



elixir

@elixirlang / elixir-lang.org

**It is not about
the syntax!**

Data
(types)

Modules

Processes

Tooling

No custom data types

- **Records aimed to add tagged tuples
but the implementation backfired**
- **Maps are an improvement (imo)
but do not officialise "tagging"**

How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

October 1988

Abstract

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the “eqtype variables” of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them for-

integers and a list of floating point numbers.

One widely accepted approach to parametric polymorphism is the Hindley/Milner type system [Hin69, Mil78, DM82], which is used in Standard ML [HMM86, Mil87], Miranda¹[Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

Collections

```
widgets.filter(b -> b.getColor() == RED)  
    .mapToInt(b -> b.getWeight())  
    .sum()
```

Laziness in collections

```
widgets.stream()  
    .filter(b -> b.getColor() == RED)  
    .mapToInt(b -> b.getWeight())  
    .sum()
```


Elixir

Goals



- Extensibility
- Productivity
- Compatibility

Extensibility

Data type polymorphism

A diagram consisting of three overlapping circles. The top-left circle is olive green and contains the text 'Data (types)'. The top-right circle is a darker olive green and contains the text 'Modules'. The bottom circle is a teal-blue color and contains the text 'Processes'. The circles overlap in the center, creating a triangular shape.

Data
(types)

