lists:usort(ItemList), <Property>) shows good results as well, meaning that the variety of items being bought also seems reasonable.

With the regular case (without specials) being taken care of, we're good to start implementing specials. That one will be a bit trickier though.

Testing Specials

Alright, before moving further, let's re-evaluate our TODO list:

- ~~items are identified by unique names (non-empty strings)~~
- ~~the list of items has no set length (though it appears non-empty), and the items it contains are in no specific order~~
- ~~the unit prices appear to be integers (not a bad idea, since floating point numbers lose precision; we can assume values are written in cents, for example)~~
- the special prices are triggered only once the right number of items of the proper type can match a given special
- ~~all items have a unit price~~

- ~~it is not mandatory for a given item to have a special price.~~

That's decent coverage in terms of features and requirements we extracted, with few lines of tests! In face, it doesn't feel like we've accomplished much for now, but most features are okay as per our spec. The specials are the one thing left to handle.

Rather than modifying the existing property we have—a property that does a fine job of checking non-special prices—we'll add a new one to do that. The separation will help narrow down problems when they happen. If the property for basic prices always works, then we know that failures in the more complex one handling specials as well is likely going to be related to bugs in specials-handling. Here's the new property:

PropertiesDrivenDevelopment/erlang/checkout/test/prop_checkout.erl

```
prop_special() ->
    ?FORALL({ItemList, ExpectedPrice, PriceList, SpecialList},
            item_price_special(),
        ExpectedPrice =:= checkout:total(ItemList, PriceList, SpecialList)).
```

Elixir translation on page 211

This property is similar to the one we wrote earlier, except that we now expect a fourth term out of the generator, which is a list of special prices (SpecialList). The easiest way to go about this would be to just come up with a static list of specials and then couple it with the previous property's generator, but that wouldn't necessarily exercise the code as well as fully dynamic lists, so let's try to do that instead.

Planning the Generator

So first things first, we'll need the basic list of items and prices. For that bit, we can reuse the price_list() generator, which gives us a fully dynamic list. Then, if we want the specials list to be effective, we should probably build it off the items in the price list. That can be done by wrapping the call to price_list() in a ?LET macro so that other generators can see it as a static value:

```
?LET(PriceList, price_list(),
     «rest of the generator»).
```

The challenge, really, will be to generate the list of items to buy, while maintaining a proper expected price (ExpectedPrice). Let's say we were to reuse the item_list/1 generator from earlier here. If the checkout counter sells 3 donuts for the price of 2, but the generator creates a list of 4 donuts without expecting a special, then our test will be wrong. If you were to patch things up by still using the generator and figure out the specials from the generated

list, then chances are the generator code would be as complex as the actual program's code as well. That's a bad testing approach, since it makes it hard to trust our tests.

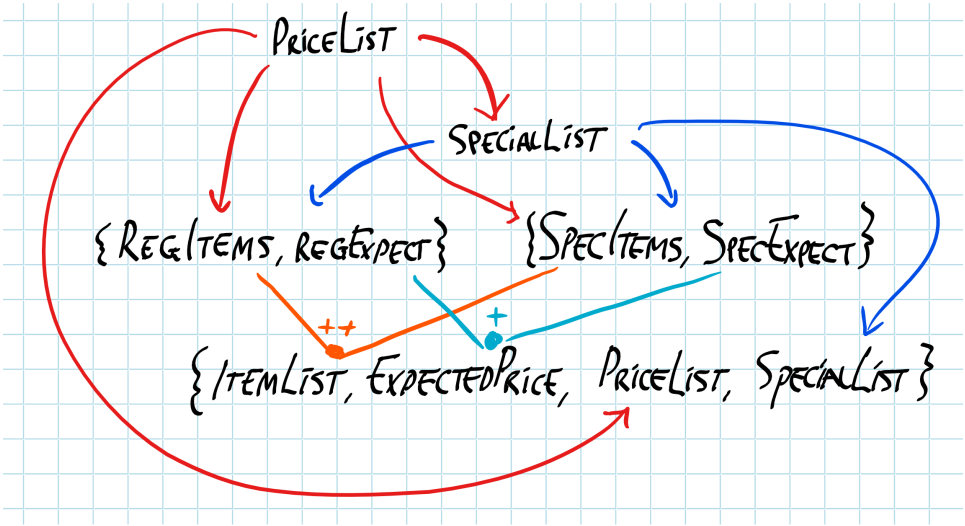
The way to go about it is to shift your perspective and look at the problem from a different direction. You'll see the trick, but be aware that it's probably the sneakiest and cleverest one in the entire book; it's a decent example of the creative thinking sometimes required to write good generators.

Here it goes. Instead of building the list one item at a time and figuring out if or when specials apply, we can try to break it up by generating either types of sequences: items that *never* amount to a special, and items that *always* amount to a special. If these types of sequences are well-generated and distinct, then they can be used independently or as one big list, and always remain consistent!

So we can have two generators. On the one hand, a list where all the items of each type are in a quantity smaller than required for the special price. We can sum up their prices together to give us one part of the expected price. Then on the other side, a generator where all the items in a least are in a quantity that is an even multiple of the quantity required for a special. For that list, the expected price is the sum of all specials we triggered. We can then merge both lists, add both expected prices, and end up with a list that covers all kinds of cases possible in a totally predictable manner.

Let's step through this with our example where 3 donuts are being sold at the price of two. If we have a list with 8 donuts in it, we'd expect the the special to be applied twice, with 2 remaining donuts being sold at full price. So let's see how this would work with our two planned generators. First we generate a list of two donuts—below the special number—and track their expected price with the non-special value. Then we generate a list of 6 donuts (twice the special quantity) with their expected price based off the special value. Then, we merge both lists together and sum up their expected prices, and end up with the same 8 donuts and the correct expected price.

This should work fine. The generator should therefore have this kind of flow:



Where ++ stands for list concatenation, and + is regular addition. Let's build it step by step.

Writing the Generator

First, we'll want to isolate the PriceList value to be standing alone and reusable by all generators:

```
item_price_special() ->
    %% first LET: freeze the PriceList
    ?LET(PriceList, price_list(),
        «rest of the generator»).
```

We can then fill it in by generating a SpecialList that will contain the specials definition:

```
item_price_special() ->
    %% first LET: freeze the PriceList
    ?LET(PriceList, price_list(),
        %% second LET: freeze the SpecialList
        ?LET(SpecialList, special_list(PriceList),
            «rest of the generator»)).
```

Okay, that part was still straightforward. The special_list/1 generator is not written yet, but we can do that later. Let's finish the current generator first. For the next level, we have to generate the two sets of prices, the regular ones and the special ones, which can each have their own dedicated generator:

```
item_price_special() ->
    %% first LET: freeze the PriceList
    ?LET(PriceList, price_list(),
        %% second LET: freeze the SpecialList
```

```
        ?LET(SpecialList, special_list(PriceList),
            %% third LET: Regular + Special items and prices
              ?LET({{RegularItems, RegularExpected},
                    {SpecialItems, SpecialExpected}},
                   {regular_gen(PriceList, SpecialList),
                    special_gen(PriceList, SpecialList)},
                   «rest of the generator»))).
```

You'll notice that each of the two new (and yet undefined) generators both have the PriceList and SpecialList passed in. From within the generators, those will look like regular Erlang terms.

All we have left to do is merge the lists and sum up the prices:

```
item_price_special() ->
    %% first LET: freeze the PriceList
    ?LET(PriceList, price_list(),
        %% second LET: freeze the SpecialList
        ?LET(SpecialList, special_list(PriceList),
            %% third LET: Regular + Special items and prices
              ?LET({{RegularItems, RegularExpected},
                    {SpecialItems, SpecialExpected}},
                   {regular_gen(PriceList, SpecialList),
                    special_gen(PriceList, SpecialList)},
                   %% And merge + return initial lists:
                   {shuffle(RegularItems ++ SpecialItems),
                    RegularExpected + SpecialExpected,
                    PriceList, SpecialList}))).

shuffle(L) ->
    %% Shuffle the list by adding a random number first,
    %% then sorting on it, and then removing it
    Shuffled = lists:sort([{rand:uniform(), X} || X <- L]),
    [X || {_, X} <- Shuffled].
```

Elixir translation on page 211

Whew. That's the whole thing. The list shuffling is added as a way to make sure the final result is truly unpredictable. You will rarely see generators of that complexity, but it's good to try your hand at it and see what can be done. Sadly, we can't use this yet, because we still need to define the special list generators, and then both the regular and special lists generators.

Generating the List of Specials

We'll start with the list of specials, since it's going to be conceptually simpler, and a pre-requisite of other generators. The way to do this one is to first extract the list of all item names from the price list, and then create a matching table of specials:

```
%% Generates specials in a list of the form
%% [{Name, Count, SpecialPrice}]
special_list(PriceList) ->
    Items = [Name || {Name, _} <- PriceList],
    ?LET(Specials, list({elements(Items), choose(2,5), integer()}),
         lists:ukeysort(1, Specials)). % no dupes
```

Elixir translation on page 211

The generated list of specials contains the name of the item, how many of them are required for the special to apply, and then the price for the whole group. The specials are generated from randomly selected entries from the item list—with the duplicates removed—so that there is always a matching non-special item to any special one. Note that it is possible for the special price to be higher than the non-special price with this generator.

Generating the List of Regular Items

The list of items *never* being on special is a bit tricky to generate. We have to make it so we generate 0 or more items, as long as the maximal value is below any specials for it:

```
Line 1  %% Generates lists of regular items, at a price below the special value.
-       regular_gen(PriceList, SpecialList) ->
-           regular_gen(PriceList, SpecialList, [], 0).
-
5       regular_gen([], _, Items, Price) ->
-           {Items, Price};
-       regular_gen([{Item, Cost}|PriceList], SpecialList, Items, Price) ->
-           CountGen = case lists:keyfind(Item, 1, SpecialList) of
-               {_, Limit, _} -> choose(0, Limit-1); % has special; set max amount
10              _ -> non_neg_integer()               % no special; generate at will
-           end,
-           %% Use the conditional generator to generate items
-           ?LET(Count, CountGen,
-                regular_gen(PriceList, SpecialList,
15                             ?LET(V, vector(Count, Item), V ++ Items),
-                             Cost*Count + Price)).
```

Elixir translation on page 212

First, on line 2, we just wrap the generator to start with a null expected price. Line 5 contains the recursive function's base case; whenever all items in the list have been iterated over, the full list and expected price are returned.

Now for the fun part, the last clause. The first thing to do is look up whether the item from PriceList can be found in SpecialList, then the count required to trigger the special is used as an upper bound to generation. Otherwise, any non-negative integer is fine. Using a lower bound of 0—explicitly with choose/2

and implicitly with non_neg_integer/0—we make it possible that not all items will be included.

You'll note that in both branches of the case ... of expression, a generator is directly returned, and bound to the variable CountGen on line 8. This generator has no value yet, and is still just an abstract piece of data waiting to be processed by PropEr. This is why we actualize it in a ?LET macro on line 13, after which we can use it multiple times with a single fixed value.

On line 15, the count is used to generate a list of fixed-size (using vector/2), being added to the accumulated items, with the expected price calculated on line 16. The generator is then called recursively until the full list of items has been used.

This gives us a full generator of items that will not be in sufficient amounts to trigger a single special offer. We are free to implement the specials next.

Generating the List of Items on Special

The specials generator is a bit simpler by comparison, although it uses a similar recursive approach:

```
Line 1  special_gen(_, SpecialList) ->
        %% actually do not need the item list
        special_gen(SpecialList, [], 0).
        
5   special_gen([], Items, Price) ->
        {Items, Price};
    special_gen([{Item, Count, Cost} | SpecialList], Items, Price) ->
        %% Generate sequences of items equal to the special, based on a
        %% multiplier. If we have a need for 3 items for a special, we can
10      %% generate 0, 3, 6, 9, ... of such items at once
        ?LET(Multiplier, non_neg_integer(),
             special_gen(SpecialList,
                         ?LET(V, vector(Count * Multiplier, Item), V ++ Items),
                         Cost * Multiplier + Price)).
```

Elixir translation on page 212

The only conceptual difference takes place after line 11. Because a special being applied requires a specific number of items (Count), any multiple of that number is going to be fair. If we need 3 bagels to get a rebate, then lists of 0, 3, 6, or 9 bagels are all equally good here.

With this, we finally have everything we need to run the property and make it work!

Implementing Specials

So we've got a bit of a complex generator for specials, and a basic checkout implementation that does not consider specials at all. We've really got everything we need to get started on implementing specials. Our first step should be making sure our test actually fails eventually, as a sanity check to know it's good at detecting bad cases:

```
$ rebar3 proper
«build information and other properties»
===> Testing prop_checkout:prop_special()
......!
Failed: After 7 test(s).
{[[0],[6,1],[0],[0],[6,1],[6,1],[6,1],[0],[6,1],[6,1],[0],[6,1],[6,1],[6,1],
[6,1],[6,1],[6,1],[6,1],[6,1],[0],[6,1],[6,1],[6,1],[0],[6,1],[0],[0],[6,1],
[6,1],[6,1],[6,1],[0],[0],[0],[6,1],[6,1],[0]],20,[{[0],-2},{[6,1],1}],
[{[0],2,3},{[6,1],5,0}]}

Shrinking ..........(10 time(s))
{[[0],[0]],1,[{[0],0}],[{[0],2,1}]}
===>
2/3 properties passed, 1 failed
===> Failed test cases:
  prop_checkout:prop_special() -> false
```

The original counterexample is pretty much incomprehensible, but the shrunken one is simpler. The term `{[[0],[0]],1,[{[0],0}],[{[0],2,1}]}`, can be imagined to stand for `{[A, A], 1, [{A,0}], [{A,2,1}]}` if `[0]` is substituted with `A`—basically any list with a special being triggered causes a test failure. This is good, since that's exactly what we want to see tested.

With our test shown to catch failures, we can start implementing the feature.

A simple way to do it can be to count how many of each items are in the list. Account for the specials first, reducing the count every time the specials apply, and then run over the list of items that are not on sale. At a high level, it should look like this:

PropertiesDrivenDevelopment/erlang/checkout/src/checkout.erl

```
-module(checkout).

-export([total/3]).

-type item() :: string().
-type price() :: integer().
-type special() :: {item(), pos_integer(), price()}.

-spec total([item()], [{item(), price()}], [special()]) -> price().
total(ItemList, PriceList, Specials) ->
    Counts = count_seen(ItemList),
```

```
    {CountsLeft, Prices} = apply_specials(Counts, Specials),
    Prices + apply_regular(CountsLeft, PriceList).
```

Elixir translation on page 212

Here, count_seen/1 should create a list of each item and how many times it is seen. We then pass that data to apply_specials/2, which returns a tuple with two elements: the number of items not processed as part of a special on the left, and the summed up prices of all specials on the right. Finally, we take that sum, and add it to the cost of the rest of the items, as defined by apply_regular/2.

This method is kind of the opposite approach from the test; instead of generating both lists and mashing them together, this splits them up to get the final count.

Let's look at the helper functions, first with count_seen/2:

```
-spec count_seen([item()]) -> [{item(), pos_integer()}].
count_seen(ItemList) ->
    Count = fun(X) -> X+1 end,
    maps:to_list(
      lists:foldl(fun(Item, M) -> maps:update_with(Item, Count, 1, M) end,
                  maps:new(), ItemList)
    ).
```

Elixir translation on page 213

A Count function is defined and passed to maps:update_with/4, which allows us to increment a counter associated with each item's name as we iterate over ItemList. The map is then turned to a list for convenience.

The next function to write is apply_specials/2, which is a little bit trickier:

```
Line 1  -spec apply_specials([{item(), pos_integer()}], [special()]) ->
        -    {[{item(), pos_integer()}], price()}.
        -apply_specials(Items, Specials) ->
        -    lists:mapfoldl(fun({Name, Count}, Price) ->
      5          case lists:keyfind(Name, 1, Specials) of
        -            false -> % not found
        -                {{Name, Count}, Price};
        -            {_, Needed, Value} ->
        -                {{Name, Count rem Needed},
     10                  Value * (Count div Needed) + Price}
        -        end
        -    end, 0, Items).
```

Elixir translation on page 213

The core component here is the lists:mapfoldl/3 function, which both applies a map operation and a fold operation over a list in a single pass. The map

operation consists of applying a function to every value of the list and building a new list from them, and has its result stored on the left-hand side of every returned tuple. It is used to build a list of the form [{Name, Count},...], consisting of the items left to be processed once the specials' prices have been processed. The fold operation consists of looking at every element of the list and combining them into one accumulator, with its result stored on the right-hand side of every returned tuple. It is used here to sum up all of the specials prices that apply into a single total value.

Line 7 shows an item for which there are no specials: both the item count and the seen price remain unchanged. By comparison, line 9 shows a matching item with its count reduced by how many times a special matches, and line 10 increments the total price as expected.

The only function left is the one to apply the non-special prices, and that one is fortunately straightforward:

```
-spec apply_regular([{item(), integer()}], [{item(), price()}]) -> price().
apply_regular(Items, PriceList) ->
    lists:sum([Count * proplists:get_value(Name, PriceList)
               || {Name, Count} <- Items]).
```

Elixir translation on page 213

With this in place, you can give our test suite a spin, and even get some coverage metrics:

```
$ rebar3 do proper -c, cover -v
«build information and other properties»
===> Testing prop_checkout:prop_special()
.....................................................................
.....................
OK: Passed 100 test(s).
===>
3/3 properties passed

===> Performing cover analysis...
  |-----------------------|------------|
  |                module  |  coverage  |
  |-----------------------|------------|
  |              checkout  |      100%  |
  |-----------------------|------------|
  |                 total  |      100%  |
  |-----------------------|------------|
```

That's good! If we look at our list, we are now feature-complete.

- ~~items are identified by unique names (non-empty strings)~~

- ~~the list of items has no set length (though it appears non-empty), and the items it contains are in no specific order~~
- ~~the unit prices appear to be integers (not a bad idea, since floating point numbers lose precision; we can assume values are written in cents, for example)~~
- ~~the special prices are triggered only once the right number of items of the proper type can match a given special~~
- ~~all items have a unit price~~
- ~~it is not mandatory for a given item to have a special price.~~

And the 100% code coverage sure makes it sound like we can trust everything here. But can we really? Negative testing is an additional safety check we can add here.

Negative Testing

An interesting aspect of the code and tests we've written so far is that the list of items being bought is generated through the list of supported items. It means that by having designed the test case from the price list, there is a lack of tests looking for unexpected use cases. Our tests are positive, happy-path tests, validating that everything is right, and good things happen. Negative tests, by comparison, try to specifically exercise underspecified scenarios and finding what happens in less happy paths.

Our current properties are good; we don't want to modify them or make them more complex. Instead, we'll write a few broad properties that test more general properties of the code, and see if the requirements we had in the first place are actually all that consistent. On their own, broad (and often vague) properties are not too useful, but as a supplement or anchor to specific ones, they start to shine. And the broader the properties, the lower chances are that we'll be searching *only* for expected problems the way it is usually done with traditional tests.

Broad Properties

Let's start with a very, very broad property: the `checkout:total/3` function should return an integer and not crash:

```
prop_expected_result() ->
    ?FORALL({ItemList, PriceList, SpecialList}, lax_lists(),
        try checkout:total(ItemList, PriceList, SpecialList) of
            N when is_integer(N) -> true
        catch
            _:_ -> false
        end).
```

This introduces a new generator, lax_lists(), defined as broadly as possible at first:

```
lax_lists() ->
     {list(string()),                          % item list
      list({string(), integer()}),             % price list
      list({string(), integer(), integer()})}.  % specials list
```

Elixir translation on page 213

That kind of property, based on the types of inputs and outputs, is usually the kind of stuff where a type system could catch most errors. If you have any type analysis available (as we do in Erlang and Elixir with Dialyzer), it's best to leave that kind of issues to it. In fact, we do this here by using integer() as a generator rather than number()—it's no use sending it wrongly typed data when another tool will catch that if it happens. But even if type analysis can find plenty of issues, not all interesting errors will be found there. Case in point, running the property finds this:

```
«commands to run the property»
===> Testing prop_checkout:prop_expected_result()
.!
Failed: After 2 test(s).
{[[]],[],[]}

Shrinking (0 time(s))
{[[]],[],[]}
```

The PropEr representation of strings is lacking, but if a single item (with name "" shown as []) is used, then the lookups fail. The price of an unknown item cannot be used in the calculation, and the function bails out. All items being bought must also be within the price list, something rather hard to encode in a type system if the set of items is not known at compile-time.

This type of error is fine, but maybe we could give a clearer exception. We could raise a descriptive error instead, alerting the caller to the true nature of the problem. Let's first make the new exception acceptable to our property:

```
prop_expected_result() ->
    ?FORALL({ItemList, PriceList, SpecialList}, lax_lists(),
        try checkout:total(ItemList, PriceList, SpecialList) of
            N when is_integer(N) -> true
        catch
            error:{unknown_item, _} -> true;
            _:_ -> false
        end).
```

Elixir translation on page 213

And attach a similar patch that adds validation:

```
-spec apply_regular([{item(), integer()}], [{item(), price()}]) -> price().
apply_regular(Items, PriceList) ->
    lists:sum([Count * cost_of_item(Name, PriceList)
               || {Name, Count} <- Items]).

cost_of_item(Name, PriceList) ->
    case proplists:get_value(Name, PriceList) of
        undefined -> error({unknown_item, Name});
        Price -> Price
    end.
```

Elixir translation on page 214

Go and run the properties, and you'll find the bug fixed. In fact, nothing else is revealed by the property. Before calling victory and considering our code bug-free, though, we want to make sure we actually did our negative property testing right.

