

Introducing HDD

@elixirlang / elixir-lang.org

Elixir v1.6

- **January/2018**
- **>702 contributors**
- **>5800 packages on hex.pm**
- **>175 000 000 downloads**

Code Formatter

```
defmodule DemoWeb.PostController do
  use DemoWeb, :controller

  alias Demo.Blog
  alias Demo.Blog.Post

  def create(conn, %{"post" => post_params}) do
    case Blog.create_post(post_params) do

      {:ok, post} ->
        conn
        |> put_flash(:info, "Post created successfully.")
        |> redirect(to: post_path(conn, :show, post))

      {:error, %Ecto.Changeset{}=changeset} ->
        render(conn, "new.html", changeset: changeset)
    end
  end
end
```

```
defmodule DemoWeb.PostController do
  use DemoWeb, :controller

  alias Demo.Blog
  alias Demo.Blog.Post

  def create(conn, %{"post" => post_params}) do
    case Blog.create_post(post_params) do
      {:ok, post} ->
        conn
        |> put_flash(:info, "Post created successfully.")
        |> redirect(to: post_path(conn, :show, post))

      {:error, %Ecto.Changeset{} = changeset} ->
        render(conn, "new.html", changeset: changeset)
    end
  end
end
```

Elixir v1.7

- July/2018
- ??? contributors
- ??? packages on hex.pm
- ??? downloads

StreamData

Data generation

```
iex> import Stream.Data
```

```
iex> Enum.take integer(), 10  
[1, -2, -1, 2, 0, 6, 1, 8, -6, 3]
```

```
iex> Enum.take string(:ascii), 10  
["C", "", "", "1I", "dkrov", "dR4",  
"T4=h4", "u", "_gjE", "K^u5eJ&7"]
```


Property-based testing

```
check all left <- string(),  
           right <- string(),  
           string = left <> right do  
  
    assert String.contains?(string, left)  
    assert String.contains?(string, right)  
  
end
```



**John
Hughes**

Code Formatter

Code Formatter

- Formats your code using a consistent style
- Helps you focus on what matters
- Works as a guide for newcomers
- Unifies code written by teams and the community

Remove guidelines that are automatically handled by a code formatter #46

Edit

[Open](#) josevalim wants to merge 1 commit into `lexmag:master` from `josevalim:patch-1`

💬 Conversation 9

🔗 Commits 1

📄 Files changed 1

+0 -443

josevalim commented 3 days ago

This PR is not intended to be merged (not yet anyway). It exists to highlight all concerns developers no longer need to worry about by adopting Elixir's upcoming code formatter.

🔗 Update README.md

43e4330


👁 josevalim reviewed 3 days ago

[View changes](#)

README.md

```
115 - # Good
116 - <<102::unsigned-big-integer, rest::binary>>
117 - ...
118 -
119 34 <<a name="leading-space-comment"></a>
120 35 Use one space between the leading `#` character of the comment and the text
of the comment.
```

Reviewers

 whatyouhide uohzxela

Assignees

No one assigned

Labels

None yet

Projects

None yet

Milestone

No milestone

Notifications

Code Formatter principles

1. It does not change the code semantics by default
2. Minimize configuration
3. No special cases

mix format

[a: [1, 2, 3], b: :ok]

[a: [1, 2, 3], b: :ok]

20

[a: [1, 2, 3], b: :ok]

```
[  
  a: [1, 2, 3],  
  b: :ok  
]
```

10

[

a: [1, 2, 3],

b: :ok

]

10

```
[  
  a: [  
    1,  
    2,  
    3  
  ],  
  b: :ok  
]
```

Document Algebra

The Design of a Pretty-printing Library

John Hughes

Chalmers Tekniska Högskola, Göteborg, Sweden.

1 Introduction

On what does the power of functional programming depend? Why are functional programs so often a fraction of the size of equivalent programs in other languages? Why are they so easy to write? I claim: because functional languages support software reuse extremely well.

Programs are constructed by putting program components together. When we discuss reuse, we should ask

- What kind of components can be given a name and reused, rather than reconstructed at each use?
- How flexibly can each component be used?