Modules

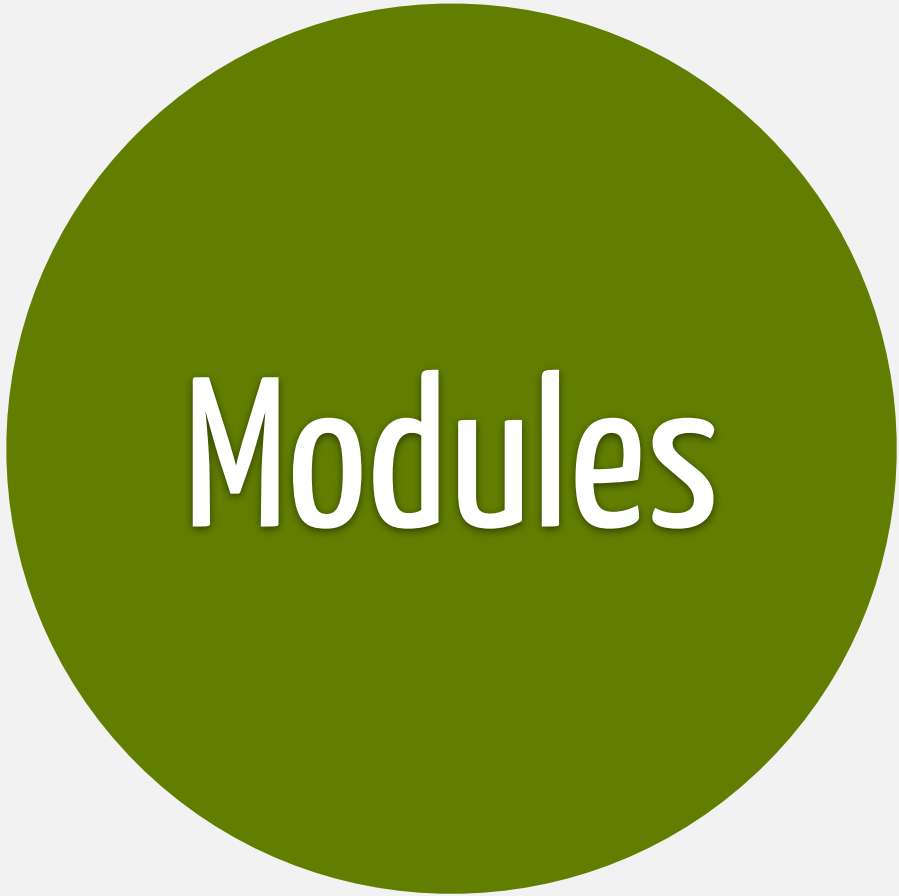
Processes



Processes

Pid ! Message

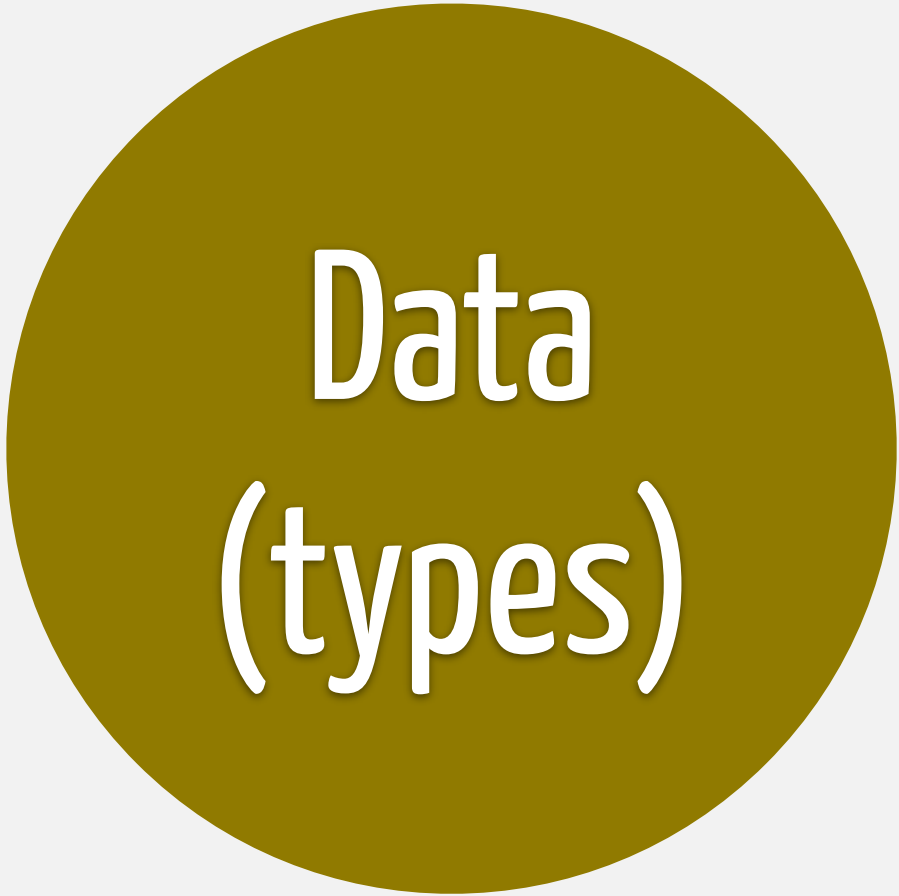
“Any process that handles this message”



Modules

`Module: function()`

“Any module that exports this function”



Data
(types)

??????

“Any data type that does ????”


```
-module(json).
```

```
encode(Item) when is_list(Item) ->  
    % ...
```

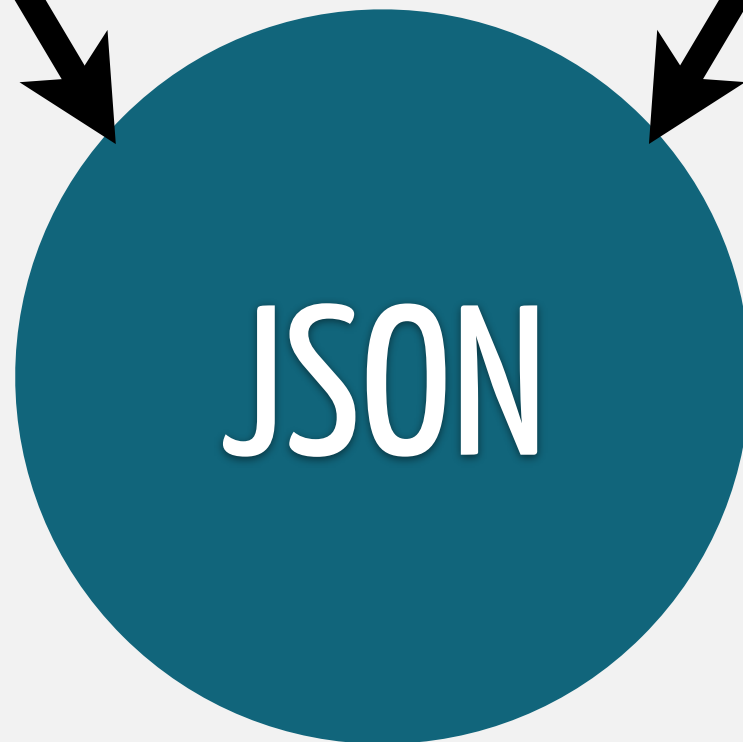
```
encode(Item) when is_binary(Item) ->  
    % ...
```

```
encode(Item) when is_number(Item) ->  
    % ...
```