Calibrating Negative Properties

The easiest and best tools you have to check whether a property is good is always going to be gathering statistics. Let's take a look again at prop_expected_result(). This time, we'll look into the type of end result we get. Right now, the poperty has two valid cases: one where the list of items is all valid, and one where at least some item is missing from the price list and would fail. Let's see what kind of split we get by using collect/2:

```
prop_expected_result() ->
    ?FORALL({ItemList, PriceList, SpecialList}, lax_lists(),
        collect(
          item_list_type(ItemList, PriceList),
          try checkout:total(ItemList, PriceList, SpecialList) of
              N when is_integer(N) -> true
          catch
              error:{unknown_item, _} -> true;
              _:_ -> false
          end)).

item_list_type(Items, Prices) ->
    case lists:all(fun(X) -> has_price(X, Prices) end, Items) of
        true -> valid;
        false -> prices_missing
    end.

has_price(Item, ItemList) ->
    proplists:get_value(Item, ItemList) =/= undefined.
```

Run this and check the stats:

```
«other properties»
===> Testing prop_checkout:prop_expected_result()
.....................................................................
.....................
OK: Passed 100 test(s).

91% prices_missing
9% valid
```

Chances are you'll get something equally lopsided when trying it. The vast majority of all failing test cases only exercise the one failing case we have identified, the one where an item isn't in the price list.

Our negative property is depressing. If we represent the test space to explore with a spectrum—where one end contains tests about the happiest of all paths where everything goes according to plan, and the other end contains tests about the terrible cases where all input is garbage and nothing works—we get something that looks a bit like this:

```
[Perfect happy case] <-a--------------b------c-> [nothing works]
```

Our current positive properties are probably sitting around point `a` right now—pretty much everything we pass them is ideal—and our negative property is around `c`, choking on predictable garbage over 90% of the time. We've got very little coverage on the gradient in-between, where things are neither perfect nor wrong. To fix this, we will have to drag our negative properties somewhere closer to `b`.

One common trick to do this is to take our very lax generator, and make it a bit stricter. We can get that with a kind of hybrid approach where we not only generate entirely random items, but also purposefully put repeating predictible items in there:

PropertiesDrivenDevelopment/erlang/checkout/test/prop_checkout.erl

```
lax_lists() ->
    KnownItems = ["A", "B", "C"],
    MaybeKnownItemGen = elements(KnownItems ++ [string()]),
    {list(MaybeKnownItemGen),                          % item list
     list({MaybeKnownItemGen, integer()}),             % price list
     list({MaybeKnownItemGen, integer(), integer()})}. % specials list
```

Elixir translation on page 214

Try the prop_expected_result() property once again and check the results:

```
$ rebar3 proper
«build info and other properties»
===> Testing prop_checkout:prop_expected_result()
.........!
```

```
Failed: After 10 test(s).
{[[44,0,2,0],[66],[65],[1,3,6,5]],[{[6,6,0],-7}],[{[65],0,-1},
 {[66],3,8},{[2],-3,0},{[67],12,0}]}

Shrinking ......(6 time(s))
{[[65]],[],[{[65],0,0}]}
```

A new interesting bug is triggered whenever a list of specials requires exactly 0 items to work. This is, in fact, due to a division by zero when calculating totals. It's a bit surprising that none of our other properties ever encountered that case through all their random walks, but at least it is caught in one of them. Fixing it will require validating the specials. First the property, and then the code:

```
prop_expected_result() ->
    ?FORALL({ItemList, PriceList, SpecialList}, lax_lists(),
        try checkout:total(ItemList, PriceList, SpecialList) of
            N when is_integer(N) -> true
        catch
            error:{unknown_item, _} -> true;
            error:invalid_special_list -> true;
            _:_ -> false
        end).
```

Elixir translation on page 214

This handles the invalid specials list. Now for the actual code:

PropertiesDrivenDevelopment/erlang/checkout/src/checkout.erl

```
-module(checkout).
-export([valid_special_list/1, total/3]).

-type item() :: string().
-type price() :: integer().
-type special() :: {item(), pos_integer(), price()}.

-spec valid_special_list([special()]) -> boolean().
valid_special_list(List) ->
    lists:all(fun({_,X,_}) -> X =/= 0 end, List).

-spec total([item()], [{item(), price()}], [special()]) -> price().
total(ItemList, PriceList, Specials) ->
    valid_special_list(Specials) orelse error(invalid_special_list),
    Counts = count_seen(ItemList),
    {CountsLeft, Prices} = apply_specials(Counts, Specials),
    Prices + apply_regular(CountsLeft, PriceList).
```

Elixir translation on page 214

The new function is valid_special_list/1, which checks that all list terms have 3-tuples and that the middle value is not a 0. This is something Dialyzer could

already handle in your code, but it wouldn't necessarily detect if the data were coming from a database. Then on line 13, we integrate the function into the regular workflow.

This patches the test up. How do we know we've covered everything with our negative tests now? We don't. We could try to play with metrics again to see what we could improve, but there's another approach that can work well: relaxing constraints further.

Relaxing Constraints

While metrics are always a good thing to keep an eye on, another very interesting way to improve our negative tests and explore the program's problem space is to play with constraints, and relax them with existing generators. Let's revisit our TODO list. The items in *italics* are properties or assumptions we made about the system that we may want to play with:

- items are identified by *unique names* (non-empty strings)
- the list of items has no set length (though it appears non-empty), and the items it contains are in no specific order
- the *unit prices appear to be integers* (not a bad idea, since floating point numbers lose precision; we can assume values are written in cents, for example)
- the special prices are triggered only once the right number of items of the proper type can match a given special
- all *items have a unit price*
- it is not mandatory for a given item to have a special price.

The way we relax constraints is usually through simple code modifications. Check your working code into source control, and do most of the changes right in place. Modify a bunch of generators, making them less strict so they trigger some unexpected case. Find out why that happened, revert the change, and then either add a unit test or a property test to validate the bug before fixing it.

Then we can rinse and repeat, gradually weeding out more and more bugs out of our code.

Let's start with the first one, checking what happens when item names are not unique in the price list, which we can do by changing the last line of the price_list() generator:

```
%% generate a list of {ItemName, Price} to configure the checkout
price_list() ->
    ?LET(PriceList, non_empty(list({non_empty(string()), integer()})),
```

```
        lists:keysort(1, PriceList)). % allow duplicates
```

Here we replaced lists:ukeysort/2 with lists:keysort/2, keeping similar semantics but without removing duplicates. Run the properties again and you should see a failure:

```
«build information»
===> Testing prop_checkout:prop_no_special1()
.....................!
Failed: After 25 test(s).
{[[4],[4],[7,9,9,4,1,3,5,12,9],[4],[4],[4],[4],[4],[5,3,3]],-13,
[{[2,2,53,3,1,0,29,3,0],4},{[4],-4},{[4],1},{[5,3,3],-3},{[5,5,3,8,40],0},
{[6,0],-7},{[7,9,9,4,1,3,5,12,9],-2},{[12,3,5,14,16,2,4],-5}]}

Shrinking ..........(10 time(s))
{[[4],[4],[4],[4],[4],[4],[4],[4],[4]],1,[{[4],0},{[4],1}]}
«other properties»
3/4 properties passed, 1 failed
===> Failed test cases:
  prop_checkout:prop_no_special1() -> false
```

Unsurprisingly, it appears that whenever a list of prices contains two identical items ([4] with both a price of 0 and 1), our system gets confused and dies. We don't support duplicates in the price list, and our generator implementation aligned itself with that fact. We baked the uniqueness assumption into our model, but did not necessarily expose that to our users, nor did we test for it explicitly. This could lead to problems. For example, someone might send malformed price lists, only to then open a support ticket once they discover the prices are wrong at checkout, and they'd be right to do so. It would in fact be nicer to let users know if the item list they submitted is valid without needing to buy anything through the checkout.

Let's revert the generator change, and add a property to replicate that behavior:

PropertiesDrivenDevelopment/erlang/checkout/test/prop_checkout.erl

```
prop_dupe_list_invalid() ->
    ?FORALL(PriceList, dupe_list(),
        false =:= checkout:valid_price_list(PriceList)).

dupe_list() ->
    ?LET(Items, non_empty(list(string())),
         vector(length(Items)+1, {elements(Items), integer()})).
```

Elixir translation on page 215

The dupe_list() generator works by generating a random list of item names, and then using it to generate the price list. By asking for more price list entries

than there are item names—that's what using vector(length(Items)+1, ...) accomplishes—we're guaranteeing duplicate entries.

You'll note that the property does not check against the checkout:total/3 function, but against a valid_special call that we'll add to the implementation module, matching what we did earlier for the specials list validation:

PropertiesDrivenDevelopment/erlang/checkout/src/checkout.erl

```
-export([valid_price_list/1, valid_special_list/1, total/3]).

-spec valid_price_list([{item(), price()}]) -> boolean().
valid_price_list(List) ->
    length(List) =:= length(lists:ukeysort(1, List)).
```

And as with the earlier case, we should also wire it into the total/3 call, just to be thorough, on line 3:

```
Line 1 -spec total([item()], [{item(), price()}], [special()]) -> price().
  2 total(ItemList, PriceList, Specials) ->
  3     valid_price_list(PriceList) orelse error(invalid_price_list),
  4     valid_special_list(Specials) orelse error(invalid_special_list),
  5     Counts = count_seen(ItemList),
  6     {CountsLeft, Prices} = apply_specials(Counts, Specials),
  7     Prices + apply_regular(CountsLeft, PriceList).
```

Elixir translation on page 215

Run the tests again and see what happens:

```
«build information»
===> Testing prop_checkout:prop_expected_result()
........!
Failed: After 9 test(s).
{[[67],[8],[2,3,1]],[{[67],-2},{[67],-1}],[{[67],-3,3}]}

Shrinking .......(7 time(s))
{[],[{[67],0},{[67],0}],[]}
«other properties»
4/5 properties passed, 1 failed
===> Failed test cases:
  prop_checkout:prop_expected_result() -> false
```

Our prop_expected_result() property fails again, this time, because of the new exception we added. It turns out that this property would sometimes generate the right kind of inputs to trigger that case, but did not know about the business rules well enough to know it a wasn't valid thing to do.

Fuzzing vs. Properties

This perfectly highlights the distinction between fuzzing—generating garbage input to see if the program fails—compared to property-based testing, where we check that the program behaves the right way given all kinds of inputs. Both scanning largely with negative tests and relaxing constraints find interesting but distinct results.

We can fix the failing test by adding a specific exception handler for the one failing case:

PropertiesDrivenDevelopment/erlang/checkout/test/prop_checkout.erl

```
prop_expected_result() ->
    ?FORALL({ItemList, PriceList, SpecialList}, lax_lists(),
        try checkout:total(ItemList, PriceList, SpecialList) of
            N when is_integer(N) -> true
        catch
            error:{unknown_item, _} -> true;
            error:invalid_price_list -> true;
            error:invalid_special_list -> true;
            _:_ -> false
        end).
```

Elixir translation on page 215

And while we're at it, add a property to deal with duplicates in the specials list:

PropertiesDrivenDevelopment/erlang/checkout/test/prop_checkout.erl

```
prop_dupe_specials_invalid() ->
    ?FORALL(SpecialList, dupe_special_list(),
        false =:= checkout:valid_special_list(SpecialList)).

dupe_special_list() ->
    ?LET(Items, non_empty(list(string())),
         vector(length(Items)+1, {elements(Items), integer(), integer()})).
```

Elixir translation on page 216

With the matching code to fix things:

PropertiesDrivenDevelopment/erlang/checkout/src/checkout.erl

```
-spec valid_special_list([special()]) -> boolean().
valid_special_list(List) ->
    lists:all(fun({_,X,_}) -> X =/= 0 end, List) andalso
    length(List) =:= length(lists:ukeysort(1, List)).
```

Elixir translation on page 216

Run the properties and you'll see that they all pass. You could still dig for more bugs; here, we found three bugs by relaxing only one of them, and relaxing more properties would likely find more interesting bugs:

- Not handling unit prices for some items (or specials) creates unexpected crashes; more interestingly, replacing integers with numbers (specifically floats) yield multiple failures because the current implementation uses `rem` and `div`, two integer-specific operators
- Using negative numbers for prices would technically mean we credit people rather than charging them to walk away with items, and currently we don't validate for that
- Passing in non-numeric values in the price list or specials list is considered valid by the code but can't logically work

Given how vague the specification was, we probably would have to discuss these discoveries with stakeholders to figure out what is acceptable or not before the code hits production. A strict interpretation of the specification will mean our current implementation is sufficient. A lax one will cause explosions for multiple cases that may or may not be preposterous.

Of course, in a statically typed language (or in the case of Erlang and Elixir, with Dialyzer's type analysis), a strict interpretation of the spec is the only one accepted. The potential bugs we could discover through relaxing the properties above do not even register as a possibility with type analysis, and the current state of affairs is very likely acceptable.

In general, when strict assumptions are made and are being enforced by manual checks, the compiler, and/or static code analysis, then the program should not get into unexpected states. It may be inflexible and frustrating for the user, but it will be less likely to go wrong. At least not on the bugs that may be preventable through type analysis.

Wrapping Up

Through properties-driven development, we've extrapolated properties from a vague spec, testing our happy paths in a test-first approach. Code coverage, while a useful metric to show code is tested, is not a great one to assess test quality. Instead, you've seen how negative testing in a fuzzing-inspired approach can help, as well as how playing with properties by relaxing constraints can uncover all kinds of bugs and underspecifications that could prove problematic.

All of this has been done while going through the most complex generators this book contains. You should now have a pretty good idea of what can and

cannot be done well with stateless properties. Most of your unit testing needs should now be covered, in fact. You can go ahead and try a bunch of that stuff in your own projects. As you get trickier and trickier generators, you may find them hard to debug. If so, the next chapter should have you covered, as it discusses *Shrinking*.

CHAPTER 7

Shrinking

A critical component of property-based testing is shrinking—the mechanism by which a property-based testing framework can be told how to simplify failure cases enough to let us figure out exactly what the minimal reproducible case is. While finding complex obtuse cases is worthwhile, being able to reduce failing inputs (data generated) to a simple counterexample truly is the killer feature here. But there are some cases where what PropEr does is not what we need. Either it can't shrink large datastructures well enough to be understandable, or it's not shrinking them way we'd want it to. In this chapter we'll see two ways to handle things: the `?SHRINK` and the `LETSHRINK` macros, which let us give the framework hints about what to do.

But first, we have to see how shrinking works at a high level. In general you can think of shrinking as the framework attempting to bring the data generator closer to its own zero point, and successfully doing so as long as the property fails. A "zero" for a generator is a bit arbitrary, but if you played with the default generators a bit by calling `proper_gen:sampleshrink/1` on them in the shell, you may notice that:

- A number tends to shrink from floating point values towards integers, and integers tend to shrink towards the number 0 (floating point numbers themselves shrink towards 0.0)
- binaries tend to shrink from things full of bytes towards the empty binary (`<<>>`)
- lists tend to shrink towards the empty list
- `elements([A,B,C])` will shrink towards the value `A`

In short, data structures that contain other data tend to empty themselves, and other values try to find a neutral point. The nice aspect of this is that as custom generators are built from other generators, the shrinking is "inherited", and a custom generator may get its own shrinking for free: a map full of

people records made of strings and numbers will see the strings get shorter and simpler, the numbers will get closer to zero, and the map will get fewer and fewer elements, until only the components essential to trigger a failure are left.

For some data types however, there is no good zero point: a vector of length 15 will always have length 15, and same with a tuple. Similarly, larger recursive data structures that have been defined by the user may not have obvious ways to shrink (such as probabilistic ones), or may require shrinking towards other values than the default one for a generator. Some examples of special shrinking points could be things such as a chessboard, which is at its neutral point not when it is empty, but when it is full, with all its pieces in their initial positions. Similarly, in chemistry, pH[1] has a neutral value of 7. In a database, an interesting boundary position is usually triggered when records reach 4 kilobytes—a common minimal page size[2] for computer memory.

For such cases, even if they are relatively rare, the `?SHRINK` macro might be what you need.

Re-centering With ?SHRINK

`?SHRINK` is conceptually the simplest of the two macros that can be used to impact shrinking. It is best used to pick a custom "zero" point towards which PropEr will try to shrink data. You can do this mainly by giving the framework a normal generator for normal cases, and then suggesting it uses other simpler generators whenever an error is discovered.

The macro takes the form `?SHRINK(DefaultGenerator, [AlternativeGenerators])` in Erlang, and `shrink(default_generator, [alternative_generators])` in Elixir. The `DefaultGenerator` will be used for all passing tests. Once a property fails however, `?SHRINK` lets us tell PropEr that any of the alternative generators in the list are interesting ways to get simpler relevant data. We can give *hints* about how the framework should search for failures, basically. And if the alternative generators are not fruitful, so be it, the shrinking will continue in other ways until no progress can be made.

To make things practical, if we are generating timestamps or dates, we may be interested in including years between 0 to 9999 to make sure we cover all kinds of weird cases. However, if you know that the underlying implementation

1. https://en.wikipedia.org/wiki/PH
2. https://en.wikipedia.org/wiki/Page_(computer_memory)#Multiple_page_sizes

of your system uses unix timestamps, then you should consider that its epoch (starting time) is on January 1st 1970. Since January 1st 1970 is the underlying system's zero value, picking 1970 as a shrinking target makes more sense than the literal year 0 (particularly since year zero is not necessarily a valid concept[3] in the first place).

Let's take a look at the following set of generators used to create strings of the form "1997-08-04T12:02:18-05:00", in accordance with the ISO 8601[4] standard. This set of generators will be centering its shrinking efforts towards January 1st 1970. One thing you should note is that the standard is somewhat lax and allows (or rather, does not forbid) the notation of a timezone that is +99:76, lagging about 4 days behind standard time, even if that is nonsensical from a human perspective.

Shrinking/erlang/pbt/test/prop_shrink.erl

```
-module(prop_shrink).
-include_lib("proper/include/proper.hrl").
-compile([export_all, {no_auto_import,[date/0]}, {no_auto_import,[time/0]}]).

strdatetime() ->
    ?LET(DateTime, datetime(), to_str(DateTime)).

datetime() ->
    {date(), time(), timezone()}.

date() ->
    ?SUCHTHAT({Y,M,D}, {year(), month(), day()}, calendar:valid_date(Y,M,D)).

year() ->
    ?SHRINK(range(0, 9999), [range(1970, 2000), range(1900, 2100)]).

month() ->
    range(1, 12).

day() ->
    range(1, 31).

time() ->
    {range(0, 24), range(0, 59), range(0, 60)}.

timezone() ->
    {elements(['+', '-']),
     ?SHRINK(range(0, 99), [range(0, 14), 0]),
     ?SHRINK(range(0, 99), [0, 15, 30, 45])}.

%% Helper to convert the internal format to a string
```

3. https://en.wikipedia.org/wiki/Year_zero
4. https://en.wikipedia.org/wiki/ISO_8601

```
- to_str({{Y,M,D}, {H,Mi,S}, {Sign,Ho,Mo}}) ->
-     FormatStr = "~4..0b-~2..0b-~2..0bT~2..0b:~2..0b:~2..0b~s~2..0b:~2..0b",
30    lists:flatten(io_lib:format(FormatStr, [Y,M,D,H,Mi,S,Sign,Ho,Mo])).
```

Elixir translation on page 216

As you can see, various generator functions call the ?SHRINK macro. First, the year() generator on line 11 uses range(0,9999) as its default generator. This covers all thousand or so years we are interested in. The alternative generators for the macro are range(1970,2000) and range(1900,2100), which means that if some generated year causes a property to fail, rather than trying years such as 73 or 8763, PropEr will try years closer to the epoch, like 1988 or 2040. ?SHRINK lets us narrow PropEr's search space down significantly to get relevant results faster.

Similarly, the timezone() generator will look for values between 0 and 99, but in any failure case, will try to settle between 0 and 14 when possible, which are ranges that we humans find more reasonable. Similarly the minutes offsets will try to match currently standard offsets of 0, 15, 30, or 45 minutes.

And as you can see, generators that use ?SHRINK can be used as any other one; the macro is adding some metadata to the underlying structure representing a generator, so they remain entirely composable. From anybody else's point of views, it's a generator as any other one.

You can see shrinking in action by calling proper_gen:sampleshrink/1 in the shell, and PropEr will generate sequences of more and more aggressive shrinks:

```
1> proper_gen:sampleshrink(prop_shrink:strdatetime()).
"1757-06-26T02:36:60-64:38"
"1995-06-26T02:36:60-64:38"
"1970-06-26T02:36:60-64:38"
"1970-01-26T02:36:60-64:38"
"1970-01-01T02:36:60-64:38"
"1970-01-01T00:36:60-64:38"
"1970-01-01T00:00:60-64:38"
"1970-01-01T00:00:00-64:38"
"1970-01-01T00:00:00+64:38"
"1970-01-01T00:00:00+09:38"
"1970-01-01T00:00:00+00:38"
"1970-01-01T00:00:00+00:00"
ok
```

Elixir's equivalent to proper_gen:sample_shrink/1 is instead PropCheck.sample_shrink(PbtTest.strdatetime()), which provides the same functionality but with a more idiomatic format.

In practice, shrinking will be done less linearly than this. A given attempt at shrinking that fails to create another failing case won't be used. That means that if a test failed on every month of July, the shrinking would end up looking like "1970-07-01T00:00:00+00:00".

That is as complex as ?SHRINK gets. For a lot of cases, you may find yourself using other generators like element([A,B,C,...,Z]), which will do something kind of equivalent by shrinking towards A rather than Z. When element/1 no longer suffices, then ?SHRINK becomes real interesting. The one thing that it won't necessarily help is with huge giant chunks of data that are hard to reduce to smaller counterexamples. That is where ?LETSHRINK shines.

Dividing With ?LETSHRINK

As you use PropEr, you will possibly find yourself stuck with generators creating huge data structures which take a long time to shrink and often don't give very interesting results back. This will often happen when some very low-probability failure is triggered, meaning that the framework had to generate a lot of data to find it, and will have limited chances of shrinking things in a significant manner.