Every programming language worthy of the name allows sections of a program with identical control flow to be shared, by defining and reusing a procedure. But ‘programming idioms’ — for example looping over an array — often cannot be defined as procedures because the repeated part (the loop construct) contains a varying part (the loop body) which is different at every instance. In a functional language there is no problem: we can define a *higher order function*, in which the varying part is

Documents

- "text"
- `nest(doc, columns)`
- `line(doc, doc)`
- `concat(doc, doc)`
- `empty()`

[1,2,3]

"["

| > concat("1,")

| > concat("2,")

| > concat("3")

| > concat("]")

```
[1,  
2,  
3]
```

```
"["
```

```
|> line("1,")
```

```
|> line("2,")
```

```
|> line("3")
```

```
|> nest(2)
```

```
|> line("]")
```

**How to choose
between layouts?**

[1,2,3]

"["

| > concat("1,")

| > concat("2,")

| > concat("3,")

| > nest(2)

| > concat("]")

```
[1,  
2,  
3]
```

```
"["
```

```
| v line("1,")
```

```
| v line("2,")
```

```
| v line("3")
```

```
| v nest(2)
```

```
| v line("]")
```

A prettier printer

Philip Wadler

Joyce Kilmer and most computer scientists agree: there is no poem as lovely as a tree. In our love affair with the tree it is parsed, pattern matched, pruned — and printed. A pretty printer is a tool, often a library of routines, that aids in converting a tree into text. The text should occupy a minimal number of lines while retaining indentation that reflects the underlying tree. A good pretty printer must strike a balance between ease of use, flexibility of format, and optimality of output.

Flexible layout

```
group(  
  "[ "  
  | > line("1, ")  
  | > line("2, ")  
  | > line("3")  
  | > nest(2)  
  | > line("] ")  
)
```

Flexible layout

```
def group(doc) do  
  choose(  
    replace_line_by_concat(doc),  
    doc  
  )  
end
```


[1,2,3]

"["


```
| > concat("1,")  
| > concat("2,")  
| > concat("3")  
| > nest(2)  
| > concat("]")
```

[
1,
2,
3
]


"["

```
| > line("1,")  
| > line("2,")  
| > line("3")  
| > nest(2)  
| > line("]")
```

Pretty printing implemented* #1047

 Closed

[manpages](#) wants to merge 11 commits into `elixir-lang:master` from `manpages:pretty`

 Conversation 58

 Commits 11

 Files changed 9



manpages commented on 11 May 2013

Contributor



Closes [#965](#).

wadler.ex implements Hughes-Wadler document algebra;
binary/inspect.ex now pretty-prints.

Commentaries about implementation are available upon the request.


Rebased properly, thanks @alco.

Fixed issue existed on case insensitive platforms, thanks @yrashk.




manpages added some commits on 6 May 2013



 Add pretty printing library to Elixir stdlib


64fb24d



 Tests for wadler; Guards for primitive types.


e035ca0



 Put pretty printing library into core

2fd4aa4



 Binary.Inspect now uses pretty printing library

a492472



alco commented on 14 May 2013

Member



Fancy some stress testing? Here's something completely nuts. <https://gist.github.com/alco/5579369>

This is documentation for Elixir in one Erlang list. You can load it into IEx like this:

```
{:ok, terms} = :file.consult 'docs.erl'  
IO.inspect terms
```

It takes a couple of seconds for the current `IO.inspect` to print. With your patch, I wasn't able to see it finish. It eats 100% CPU and gradually eats up memory, then release a piece of memory, then it's it up again -- in a cycle. So it seems to me that it gets into an infinite loop somewhere during the process.

If it's not possible to pretty-print this much data, I would expect to at least be presented with something like initial 100 characters of output and then a message "too much data to format. Use `inspect: raw`".

Lazy vs Eager

```
def group(doc) do  
  choose(  
    replace_line_by_concat(doc),  
    doc  
  )  
end
```

[a: [1, 2, 3], b: :ok]

10

```
[  
  a: [  
    1,  
    2,  
    3  
  ],  
  b: :ok  
]
```

[a: [1, 2, 3], b: :ok]

Lazy vs Eager

```
def group(doc) do  
  choose(  
    replace_line_by_concat(doc),  
    doc  
  )  
end
```


Strictly Pretty

Christian Lindig
Gärtner Datensysteme GbR
Hamburger Str. 273a
D-38 114 Braunschweig, Germany
`lindig@gaertner.de`

March 6, 2000


Abstract

Pretty printers are tools for formatting structured text. A recently taken algebraic approach has lead to a systematic design of pretty printers. Wadler has proposed such an algebraic pretty printer together with an implementation for the lazy functional language Haskell. The original design causes exponential complexity when literally used in a strict language. This note recalls some properties of Wadler's pretty printer on an operational level and presents an efficient implementation for the strict functional language Objective Caml.