fork

fork



```
-module(json).
```

```
encode(Item) when is_list(Item) ->  
    % ...
```

```
encode(Item) when is_binary(Item) ->  
    % ...
```

```
encode(Item) when is_number(Item) ->  
    % ...
```

The data type is the one
that knows how to
convert itself to JSON

```
defprotocol JSON do  
  def encode(item)  
end
```

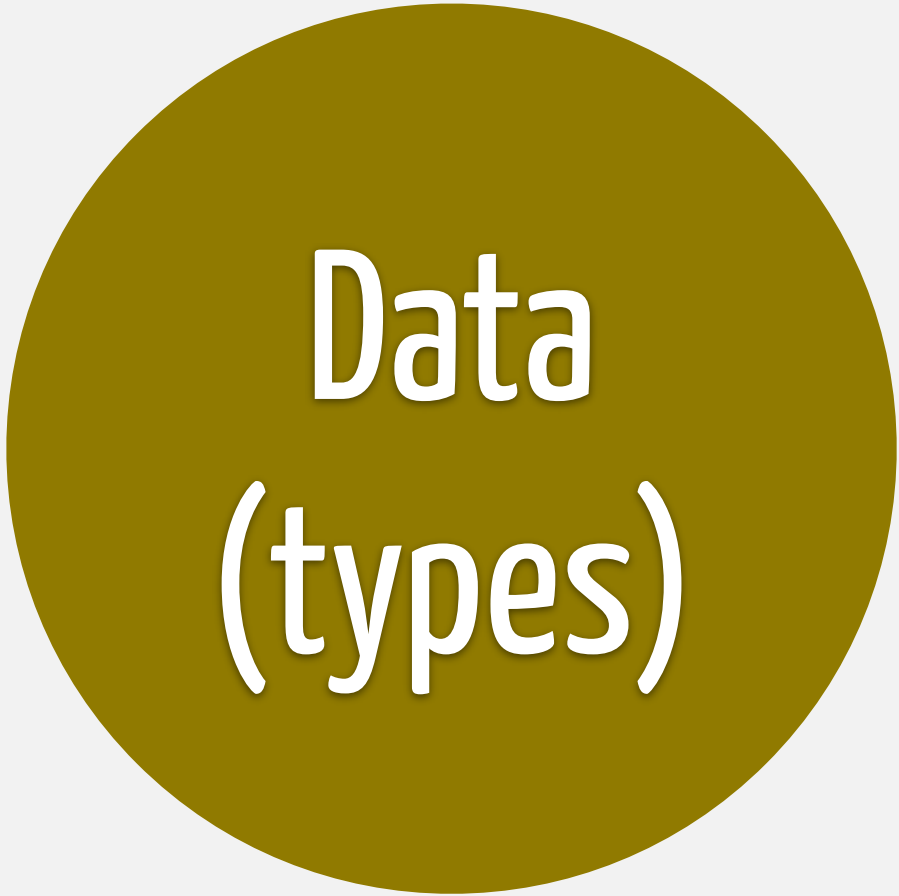
```
JSON.encode(item)
```

```
defimpl JSON, for: List do
  def encode(item) # ...
end
```

```
defimpl JSON, for: BitString do
  def encode(item) # ...
end
```

```
defimpl JSON, for: Number do
  def encode(item) # ...
end
```