Whenever that happens, the ?LETSHRINK([Pattern, ...], [Generator, ...], Expression) is what we need. In practice, you use the generator like this:

```
?LETSHRINK([A,B,C], [list(number()), list(number()), list(number())],
           A ++ B ++ C)
```

Elixir translation on page 217

The macro looks a lot like a regular ?LET macro, but with a few constraints: the first two arguments must always be lists, and the third argument is an operation where all list elements get combined into one. Here, A, B, and C are three lists filled with integers, and A ++ B ++ C is just a bigger list of integers. The important part is that any of A, B or C can be used by the program instead of A ++ B ++ C.

The reason for that is that once a property fails and that PropEr tries to shrink the data set, it will instead pick just one of A, B, or C *without applying the transformation* and return that directly. ?LETSHRINK is particularly appropriate for recursive structures, data made through branching, and just all kinds of pieces of data that are generated by smashing others together and applying transformations, since taking a part of it is a legitimate way to get a simpler version.

Basically, we're giving PropEr a way to divide the data up to isolate a failing subset more efficiently.

The most common form of ?LETSHRINK is the one you'd use on tree data structures. For a binary tree generator of size N, we'd write something like:

```
tree(N) when N =< 1 ->
    {leaf, number()};
tree(N) ->
    PerBranch = N div 2,
    {branch, tree(PerBranch), tree(PerBranch)}.
```

Elixir translation on page 217

If you run sampleshrink/1 on it, you'll find out that the elements within the tree shrink, but the tree itself stays the same size:

```
1> proper_gen:sampleshrink(prop_shrink:tree(4)).
{branch,{branch,{leaf,13},{leaf,0.6154862580810709}},
        {branch,{leaf,8},{leaf,-3}}}
{branch,{branch,{leaf,0},{leaf,0.6154862580810709}},
        {branch,{leaf,8},{leaf,-3}}}
{branch,{branch,{leaf,0},{leaf,-6}},{branch,{leaf,8},{leaf,-3}}}
{branch,{branch,{leaf,0},{leaf,0}},{branch,{leaf,8},{leaf,-3}}}
{branch,{branch,{leaf,0},{leaf,0}},{branch,{leaf,0},{leaf,-3}}}
{branch,{branch,{leaf,0},{leaf,0}},{branch,{leaf,0},{leaf,0}}}
```

Each of the trees in the sample contain exactly 4 entries: all the integers tend towards 0, but the structure itself has a fixed size. The obvious way to get the tree to shrink is through parametrizing it with the Size variable obtained from the ?SIZED(Var, Exp) macro. However, if the failure requires internal tree elements to remain large while the tree structure itself is small, then there's fewer chances that shrinking by size only would work well.

Instead, if we use ?LETSHRINK, we can get a much shrink-friendly version:

```
tree_shrink(N) when N =< 1 ->
    {leaf, number()};
tree_shrink(N) ->
    PerBranch = N div 2,
    ?LETSHRINK([L, R], [tree_shrink(PerBranch), tree_shrink(PerBranch)],
               {branch, L, R}).
```

Elixir translation on page 217

The leaf clause is left unchanged. However, the inner-node (branch) clause is modified. Instead of generating {branch, Left, Right} right away, we generate both the left and right side in a list within ?LETSHRINK. In the third argument, we

assemble both parts within the branch tuple. Effectively, we take the same exact approach, but with the macro as a layer of indirection.

This is a small change, but it has a large impact. We can now start from larger trees to initially find bugs, without losing clarity when getting good counterexamples:

```
2> proper_gen:sampleshrink(prop_shrink:tree_shrink(16)).
{branch,{branch,{branch,{branch,{leaf,28},{leaf,-0.039220389013186946}},
                        {branch,{leaf,3.9013940456284684},{leaf,-3}}},
                {branch,{branch,{leaf,14.576812882147989},{leaf,-16}},
                        {branch,{leaf,-2.0345272435474966},
                                {leaf,-8.151564195158691}}}},
        {branch,{branch,{branch,{leaf,-12},{leaf,-85}},
                        {branch,{leaf,16.829645166380576},{leaf,-1}}},
                {branch,{branch,{leaf,1},{leaf,5.058669843388856}},
                        {branch,{leaf,7},{leaf,2}}}}}
{branch,{branch,{branch,{leaf,28},{leaf,-0.039220389013186946}},
                {branch,{leaf,3.9013940456284684},{leaf,-3}}},
        {branch,{branch,{leaf,14.576812882147989},{leaf,-16}},
                {branch,{leaf,-2.0345272435474966},
                        {leaf,-8.151564195158691}}}}
{branch,{branch,{leaf,28},{leaf,-0.039220389013186946}},
        {branch,{leaf,3.9013940456284684},{leaf,-3}}}
{branch,{leaf,28},{leaf,-0.039220389013186946}}
{leaf,28}
{leaf,0}
```

Rather than a constant tree size, each shrink subsequently makes the tree smaller (if the property still fails). If you pay close attention, you'll also notice that the values within the tree are initially *not modified*. Therefore, if the tree size is at play, the tree will remain large by failing to shrink that way, telling PropEr to instead try with its contents. If the bug is due to contents before structure and the size does not matter, we'll know rapidly as well. We're giving a good search stategy to the framework here by telling it how the recursive aspects of our generators work.

In general, for a framework like PropEr or Quickcheck, adding shrinking instructions is not something that needs to be done as part of writing the generator the first time around. Instead, it's something that will be worth doing once a confusing counter-example is found and the minimal counter-example given by the framework is not understandable on its own.

As with other property-based testing debugging practices, improving generators is likely going to be iterative and a bit explorative. The generator, its shrinking, the test cases, and our understanding of the program itself will all get to

improve as we discover the hidden properties embedded in the code that was written.

Wrapping Up

We've been over what is pretty much an optimization when we have large counterexamples that PropEr does not necessarily know how to handle. By using the ?SHRINK macro, you can let it know how to re-target shrinking towards more meaningful neutral values for a given generator. Then you've seen that with ?LETSHRINK, we can give PropEr tips on how to divide up a data structure to find problems with more ease.

With shrinking under your belt, you now have pretty much all of the tools you will need to handle stateless property tests. In fact, we're almost done with stateless properties. Right now, the only thing left to see is a brand new PropEr feature called *Targeted Property-Based Testing*, which at this point will just be a bonus for some amazing flexibility in property testing.

Exercises

Question 1

What are the two macros used for shrinking and what do their arguments stand for?

Question 2

What are the differences between the ?LETSHRINK macro and the ?LET macro?

Question 3

In the following property, a list of servings for a meal is generated. If any serving contains dairy, the property fails:

```
prop_too_much_dairy() ->
    ?FORALL(Food, meal(), dairy_count(Food) =:= 0).

dairy_count(L) ->
    length([X || X <- L, is_dairy(X)]).

is_dairy(cheesesticks) -> true;
is_dairy(lasagna) -> true;
is_dairy(icecream) -> true;
is_dairy(milk) -> true;
is_dairy(_) -> false.
```

The generator looks like this:

```
meal() ->
    ?LETSHRINK([Appetizer, Drink, Entree, Dessert],
```

```
    [elements([soup, salad, cheesesticks]),
     elements([coffee, tea, milk, water, juice]),
     elements([lasagna, tofu, steak]),
     elements([cake, chocolate, icecream])],
    [Appetizer, Drink, Entree, Dessert]).
```

Elixir translation on page 217

However, whenever the test case fails, we instead get a result set that always contain the four courses. Fix the `?LETSHRINK` usage in the generator so that data can be appropriately shrunk.

Question 4

In *Writing the Generator*, on page 115, the following generator was introduced to create lists of items with expected prices and specials:

```
item_price_special() ->
    %% first LET: freeze the PriceList
    ?LET(PriceList, price_list(),
        %% second LET: freeze the SpecialList
        ?LET(SpecialList, special_list(PriceList),
            %% third LET: Regular + Special items and prices
              ?LET({{RegularItems, RegularExpected},
                    {SpecialItems, SpecialExpected}},
                   {regular_gen(PriceList, SpecialList),
                    special_gen(PriceList, SpecialList)},
                   %% And merge + return initial lists:
                   {shuffle(RegularItems ++ SpecialItems),
                    RegularExpected + SpecialExpected,
                    PriceList, SpecialList}))).

shuffle(L) ->
    %% Shuffle the list by adding a random number first,
    %% then sorting on it, and then removing it
    Shuffled = lists:sort([{rand:uniform(), X} || X <- L]),
    [X || {_, X} <- Shuffled].
```

Elixir translation on page 218

This generator merges two types of data: those without a special, and those with a special. The two types are joined together into one larger item list with prices, which can then be passed to the property.

Modify the generator so that both types of pricing can shrink independently.

CHAPTER 8

Targeted Properties

Content to be supplied later.

Part III

Stateful Properties

Now we're cooking. Property-based testing becomes really amazing once you deal with testing complex interactions with stateful system, and this part of the book will show you everything you need to be comfortable with these advanced features.

CHAPTER 9

Stateful Properties

Most of the amazing stories of property-based testing—those that make you go *holy crap I need to get in on this*—involve large and complex stateful systems where tricky bugs are found with a relatively tiny test. Those usually turn out to be stateful properties.

Stateful property tests are particularly useful when "what the code should do"—what the user perceives—is simple, but "how the code does it"—how it is implemented—is complex. A large amount of integration and system tests fit this description, and stateful property-based testing will become one of the most interesting tools to have in your toolbox since it can exercise major parts of your systems with little code.

Stateful property tests are a non-formal variation on model checking,[1] something a bit fancier than the modelling approach we used in stateless properties. The core concept is that you must define a (mostly) predictable model of the system, and then use PropEr to generate a series of commands that represent operations that can be applied to the system. PropEr then runs these operations on both the model and the actual system, and compares them to each other. If they agree, the test passes; if they disagree, it fails.

This chapter will cover the basic structure of properties, and expand on how exactly PropEr executes them. This will be critical to understanding how to write your own stateful properties, which we'll explore by testing a cache implementation. As a bonus, we'll see how stateful properties can be used to find concurrency bugs. We'll have to start with a bit of theory first, to prevent some major confusion.

1. https://en.wikipedia.org/wiki/Model_checking

Laying Out Stateful Properties

Stateful properties all have a few parts in common, with a bit more scaffolding than what we had in stateless ones. It's important to keep these parts in mind, because they'll be interacting with each other for all test executions. The three major components are:

- a model, which represents what the system should do at a high level
- a generator for commands, which represent the execution flow of the program
- an actual system, which is validated against our model

The Model

A core part of a stateful property is the model. It represents a simple and straightforward version of what our actual system should be doing. By ensuring that the real system behaves like the model does, we show that our programs are most likely correct. The model itself is made of two important parts:

1. A data structure that represents the expected state of the system: the data it should contain and that you'd expect to be able to get from it
2. A function that transforms the model's state based on commands that could be applied to the system (named `next_state`)

This sounds a bit tricky, but an example should help. Let's say we have a web service where we can upload files, which get to be replicated in multiple datacenters. There are a lot of complex operations taking place, and a command like `upload_file(Name, Contents)` involves the network, various computers, and a bunch of protocols with multiple data representations. There are lots of places where things can go wrong.

Our model, by comparison, could have its data structure (the first important part) be a map of the form `#{Name => Contents}` that represents what files the service should know about. The `next_state` function that transforms the model state (the second important bit) would just be a function that adds a `(Name, Contents)` pair to the map based on the command, something like `next_state(Map, [Name, Contents]) -> Map#{Name => Contents}`.

That's it. We have a very simple abstract representation of what the system should do, and that's a model!

The Commands

The next core part is a bunch of commands that can be generated by the framework. These represent operations that can be run against the actual system. Their generation has two components:

1. A list of potential *symbolic calls*, with generators defining their arguments
2. A series of functions that defines whether a given symbolic call would make sense to apply according to the current model state.

The first point is rather simple. We've seen symbolic calls in Custom Generators on page 67. They are tuples that represent function calls of the form `{call, Module, Function, [Arg1, Arg2, ..., ArgN]}`. All the arguments can use regular PropEr generators, and the function calls map to actual system calls. So in our example file upload, we may have something like `{call, actual_system, upload_file, [Name, Contents]}`.

The second point is interesting. The functions that validate if a call is acceptable in a sequence are called *preconditions*, and they define invariants that should hold true in the system for the current test. For example, it is possible that an ATM only exposes functionality to deposit money if a debit card is inserted; a precondition of depositing money would therefore be that a valid debit card is currently in the ATM. You can think of preconditions are the stateful properties' equivalent to `?SUCHTHAT` macros in stateless properties.

The Validation

Finally, we have the validation of the system against the model. This is done through *postconditions*, which are invariants that should hold true after a given operation has been applied. They are how we check that things are right.

For example, if our model's state for an ATM says a user with the PIN "1984" has inserted their card, and that the last operation against the real system was the user typing in the password "2421", then the postcondition would validate that the actual system properly returns a failure to log in. Such postcondition validation can be done by checking global invariants that are expected to always be true, but also frequently takes place by comparing the system's output with the expected result based on the model state. In this example, we did the latter: it does not matter what the actual system stores as a PIN or how it stores it. We just want it to return what our model says it should.

We'll get to put that into practice real soon, but in a nutshell, remember that:

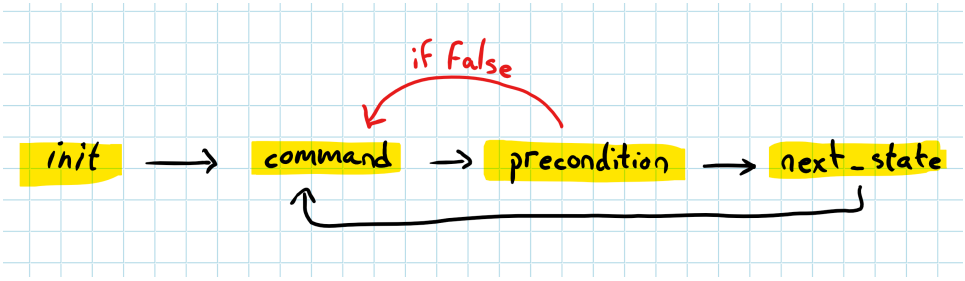
- we need a model, which is comprised of some data (the state) that gets modified by a next_state function
- we need a bunch of symbolic calls that represent operations that can be applied to the real system, and which can be constrained with preconditions
- we need to ensure that the results from the actual system match those we would expect from our model, which is done through postconditions

With this in mind, we can go figure out how PropEr is going to line all of those things up.

How Stateful Properties Run

PropEr divides the execution of a stateful test in two phases, one abstract and one real. The abstract phase is used to create a test scenario, and is executed without any code from your actual system running. Its whole objective is to take the model and command generation callbacks, and call these to build out the sequence of calls that will later be applied to the system.

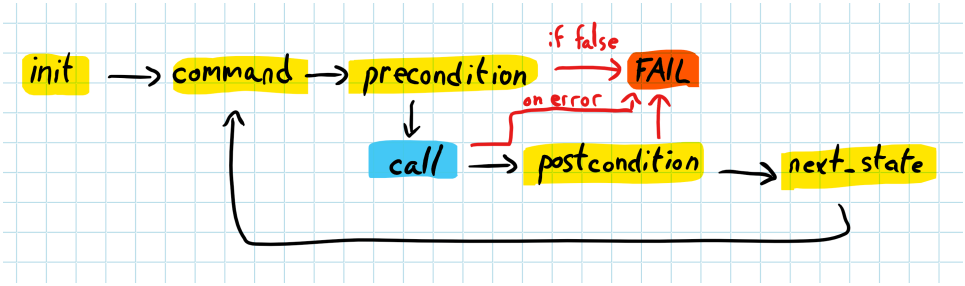
Put visually, it looks a bit like this:



Note the lack of postcondition or calls to the actual system. In the abstract mode, a command generator creates a symbolic call with its arguments based on an initial model state. PropEr then applies the preconditions to that command to know if it would be valid. If the validation fails, PropEr tries again with a new generated command. Once a suitable command is found, we can move forward. The next_state function takes the command and the current state, and has to return a new state data structure. Then the whole process is repeated over and over, until PropEr decides it has enough commands.

Once this is done, we're left with a valid and legitimate sequence of commands, with all its expected state transitions. Our model is ready.

With our model in hand, PropEr can start applying the commands to the real system, and our postconditions can check that things all remain valid:



The execution is repeated, except that now, at every step of the way, PropEr also runs the commands against the real system. The preconditions are still re-evaluated to ensure consistency so that if a generated precondition that used to work suddenly fails, the entire test also fails. The next symbolic call in the list is executed, with its result stored. The postcondition is then evaluated, and if it succeeds, the state transition for the command is applied to the state and the next command can be processed.

In case of a failure, shrinking is done by modifying the command sequence as required, mostly by removing operations and seeing if things still work. Preconditions will be used by the framework to make sure that the various attempts are valid.

So with all of this theory, we can start putting it all in practice, and seeing what the implementation looks like.

Writing Properties

In earlier chapters, you've seen that stateless properties all follow a pretty similar structure. The layout of code around files may vary from project to project, but overall, most test suites do share a separation between properties, generators, and helper functions. When it comes to stateful properties, there is far less of a standard: some people will put properties in one file, models in another one, with helper functions and wrappers around the actual system in a third module. Some developers will rather prefer to have everything in one spot.

In this book, we'll stick to having the properties and the model in one file. As with basic properties, we can make use of the rebar3 plugin's templating facilities to get the file we need within any standard Erlang project. Call the following within an existing project:

```
$ rebar3 new proper_statem base
===> Writing test/prop_base.erl
```

The generated file contains the prop_base module, a test suite that is divided in two sections: one section for the stateful property we will want to execute, and one for the model, which is a mix of callbacks and generators. Let's start by looking at the property:

```
Line 1  -module(prop_base).
        -include_lib("proper/include/proper.hrl").
        -
        %% Model Callbacks
      5 -export([command/1, initial_state/0, next_state/3,
        -         precondition/2, postcondition/3]).
        -
        %%%%%%%%%%%%%%%%%%
        %%% PROPERTIES %%%
     10 %%%%%%%%%%%%%%%%%%
        prop_test() ->
            ?FORALL(Cmds, commands(?MODULE),
                    begin
                        actual_system:start_link(),
     15                 {History, State, Result} = run_commands(?MODULE, Cmds),
                        actual_system:stop(),
                        ?WHENFAIL(io:format("History: ~p\nState: ~p\nResult: ~p\n",
                                            [History,State,Result]),
                                  aggregate(command_names(Cmds), Result =:= ok))
     20             end).
```

Elixir translation on page 219

This looks similar to standard properties, but with a few differences. We do have a bunch of model callbacks to export, but that's expected. The change starts in the property itself. First, the generator is the commands/1 function on line 12. This is a generator automatically imported by PropEr, which calls the model functions to create the command sequence that will be used. This includes the symbolic execution only.

The commands will then be run against the real system on line 15, which is where the real execution (with validation of postconditions) takes place.

On lines 14 and 16, we have functions having to do with the setup and teardown of tests. There is no specific place or construct provided by PropEr to do it for each iteration, so it has to be done inline within the property.

Setting Up and Tearing Down Tests

While PropEr offers no special mechanism to let you set up and tear down some state before specific iterations of a test, it does allow you to set things up before all iterations of a given property.

This can be done with the ?SETUP macro, of the form:

```
prop_example() ->
    ?SETUP(fun() ->
        %% setup phase as any code running within the macro
        OptionalData = do_setup(),
        %% teardown phase as a no-argument function returned
        %% by the setup function
        fun() -> do_teardown(OptionalData) end
    end,
    ?FORALL(«property»)
).
```

Multiple macros of this kind can be nested together. But do remember: the setup will be run only once for all iterations for any given property. If you want to run something equivalent for each individual iteration, it has to be done inline.

The rest works as usual. Let's take a look at the callbacks now:

```
%%%%%%%%%%%%%
%%% MODEL %%%
%%%%%%%%%%%%%
%% @doc Initial model value at system start. Should be deterministic.
initial_state() ->
    #{}.

%% @doc List of possible commands to run against the system
command(_State) ->
    oneof([
        {call, actual_system, some_call, [term(), term()]}
    ]).

%% @doc Determines whether a command should be valid under the
%% current state.
precondition(_State, {call, _Mod, _Fun, _Args}) ->
    true.

%% @doc Given the state `State' *prior* to the call
%% `{call, Mod, Fun, Args}', determine whether the result
%% `Res' (coming from the actual system) makes sense.
postcondition(_State, {call, _Mod, _Fun, _Args}, _Res) ->
    true.

%% @doc Assuming the postcondition for a call was true, update the model
%% accordingly for the test to proceed.
next_state(State, _Res, {call, _Mod, _Fun, _Args}) ->
    NewState = State,
    NewState.
```

Elixir translation on page 219

Every model must have an initial state. The callback on line 5 lets you pick it. The state has to be deterministic, always the same. If the initial state is unpredictable, then there will be no way to know whether shrinking can be effective or not. The program exploration comes from various commands being applied, not the initial state.

The commands themselves are generated through the `command/1` callback (on line 9). Do note that the model's `State` is available, and can be used to generate commands and their arguments—such as trying to read an entry that is already existing within the model. In fact, the `State` variable can even be used to generate context-specific commands. For example, you could decide that when the state is empty, only commands about initializing it can run and no other. This can help in generating valid command chains faster and boost the speed of your model quite a bit.

Preconditions (on line 16) can be used to constrain whenever a command is acceptable or not: you could decide to limit their frequency, or that "inserting a new entry" does not work if the entry is already in the `State` variable. Much like the `?SUCHTHAT` macro, too much filtering in preconditions can tend to slow down the model generation as the framework has to try building more commands to find something that works.

You might be asking why use preconditions when matching within `command/1` should be faster and has access to the same data. The reason is that preconditions are used when shrinking as well as generating, and basically help ensure that whatever the framework tries to do with the command sequence remains valid. Whatever matching rule or constraint you put in `command/1` must be duplicated in `precondition/2` (or live *only* in the latter) in order for shrinking to work best.