Pretty printing using Wadler/Lindig algorithm #1267

 Closed

brunoro wants to merge 23 commits into `elixir-lang:master` from `unknown repository`

 Conversation 26

 Commits 23

 Files changed 18



brunoro commented on 16 Jun 2013





















Contributor



A follow-up from the discussion on [@manpages pull request](#).

`wadler.ex` contains a strict implementation of the Wadler document algebra as described by [Lindig \(2000\)](#).

 **manpages** and others added some commits on 6 May 2013

-   Add pretty printing library to Elixir stdlib 1f7a806
-   Tests for wadler; Guards for primitive types. 3e2af17
-   Merged (Fri May 10 06:22:13 EEST 2013) 7f28627
-   Put pretty printing library into core 1da4844
-   Binary.Inspect now uses pretty printing library  0060190
-   using Lindig's strict pretty printer  a41528d
-   adapted IO.inspect calls to Lindig interface  54097e1
-   Optimizing string concatenation calls on wadler.ex  f7c8ed0

Document Algebra

- Implemented by `Inspect.Algebra`
- Used to Inspect data structures
- Used by the Code Formatter

```
code([1, 2, 3], :ok)
```

20

```
code([1, 2, 3], :ok)
```

```
code(  
  [1, 2, 3],  
  :ok  
)
```

10

```
code(  
  [1, 2, 3],  
  :ok  
)
```

10

```
code(  
  [  
    1,  
    2,  
    3  
  ],  
  :ok  
)
```


Extensions

- `color(doc, color)`
- `nest(doc, :cursor)`
- `force_unfit(doc)`
- `etc`

force_unfit(doc)

```
code( :ok, """  
foo  
""", some_arg)
```

force_unfit(doc)

```
code(  
  :ok,  
  |||||  
  foo  
  |||||,  
  some_arg  
)
```

StreamData

Example-based testing

```
assert String.contains?("foobar", "foo")
assert String.contains?("foobar", "bar")
assert String.contains?("foobar", "ob")
refute String.contains?("foobar", "oops")
```

Example-based testing

- How to find corner cases?
- And how to reason about them?

Example-based testing

```
String.contains?("foobar", "")  
String.contains?("", "foobar")  
String.contains?("", "")
```

Property-based testing

```
check all left <- string(),  
          right <- string(),  
          string = left <> right do  
  
  assert String.contains?(string, left)  
  assert String.contains?(string, right)  
  
end
```


Property-based testing

- Described as generative testing
- Useful to describe the invariants in the system
- Leads to thoroughly tested software that is designed with intent

QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

ABSTRACT

QuickCheck is a tool which aids the Haskell programmer in formulating and testing properties of programs. Properties are described as Haskell functions, and can be automatically tested on random input, but it is also possible to define custom test data generators. We present a number of case studies, in which the tool was successfully used, and also point out some pitfalls to avoid. Random testing is especially suitable for functional programs because properties can be stated at a fine grain. When a function is built from separately tested components, then random testing suffices to obtain good coverage of the definition under test.

1. INTRODUCTION

Testing is by far the most commonly used approach to ensuring software quality. It is also very labour intensive, accounting for up to 50% of the cost of software development. Despite anecdotal evidence that functional programs require somewhat less testing ('Once it type-checks, it usually works'), in practice it is still a major part of functional program development.

(monad are hard to test), and so testing can be done at a fine grain.


A testing tool must be able to determine whether a test is passed or failed; the human tester must supply an automatically checkable criterion of doing so. We have chosen to use formal specifications for this purpose. We have designed a simple domain-specific language of *testable specifications* which the tester uses to define expected properties of the functions under test. QuickCheck then checks that the properties hold in a large number of cases. The specification language is embedded in Haskell using the class system. Properties are normally written in the same module as the functions they test, where they serve also as checkable documentation of the behaviour of the code.