I can write a JSON library that
is **extensible** to any data type



Data
(types)

`JSON.encode(data)`

“Any data type that implements a protocol”

Enumerable Protocol

```
Enum.map [1,2,3], fn(x) ->  
  x * 2  
end  
#=> [2,4,6]
```


Enumerable Protocol

```
Enum.map 1..5, fn(x) ->  
  x * 2  
end  
#=> [2,4,6,8,10]
```

Enumerable Protocol

- Based on Haskell Iteratees
- Works with in-memory collections and resources (like I/O, File, etc)

Enumerable Protocol

```
# Uses raw & read_ahead  
file = File.stream(path)  
Enum.take(file, 5)
```

Inspect Protocol

```
1> dict:from_list([{a,1}]).  
{dict,1,16,16,8,80,48,  
  {[],[],[],[],[],[],[],[],[],[],  
   [],[],[],[],[],[]},  
  {{{[],  
    [[a|1]],  
    [],[],[],[],[],[],[],[],[],  
    [],[],[],[]}}}}
```

Inspect Protocol

```
iex> HashDict.new(a: 1)  
#HashDict<[a: 1]>
```

Productivity

Mix + Hex + Docs


```
$ mix new foo
```

```
$ cd foo
```

```
$ mix test
```

```
$ mix hex.publish
```

```
$ mix hex.docs
```


Compatibility

OTP & Elixir

- GenServer (plus Task and Agents)
- GenRouter

GenServer

Task

GenServer

Agent



Only
Computation

Only
State

Task

- Task.start_link/3 is similar to
proc_lib:spawn_link/3

async / await

```
Top = self(),  
Ref = make_ref(),  
  
Pid = spawn_link(fun ->  
    Top ! {Ref, ...}  
end),  
  
receive  
    {Ref, Value} -> Value  
end
```

async / await

```
task = Task.async(&calculate_x/0)  
# Do something else  
Task.await(task)
```

Distributed Tasks

In the remote node

```
Task.Supervisor.start_link(name: :tasks_sup)
```

In the client

```
Task.Supervisor.async(  
  {:tasks_sup, :remote@local},  
  &calculate_x/0)
```


Agent

```
agent = Agent.start_link(&initial_state/0)  
Agent.update(agent, &increment/1)  
Agent.get(agent, &identity/1)
```


LVars: Lattice-based Data Structures for Deterministic Parallelism

Lindsey Kuper Ryan R. Newton

Indiana University

{lkuper, rrnewton}@cs.indiana.edu

Abstract

Programs written using a *deterministic-by-construction* model of parallel computation are guaranteed to always produce the same observable results, offering programmers freedom from subtle, hard-to-reproduce nondeterministic bugs that are the scourge of parallel software. We present *LVars*, a new model for deterministic-by-construction parallel programming that generalizes existing single-assignment models to allow multiple assignments that are

binators can provide real speedups on practical programs while guaranteeing determinism [22].¹ Yet pure programming with futures is not ideal for all problems. Consider a *producer/consumer* computation in which producers and consumers can be scheduled onto separate processors, each able to keep their working sets in cache. Such a scenario enables *pipeline parallelism* and is common, for instance, in stream processing. But a clear separation of producers and consumers is difficult with futures, because when-

Agent.Lattices

- A set of agents/operations guaranteed to be deterministic for parallelism?



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

CRDTs: Consistency without concurrency control

Mihai Leția — Nuno Preguiça — Marc Shapiro

Agent.CRDT

- A set of agents/operations guaranteed to replicatable across nodes?