Postconditions are where the validation takes place. The `_Res` variable on line 22 contains the result of a command being applied to the actual system, with the model's `State` variable containing the state *before* the call was made to the system and the `_Args` passed to the system. This gives us all the ingredients to check everything: given the model state and the function called, does the actual return value match what we think it should be?

The last callback is `next_state/3`, on line 27. This one is a bit tricky because it accepts a `_Res` value *even during symbolic execution, where no result exists* to put in `_Res`. As such you cannot easily use the return value for any transformations or comparisons (a workaround exists with symbolic calls, but it's not all that obvious). Instead, it's easier to pretend `next_state/3` is only run during symbolic execution: assume that `_Res` contains opaque placeholders,

and only use its value to blindly update your model's state, without the ability to look at what it contains and reveals about the running system.

The two execution flows are important to keep in mind: since the calls to preconditions, commands, and state transitions are executed both when generating commands and when running the actual system, side-effects should be avoided in the model. And any value coming from the actual system that gets transferred to the model should be treated as an opaque blob that cannot be inspected, matched against, or used in a function call that aims to transform it.

The abstract phase has to pass some of these values in to make them available to the model, but since the actual system is not running, it cannot provide the data. PropEr instead passes in abstract placeholders. These do not look like the actual expected data; they can be of an entirely different type than what you expect, which is why the data must be treated as opaque.

The Model Decides

This is very abstract for now, but the core concept to keep in mind is that the model is the source of authority that leads the test execution, not the system; the system is being tested and is not in the driver's seat.

That's a lot of theory, but it will really make things simpler when it comes to putting it in practice. Not being aware of the execution model makes for a difficult learning experience. Feel free to come back to the diagrams and descriptions as often as you need them, until it becomes natural. You might need them a bit in the following example, which uses stateful testing with a cache process.

Testing a Basic Concurrent Cache

To use stateful tests, we'll first need a stateful system to validate. In this section, we'll use a cache implemented as an OTP `gen_server`. A common optimization pattern in Erlang is to use an ETS table for reads, and to make the writes sequential through calls to the `gen_server`, which ensures they're safe. This creates a bit of contention on the write operations, so instead, we'll try to write a cache that *only* uses ETS for all operations, and the `gen_server`'s job is just to keep the ETS table alive. The simple conceptual model—a cache handling data like a key-value store—along with an implementation dangerously accessing ETS tables concurrently means that this is a great candidate to demonstrate stateful property tests. We'll see the cache implementation, and then how to approach modeling it to find potential bugs it may hide.

Our cache will have a rather simple set of requirements:

- Values can be read by searching for their key
- The cache can be emptied on demand
- The cache can be configured with a maximum number of items to hold in memory
- Once the maximal size is reached, the oldest written value is replaced
- If an item is overwritten, even with a changed value, the cache entry remains in the same position

Those are a bit unconventional: most caches care about evicting entries that were not *accessed* for a long time, whereas ours really focuses on writes, and does not even care about updates in its eviction policy. But that's fine because we want to show how to model that cache, not necessarily how to write a good one, so we'll stick with these requirements that are friendlier to a succint implementation.

Implementing the Cache

In general, stateful tests are often used during integration tests. This means that you will likely use stateful properties later in a project's lifetime, and will likely write tests after the program has been written. We will therefore respect this by writing the cache implementation itself first, then we'll put the system in place and add tests after the fact.

We'll start with a standard `gen_server` set of callbacks and public exports:

StatefulProperties/erlang/pbt/src/cache.erl

```
-module(cache).
-export([start_link/1, stop/0, cache/2, find/1, flush/0]).
-behaviour(gen_server).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2]).

start_link(N) ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, N, []).

stop() ->
    gen_server:stop(?MODULE).
```

Elixir translation on page 220

The process will be unique to the entire node by virtue of having the `{local, ?MODULE}` name. Since all operations will be done in an ETS table, we can read from the cache using the table directly, assuming the table is named `cache`. We'll give the table's records a structure of the form `{Index, {Key, Val}}`, where `Index` ranges from `1` to the max value allowed, basically forcing the table to be used like a big 1-indexed array. Whenever we write to the table, we increment

the Index value before doing so, wrapping around to the first entry whenever we fill the array.

Unfortunately, this does mean that we'll need to scan the table on every read operation, but optimizing is not the point here. Here's how the table is initialized:

```
init(N) ->
    ets:new(cache, [public, named_table]),
    ets:insert(cache, {count, 0, N}),
    {ok, nostate}.

handle_call(_Call, _From, State) -> {noreply, State}.

handle_cast(_Cast, State) -> {noreply, State}.

handle_info(_Msg, State) -> {noreply, State}.
```

Elixir translation on page 220

There's a magic record {count, 0, Max} inserted in the table. That's basically our index-tracking mechanism. Each writer will be able to increment it before writing their own data, ensuring the index is always moving forwards. You'll also note that the gen_server callbacks are otherwise empty, since we don't need them. Let's see how reads work:

```
find(Key) ->
    case ets:match(cache, {'_', {Key, '$1'}}) of
        [[Val]] -> {ok, Val};
        [] -> {error, not_found}
    end.
```

Elixir translation on page 220

Here the ets:match/2 pattern basically means "ignore the index" ('_'), "match the key we want" (Key), and "return the value" ('$1'). The documentation for ets:match/2[2] contains more details if you need further explanations.

Writing to the cache is a bit more complex:

```
Line 1  cache(Key, Val) ->
-           case ets:match(cache, {'$1', {Key, '_'}}) of % find dupes
-               [[N]] ->
-                   ets:insert(cache, {N,{Key,Val}}); % overwrite dupe
5           [] ->
-                   case ets:lookup(cache, count) of % insert new
-                       [{count,Max,Max}] ->
-                           ets:insert(cache, [{1,{Key,Val}}, {count,1,Max}]);
-                       [{count,Current,Max}] ->
```

2. http://erlang.org/doc/man/ets.html#match-2

```
10                      ets:insert(cache, [{Current+1,{Key,Val}},
-                                          {count,Current+1,Max}])
-               end
-       end.
```

Elixir translation on page 220

There are 3 cases considered here:

1. When the value to insert matches a key that already exists (on line 3), we just overwrite
2. When the value is inserted in a regular case (on line 9), we insert it after having incremented the index
3. When we reach the max point of the index (on line 7), we reset the index and start writing from the start, wrapping the cache around

That's all a bit convoluted—and probably feels risky when thinking of concurrent code execution—but let's keep going with the last function, the one to flush the cache:

```
flush() ->
    [{count,_,Max}] = ets:lookup(cache, count),
    ets:delete_all_objects(cache),
    ets:insert(cache, {count, 0, Max}).
```

Elixir translation on page 221

This basically empties the cache table and resets the `{count, CurrentIndex, Max}` entry.

You can play with the code in the shell a bit if you want to see if it works, but we'll otherwise get started right away with the stateful property tests.

Writing the Tests

With the cache complete, we can write tests for it that will show whether it works correctly. As we've seen earlier, stateful properties are executed in two big phases: a symbolic one to generate the command set, and a real one, where the commands are run against the real system for validation purposes.

We're going to follow the same pattern here, and we'll start with a focus on the symbolic execution, by setting up the model before adding validation rules, and then running the property to see if it seems sound.

Building the Model

The first step in coming up with a good stateful model is to think like an operator, someone in charge of running or debugging your code in production.

If they are to operate and run your code, they have to be able to understand what to expect out of it. Whatever expectations people form as operators turn out to be a bit of a *mental model*: they play the operations in their heads, and make guesses as to what data or behavior they'll get back out of the system. Whenever their mental model is wrong ("the system doesn't do what I think it should"), you get a bug report, or a production incident.

If you can figure out how you'd explain how the system works to an operator in a way that is both realistic and simple, you've given them a reliable mental model to work with. That mental model is something we can try to encode as a property.

Interestingly, if a good way to come up with a model is to try to figure out how you'd explain your component to a human operator having to run it in production, the opposite is true as well: if you have to explain your system to someone, the model you used in your tests could be a good starting point. If your tests are complex, convoluted, and hard to explain, then know that your testing experience is likely to match their operational experience as well.

Since our cache works a bit like a big array where old entries are evicted to make place for new ones, we can use any data structure or implementation with First-In-First-Out (FIFO) semantics as a model, and it should be accurate. We'll use all of proper_statem's callbacks to write our simpler FIFO structure to show whether the real cache works or not. Let's set this up:

StatefulProperties/erlang/pbt/test/prop_cache.erl

```
-module(prop_cache).
-include_lib("proper/include/proper.hrl").
-behaviour(gen_statem).
-export([command/1, initial_state/0, next_state/3,
         precondition/2, postcondition/3]).

-define(CACHE_SIZE, 10).

prop_test() ->
    ?FORALL(Cmds, commands(?MODULE),
        begin
            cache:start_link(?CACHE_SIZE),
            {History, State, Result} = run_commands(?MODULE, Cmds),
            cache:stop(),
            ?WHENFAIL(io:format("History: ~p\nState: ~p\nResult: ~p\n",
                                [History,State,Result]),
                      aggregate(command_names(Cmds), Result =:= ok))
        end).
```

Elixir translation on page 221

An arbitrary ?CACHE_SIZE value has been chosen for the sake of simplicity. A generator could also have been used for more thorough testing, but we'll get the basics of stateful testing without that. Important to note is that the setup and teardown functions (cache:start_link/1 and cache:stop/0) run as part of the property, every time. Had we used the ?SETUP macro instead, we'd have needed to only call cache:flush() after every run to ensure it's always empty, but the current form is just a bit longer, and there are few enough setup and teardown requirements that it will do fine for our example.

For our model's state, we'll try to use as little data as possible, carrying only what's strictly necessary to validate everything:

```
-record(state, {max=?CACHE_SIZE, count=0, entries=[]}).

%% Initial model value at system start. Should be deterministic.
initial_state() ->
    #state{}.
```

Elixir translation on page 221

We'll use a list to contain the model's data, along with a count of how many entries have been seen. The list is going to contain {Key,Value} pairs, and the counter will know when to drop pairs from the list—as simple as it can be.

The command generation is straightforward as well. We put emphasis on writes for the tests through the use of the frequency/1 generator:

```
command(_State) ->
    frequency([
        {1, {call, cache, find, [key()]}},
        {3, {call, cache, cache, [key(), val()]}},
        {1, {call, cache, flush, []}}
    ]).
```

We can then use the precondition to add constraints, such as preventing calls to empty the cache when it's already empty:

```
%% Picks whether a command should be valid under the current state.
precondition(#state{count=0}, {call, cache, flush, []}) ->
    false; % don't flush an empty cache for no reason
precondition(#state{}, {call, _Mod, _Fun, _Args}) ->
    true.
```

You can define the generators for key() and val() as such:

```
key() ->
    oneof([range(1,?CACHE_SIZE), % reusable keys, raising chance of dupes
           integer()]).          % random keys

val() ->
    integer().
```

The generator for keys is designed to allow some keys to be repeated multiple times: by using a restricted set of keys (with the range/2 generator) along with an unrestricted integer() we ensure some keys will be reused, which forces our property to exercise any code related to key reuse or matching, but without losing the ability to "fuzz" the system with occasionally unexpected new keys.

The next_state callback completes command generation by allowing the model to stay up to date with what the system state should be:

```
Line 1  %% Assuming the postcondition for a call was true, update the model
      - %% accordingly for the test to proceed.
      - next_state(State, _, {call, cache, flush, _}) ->
      -     State#state{count=0, entries=[]};
    5 next_state(S=#state{entries=L, count=N, max=M}, _Res,
      -            {call, cache, cache, [K, V]}) ->
      -     case lists:keyfind(K, 1, L) of
      -         false when N =:= M -> S#state{entries = tl(L) ++ [{K,V}]};
      -         false when N < M -> S#state{entries = L ++ [{K,V}], count=N+1};
   10         {K,_} -> S#state{entries = lists:keyreplace(K,1,L,{K,V})}
      -     end;
      - next_state(State, _Res, {call, _Mod, _Fun, _Args}) ->
      -     State.
```

Elixir translation on page 222

The first clause says that whenever we flush the cache, we must empty the model by dropping all its entries and returning the count to 0. The second clause is about adding items to the cache. There are three cases identified. First, line 8 concerns itself with whenever the FIFO list is full. Whenever that happens, we drop the oldest element of the list by calling tl(L), and then add the new entry at the end. The case just after that, on line 9, deals with a cache that still has space and so it just adds the term at the end of the list and increments the counter. The last branch (line 10) replaces an existing entry wherever it was in the list.

Finally, the last function clause tells us that any other call (like lookups) have no impact on the model state; it remains unchanged.

With this in place, commands can be generated. If you stub out postcondition/3 (write it as something like postcondition(_, _, _) -> true.) and try it in the shell, you can see the kinds of commands PropEr generates:

```
$ rebar3 as test shell
«build output»
1> proper_gen:sample(proper_statem:commands(prop_cache)).
[{set,{var,1},{call,cache,cache,[3,63]}},
 {set,{var,2},{call,cache,find,[2]}},
 {set,{var,3},{call,cache,find,[5]}}]
```

```
[{set,{var,1},{call,cache,cache,[2,-1]}},
 {set,{var,2},{call,cache,flush,[]}},
 {set,{var,3},{call,cache,cache,[9,-9]}},
 {set,{var,4},{call,cache,cache,[3,1]}},
 {set,{var,5},{call,cache,cache,[8,18]}},
 {set,{var,6},{call,cache,cache,[-14,6]}},
 {set,{var,7},{call,cache,cache,[8,-12]}}]
«more runs»
[{set,{var,1},{call,cache,cache,[-4,-1]}},
 {set,{var,2},{call,cache,flush,[]}},
 {set,{var,3},{call,cache,cache,[3,2]}},
 {set,{var,4},{call,cache,cache,[-25,-43]}},
 {set,{var,5},{call,cache,cache,[3,3]}},
 {set,{var,6},{call,cache,find,[3]}}]
```

Don't worry about the format it has ({set, VarNum, Call}) since that's something PropEr deals with internally. Just know that you can see the sequence of calls it would run, and in what order. The run_commands/2 function provided by PropEr and used in our property will deal with the rest. Now let's see how we can validate the actual system.

Validating the System

When dealing with a cache like this, we expect all writes to always succeed. Therefore the only operation we can really use to validate the system is find/1. By observing the results we get when reading values, we can see if they match those the model predicts we'd return based on the write sequences we applied. If the key we're looking up is in the model state, then the actual system better return the value we expect, and return nothing when it's not there.

This is rather straightforward to implement:

StatefulProperties/erlang/pbt/test/prop_cache.erl

```
%% Given the state `State' *prior* to the call `{call, Mod, Fun, Args}',
%% determine whether the result `Res' (coming from the actual system)
%% makes sense.
postcondition(#state{entries=L}, {call, cache, find, [Key]}, Res) ->
    case lists:keyfind(Key, 1, L) of
        false      -> Res =:= {error, not_found};
        {Key, Val} -> Res =:= {ok, Val}
    end;
postcondition(_State, {call, _Mod, _Fun, _Args}, _Res) ->
    true.
```

Elixir translation on page 222

The lookup is done on the model's state (the L list), and based on this expected value, we compare the result Res from the actual system. Only if they agree do we say the operation was valid.

You can go ahead and run the property;

```
$ rebar3 proper -n 1000
«build information»
===> Testing prop_cache:prop_test()
.....................................................«more tests»
OK: Passed 1000 test(s).

63% {cache,cache,2}
21% {cache,find,1}
14% {cache,flush,0}
```

This looks decent enough. In a real project, we may want to repeat the steps we've taken in earlier chapters: measure whether the operations executed are those we really want. Are the reads numerous enough? Are we only reading non-existing keys? Do we ever have a read after a flush to validate that it worked? And so on. You know how to do this by now, so let's just do a simple sanity check to make sure our property works fine by injecting a bug and seeing if it picks up on it.

Replace the init/1 function with this one:

```
init(N) ->
    ets:new(cache, [public, named_table]),
    ets:insert(cache, {count, 0, N-1}),
    {ok, nostate}.
```

This simply makes sure that the maximum cache size is one less than what is asked for, which means we should start dropping entries earlier than expected by the model. Run the property a bunch of times and you should eventually trigger a failure like:

```
Line 1  $ rebar3 proper -n 10000
  -     «build information»
  -     ===> Testing prop_cache:prop_test()
  -     .....«more tests».....!
  5     Failed: After 747 test(s).
  -     [{set,{var,1},{call,cache,find,[8]}},[...]]
  -     History: [{{state,10,0,[]},{error,not_found}},
  -               «more history»]
  -     State: {state,10,10, «cache model list»}
 10     Result: {postcondition,false}
  -
  -     Shrinking ......(6 time(s))
  -     [{set,{var,5},{call,cache,cache,[10,36]}},«more commands»,
  -      {set,{var,18},{call,cache,find,[10]}}]
 15     History: [{{state,10,0,[]},true},
  -              {{state,10,1,[{10,36}]},true},
  -              {{state,10,2,[{10,36},{-19,-6}]},true},
  -              {{state,10,3,[{10,36},{-19,-6},{12,5}]},true},
```

```
-           «more history»
20          {{state,10,10,[{10,36},{-19,-6},{12,5},{9,26},{-4,-30},{-75,17},
-                          {13,-22},{42,5},{31,55},{197,18}]},
-            {error,not_found}}]
- State: {state,10,10, [{10,36},{-19,-6},{12,5},{9,26},{-4,-30},{-75,17},{13,-22},
-                       {42,5},{31,55},{197,18}]}
25 Result: {postcondition,false}
```

This is very noisy output, with a lot of data. Let's go through it and figure it out. First, on line 6, we get the initial failing case. It can be huge with a high number of operations, so when possible, we ignore that one. Instead you should move past the shrinking step (on line 12), so that we get a simpler minimal counterexample.

The shortened output, based on the `?WHENFAIL` macro we have in our property, is divided in 4 categories: the failing list of commands, the state and call history, the final state, and the failing result. Line 13 shows the final set of commands that caused the failure as output by PropEr. It's all on one line and a bit hard to read. You can cross-reference this set of commands with the history, starting at line 15. The history section tracks the progression of all of our model's states. The `{state, 10, N, List}` format is the underlying representation of the record `#state{max=10, count=N, entries=List}`, followed with the result of the command that caused it. The history starts at the initial state, so there should be one more entry in there than there are commands.

Checking the sequence of commands against the history will prove helpful in most cases, but it requires quite a bit of care and attention to detail. To help a bit, the final model state is output on line 24, and the last line contains the error for the failure—usually just `false`, unless you make your postcondition fails with something more descriptive.

In this case, data shows that the last call to `find(10)` had the system return `{error, not_found}` whereas the model contains a tuple `{10,36}` in its state list, meaning that we expected that key to be found. The pair is the first one of the list (the next to be dropped!), and the shrinking also managed to drop all commands it could until the counter hit the value `10`, which should also be seen as a clue. With all of that data, identifying the bug and fixing it is hopefully doable.

In this case we know what the error is, but it's nice to see that our test picks it up, albeit with many iterations. Perhaps improving it by altering generators to get longer sequences or more repeated reads would prove useful.

This is pretty neat. Anything we can model, we can likely test with this kind of property. If the actual system used geodistributed databases with complex

protocols instead of just ETS tables, the same model could be used to validate it, since the expectations it fulfills are the same—albeit with more tolerance to local datacenter outages. That's pretty amazing stuff.

One thing this property does not do, however, is find concurrency errors that could happen with our ETS usage. Our model is purely sequential, but we decided to use ETS specifically for concurrency. The good news is that PropEr gives us one more tool just for that.

Testing Parallel Executions

One of the most interesting features of stateful tests with PropEr is the ability to take the models we write for sequential ones, and automatically turn them into parallel tests that have a decent chance at finding concurrency bugs in your code at nearly no cost. It requires just a few minor changes to the property and the rest is done for you.

The way it works is conceptually simple. The framework starts by taking the existing command generation mechanism, and then builds a sequence. Something like the following, in abstract terms:

```
A -> B -> C -> D -> E -> F -> G
```

What it will do then is pick a common root of operations. Here for example, `A -> B` is shared by all operations that follow them both. PropEr will take the remaining chain (`C -> D -> E -> F -> G`) and split it up in concurrent timelines based on some fancypants analysis (your preconditions will help drive this process), giving something like:

```
      ,-> C -> E -> G
A -> B
      '-> D -> F
```

This new sequence is going to be represented as a tuple of the form `{Sequential-Root, [LeftBranch, RightBranch]}`. PropEr will run the common sequential root, and then run both alternative branches in parallel in an attempt to cause bugs to surface.

The only changes required will be in the generator and command used to run the tests, and some adjustments to the `?WHENFAIL` macro:

StatefulProperties/erlang/pbt/test/prop_cache.erl

```
prop_parallel() ->
    ?FORALL(Cmds, parallel_commands(?MODULE),
       begin
           cache:start_link(?CACHE_SIZE),
           {History, State, Result} = run_parallel_commands(?MODULE, Cmds),
```

```
        cache:stop(),
        ?WHENFAIL(io:format("=======~n"
                            "Failing command sequence:~n~p~n"
                            "At state: ~p~n"
                            "=======~n"
                            "Result: ~p~n"
                            "History: ~p~n",
                            [Cmds,State,Result,History]),
                  aggregate(command_names(Cmds), Result =:= ok))
    end).
```

Elixir translation on page 223

As you can see here, we replaced commands/1 with parallel_commands/1, and run_commands/2 with run_parallel_commands/2. Our model remains exactly the same. If you run this test, it will likely pass even with thousands of runs:

```
$ rebar3 proper
«build info and other test runs»
===> Testing prop_cache:prop_parallel()
..............f.................................................f........
.......................
OK: Passed 100 test(s).

65% {cache,cache,2}
19% {cache,find,1}
14% {cache,flush,0}
```

Here, each f stands for a non-parallelizable command generation that failed and had to be retried. PropEr tried to make parallel branches of the command, but could not do it without breaking some precondition. This lets you know if it's hard to make good parallel executions or not for your property.