A testing tool must also be able to generate test cases automatically. We have chosen the simplest method, random testing [11], which competes surprisingly favourably with systematic methods in practice. However, it is meaningless to talk about random testing without discussing the distribution of test data. Random testing is most effective when the distribution of test data follows that of actual data, but when testing reusable code units as opposed to whole sys-

QuickCheck for Clojure

555 commits
38 branches
41 releases
28 contributors

Branch: master
New pull request
Create new file
Upload files
Find file
Clone or download


gfredericks committed 4 days ago Unify naming between ret & reporter-fn in quick-check ...
 Latest commit ccf670b 4 days ago

doc	doc typo	10 months ago
script	TCHECK-137: Self-host tests not runnable	2 months ago
src	Unify naming between ret & reporter-fn in quick-check	4 days ago
test-runners	Move test runners out of `resources` so they aren't packaged with rel...	3 months ago
.gitignore	Don't ignore the doc directory	2 years ago
CHANGELOG.markdown	Annotate simple-check versions in changelog	3 months ago
CONTRIBUTING.md	More detailed contribution instructions	4 years ago
Makefile	Update Makefile with test.check name change	4 years ago
README.md	TCHECK-136: dd self-host ClojureScript test instructions	2 months ago
pom.xml	Update cljs version in pom.xml to match project.clj	5 months ago
project.clj	Unify naming between ret & reporter-fn in quick-check	4 days ago

[README.md](#)

Generators...

```
iex> import Stream.Data
```

```
iex> Enum.take integer(), 10  
[1, -2, -1, 2, 0, 6, 1, 8, -6, 3]
```

```
iex> Enum.take string(:ascii), 10  
["C", "", "", "1I", "dkrov", "dR4",  
"T4=h4", "u", "_gjE", "K^u5eJ&7"]
```

...are lazy

```
iex> integer()  
#StreamData<45.79517404/2 in  
StreamData.integer/0>
```

...are infinite

```
iex> integer() |> Enum.to_list()  
...takes forever...
```

...are random

```
iex> integer() |> Enum.take(10)  
[1, -2, -1, 2, 0, 6, 1, 8, -6, 3]
```

```
iex> integer() |> Enum.take(10)  
[-1, 1, -1, 3, 5, 5, -1, -8, 3, 9]
```

...grow in size

```
iex> integer()  
...> |> Stream.drop(100)  
...> |> Enum.take(10)  
[-96, -11, 100, -65, -41, -71, -81,  
88, -54, -9]
```


Property-based testing

```
check all left <- string(),  
           right <- string(),  
           string = left <> right do  
  
    assert String.contains?(string, left)  
    assert String.contains?(string, right)  
  
end
```

Generators are shrinkable

```
property "element not in list" do
  check all list <- list_of(integer()) do
    assert 22 not in list
  end
end
```

Generators are shrinkable

1) property element not in list (Test)
examples/examples_test.exs:15
Failed with generated values
(after 29 successful run(s)):

```
Clause: list <- list_of(integer())  
Generated: [22]
```

```
Expected truthy, got false  
code: assert 22 not in list
```

Generators are functions

```
fn seed, size ->  
  {current_element, shrinking_recipe}  
end
```

StreamData

- Provides data generation primitives
- Brings stateless property-based testing to ExUnit
- QuickCheck provides more advanced features such as model checking

Tired of writing and maintaining thousands of automated tests? Did you know that repeating tests finds only 15% of your bugs anyway? Let QuickCheck generate *new* tests for you daily, saving you effort and nailing your bugs earlier!

QuickCheck takes you quickly from specification to identified bug.

Three steps to QuickCheck

- Write a QuickCheck specification instead of test cases— general properties your system should always satisfy.
- QuickCheck uses controlled random generation to test your code against the spec.
- Failing cases are automatically simplified before they are presented to you.

Concise Specifications

Your QuickCheck specifications consist of properties and generators which describe system behaviour in a specified set of cases. Each property generates many different test cases—so specifications can be much more concise and maintainable than test suites. At the same time, many cases can be generated, so testing is more thorough. QuickCheck uses the power of functional programming to keep specifications concise and readable.

Controlled Randomness

QuickCheck tests your properties in randomly generated cases, either interactively or invoked from your test server. Pure random generation makes for poor test data, but QuickCheck's simple and powerful interface puts you in control, making it easy to generate complex data with

Hughes Driven Development



plataformatec
consulting and software engineering



elixir

@elixirlang / elixir-lang.org