Parallelism

Parallelism

- Laziness
- Pipeline parallelism
- Data parallelism

Collections

widgets

```
|> Enum.filter(fn b -> b[:color] == RED end)
```

```
|> Enum.map(fn b -> b[:weight] end)
```

```
|> Enum.take(5)
```


Streams / Laziness

widgets

```
|> Stream.filter(fn b -> b[:color] == RED end)
```

```
|> Stream.map(fn b -> b[:weight] end)
```

```
|> Stream.take(5)
```

```
|> Enum.to_list()
```


Pipeline Parallelism

widgets

```
|> Stream.filter(fn b -> b[:color] == RED end)
|> Stream.map(fn b -> b[:weight] end)
|> Stream.take(5)
|> Stream.async()
|> Enum.to_list()
```



Pipeline Parallelism

data

|> ...

|> Stream.async()

|> ...

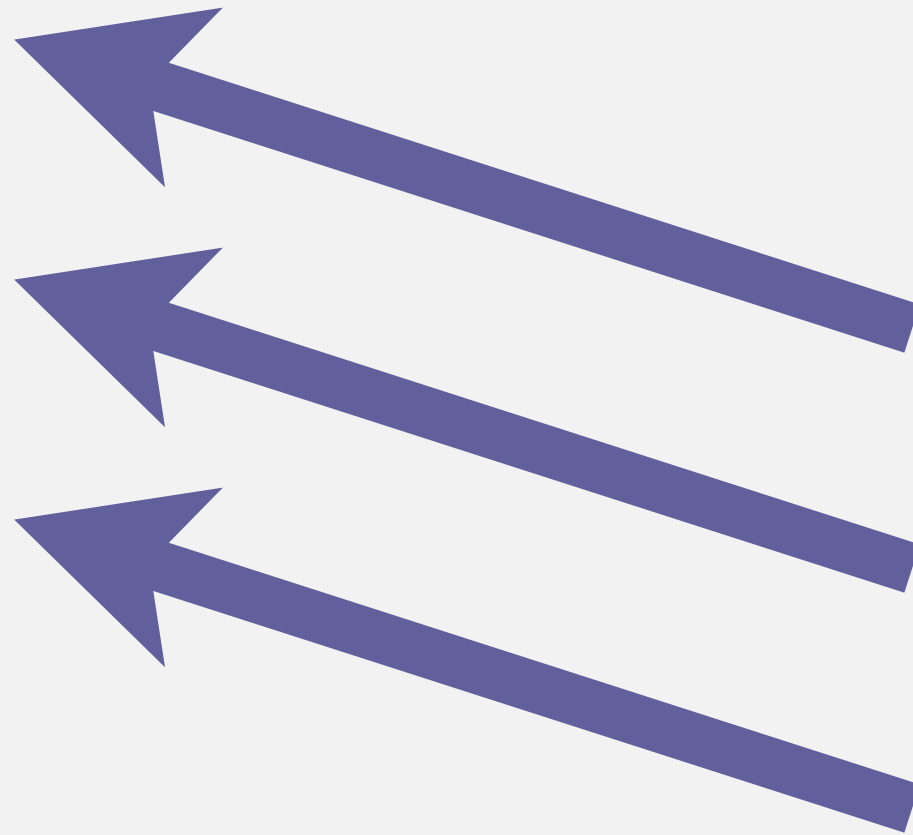
|> Stream.async()

|> ...

|> Stream.async()

|> ...

|> Enum.to_list()



Pipeline Parallelism



Pipeline Parallelism



GenRouter

- Specifies the message format for connecting different processes
- GenEvent replacement where we have one process per handler
- While allowing custom dispatch rules

GenRouter

```
@spec init(args)  
@spec handle_up(pid, ref, state)  
@spec handle_down(reason, pid, ref, state)  
@spec handle_dispatch(event, state)  
@spec handle_info(message, state)  
@spec code_change(old_vsn, state, extra)  
@spec terminate(reason, state)
```