Unfortunately for us, the Erlang scheduler is rather predictable, and making it test random interleavings is not really obvious nor something we can control. This means that if it doesn't fail right away, it will need more luck finding bugs, which means more executions. Only after an ungodly amount of iterations might it find something in this cache, if at all:

```
$ rebar3 proper -n 10000
«build info and other test runs»
===> Testing prop_cache:prop_parallel()
«lots of dots»
Failed: After 3850 test(s).
An exception was raised:
    error:{'EXIT',{{case_clause,[[-27],[3]]},[{cache,find,«stacktrace»]
Stacktrace: [«stacktrace»].
«huge command dump»
Shrinking ......(6 time(s))
{[{set,{var,2},{call,cache,find,[-4]}},{set,{var,3},{call,cache,cache,
```

```
[7,22]}},{set,{var,4},{call,cache,cache,[2,3]}}],[[{set,{var,5},{call,cache,
cache,[9,15]}},{set,{var,6},{call,cache,find,[2]}},{set,{var,7},{call,cache,
cache,[2,14]}},{set,{var,8},{call,cache,cache,[-1,2]}},{set,{var,12},{call,
cache,find,[4]}}],[{set,{var,11},{call,cache,flush,[]}},{set,{var,14},{call,
cache,cache,[2,-27]}}]]}
===>
1/2 properties passed, 1 failed
===> Failed test cases:
prop_cache:prop_parallel() -> false
```

Ouch. We'll get back to this one a bit later, because first we should talk about *why* it took so many tests to find the bug. As mentioned earlier, the Erlang scheduler is quite predictable even if it is not deterministic. The best you can do to help from the outside is passing some flags to the Erlang VM that will force it to preempt processes more often, even if that has no guarantees of finding anything. This can be done through the `+T0` to `+T9` emulator flags, which allow you to play with timing values such as how long a process takes to spawn, the number of work it can do before being scheduled out, or the perceived cost of IO operations. Those are intended for testing only and can be enabled by setting them like this: `ERL_ZARGS="+T4" rebar3 proper -n 10000`.

This is unlikely to help much with our cache since there's not that much work going on for it (but in larger systems it may prove useful). Instead, we can go the manual way and tell Erlang when to deschedule processes by peppering calls to `erlang:yield()` around the places in code that seem worrisome. This will tend to generate errors fast:

```
Line 1  cache(Key, Val) ->
        -   case ets:match(cache, {'$1', {Key, '_'}}) of % find dupes
        -       [[N]] ->
        -           ets:insert(cache, {N,{Key,Val}}); % overwrite dupe
        5       [] ->
        -           erlang:yield(),
        -           case ets:lookup(cache, count) of % insert new
        -               [{count,Max,Max}] ->
        -                   ets:insert(cache, [{1,{Key,Val}}, {count,1,Max}]);
        10              [{count,Current,Max}] ->
        -                   ets:insert(cache, [{Current+1,{Key,Val}},
        -                                      {count,Current+1,Max}])
        -           end
        -   end.
        15
        - flush() ->
        -   [{count,_,Max}] = ets:lookup(cache, count),
        -   ets:delete_all_objects(cache),
        -   erlang:yield(),
        20  ets:insert(cache, {count, 0, Max}).
```

Elixir translation on page 223

Here we've added calls on lines 6 and 19. These calls basically tell the Erlang VM: "once you reach this point, de-schedule my process and run another one if you can." Since they are placed in code we are suspicious about—the places where various reads and writes take place on a shared resource—PropEr finds issues almost instantly:

```
$ rebar3 proper -n 10000 -p prop_parallel
===> Testing prop_cache:prop_parallel()
..!
Failed: After 3 test(s).
An exception was raised:
    error:{'EXIT',{{case_clause,[[1],[2]]},[{cache,cache,2,[«stacktrace»
Stacktrace: «stacktrace»
{[],[[{set,{var,1},{call,cache,find,[0]}},«commands»

Shrinking ....(4 time(s))
{[],[[{set,{var,2},{call,cache,cache,[0,2]}}],
     [{set,{var,5},{call,cache,cache,[0,1]}}]]}
```

And just like that, in merely 3 runs, it found conflicts between two cache write operations. The most basic functionality of our cache is not safe for concurrency. Put in a graphical form, this is what breaks:

```
     ,-> cache(0,2)
[] --+
     '-> cache(0,1)
```

With the error being a case_clause in cache:cache/2 as per the exception, we know the bug to be in:

```
cache(Key, Val) ->
    case ets:match(cache, {'$1', {Key, '_'}}) of % find dupes
        [[N]] ->
            ets:insert(cache, {N,{Key,Val}}); % overwrite dupe
        [] ->
            «insertion code»
    end.
```

Whenever two insertions happen concurrently, there is a slight chance that our two writes check for an existing key before either of them is written, and that both of them increment the counter one after the other, causing two records to exist: {0,{Key,Val1}} and {1,{Key,Val2}}. We have two items in the cache sharing the same key, but with two distinct values.

Alternative Tools

If finding concurrency bugs is a big concern for you, looking into a QuickCheck license may be worthwhile since it comes with PULSE, a user-level scheduler that can be used to augment concurrency in the VM to weed out these bugs in property tests.

Otherwise, look into tools like Concuerror (at concuerror.com), which aims to focus solely on concurrency bugs in Erlang. In fact it can be used as a full formal proof that some execution paths are not going to be sensitive to concurrency bugs

Fixing the bug requires changing our approach fundamentally, by making sure all destructive updates to the cache are done in a mutually exclusive manner. This could be done with locks, but an easier fix is to move write operations within the process that owns the table so that it will force all destructive updates to be sequential:

```
-module(cache).

-export([start_link/1, stop/0, cache/2, find/1, flush/0]).
-behaviour(gen_server).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2]).

start_link(N) ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, N, []).

stop() ->
    gen_server:stop(?MODULE).

find(Key) ->
    case ets:match(cache, {'_', {Key, '$1'}}) of
        [[Val]] -> {ok, Val};
        [] -> {error, not_found}
    end.

cache(Key, Val) ->
    gen_server:call(?MODULE, {cache, Key, Val}).

flush() ->
    gen_server:call(?MODULE, flush).

%%%%%%%%%%%%%%%
%%% Private %%%
%%%%%%%%%%%%%%%
init(N) ->
    ets:new(cache, [public, named_table]),
    ets:insert(cache, {count, 0, N}),
    {ok, nostate}.

handle_call({cache, Key, Val}, _From, State) ->
    case ets:match(cache, {'$1', {Key, '_'}}) of % find dupes
        [[N]] ->
```

```
            ets:insert(cache, {N,{Key,Val}}); % overwrite dupe
        [] ->
            erlang:yield(),
            case ets:lookup(cache, count) of % insert new
                [{count,Max,Max}] ->
                    ets:insert(cache, [{1,{Key,Val}}, {count,1,Max}]);
                [{count,Current,Max}] ->
                    ets:insert(cache, [{Current+1,{Key,Val}},
                                       {count,Current+1,Max}])
            end
    end,
    {reply, ok, State};
handle_call(flush, _From, State) ->
    [{count,_,Max}] = ets:lookup(cache, count),
    ets:delete_all_objects(cache),
    erlang:yield(),
    ets:insert(cache, {count, 0, Max}),
    {reply, ok, State}.

handle_cast(_Cast, State) -> {noreply, State}.

handle_info(_Msg, State) -> {noreply, State}.
```

Elixir translation on page 223

This being done, you can re-run the property and see that it always works:

```
$ rebar3 proper -p prop_parallel -n 10000
.......f.....................................f...«more tests»
OK: Passed 10000 test(s).

63% {cache,cache,2}
21% {cache,find,1}
15% {cache,flush,0}
```

You can then take out the erlang:yield() calls from the code before committing it, and be more confident that the cache works well.

Wrapping Up

This chapter has shown you the basics of stateful property testing, based on a single simple model. We've been through the two phases of the test: the symbolic one, where the model is used to generate a sequence of commands that represents what the system should do, and the real one, where the symbolic sequence of commands is applied to the real system, and the results compared with what the model expects.

We've put these concepts in practice through a cache server modeled as a First-In-First-Out list and showed that the model seems to hold fine under sequential operations. You have then seen how PropEr can take that same

exact model, and create a parallel version of it that could, with some amount of help, find concurrency bugs in our cache program.

You've seen the base mechanism of how stateful properties work, but truth be told, the cache example we used resulted in a rather simple model. In the next chapter, we'll experiment with a more complex stateful system based on SQL queries, with varying amounts of determinism. This will let us go past the basics of stateful properties, and let us explore more advanced techniques to handle the complexity of modeling large systems for integration tests.

Exercises

Question 1

Which callbacks for stateful tests belong in which execution phases?

Question 2

Pattern matching on the model state can be used to direct the generation of commands based on the current context, but preconditions are still required. Why are commands alone not sufficient and why do preconditions need to be used?

Question 3

The three ways to initialize a stateful property test are seen in a file:

```
prop_test1() ->
    ?FORALL(Cmds, commands(?MODULE),
        begin
            actual_system:start_link(),
            {_History, _State, Result} = run_commands(?MODULE, Cmds),
            actual_system:stop(),
            Result =:= ok
        end).

prop_test2() ->
    ?SETUP(fun() ->
        actual_system:start_link(),
        fun() -> actual_system:stop() end
    end,
      ?FORALL(Cmds, commands(?MODULE),
        begin
            {_History, _State, Result} = run_commands(?MODULE, Cmds),
            Result =:= ok
        end)
    end).

prop_test3() ->
    actual_system:start_link(),
```

```
?FORALL(Cmds, commands(?MODULE),
    begin
        {_History, _State, Result} = run_commands(?MODULE, Cmds),
        Result =:= ok
    end),
actual_system:stop().
```

Elixir translation on page 225

What will happen to the "actual system" in every one of them?

Question 4

What is the caveat of using the `Res` value—the result from the actual system—in the `next_state/3` callback? Is there a scenario where it is absolutely useful to use it?

CHAPTER 10

Case Study: Bookstore

Content to be supplied later.

CHAPTER 11

State Machine Properties

Content to be supplied later.

APPENDIX 1

Solutions

Writing Properties

Question 1

```
proper_gen:pick(proper_type:Type())
```

Question 2

What the property is doing is validating the `lists:seq(Start, Stop)` function, which would be expected to return a list of integers in the range `[Start, ..., Stop]`. For example, running `lists:seq(2,5)` should return `[2,3,4,5]`. The property does the validation of this by looking at two aspects of such a list:

- the list should contain as many entries as the range covered by both terms (`2..5` has 4 entries, or just `(5-2)+1`)
- to avoid having the test succeed on outputs such as `[1,1,1,1]`, the `increments/1` function is used to ensure that each number is greater than the following one.

Thinking In Properties

Question 1

Any three of:

- Modeling; comparing the implementation with a simpler but obviously correct one
- Generalizing traditional tests by automating the steps we would do by hand
- Finding program invariants to validate

- Using symmetric properties

Question 2

Two example solutions:

ThinkingInProperties/erlang/pbt/test/prop_solutions.erl

```
%% @doc this function tests that any lists of `{Key,Val}' pairs
%% end up being able to be sorted by the key by using `lists:keysort/2'.
prop_keysort1() ->
    ?FORALL(List, list({term(),term()}),
        begin
            %% is_key_ordered checks that all tuples' keys are ordered.
            is_key_ordered(lists:keysort(1, List))
        end).

is_key_ordered([{A,_},{B,_}=BTuple|T]) ->
    A =< B andalso is_key_ordered([BTuple|T]);
is_key_ordered(_) -> % smaller lists
    true.

%% @doc This function instead works by using random tuples with various
%% sizes, and picking a random key to test it.
%% This tests broader usages of lists:keysort, such as
%% `lists:keysort(2, [{a,b},{e,f,g},{t,a,n,e}])' yielding the list
%% `[{t,a,n,e},{a,b},{e,f,g}]', where the comparison takes place
%% on the second element of each tuple.
%%
%% While more complete than the previous one, this function
%% does not accurately portray the need for stability in the
%% function as documented: `[{a,b}, {a,a}]' being sorted will
%% not be tested here!
%% Those can either be added in a regular test case, or would
%% require devising a different property.
prop_keysort2() ->
    ?FORALL(List, non_empty(list(non_empty(list()))),
        begin
            %% Since the default built-in types do not let us easily
            %% create random-sized tuples that do not include `{}',
            %% which is not working with `lists:keysort/2', we
            %% create variable-sized tuples ourselves.
            Tuples = [list_to_tuple(L) || L <- List],
            %% To know what position to use, we're going to use
            %% the smallest, to avoid errors
            Pos = lists:min([tuple_size(T) || T <- Tuples]),
            Sorted = lists:keysort(Pos, Tuples),
            Keys = extract_keys(Pos, Sorted),
            %% The keys returned by keysort have to be in the
            %% same order as returned by `lists:sort/1', which
            %% we now trust.
            Keys == lists:sort(Keys)
```

```
        end).
extract_keys(Pos, List) -> [element(Pos,T) || T <- List].
```

Elixir translation on page 225

Question 3

The problem is with the model; sets do not generally allow duplicate elements. In this case, the call that generates ModelUnion adds both lists together. This inadvertently maintains duplicate elements, which the actual sets module avoids.

If the call to lists:sort/1 is changed to lists:usort/1 (which removes duplicated elements) when handling ModelUnion, the property will adequately represent sets by removing duplicate elements and pass.

Question 4

The function is shaky because it validates only the keys portion of dictionary merging. The values resulting from the merge operation are untouched, and the test makes no mention of how it may resolve or report conflicts. To be safe, the property should either take into account the conflict-resolution function for merging, or a second property should be added to cover it.

Question 5

First the function:

ThinkingInProperties/erlang/pbt/test/prop_solutions.erl

```
word_count(String) ->
    Stripped = string:trim(dedupe_spaces(String), both, " "),
    Spaces = lists:sum([1 || Char <- Stripped, Char =:= $\s]),
    case Stripped of
        "" -> 0;
        _ -> Spaces + 1
    end.

dedupe_spaces([]) -> [];
dedupe_spaces([$\s,$\s|Rest]) -> dedupe_spaces([$\s|Rest]);
dedupe_spaces([H|T]) -> [H|dedupe_spaces(T)].
```

And the test, using an alternative implementation:

```
prop_word_count() ->
    ?FORALL(String, non_empty(string()),
            word_count(String) =:= alt_word_count(String)
    ).

alt_word_count(String) -> space(String).
```

```
space([]) -> 0;
space([$\s|Str]) -> space(Str);
space(Str) -> word(Str).

word([]) -> 1;
word([$\s|Str]) -> 1+space(Str);
word([_|Str]) -> word(Str).
```

Elixir translation on page 226

Custom Generators

Question 1

- collect(Term, BoolExpression) (in Elixir, BoolExpression |> collect(Term))
- aggregate([Term], BoolExpression) (in Elixir, BoolExpression |> aggregate(Term))

Question 2

?LET(Var, Generator, ErlangExpr); otherwise, the generators are abstract representations that won't be modifiable the way the generated data would be. The same syntax in Elixir is let var <- Generator do ElixirExpr end instead.

Question 3

Whenever a generator that may probabilistically choose multiple branches is called, using eager evaluation means that all alternative paths for it will be evaluated at once. If the generator is made recursive, this can quickly blow the size of the computation to very large levels. The ?LAZY(Generator) macro allows to only evaluate a given branch when required, ensuring faster execution with more predictable memory usage.

Question 4

The first step will be to ensure the generator can terminate through by using ?LAZY macros:

CustomGenerators/erlang/pbt/test/prop_solutions.erl

```
tree() ->
    tree(term()).

tree(Type) ->
    frequency([
        {3, {node, Type, undefined, undefined}},
        {2, {node, Type, ?LAZY(tree(Type)), undefined}},
        {2, {node, Type, undefined, ?LAZY(tree(Type))}},
        {3, {node, Type, ?LAZY(tree(Type)), ?LAZY(tree(Type))}}
    ]).
```

Elixir translation on page 227

Although the tree may terminate, finding a good balance can still prove tricky. The numbers have been modified a bit to fit better, but using the Size variable proves more effective to put a predictable boundary on growth:

CustomGenerators/erlang/pbt/test/prop_solutions.erl

```
limited_tree(Type) ->
    ?SIZED(Size, limited_tree(Size, Type)).

limited_tree(Size, Type) when Size =< 1 ->
    {node, Type, undefined, undefined};
limited_tree(Size, Type) ->
    frequency([
        {1, {node, Type, ?LAZY(limited_tree(Size-1, Type)), undefined}},
        {1, {node, Type, undefined, ?LAZY(limited_tree(Size-1, Type))}},
        {5, {node, Type,
             %% Divide to avoid exponential growth
            ?LAZY(limited_tree(Size div 2, Type)),
            ?LAZY(limited_tree(Size div 2, Type))}}
    ]).
```

Elixir translation on page 227

Although more verbose, it behaves much better.

Question 5

For morning stamps

The first implementation, while clear to its intent, is likely to be far less efficient as it is going to require dropping roughly half of all the generated terms to replace them with new ones, assuming a roughly equal distribution for digits between 0 and 23 for the H position.

By comparison, the generator with a ?LET uses a modulus to ensure that any hour greater than 11 starts over at 0. It will need to do one operation per generator, and will never have to retry. It is the better implementation.

For ordered stamps

Similarly in the second one, chances are roughly 50% that the first generator will need to be discarded because the probability that one stamp is greater than the other should be fairly even. Using the min/2 and max/2 functions gives a proper ordering without needing to generate newer stamps. The second solution is better.

For meeting hours

In this set of generators, there is no clearly defined way to use a ?LET to transform data that is not acceptable into acceptable one. The filtering done there is roughly the same as the one in ?SUCHTHAT and some ad-hoc procedure is used to generate alternative data.

Because the range of restricted samples is fairly limited when compared to the whole problem space, the readability of the first solution is likely better in the long run.

Question 6

Using the provided wrapper functions, it is enough to export them and then use them with {'$call, ...} symbolic calls:

CustomGenerators/erlang/pbt/test/prop_solutions.erl

```
file(Name) ->
    ?SIZED(
       Size,
       lines(Size, {'$call', ?MODULE, file_open, [Name, [read,write,raw]]})
    ).

lines(Size, Fd) ->
    if Size =< 1 -> Fd
     ; Size > 1 -> lines(Size-1, {'$call', ?MODULE, file_write, [Fd,bin()]})
    end.

bin() ->
    non_empty(binary()).
```

Elixir translation on page 228

The file is opened in both read/write mode to be more flexible to whoever will use it when running the test. The raw mode is entirely optional. It makes things go faster, but removes some flexibility when using the file module regarding encoding and buffering. Then the file is created. The generator asks for a name (which may or may not be a generator) and uses the Size parameter to know how big of a file to generate.

Through the usage of the shim functions, as many bytes can be added as desired over multiple calls. In this case, non-empty binaries are just sufficient, but the generator could as well have been written to let the user pass a generator for the data to file() and lines(), becoming fully configurable.

Shrinking

Question 1

- ?SHRINK(Generator, [AltGenerators, ...])
- ?LETSHRINK([Pattern, ...], [Generator, ...], Expression)

Question 2

?LETSHRINK always takes a list of arguments and generators, whereas ?LET takes any pattern and any generator in its first two arguments. When failing in a test case and needing a new simplified generator, one of the generators in the list will be used, but without the transformations being applied. A ?LET macro has no such rule and the transformation is always applied.

Question 3

The problem here is that the property expects a list coming out of the generator, but the way ?LETSHRINK works is that every single of the variable between Appetizer, Drink, Entree, or Dessert is an atom put in a list. During a failing test case, the shrinking attempt will fail as the property cannot work through receiving an atom when it expects a list.

Instead, the generator should be reworked as follows:

Shrinking/erlang/pbt/test/prop_solutions.erl

```
meal() ->
    ?LETSHRINK([Appetizer, Drink, Entree, Desert],
               [[elements([soup, salad, cheesesticks])],
                [elements([coffee, tea, milk, water, juice])],
                [elements([lasagna, tofu, steak])],
                [elements([cake, chocolate, icecream])]],
               Appetizer ++ Drink ++ Entree ++ Desert).
```

Elixir translation on page 228

This ensures that every single variable is always a list, and the final result passed to the property in a successful case also remains a list. The elements generated should always have the same type as the one returned for ?LETSHRINK to work well.

Question 4

Since the special list and the price list are both required for the rest of operations, those two ?LET will be left alone. The third ?LET is where the merging takes place, and that's where we'll operate.

The gotcha here is once again that the types generated by the ?LET expression as a whole ({Items, Expected, PriceList, SpecialList}) do not match what the generators produce inside the ?LET ({{RegularItems, RegularExpected}, {SpecialItems, SpecialExpected}}). We can't just replace it wholesale for a ?LETSHRINK.

To do so, we must ensure that the values match. This can be done by adding an intermediary step where we use ?LETSHRINK to build the merged lists of items and the merged expected price, and use ?LET to join them in the final 4-tuple expected by the property rather than doing it all at once:

Shrinking/erlang/pbt/test/prop_solutions.erl

```
item_price_special() ->
    %% first LET: PriceList
    ?LET(PriceList, price_list(),
         %% second LET: SpecialList
         ?LET(SpecialList, special_list(PriceList),
              %% third LET: Regular + Special items and prices
              ?LET({Items, Price},
                   %% Help shrinking by first trying only one of the
                   %% two possible lists in case a specific form causes
                   %% the problems on its own
                   ?LETSHRINK([{RegularItems, RegularExpected},
                               {SpecialItems, SpecialExpected}],
                              [regular_gen(PriceList, SpecialList),
                               special_gen(PriceList, SpecialList)],
                              %% And we merge:
                              {RegularItems ++ SpecialItems,
                               RegularExpected + SpecialExpected}),
                    {Items, Price, PriceList, SpecialList}))).
```

Elixir translation on page 228

This transformation ensures that the inputs and outputs of ?LETSHRINK are always 2-tuples that include a list of items and a price expected. The surrounding ?LET is in charge of taking these values and pairing them up in a 4-tuple with the price list and special list. By doing so, we give PropEr hints in how to isolate whether the problem is with regular items, special items, or only both at once.

Stateful Properties

Question 1

- The abstract phase contains: initial_state/0, command/1, precondition/2, and next_state/3
- The real interacting with the actual system contains: initial_state/0, command/1, precondition/2, next_state/3, and postcondition/3

Question 2

The pattern matching in the generator works fine to create an initial list of commands to run on the model system, but as soon as a failure happens–or a constraint needs to be enforced when parallelizing commands–the modification of the command list is done without regards to the initial patterns in command generations. Only the preconditions can be used to ensure the validity of the command sequence.

Without preconditions, the framework is not able to manipulate the sequence in any way, and parallelism and shrinking cannot be effective.

Question 3

1. For the first one, the actual system will be started and stopped before every single test iteration.
2. The actual system will be booted once before the tests run, and the system instance will be shared for all iterations, before being shut down after the whole run.
3. The actual system will be started, then an abstract representation of the test case will be created. After this, the system will be stopped. However, the test itself will not have run once at that point, so by the time the framework would try to execute the test, the actual system has already shut down.

Question 4

The value must be treated as fully opaque since placeholders will be generated by the framework during the abstract phase of command generation. Symbolic calls may be added to the command when used later in an actual execution context.

It is simpler to avoid it entirely, but one of the useful cases comes when interaction with the actual system relies on values it has returned and are not predictable. Those may include process identifiers, sockets, or any other unique, unpredictable, or transient resource. Whenever the system returns one of these and expects them back, you may not be able to plan the value ahead of time from the model state. The model is then better off holding a reference to that value without altering it.

APPENDIX 2

Elixir Translations

Writing Properties

Putting it All Together

WritingProperties/elixir/pbt/test/pbt_test.exs

```
use ExUnit.Case
use PropCheck

property "finds biggest element" do
  forall x <- non_empty(list(integer())) do
    biggest(x) == List.last(Enum.sort(x))
  end
end
```

WritingProperties/elixir/pbt/test/pbt_test.exs

```
def biggest([head | tail]) do
  biggest(tail, head)
end

defp biggest([], max) do
  max
end

defp biggest([head | tail], max) when head >= max do
  biggest(tail, head)
end

defp biggest([head | tail], max) when head < max do
  biggest(tail, max)
end
```

Exercises

Question 2

WritingProperties/elixir/pbt/test/exercises_test.exs

```
property "exercise 2: a sample" do
  forall {start, count} <- {integer(), non_neg_integer()} do
    list = Enum.to_list(start..(start + count))
    count + 1 == length(list) and increments(list)
  end
end

def increments([head | tail]), do: increments(head, tail)

defp increments(_, []), do: true

defp increments(n, [head | tail]) when head == n + 1,
  do: increments(head, tail)

defp increments(_, _), do: false
```

Thinking In Properties

Modeling

ThinkingInProperties/elixir/pbt/lib/pbt.ex

```
defmodule Pbt do
  def biggest([head | tail]) do
    biggest(tail, head)
  end

  defp biggest([], max) do
    max
  end

  defp biggest([head | tail], max) when head >= max do
    biggest(tail, head)
  end

  defp biggest([head | tail], max) when head < max do
    biggest(tail, max)
  end

end
```

ThinkingInProperties/elixir/pbt/test/pbt_test.exs

```
property "finds biggest element" do
  forall x <- non_empty(list(integer())) do
    Pbt.biggest(x) == model_biggest(x)
  end
end

def model_biggest(list) do
  List.last(Enum.sort(list))
```

```
end
```

Generalizing Example Tests

ThinkingInProperties/elixir/pbt/test/pbt_test.exs

```
property "picks the last number" do
  forall {list, known_last} <- {list(number()), number()} do
    known_list = list ++ [known_last]
    known_last == List.last(known_list)
  end
end
```

Invariants

ThinkingInProperties/elixir/pbt/test/pbt_test.exs

```
property "a sorted list has ordered pairs" do
  forall list <- list(term()) do
    is_ordered(Enum.sort(list))
  end
end

def is_ordered([a, b | t]) do
  a <= b and is_ordered([b | t])
end

# lists with fewer than 2 elements
def is_ordered(_) do
  true
end
```

ThinkingInProperties/elixir/pbt/test/pbt_test.exs

```
property "a sorted list keeps its size" do
  forall l <- list(number()) do
    length(l) == length(Enum.sort(l))
  end
end

property "no element added" do
  forall l <- list(number()) do
    sorted = Enum.sort(l)
    Enum.all?(sorted, fn element -> element in l end)
  end
end

property "no element deleted" do
  forall l <- list(number()) do
    sorted = Enum.sort(l)
    Enum.all?(l, fn element -> element in sorted end)
  end
end
```

Symmetric Properties

ThinkingInProperties/elixir/pbt/test/pbt_test.exs

```
property "symmetric encoding/decoding" do
  forall data <- list({atom(), any()}) do
    encoded = encode(data)
    is_binary(encoded) and data == decode(encoded)
  end
end

def encode(t), do: :erlang.term_to_binary(t)
def decode(t), do: :erlang.binary_to_term(t)
```

Putting it All Together

ThinkingInProperties/elixir/pbt/test/pbt_test.exs

```
defmodule PbtTest do
  use ExUnit.Case
  use PropCheck

  property "finds biggest element" do
    forall x <- non_empty(list(integer())) do
      Pbt.biggest(x) == model_biggest(x)
    end
  end

  def model_biggest(list) do
    List.last(Enum.sort(list))
  end

  property "picks the last number" do
    forall {list, known_last} <- {list(number()), number()} do
      known_list = list ++ [known_last]
      known_last == List.last(known_list)
    end
  end

  property "a sorted list has ordered pairs" do
    forall list <- list(term()) do
      is_ordered(Enum.sort(list))
    end
  end

  def is_ordered([a, b | t]) do
    a <= b and is_ordered([b | t])
  end

  # lists with fewer than 2 elements
  def is_ordered(_) do
    true
  end

  property "a sorted list keeps its size" do
```

```
    forall l <- list(number()) do
      length(l) == length(Enum.sort(l))
    end
  end

  property "no element added" do
    forall l <- list(number()) do
      sorted = Enum.sort(l)
      Enum.all?(sorted, fn element -> element in l end)
    end
  end

  property "no element deleted" do
    forall l <- list(number()) do
      sorted = Enum.sort(l)
      Enum.all?(l, fn element -> element in sorted end)
    end
  end

  property "symmetric encoding/decoding" do
    forall data <- list({atom(), any()}) do
      encoded = encode(data)
      is_binary(encoded) and data == decode(encoded)
    end
  end

  def encode(t), do: :erlang.term_to_binary(t)
  def decode(t), do: :erlang.binary_to_term(t)
end
```

Exercises

Question 3

ThinkingInProperties/elixir/pbt/test/exercise_test.exs

```
property "set union" do
  forall {list_a, list_b} <- {list(number()), list(number())} do
    set_a = MapSet.new(list_a)
    set_b = MapSet.new(list_b)
    model_union = Enum.sort(list_a ++ list_b)

    res =
      MapSet.union(set_a, set_b)
      |> MapSet.to_list()
      |> Enum.sort()

    res == model_union
  end
end
```

Question 4

ThinkingInProperties/elixir/pbt/test/exercise_test.exs

```
property "merge dictionaries" do
  forall {list_a, list_b} <-
           {list({term(), term()}), list({term(), term()})} do
    merged =
      Map.merge(Map.new(list_a), Map.new(list_b), fn _k,v1,_v2 -> v1 end)

    extract_keys(Enum.sort(Map.to_list(merged))) ==
      Enum.sort(Enum.uniq(extract_keys(list_a ++ list_b)))
  end
end

def extract_keys(list), do: for({k, _} <- list, do: k)
```

Custom Generators

The Limitations of Default Generators

```
property "find all keys in a map even when dupes are used" do
  forall kv <- list({key(), val()}) do
    m = Map.new(kv)
    for {k,_v} <- kv, do: Map.fetch!(m, k)
    true
  end
end

def key(), do: integer()
def val(), do: term()
```

Gathering Statistics

Collecting

CustomGenerators/elixir/pbt/test/generators_test.exs

```
# make verbose for metrics
property "collect 1", [:verbose] do
  forall bin <- binary() do
    #       test            metric
    collect(is_binary(bin), byte_size(bin))
  end
end
```

CustomGenerators/elixir/pbt/test/generators_test.exs

```
# make verbose for metrics
property "collect 2", [:verbose] do
  forall bin <- binary() do
    #       test            metric
    collect(is_binary(bin), to_range(10, byte_size(bin)))
  end
end
```

```
def to_range(m, n) do
  base = div(n, m)
  {base * m, (base + 1) * m}
end
```

CustomGenerators/elixir/pbt/test/generators_test.exs

```
property "find all keys in a map even when dupes are used", [:verbose] do
  forall kv <- list({key(), val()}) do
    m = Map.new(kv)
    for {k,_v} <- kv, do: Map.fetch!(m, k)
    uniques =
      kv
        |> List.keysort(0)
        |> Enum.dedup_by(fn {x, _} -> x end)
    collect(true, {:dupes, to_range(5, length(kv) - length(uniques))})
  end
end
```

CustomGenerators/elixir/pbt/test/generators_test.exs

```
def key(), do: oneof([range(1,10), integer()])
def val(), do: term()
```

Aggregating

CustomGenerators/elixir/pbt/test/generators_test.exs

```
property "aggregate", [:verbose] do
  suits = [:club, :diamond, :heart, :spade]

  forall hand <- vector(5, {oneof(suits), choose(1, 13)}) do
    # always pass
    aggregate(true, hand)
  end
end
```

CustomGenerators/elixir/pbt/test/generators_test.exs

```
property "fake escaping test showcasing aggregation", [:verbose] do
  forall str <- utf8() do
      aggregate(escape(str), classes(str))
  end
end

# this is a check we don't care about
defp escape(_), do: true

def classes(str) do
  l = letters(str)
  n = numbers(str)
  p = punctuation(str)
  o = String.length(str) - (l+n+p)
  [{:letters, to_range(5, l)},
   {:numbers, to_range(5, n)},
   {:punctuation, to_range(5, p)},
   {:others, to_range(5, o)}]
```

```
end

def letters(str) do
  is_letter = fn c -> (c >= ?a && c <= ?z) || (c >= ?A && c <= ?Z) end
  length(for <<c::utf8 <- str>>, is_letter.(c), do: 1)
end

def numbers(str) do
  is_num = fn c -> c >= ?0 && c <= ?9 end
  length(for <<c::utf8 <- str>>, is_num.(c), do: 1)
end

def punctuation(str) do
  is_punctuation = fn c -> c in '.,;:\'"-' end
  length(for <<c::utf8 <- str>>, is_punctuation.(c), do: 1)
end
```

Basic Custom Generators

Resizing Generators

CustomGenerators/elixir/pbt/test/generators_test.exs

```
property "resize", [:verbose] do
  forall bin <- resize(150, binary()) do
    collect(is_binary(bin), to_range(10, byte_size(bin)))
  end
end
```

CustomGenerators/elixir/pbt/test/generators_test.exs

```
property "profile 1", [:verbose] do
  forall profile <- [
           name: resize(10, utf8()),
           age: pos_integer(),
           bio: resize(350, utf8())
         ] do
    name_len = to_range(10, String.length(profile[:name]))
    bio_len = to_range(300, String.length(profile[:bio]))
    aggregate(true, name: name_len, bio: bio_len)
  end
end
```

CustomGenerators/elixir/pbt/test/generators_test.exs

```
property "profile 2", [:verbose] do
  forall profile <- [
           name: utf8(),
           age: pos_integer(),
           bio: sized(s, resize(s * 35, utf8()))
         ] do
    name_len = to_range(10, String.length(profile[:name]))
    bio_len = to_range(300, String.length(profile[:bio]))
    aggregate(true, name: name_len, bio: bio_len)
  end
```

```
end
```

Transforming Generators

CustomGenerators/elixir/pbt/test/generators_test.exs

```
property "naive queue generation" do
  forall list <- list({term(), term()}) do
    q = :queue.from_list(list)
    :queue.is_queue(q)
  end
end
```

CustomGenerators/elixir/pbt/test/generators_test.exs

```
property "queue with let macro" do
  forall q <- queue() do
    :queue.is_queue(q)
  end
end

def queue() do
  let list <- list({term(), term()}) do
    :queue.from_list(list)
  end
end
```

Imposing Restrictions

```
def non_empty(list_type) do
  such_that l <- list_type, when: l != [] and l != <<>>
end

def non_empty_map(gen) do
  such_that g <- gen, when: g != %{}
end

def even(), do: such_that n <- integer(), when: rem(n, 2) == 0
def uneven(), do: such_that n <- integer(), when: rem(n, 2) != 0

def even(), do: let n <- integer(), do: n * 2
def uneven(), do: let n <- integer(), do: n * 2 + 1
```

Changing Probabilities

CustomGenerators/elixir/pbt/test/generators_test.exs

```
def text_like() do
  let l <-
        list(
          frequency([
            {80, range(?a, ?z)},
            {10, ?\s},
            {1, ?\n},
            {1, oneof([?., ?-, ?!, ??, ?,])},
            {1, range(?0, ?9)}
```

```
          ])
        ) do
    to_string(l)
  end
end
```

CustomGenerators/elixir/pbt/test/generators_test.exs

```
def mostly_sorted() do
  gen = list(
    frequency([
      {5, sorted_list()},
      {1, list()}
    ])
  )
  let lists <- gen, do: Enum.concat(lists)
end

def sorted_list() do
  let l <- list(), do: Enum.sort(l)
end
```

Fancy Custom Generators

Recursive Generators

```
def path(), do: list(oneof([:left, :right, :up, :down]))

def path() do
  path({0,0}, [], %{{0,0} => :seen}, [])
end

def path(_current, acc, _seen, [_,_,_,_]) do
  acc
end
def path(current, acc, seen, ignore) do
  frequency([
    {1, acc},
    {15, increase_path(current, acc, seen, ignore)}
  ])
end

def increase_path(current, acc, seen, ignore) do
  let direction <- oneof([:left, :right, :up, :down] -- ignore) do
    new_pos = move(direction, current)
    case seen do
      %{^new_pos => _} ->
        path(current, acc, seen, [direction|ignore])
      _ ->
        path(new_pos, [direction|acc], Map.put(seen, new_pos, :seen), [])
    end
  end
```

```
end

def move(:left, {x, y}), do: {x-1, y}
def move(:right, {x, y}), do: {x+1, y}
def move(:up, {x, y}), do: {x, y+1}
def move(:down, {x, y}), do: {x, y-1}

def path(_current, acc, _seen, [_,_,_,_]) do
  acc
end
def path(current, acc, seen, ignore) do
  frequency([
    {1, acc},
    {15, lazy(increase_path(current, acc, seen, ignore))}
  ])
end
```

CustomGenerators/elixir/pbt/test/generators_test.exs

```
def path() do
  sized(
    size,
    path(size, {0,0}, [], %{{0,0} => :seen}, [])
  )
end

def path(0, _current, acc, _seen, _ignore) do
  acc
end
def path(_max, _current, acc, _seen, [_,_,_,_]) do
  acc
end
def path(max, current, acc, seen, ignore) do
    increase_path(max, current, acc, seen, ignore)
end

def increase_path(max, current, acc, seen, ignore) do
  let direction <- oneof([:left, :right, :up, :down] -- ignore) do
    new_pos = move(direction, current)
    case seen do
      %{^new_pos => _} ->
        path(max, current, acc, seen, [direction|ignore])
      _ ->
        path(
          max-1,
          new_pos,
          [direction|acc],
          Map.put(seen, new_pos, :seen),
          []
        )
    end
  end
end
```

Symbolic Calls

CustomGenerators/elixir/pbt/test/generators_test.exs

```
def dict_gen() do
  let(x <- list({integer(), integer()}), do: :dict.from_list(x))
end

def dict_symb(),
  do: sized(size, dict_symb(size, {:call, :dict, :new, []}))

def dict_symb(0, dict), do: dict

def dict_symb(n, dict) do
  dict_symb(n - 1, {:call, :dict, :store, [integer(), integer(), dict]})
end

def dict_autosymb() do
  sized(size, dict_autosymb(size, {:"$call", :dict, :new, []}))
end

def dict_autosymb(0, dict), do: dict

def dict_autosymb(n, dict) do
  dict_autosymb(
    n - 1,
    {:"$call", :dict, :store, [integer(), integer(), dict]}
  )
end
```

CustomGenerators/elixir/pbt/test/generators_test.exs

```
property "dict generator" do
  forall d <- dict_gen() do
    :dict.size(d) < 5
  end
end

property "symbolic generator" do
  forall d <- dict_symb() do
    # propcheck does not automatically handle eval() calls
    :proper_gen.eval(:dict.size(d)) < 5
  end
end

property "automated symbolic generator" do
  forall d <- dict_autosymb() do
    :dict.size(d) < 5
  end
end
```

Exercises

Question 4

CustomGenerators/elixir/pbt/test/exercises_test.exs

```
def tree(), do: tree(term())
```

```
def tree(type) do
  frequency([
    {1, {:node, type, tree(type), nil}},
    {1, {:node, type, nil, tree(Type)}},
    {5, {:node, type, tree(type), tree(type)}}
  ])
end
```

Question 5

CustomGenerators/elixir/pbt/test/exercises_test.exs

```
def stamp(), do: {hour(), min(), sec()}
def hour(), do: choose(0, 23)
def min(), do: choose(0, 59)
def sec(), do: choose(0, 59)

# returning hours in the morning
def am_stamp1() do
  such_that({h, _, _} <- stamp(), when: h < 12)
end

def am_stamp2() do
  let({h, m, s} <- stamp(), do: {rem(h, 12), m, s})
end

# Return two ordered timestamps
def stamps1() do
  such_that({s1, s2} <- {stamp(), stamp()}, when: s1 <= s2)
end

def stamps2() do
  let({s1, s2} <- {stamp(), stamp()}, do: {min(s1, s2), max(s1, s2)})
end

# Return any time that does not overlap standup meetings
def no_standup1() do
  such_that({h, m, _} <- stamp(), when: h != 9 or m > 10)
end

def no_standup2() do
  let {h, m, s} <- stamp() do
    case h do
      9 when m <= 10 -> {8, m, s}
      _ -> {h, m, s}
    end
  end
end
```

Question 6

CustomGenerators/elixir/pbt/test/exercises_test.exs

```
def file_open(name, opts) do
  {:ok, fd} = File.open(name, opts)
```

```
  # ensure the file is refreshed on each run
  :file.truncate(fd)
  fd
end

def file_write(fd, data) do
  IO.write(fd, data)
  fd
end
```

Responsible Testing

CSV Parsing

ResponsibleTesting/elixir/bday/test/csv_test.exs

```
def field() do
  oneof([unquoted_text(), quotable_text()])
end

def unquoted_text() do
  let chars <- list(elements(textdata())) do
    to_string(chars)
  end
end

# using charlists for the easy generation
def quotable_text() do
  let chars <- list(elements('\r\n",' ++ textdata())) do
    to_string(chars)
  end
end

def textdata() do
  'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789' ++
    ':;<=>?@ !#$%&\'()*+-./[\\]^_`{|}~'
end
```

ResponsibleTesting/elixir/bday/test/csv_test.exs

```
def header(size) do
  vector(size, name())
end

def record(size) do
  vector(size, field())
end

def name() do
  field()
end

def csv_source() do
  let size <- pos_integer() do
    let keys <- header(size) do
```

```
      list(entry(size, keys))
    end
  end
end

def entry(size, keys) do
  let vals <- record(size) do
    Map.new(Enum.zip(keys, vals))
  end
end
```

ResponsibleTesting/elixir/bday/test/csv_test.exs

```
property "roundtrip encoding/decoding" do
  forall maps <- csv_source() do
    maps == Csv.decode(Csv.encode(maps))
  end
end
```

ResponsibleTesting/elixir/bday/lib/csv.ex

```
defmodule Bday.Csv do
  def encode([]), do: ""

  def encode(maps) do
    keys = Enum.map_join(Map.keys(hd(maps)), ",", &escape(&1))

    vals =
      for map <- maps, do: Enum.map_join(Map.values(map), ",", &escape(&1))

    to_string([keys, "\r\n", Enum.join(vals, "\r\n")])
  end

  def decode(""), do: []

  def decode(csv) do
    {headers, rest} = decode_header(csv, [])
    rows = decode_rows(rest)
    for row <- rows, do: Map.new(Enum.zip(headers, row))
  end

  defp escape(field) do
    if escapable(field) do
      ~s|"| <> do_escape(field) <> ~s|"|
    else
      field
    end
  end

  defp escapable(string) do
    String.contains?(string, [~s|"|, ",", "\r", "\n"])
  end

  defp do_escape(""), do: ""
  defp do_escape(~s|"| <> str), do: ~s|""| <> do_escape(str)
  defp do_escape(<<char>> <> rest), do: <<char>> <> do_escape(rest)

  defp decode_header(string, acc) do
```

```
    case decode_name(string) do
      {:ok, name, rest} -> decode_header(rest, [name | acc])
      {:done, name, rest} -> {[name | acc], rest}
    end
  end

  defp decode_rows(string) do
    case decode_row(string, []) do
      {row, ""} -> [row]
      {row, rest} -> [row | decode_rows(rest)]
    end
  end

  defp decode_row(string, acc) do
    case decode_field(string) do
      {:ok, field, rest} -> decode_row(rest, [field | acc])
      {:done, field, rest} -> {[field | acc], rest}
    end
  end

  defp decode_name(~s|"| <> rest), do: decode_quoted(rest)
  defp decode_name(string), do: decode_unquoted(string)

  defp decode_field(~s|"| <> rest), do: decode_quoted(rest)
  defp decode_field(string), do: decode_unquoted(string)

  defp decode_quoted(string), do: decode_quoted(string, "")

  defp decode_quoted(~s|"|, acc), do: {:done, acc, ""}
  defp decode_quoted(~s|"\r\n| <> rest, acc), do: {:done, acc, rest}
  defp decode_quoted(~s|",| <> rest, acc), do: {:ok, acc, rest}

  defp decode_quoted(~s|""| <> rest, acc) do
    decode_quoted(rest, acc <> ~s|"|)
  end

  defp decode_quoted(<<char>> <> rest, acc) do
    decode_quoted(rest, acc <> <<char>>)
  end

  defp decode_unquoted(string), do: decode_unquoted(string, "")

  defp decode_unquoted("", acc), do: {:done, acc, ""}
  defp decode_unquoted("\r\n" <> rest, acc), do: {:done, acc, rest}
  defp decode_unquoted("," <> rest, acc), do: {:ok, acc, rest}

  defp decode_unquoted(<<char>> <> rest, acc) do
    decode_unquoted(rest, acc <> <<char>>)
  end
end
```