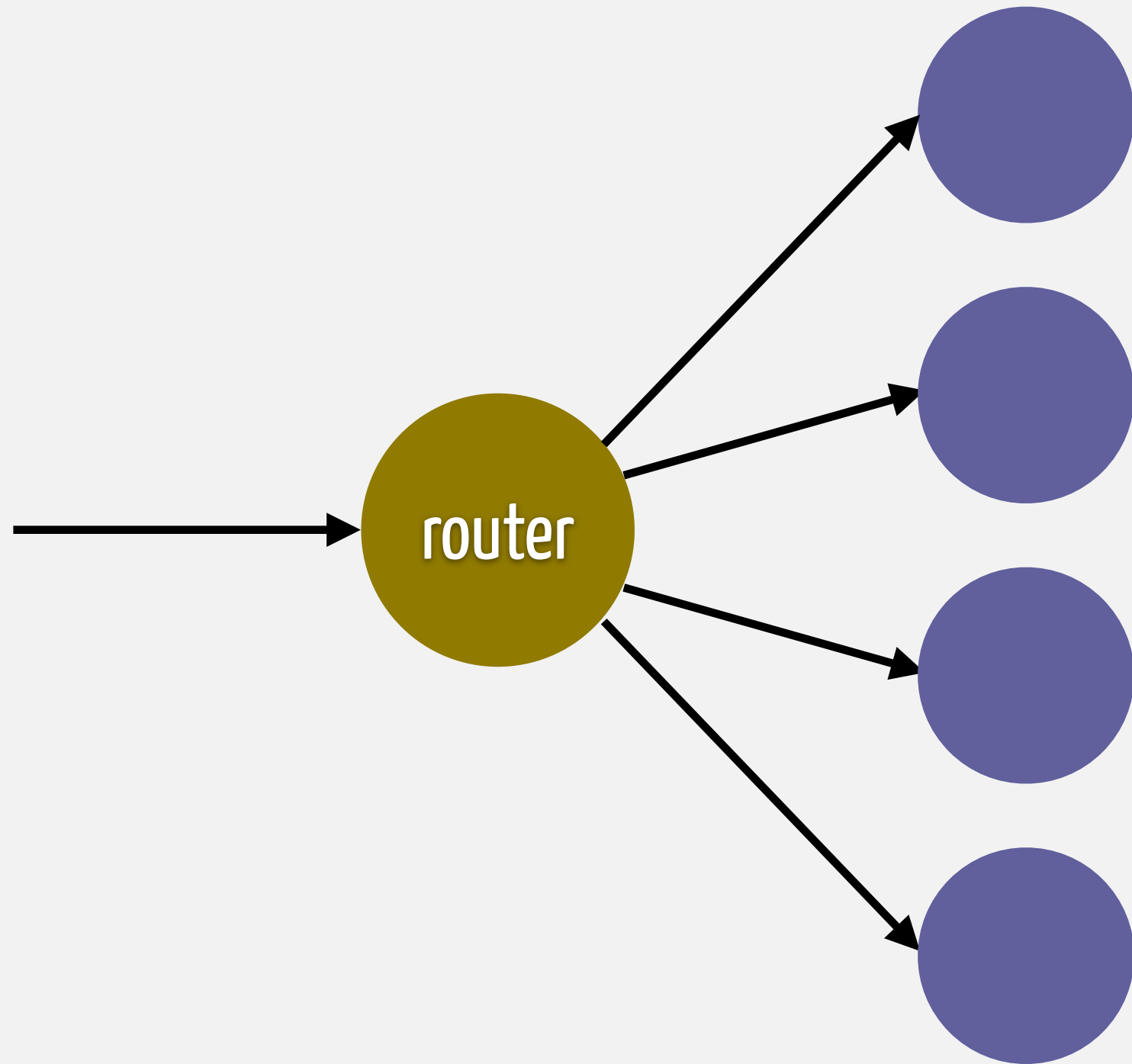
GenRouter

```
@spec init(args)  
@spec handle_up(pid, ref, state)  
@spec handle_down(reason, pid, ref, state)  
@spec handle_dispatch(event, state)  
@spec handle_info(message, state)  
@spec code_change(old_vsn, state, extra)  
@spec terminate(reason, state)
```