ResponsibleTesting/elixir/bday/test/csv_test.exs

```
def csv_source() do
  let size <- pos_integer() do
    let keys <- header(size + 1) do
      list(entry(size + 1, keys))
```

```
    end
  end
end
```

ResponsibleTesting/elixir/bday/test/csv_test.exs

```
## Unit Tests ##
test "one column CSV files are inherently ambiguous" do
  assert "\r\n\r\n" == Csv.encode([%{"" => ""}, %{"" => ""}])
  assert [%{"" => ""}] == Csv.decode("\r\n\r\n")
end
```

ResponsibleTesting/elixir/bday/test/csv_test.exs

```
test "one record per line" do
  assert [%{"aaa" => "zzz", "bbb" => "yyy", "ccc" => "xxx"}] ==
           Csv.decode("aaa,bbb,ccc\r\nzzz,yyy,xxx\r\n")
end

test "optional trailing CRLF" do
  assert [%{"aaa" => "zzz", "bbb" => "yyy", "ccc" => "xxx"}] ==
           Csv.decode("aaa,bbb,ccc\r\nzzz,yyy,xxx")
end

test "double quotes" do
  assert [%{"aaa" => "zzz", "bbb" => "yyy", "ccc" => "xxx"}] ==
           Csv.decode("\"aaa\",\"bbb\",\"ccc\"\r\nzzz,yyy,xxx")
end

test "escape CRLF" do
  assert [%{"aaa" => "zzz", "b\r\nbb" => "yyy", "ccc" => "xxx"}] ==
           Csv.decode("\"aaa\",\"b\r\nbb\",\"ccc\"\r\nzzz,yyy,xxx")
end

test "double quote escaping" do
  # Since we decided headers are mandatory, this test adds a line
  # with empty values (CLRF,,) to the example from the RFC.
  assert [%{"aaa" => "", "b\"bb" => "", "ccc" => ""}] ==
           Csv.decode("\"aaa\",\"b\"\"bb\",\"ccc\"\r\n,,")
end

# this counterexample is taken literally from the RFC and
# cannot work with the current implementation because maps
# do not allow duplicate keys
test "dupe keys unsupported" do
  csv =
    "field_name,field_name,field_name\r\n" <>
      "aaa,bbb,ccc\r\n" <> "zzz,yyy,xxx\r\n"

  [map1, map2] = Csv.decode(csv)
  assert ["field_name"] == Map.keys(map1)
  assert ["field_name"] == Map.keys(map2)
end
```

Filtering Records

ResponsibleTesting/elixir/bday/test/filter_test.exs

```
defmodule FilterTest do
  use ExUnit.Case
  alias Bday.Filter, as: Filter

  test "property-style filtering test" do
    years = generate_years_data(2018, 2038)
    people = generate_people_for_year(3)

    for yeardata <- years do
      birthdays = find_birthdays_for_year(people, yeardata)
      every_birthday_once(people, birthdays)
      on_right_date(people, birthdays)
    end
  end

  defp find_birthdays_for_year(_, []), do: []

  defp find_birthdays_for_year(people, [day | year]) do
    found = Filter.birthday(people, day) # <- function being tested
    [{day, found} | find_birthdays_for_year(people, year)]
  end
```

ResponsibleTesting/elixir/bday/test/filter_test.exs

```
# Generators
defp generate_years_data(stop, stop), do: []

defp generate_years_data(start, stop) do
  [generate_year_data(start) | generate_years_data(start + 1, stop)]
end

defp generate_year_data(year) do
  {:ok, date} = Date.new(year, 1, 1)

  days_in_feb =
    case Date.leap_year?(date) do
      true -> 29
      false -> 28
    end

  month(year, 1, 31) ++
    month(year, 2, days_in_feb) ++
    month(year, 3, 31) ++
    month(year, 4, 30) ++
    month(year, 5, 31) ++
    month(year, 6, 30) ++
    month(year, 7, 31) ++
    month(year, 8, 31) ++
    month(year, 9, 30) ++
    month(year, 10, 31) ++ month(year, 11, 30) ++ month(year, 12, 31)
end

defp month(y, m, 1) do
```

```
  {:ok, date} = Date.new(y, m, 1)
  [date]
end

defp month(y, m, n) do
  {:ok, date} = Date.new(y, m, n)
  [date | month(y, m, n - 1)]
end
```

ResponsibleTesting/elixir/bday/test/filter_test.exs

```
defp generate_people_for_year(n) do
  # leap year so all days are covered
  year_seed = generate_year_data(2016)
  Enum.flat_map(1..n, fn _ -> people_for_year(year_seed) end)
end

defp people_for_year(year) do
  for date <- year do
    person_for_date(date)
  end
end

defp person_for_date(%Date{month: m, day: d} = date) do
  case Date.new(:rand.uniform(100) + 1900, m, d) do
    {:error, :invalid_date} ->
      person_for_date(date)

    {:ok, date} ->
      %{"name" => make_ref(), "date_of_birth" => date}
  end
end
```

ResponsibleTesting/elixir/bday/test/filter_test.exs

```
defp every_birthday_once(people, birthdays) do
  found =
    birthdays
    |> Enum.flat_map(fn {_, found} -> found end)
    |> Enum.sort()

  not_found = people -- found
  found_many_times = found -- Enum.uniq(found)
  assert [] == not_found
  assert [] == found_many_times
end

defp on_right_date(_people, birthdays) do
  for {date, found} <- birthdays do
    for %{"date_of_birth" => dob} <- found do
      case Date.new(date.year, dob.month, dob.day) do
        {:error, :invalid_date} -> :skip
        _ -> assert {date.month, date.day} == {dob.month, dob.day}
      end
    end
  end
```

```
end
```

ResponsibleTesting/elixir/bday/lib/filter.ex

```
defmodule Bday.Filter do
  def birthday(people, date = %Date{month: 2, day: 28}) do
    case Date.leap_year?(date) do
      true -> filter_dob(people, 2, 28)
      false -> filter_dob(people, 2, 28) ++ filter_dob(people, 2, 29)
    end
  end

  def birthday(people, %Date{month: m, day: d}) do
    filter_dob(people, m, d)
  end

  defp filter_dob(people, month, day) do
    Enum.filter(
      people,
      fn %{"date_of_birth" => %Date{month: m, day: d}} ->
        {month, day} == {m, d}
      end
    )
  end
end
```

EmployeeModule

ResponsibleTesting/elixir/bday/test/employee_test.exs

```
defp raw_employee_map() do
  let proplist <- [
        {"last_name", CsvTest.field()},
        {" first_name", whitespaced_text()},
        {" date_of_birth", text_date()},
        {" email", whitespaced_text()}
      ] do
    Map.new(proplist)
  end
end

defp whitespaced_text() do
  let(txt <- CsvTest.field(), do: " " <> txt)
end

defp text_date() do
  rawdate = {choose(1900, 2020), choose(1, 12), choose(1, 31)}
  # only generate valid dates
  date =
    such_that(
      {y, m, d} <- rawdate,
      when: {:error, :invalid_date} != Date.new(y, m, d)
    )

  let {y, m, d} <- date do
```

```
    IO.chardata_to_string(:io_lib.format(" ~w/~2..0w/~2..0w", [y, m, d]))
  end
end
```

ResponsibleTesting/elixir/bday/lib/employee.ex

```
@opaque employee() :: %{required(String.t()) => term()}
@opaque handle() :: {:raw, [employee()]}

def from_csv(string) do
  {:raw,
   for map <- Bday.Csv.decode(string) do
     adapt_csv_result(map)
   end}
end
```

ResponsibleTesting/elixir/bday/lib/employee.ex

```
defp adapt_csv_result(map) do
  map =
    for {k, v} <- map, into: %{} do
      {trim(k), maybe_null(trim(v))}
    end

  dob = Map.fetch!(map, "date_of_birth")
  %{map | "date_of_birth" => parse_date(dob)}
end

defp trim(str), do: String.trim_leading(str, " ")

defp maybe_null(""), do: nil
defp maybe_null(str), do: str

defp parse_date(str) do
  [y, m, d] = Enum.map(String.split(str, "/"), &String.to_integer(&1))
  {:ok, date} = Date.new(y, m, d)
  date
end
```

ResponsibleTesting/elixir/bday/lib/employee.ex

```
@spec last_name(employee()) :: String.t() | nil
def last_name(%{"last_name" => name}), do: name

@spec first_name(employee()) :: String.t() | nil
def first_name(%{"first_name" => name}), do: name

@spec date_of_birth(employee()) :: Date.t()
def date_of_birth(%{"date_of_birth" => dob}), do: dob

@spec email(employee()) :: String.t()
def email(%{"email" => email}), do: email
```

ResponsibleTesting/elixir/bday/lib/employee.ex

```
@spec fetch(handle()) :: [employee()]
def fetch({:raw, maps}), do: maps
```

ResponsibleTesting/elixir/bday/lib/employee.ex

```
@spec filter_birthday(handle(), Date.t()) :: handle()
```

```
def filter_birthday({:raw, employees}, date) do
  {:raw, Bday.Filter.birthday(employees, date)}
end
```

ResponsibleTesting/elixir/bday/test/employee_test.exs

```
property "check access through the handle" do
  forall maps <- non_empty(list(raw_employee_map())) do
    handle =
      maps
      |> Csv.encode()
      |> Employee.from_csv()

    partial = Employee.filter_birthday(handle, ~D[1900-01-01])
    list = Employee.fetch(partial)
    # check for absence of crash
    for x <- list do
      Employee.first_name(x)
      Employee.last_name(x)
      Employee.email(x)
      Employee.date_of_birth(x)
    end

    true
  end
end
```

Templating

ResponsibleTesting/elixir/bday/test/mail_tpl_test.exs

```
defmodule MailTplTest do
  use ExUnit.Case
  use PropCheck
  alias Bday.MailTpl, as: MailTpl

  property "email template has first name" do
    forall employee <- employee_map() do
      String.contains?(
        MailTpl.body(employee),
        Map.fetch!(employee, "first_name")
      )
    end
  end

  defp employee_map() do
    let proplist <- [
          {"last_name", non_empty(CsvTest.field())},
          {"first_name", non_empty(CsvTest.field())},
          {"date_of_birth", date()},
          {"email", non_empty(CsvTest.field())}
        ] do
      Enum.reduce(proplist, %{}, fn {k, v}, m -> Map.put(m, k, v) end)
    end
  end
```

```
  defp date() do
    rawdate = {choose(1900, 2020), choose(1, 12), choose(1, 31)}
    # only generate valid dates
    date =
      such_that(
        {y, m, d} <- rawdate,
        when: {:error, :invalid_date} != Date.new(y, m, d)
      )

    let {y, m, d} <- date do
      {:ok, val} = Date.new(y, m, d)
      val
    end
  end
end
```

ResponsibleTesting/elixir/bday/lib/mail_tpl.ex

```
defmodule Bday.MailTpl do
  def body(employee) do
    name = Bday.Employee.first_name(employee)
    "Happy birthday, dear #{name}!"
  end

  def full(employee) do
    {[Bday.Employee.email(employee)], "Happy birthday!", body(employee)}
  end
end
```

Plumbing it all Together

ResponsibleTesting/elixir/bday/lib/bday.ex

```
defmodule Bday do
  def run(path) do
    set =
      path
      |> File.read!()
      |> Bday.Employee.from_csv()
      |> Bday.Employee.filter_birthday(DateTime.to_date(DateTime.utc_now()))
      |> Bday.Employee.fetch()

    for employee <- set do
      employee
      |> Bday.MailTpl.full()
      |> send_email()
    end

    :ok
  end

  defp send_email({to, _topic, _body}) do
    IO.puts("sent birthday email to #{to}")
  end
end
```

Properties-Driven Development

Writing the First Test

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
defmodule CheckoutTest do
  use ExUnit.Case
  use PropCheck

  ## Properties
  property "sums without specials" do
    forall {item_list, expected_price, price_list} <- item_price_list() do
      expected_price == Checkout.total(item_list, price_list, [])
    end
  end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
## Generators
defp item_price_list() do
  let price_list <- price_list() do
    let {item_list, expected_price} <- item_list(price_list) do
      {item_list, expected_price, price_list}
    end
  end
end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
defp price_list() do
  let price_list <- non_empty(list({non_empty(utf8()), integer()})) do
    sorted = Enum.sort(price_list)
    Enum.dedup_by(sorted, fn {x, _} -> x end)
  end
end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
defp item_list(price_list) do
  sized(size, item_list(size, price_list, {[], 0}))
end

defp item_list(0, _, acc), do: acc

defp item_list(n, price_list, {item_acc, price_acc}) do
  let {item, price} <- elements(price_list) do
    item_list(n - 1, price_list, {[item | item_acc], price + price_acc})
  end
end
```

PropertiesDrivenDevelopment/elixir/checkout/lib/checkout.ex

```
def total(item_list, price_list, _specials) do
  Enum.sum(
    for item <- item_list do
      elem(List.keyfind(price_list, item, 0), 1)
```

```
    end
  )
end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
property "sums without specials (but with metrics)", [:verbose] do
  forall {item_list, expected_price, price_list} <- item_price_list() do
    (expected_price == Checkout.total(item_list, price_list, []))
    |> collect(bucket(length(item_list), 10))
  end
end

## Helpers
defp bucket(n, unit) do
  div(n, unit) * unit
end
```

Testing Specials

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
property "sums including specials" do
  forall {items, expected_price, prices, specials}
         <- item_price_special() do
    expected_price == Checkout.total(items, prices, specials)
  end
end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
defp item_price_special() do
  # first let: freeze the price list
  let price_list <- price_list() do
    # second let: freeze the list of specials
    let special_list <- special_list(price_list) do
      # third let: regular + special items and prices
      let {{regular_items, regular_expected},
           {special_items, special_expected}} <-
            {regular_gen(price_list, special_list),
             special_gen(price_list, special_list)} do
        # and merge + return initial lists:
        {Enum.shuffle(regular_items ++ special_items),
         regular_expected + special_expected, price_list, special_list}
      end
    end
  end
end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
defp special_list(price_list) do
  items = for {name, _} <- price_list, do: name

  let specials <- list({elements(items), choose(2, 5), integer()}) do
    sorted = Enum.sort(specials)
```

```
    Enum.dedup_by(sorted, fn {x, _, _} -> x end)
  end
end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
defp regular_gen(price_list, special_list) do
  regular_gen(price_list, special_list, [], 0)
end

defp regular_gen([], _, list, price), do: {list, price}

defp regular_gen([{item, cost} | prices], specials, items, price) do
  count_gen =
    case List.keyfind(specials, item, 0) do
      {_, limit, _} -> choose(0, limit - 1)
      _ -> non_neg_integer()
    end

  let count <- count_gen do
    regular_gen(
      prices,
      specials,
      let(v <- vector(count, item), do: v ++ items),
      cost * count + price
    )
  end
end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
defp special_gen(_, special_list) do
  special_gen(special_list, [], 0)
end

defp special_gen([], items, price), do: {items, price}

defp special_gen([{item, count, cost} | specials], items, price) do
  let multiplier <- non_neg_integer() do
    special_gen(
      specials,
      let(v <- vector(count * multiplier, item), do: v ++ items),
      cost * multiplier + price
    )
  end
end
```

Implementing Specials

PropertiesDrivenDevelopment/elixir/checkout/lib/checkout.ex

```
def total(item_list, price_list, specials) do
  counts = count_seen(item_list)
  {counts_left, prices} = apply_specials(counts, specials)
  prices + apply_regular(counts_left, price_list)
end
```

PropertiesDrivenDevelopment/elixir/checkout/lib/checkout.ex
```
defp count_seen(item_list) do
  count = fn x -> x + 1 end

  Map.to_list(
    Enum.reduce(item_list, Map.new(), fn item, m ->
      Map.update(m, item, 1, count)
    end)
  )
end
```

PropertiesDrivenDevelopment/elixir/checkout/lib/checkout.ex
```
defp apply_specials(items, specials) do
  Enum.map_reduce(items, 0, fn {name, count}, price ->
    case List.keyfind(specials, name, 0) do
      nil ->
        {{name, count}, price}

      {_, needed, value} ->
        {{name, rem(count, needed)}, value * div(count, needed) + price}
    end
  end)
end
```

PropertiesDrivenDevelopment/elixir/checkout/lib/checkout.ex
```
defp apply_regular(items, price_list) do
  Enum.sum(
    for {name, count} <- items do
      {_, price} = List.keyfind(price_list, name, 0)
      count * price
    end
  )
end
```

Negative Testing

```
defp lax_lists() do
  {list(utf8()),
   list({utf8(), integer()}),
   list({utf8(), integer(), integer()})}
end

property "negative testing for expected results" do
  forall {items, prices, specials} <- lax_lists() do
    try do
      is_integer(Checkout.total(items, prices, specials))
    rescue
      e in [RuntimeError] ->
        String.starts_with?(e.message, "unknown item:")
      _ ->
        false
    end
  end
```

```
end
```

PropertiesDrivenDevelopment/elixir/checkout/lib/checkout.ex

```
defp apply_regular(items, price_list) do
  Enum.sum(
    for {name, count} <- items do
      count * cost_of_item(price_list, name)
    end
  )
end

defp cost_of_item(price_list, name) do
  case List.keyfind(price_list, name, 0) do
    nil -> raise RuntimeError, message: "unknown item: #{name}"
    {_, price} -> price
  end
end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
defp lax_lists() do
  known_items = ["A", "B", "C"]
  maybe_known_item_gen = elements(known_items ++ [utf8()])

  {list(maybe_known_item_gen), list({maybe_known_item_gen, integer()}),
   list({maybe_known_item_gen, integer(), integer()})}
end
```

Additional iteration of negative tests:

```
property "negative testing for expected results" do
  forall {items, prices, specials} <- lax_lists() do
    try do
      is_integer(Checkout.total(items, prices, specials))
    rescue
      e in [RuntimeError] ->
        e.message == "invalid list of specials" ||
          String.starts_with?(e.message, "unknown item:")

      _ ->
        false
    end
  end
end
```

PropertiesDrivenDevelopment/elixir/checkout/lib/checkout.ex

```
def valid_special_list(list) do
  Enum.all?(list, fn {_, x, _} -> x != 0 end)
end

def total(item_list, price_list, specials) do
  if not valid_special_list(specials) do
    raise RuntimeError, message: "invalid list of specials"
  end
```

```
  counts = count_seen(item_list)
  {counts_left, prices} = apply_specials(counts, specials)
  prices + apply_regular(counts_left, price_list)
end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
property "list of items with duplicates" do
  forall price_list <- dupe_list() do
    false == Checkout.valid_price_list(price_list)
  end
end

defp dupe_list() do
  let items <- non_empty(list(utf8())) do
    vector(length(items) + 1, {elements(items), integer()})
  end
end
```

PropertiesDrivenDevelopment/elixir/checkout/lib/checkout.ex

```
def valid_price_list(list) do
  sorted = Enum.sort(list)
  length(list) == length(Enum.dedup_by(sorted, fn {x, _} -> x end))
end

def total(item_list, price_list, specials) do
  if not valid_price_list(price_list) do
    raise RuntimeError, message: "invalid list of prices"
  end

  if not valid_special_list(specials) do
    raise RuntimeError, message: "invalid list of specials"
  end

  counts = count_seen(item_list)
  {counts_left, prices} = apply_specials(counts, specials)
  prices + apply_regular(counts_left, price_list)
end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
property "negative testing for expected results" do
  forall {items, prices, specials} <- lax_lists() do
    try do
      is_integer(Checkout.total(items, prices, specials))
    rescue
      e in [RuntimeError] ->
        e.message == "invalid list of prices" ||
          e.message == "invalid list of specials" ||
          String.starts_with?(e.message, "unknown item:")

      _ ->
        false
    end
  end
```

```
end
```

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
property "list of items with specials" do
  forall special_list <- dupe_special_list() do
    false == Checkout.valid_special_list(special_list)
  end
end

defp dupe_special_list() do
  let items <- non_empty(list(utf8())) do
    vector(length(items) + 1, {elements(items), integer(), integer()})
  end
end
```

PropertiesDrivenDevelopment/elixir/checkout/lib/checkout.ex

```
def valid_special_list(list) do
  sorted = Enum.sort(list)

  Enum.all?(list, fn {_, x, _} -> x != 0 end) &&
    length(list) == length(Enum.dedup_by(sorted, fn {x, _, _} -> x end))
end
```

Shrinking

?SHRINK

Shrinking/elixir/pbt/test/pbt_test.exs

```
def strdatetime() do
  let(date_time <- datetime(), do: to_str(date_time))
end

def datetime() do
  {date(), time(), timezone()}
end

def date() do
  such_that(
    {y, m, d} <- {year(), month(), day()},
    when: :calendar.valid_date(y, m, d)
  )
end

def year() do
  shrink(range(0, 9999), [range(1970, 2000), range(1900, 2100)])
end

def month(), do: range(1, 12)

def day(), do: range(1, 31)

def time(), do: {range(0, 24), range(0, 59), range(0, 60)}

def timezone() do
```

```
  {elements([:+, :-]), shrink(range(0, 99), [range(0, 14), 0]),
   shrink(range(0, 99), [0, 15, 30, 45])}
end

def to_str({{y, m, d}, {h, mi, s}, {sign, ho, mo}}) do
  format_str = "~4..0b-~2..0b-~2..0bT~2..0b:~2..0b:~2..0b~s~2..0b:~2..0b"

  :io_lib.format(format_str, [y, m, d, h, mi, s, sign, ho, mo])
  |> to_string()
end
```

?LETSHRINK

```
let_shrink([
    a <- list(number()),
    b <- list(number()),
    c <- list(number())
]) do
    a ++ b ++ c
end
```

Shrinking/elixir/pbt/test/pbt_test.exs

```
def tree(n) when n <= 1 do
  {:leaf, number()}
end

def tree(n) do
  per_branch = div(n, 2)
  {:branch, tree(per_branch), tree(per_branch)}
end
```

Shrinking/elixir/pbt/test/pbt_test.exs

```
def tree_shrink(n) when n <= 1 do
  {:leaf, number()}
end

def tree_shrink(n) do
  per_branch = div(n, 2)

  let_shrink([
    left <- tree_shrink(per_branch),
    right <- tree_shrink(per_branch)
  ]) do
    {:branch, left, right}
  end
end
```

Exercises

Question 3

Shrinking/elixir/pbt/test/pbt_test.exs

```
defmodule PbtTest do
  use ExUnit.Case
```

```
  use PropCheck

  property "dairy" do
    forall food <- meal() do
      dairy_count(food) == 0
    end
  end

  defp dairy_count(list) do
    Enum.count(list, fn x -> is_dairy(x) end)
  end

  defp is_dairy(:cheesesticks), do: true
  defp is_dairy(:lasagna), do: true
  defp is_dairy(:icecream), do: true
  defp is_dairy(:milk), do: true
  defp is_dairy(_), do: false

def meal() do
  let_shrink([
    appetizer <- elements([:soup, :salad, :cheesesticks]),
    drink <- elements([:coffee, :tea, :milk, :water, :juice]),
    entree <- elements([:lasagna, :tofu, :steak]),
    dessert <- elements([:cake, :chocolate, :icecream])
  ]) do
    [appetizer, drink, entree, dessert]
  end
end
```

Question 4

PropertiesDrivenDevelopment/elixir/checkout/test/checkout_test.exs

```
defp item_price_special() do
  # first let: freeze the price list
  let price_list <- price_list() do
    # second let: freeze the list of specials
    let special_list <- special_list(price_list) do
      # third let: regular + special items and prices
      let {{regular_items, regular_expected},
           {special_items, special_expected}} <-
            {regular_gen(price_list, special_list),
             special_gen(price_list, special_list)} do
        # and merge + return initial lists:
        {Enum.shuffle(regular_items ++ special_items),
         regular_expected + special_expected, price_list, special_list}
      end
    end
  end
end
```

Stateful Properties

Structure of Properties

```
defmodule PbtTest do
  use ExUnit.Case
  use PropCheck
  use PropCheck.StateM # <-- this is a new one to use

  property "stateful property" do
    forall cmds <- commands(__MODULE__) do
      ActualSystem.start_link()
      {history, state, result} = run_commands(__MODULE__, cmds)
      ActualSystem.stop()

      (result == :ok)
      |> aggregate(command_names(cmds))
      |> when_fail(
        IO.puts("""
        History: #{history}
        State: #{state}
        Result: #{result}
        """)
      )
    end
  end

  # Initial model value at system start. Should be deterministic.
  def initial_state() do
    %{}
  end

  # List of possible commands to run against the system
  def command(_state) do
    oneof([
      {:call, ActualSystem, :some_call, [term(), term()]}}
    ])
  end

  # Determines whether a command should be valid under the current state
  def precondition(_state, {:call, _mod, _fun, _args}) do
    true
  end

  # Given that state prior to the call `{:call, mod, fun, args}`,
  # determine whether the result (res) coming from the actual system
  # makes sense according to the model
  def postcondition(_state, {:call, _mod, _fun, _args}, _res) do
    true
  end

  # Assuming the postcondition for a call was true, update the model
  # accordingly for the test to proceed
```

```
  def next_state(state, _res, {:call, _mod, _fun, _args}) do
    newstate = state
    newstate
  end
end
```

Implementing the Cache

StatefulProperties/elixir/pbt/lib/cache.ex

```
defmodule Cache do
  use GenServer

  def start_link(n) do
    GenServer.start_link(__MODULE__, n, name: __MODULE__)
  end

  def stop() do
    GenServer.stop(__MODULE__)
  end

def init(n) do
  :ets.new(:cache, [:public, :named_table])
  :ets.insert(:cache, {:count, 0, n})
  {:ok, :nostate}
end

def handle_call(_call, _from, state), do: {:noreply, state}

def handle_case(_cast, state), do: {:noreply, state}

def handle_info(_msg, state), do: {:noreply, state}

def find(key) do
  case :ets.match(:cache, {:_, {key, :"$1"}}) do
    [[val]] -> {:ok, val}
    [] -> {:error, :not_found}
  end
end

def cache(key, val) do
  case :ets.match(:cache, {:"$1", {key, :_}}) do
    [[n]] ->
      :ets.insert(:cache, {n, {key, val}})

    [] ->
      case :ets.lookup(:cache, :count) do
        [{:count, max, max}] ->
          :ets.insert(:cache, [{1, {key, val}}, {:count, 1, max}])

        [{:count, current, max}] ->
          :ets.insert(:cache, [
            {current + 1, {key, val}},
            {:count, current + 1, max}
          ])
      end
```

```
  end
end

def flush() do
  [{:count, _, max}] = :ets.lookup(:cache, :count)
  :ets.delete_all_objects(:cache)
  :ets.insert(:cache, {:count, 0, max})
end
```

Writing the Tests

StatefulProperties/elixir/pbt/test/cache_test.exs

```
defmodule CacheTest do
  use ExUnit.Case
  use PropCheck
  use PropCheck.StateM
  doctest Cache

  @cache_size 10

  property "stateful property", [:verbose] do
    forall cmds <- commands(__MODULE__) do
      Cache.start_link(@cache_size)
      {history, state, result} = run_commands(__MODULE__, cmds)
      Cache.stop()

      (result == :ok)
      |> aggregate(command_names(cmds))
      |> when_fail(
        IO.puts("""
        History: #{history}
        State: #{state}
        Result: #{result}
        """)
      )
    end
  end
```

StatefulProperties/elixir/pbt/test/cache_test.exs

```
defmodule State do
  @cache_size 10
  defstruct max: @cache_size, count: 0, entries: []
end

def initial_state(), do: %State{}

def command(_state) do
  frequency([
    {1, {:call, Cache, :find, [key()]}},
    {3, {:call, Cache, :cache, [key(), val()]}},
    {1, {:call, Cache, :flush, []}}
  ])
end
```

```
def precondition(%State{count: 0}, {:call, Cache, :flush, []}) do
  false
end

def precondition(%State{}, {:call, _mod, _fun, _args}) do
  true
end

# Assuming the postcondition for a call was true, update the model
# accordingly for the test to proceed
def next_state(state, _res, {:call, Cache, :flush, _}) do
  %{state | count: 0, entries: []}
end

def next_state(
      s = %State{entries: l, count: n, max: m},
      _res,
      {:call, Cache, :cache, [k, v]}
    ) do
  case List.keyfind(l, k, 0) do
    nil when n == m ->
      %{s | entries: tl(l) ++ [{k, v}]}

    nil when n < m ->
      %{s | entries: l ++ [{k, v}], count: n + 1}

    {^k, _} ->
      %{s | entries: List.keyreplace(l, k, 0, {k, v})}
  end
end

def next_state(state, _res, {:call, _mod, _fun, _args}) do
  state
end
```

StatefulProperties/elixir/pbt/test/cache_test.exs

```
# Given that state prior to the call `{:call, mod, fun, args}`,
# determine whether the result (res) coming from the actual system
# makes sense according to the model
def postcondition(%State{entries: l}, {:call, _, :find, [key]}, res) do
  case List.keyfind(l, key, 0) do
    nil ->
      res == {:error, :not_found}

    {^key, val} ->
      res == {:ok, val}
  end
end

def postcondition(_state, {:call, _mod, _fun, _args}, _res) do
  true
end
```

Parallel Stateful Testing

StatefulProperties/elixir/pbt/test/cache_test.exs

```
property "parallel stateful property", numtests: 10000 do
  forall cmds <- parallel_commands(__MODULE__) do
    Cache.start_link(@cache_size)
    {history, state, result} = run_parallel_commands(__MODULE__, cmds)
    Cache.stop()

    (result == :ok)
    |> aggregate(command_names(cmds))
    |> when_fail(
      IO.puts("""
      =======
      Failing command sequence
      #{cmds}
      At state: #{state}
      =======
      Result: #{result}
      History: #{history}
      """)
    )
  end
end
```

Note the added :erlang.yield() calls:

```
def cache(key, val) do
  case :ets.match(:cache, {:'$1', {key, :'_'}}) do
    [[n]] ->
      :ets.insert(:cache, {n,{key,val}})
    [] ->
      :erlang.yield()
      case :ets.lookup(:cache, :count) do
        [{:count,max,max}] ->
          :ets.insert(:cache, [{1,{key,val}}, {:count,1,max}])
        [{:count,current,max}] ->
          :ets.insert(:cache, [{current+1, {key,val}},
                               {:count,current+1,max}])
      end
  end
end

def flush() do
  [{:count,_,max}] = :ets.lookup(:cache, :count)
  :ets.delete_all_objects(:cache)
  :erlang.yield()
  :ets.insert(:cache, {:count, 0, max})
end

defmodule Cache do
```

```
use GenServer

## Public API ##
def start_link(n) do
  GenServer.start_link(__MODULE__, n, name: __MODULE__)
end

def stop() do
  GenServer.stop(__MODULE__)
end

def find(key) do
  case :ets.match(:cache, {:_, {key, :"$1"}}) do
    [[val]] -> {:ok, val}
    [] -> {:error, :not_found}
  end
end

def cache(key, val) do
  GenServer.call(__MODULE__, {:cache, key, val})
end

def flush() do
  GenServer.call(__MODULE__, :flush)
end

## GenServer callbacks ##
def init(n) do
  :ets.new(:cache, [:public, :named_table])
  :ets.insert(:cache, {:count, 0, n})
  {:ok, :nostate}
end

def handle_call({:cache, key, val}, _from, state) do
  case :ets.match(:cache, {:"$1", {key, :_}}) do
    [[n]] ->
      :ets.insert(:cache, {n, {key, val}})

    [] ->
      :erlang.yield()

      case :ets.lookup(:cache, :count) do
        [{:count, max, max}] ->
          :ets.insert(:cache, [{1, {key, val}}, {:count, 1, max}])

        [{:count, current, max}] ->
          :ets.insert(:cache, [
            {current + 1, {key, val}},
            {:count, current + 1, max}
          ])
      end
  end

  {:reply, :ok, state}
end
```

```
  def handle_call(:flush, _from, state) do
    [{:count, _, max}] = :ets.lookup(:cache, :count)
    :ets.delete_all_objects(:cache)
    :erlang.yield()
    :ets.insert(:cache, {:count, 0, max})
    {:reply, :ok, state}
  end

  def handle_case(_cast, state), do: {:noreply, state}

  def handle_info(_msg, state), do: {:noreply, state}
end
```

Exercises

```
property "first example" do
    forall cmds <- commands(__MODULE__) do
        ActualSystem.start_link()
        {_history, _state, result} = run_commands(__MODULE__, cmds)
        ActualSystem.stop()
        result == :ok
    end
end

# The second example cannot be translated in Elixir since PropCheck
# Does not support the setup macro at the time of this writing

property "third example" do
    ActualSystem.start_link()
    forall cmds <- commands(__MODULE__) do
        {_history, _state, result} = run_commands(__MODULE__, cmds)
        result == :ok
    end
    ActualSystem.stop()
end
```

Solutions

Thinking In Properties

Question 2

ThinkingInProperties/elixir/pbt/test/solutions_test.exs

```
property "pair keysort approach" do
  # This function tests that any list of {key,val} pairs
  # end up being able to be sorted by the key by using List.keysort
  forall list <- list({term(), term()}) do
    # is_key_ordered checks that all tuples' keys are ordered.
    is_key_ordered(List.keysort(list, 0))
  end
end
```

```
def is_key_ordered([{a, _}, {b, _} = btuple | t]) do
  a <= b and is_key_ordered([btuple | t])
end

def is_key_ordered(_) do
  true
end

# This function instead works by using random tuples with various sizes,
# and picking a random key to test it.
# This tests broader usages of List.keysort/2, such as
# List.keysort([{:a,:b},{:e,:f,:g},{:t,:a,:n,:e}], 2) yielding the list
# [{:t,:a,:n,:e},{:a,:b},{:e,:f,:g}], where the comparison takes place
# on the second element of each tuple.
#
# While more complete than the previous one, this function does not
# accurately portray the need for stability in the function:
# [{:a,:b}, {:a,:a}] being sorted in the same order will not be tested
# here!
#
# Those can either be added in a regular test case, or would require
# devising a different property.
property "multi-sized tuple keysort approach" do
  forall list <- non_empty(list(non_empty(list()))) do
    # Since the default built-in types do not let us easily create
    # random-sized tuples that do not include {}, which wouldn't work
    # with List.keysort/2, we create variable-sized tuples ourselves
    tuples = for l <- list, do: List.to_tuple(l)
    # To know what position to use, we're going to use the smallest,
    # to avoid errors
    pos = Enum.min(for t <- tuples, do: tuple_size(t)) - 1
    sorted = List.keysort(tuples, pos)
    keys = extract_keys(sorted, pos)
    # The keys returned by keysort have to be in the same order as
    # returned by Enum.sort/1, which we now trust.
    keys == Enum.sort(keys)
  end
end

def extract_keys(list, pos) do
  for t <- list, do: :erlang.element(pos + 1, t)
end
```

Question 5

ThinkingInProperties/elixir/pbt/test/solutions_test.exs

```
def word_count(chars) do
  stripped = :string.trim(dedupe_spaces(chars), :both, ' ')
  spaces = Enum.sum(for char <- stripped, char == ?\s, do: 1)

  case stripped do
    '' -> 0
    _ -> spaces + 1
```

```
  end
end

defp dedupe_spaces([]), do: []
defp dedupe_spaces([?\s, ?\s | rest]), do: dedupe_spaces([?\s | rest])
defp dedupe_spaces([h | t]), do: [h | dedupe_spaces(t)]

property "word counting" do
  forall chars <- non_empty(char_list()) do
    word_count(chars) == alt_word_count(chars)
  end
end

defp alt_word_count(string), do: space(to_charlist(string))

defp space([]), do: 0
defp space([?\s | str]), do: space(str)
defp space(str), do: word(str)

defp word([]), do: 1
defp word([?\s | str]), do: 1 + space(str)
defp word([_ | str]), do: word(str)
```

Custom Generators

Question 4

CustomGenerators/elixir/pbt/test/solutions_test.exs

```
def tree(), do: tree(term())

def tree(type) do
  frequency([
    {3, {:node, type, nil, nil}},
    {2, {:node, type, lazy(tree(type)), nil}},
    {2, {:node, type, nil, lazy(tree(Type))}},
    {3, {:node, type, lazy(tree(type)), lazy(tree(type))}}
  ])
end
```

CustomGenerators/elixir/pbt/test/solutions_test.exs

```
def limited_tree(type) do
  sized(size, limited_tree(size, type))
end

def limited_tree(size, type) when size <= 1, do: {:node, type, nil, nil}

def limited_tree(size, type) do
  frequency([
    {1, {:node, type, lazy(limited_tree(size - 1, type)), nil}},
    {1, {:node, type, nil, lazy(limited_tree(size - 1, type))}},
    {5,
     {
       :node,
       type,
       # divide to avoid exponential growth
```

```
      lazy(limited_tree(div(size, 2), type)),
      lazy(limited_tree(div(size, 2), type))
    }}
  ])
end
```

Question 6

CustomGenerators/elixir/pbt/test/solutions_test.exs

```
def file(name) do
  sized(
    size,
    lines(
      size,
      {:"$call", __MODULE__, :file_open, [name, [:read, :write, :utf8]]}
    )
  )
end

def lines(size, fd) when size <= 1, do: fd

def lines(size, fd) do
  lines(
    size - 1,
    {:"$call", __MODULE__, :file_write, [fd, non_empty(utf8())]}
  )
end
```

In the Elixir version, the raw mode is replaced with the utf8 mode and the binary() generator with utf8(). The reason for that is that the write wrappers for this solution use the IO module, which forbids using the raw mode.

Shrinking

Question 3

Shrinking/elixir/pbt/test/solutions_test.exs

```
def meal() do
  let_shrink([
    appetizer <- [elements([:soup, :salad, :cheesesticks])],
    drink <- [elements([:coffee, :tea, :milk, :water, :juice])],
    entree <- [elements([:lasagna, :tofu, :steak])],
    dessert <- [elements([:cake, :chocolate, :icecream])]
  ]) do
    appetizer ++ drink ++ entree ++ dessert
  end
end
```

Question 4

Shrinking/elixir/pbt/test/solutions_test.exs

```
def item_price_special() do
```

```
  # first let: freeze the price list
  let price_list <- price_list() do
    # second let: freeze the list of specials
    let special_list <- special_list(price_list) do
      # third let: regular + special items and prices
      # help shrinking by first trying only one of the
      # two possible lists in case a specific form causes
      # the problems on its own
      # and we merge
      let {items, price} <-
            (let_shrink([
               {regular_items, regular_expected} <-
                 regular_gen(price_list, special_list),
               {special_items, special_expected} <-
                 special_gen(price_list, special_list)
             ]) do
               {regular_items ++ special_items,
                regular_expected + special_expected}
             end) do
        {items, price, price_list, special_list}
      end
    end
  end
end
```

APPENDIX 3

Generators Reference

Generator	Data Generated	Sample
any()	Any Erlang term PropEr can produce	any of the samples below
atom()	Erlang atoms	'ós43Úrcá\200'
binary()	Binary data aligned on bytes	<<2,203,162,42,84,141>>
binary(Len)	Binary data of fixed length Len, in bytes	<<98,126,144,175,175>>
bitstring()	Binary data without byte alignment	<<10,74,2:3>>
bitstring(Len)	Binary data of fixed length Len, in bits	<<11:5>>
boolean()	Atoms true or false. Also bool()	true, false
char()	Character codepoint, between 0 and 1114111 inclusively	23
choose(Min, Max)	Any integer between Min and Max inclusively	choose(1, 1000) => 596
fixed_list([Type])	A list where all entries match a generator from the list of arguments	fixed_list([boolean(), byte(), atom()]) => [true,2,b]
float()	Floating point number. Also real()	4.982972307245969
float(Min, Max)	Floating point number between Min and Max inclusively	float(0.0, 10.0) => 5.934602482212932
frequency([{N, Type}])	The value matching the generator of one of those in the second tuple element, with a probability similar to the value N. Also weighed_union/1	frequency([{1,atom()}, {10,char()}, {3,binary()}]) => 23 (chances of getting an atom are $^{1}/_{14}$)

Generator	Data Generated	Sample
function([Arg], Ret])	An anonymous function	function([char(), bool()], atom()) => #Fun<proper_gen.25.96459342>
integer()	An integer	89234
list()	A list of terms, equivalent to list(any())	any of the samples, as list elements
list(Type)	A list of terms of type Type	list(boolean()) => [true, true, false]
loose_tuple(Type)	A tuple with terms of type Type	loose_tuple(boolean()) => {true, true, false}
map(KeyType, ValType)	A map with keys of type KeyType and values of type ValType	map(integer(), boolean()) => #{0 => true, -4 => true, 18 => false}
neg_integer()	An integer smaller than 0	-1231
non_empty(Gen)	Constrains a binary or a list generator into not being empty	non_empty(list()) => [abc]
non_neg_float()	A float greater than or equal to 0.0	98.213012312
non_neg_integer()	An integer greater than or equal to 0	98
number()	A float or integer	123
oneof(Types)	The value created by the generator of one of those in Types, also union(Types) and elements(Types). In QuickCheck, oneof() shrinks towards a failing element whereas elements() shrinks towards the first element of the list. PropEr does not make that distinction.	oneof([atom(), char()]) => a
pos_integer()	Integer greater than 0	1
range(Min, Max)	Any integer between Min and Max inclusively	range(1, 1000) => 596
string()	Equivalent to list(char())	"^DQ^W^R/D" (may generate weird escape sequences!)
term()	Same as any()	any of the samples in this table

Generator	Data Generated	Sample
timeout()	Union of non_neg_integer() generator and the atom infinity	312
tuple()	A tuple of random terms	tuple() => {true, 13.321123, -67}
{T1, T2, ...}	A literal tuple with types in it	{boolean(), char()} => {true, 1}
utf8()	Generates to utf8-encoded text as a binary data structure	<<"□□ᵽ_3ʳ"/utf8>> (may generate weird escape sequences!)
vector(Len, Type)	A list of length Len of type Type	vector(5, integer()) => [17,0,1,3,8]

Thank you!

How did you enjoy this book? Please let us know. Take a moment and email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line "Book Feedback."

Ready for your next great Pragmatic Bookshelf book? Come on over to https://pragprog.com and use the coupon code BUYANOTHER2018 to save 30% on your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn't apply to *The Pragmatic Programmer* ebook because it's older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

And thank you for your continued support,

Andy Hunt, Publisher

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

https://pragprog.com/book/fhproper

Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up to Date

https://pragprog.com

Join our announcement mailing list (low volume) or follow us on twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy

https://pragprog.com/news

Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/fhproper*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764