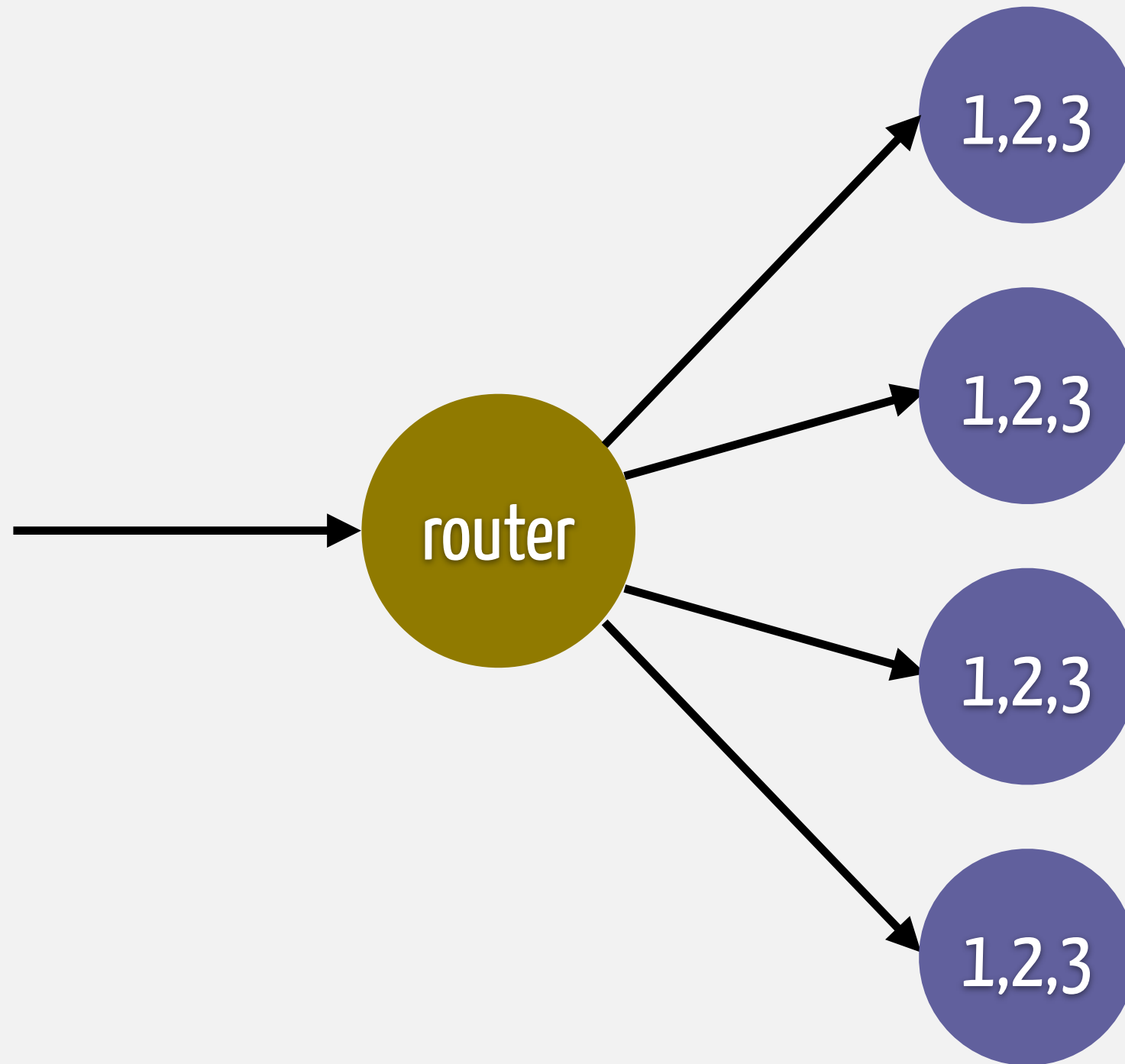

GenRouter

```
@spec handle_dispatch(event, state) ::  
  {:ok, [pid], state} |  
  {:error, reason, state} |  
  {:stop, reason, state}
```

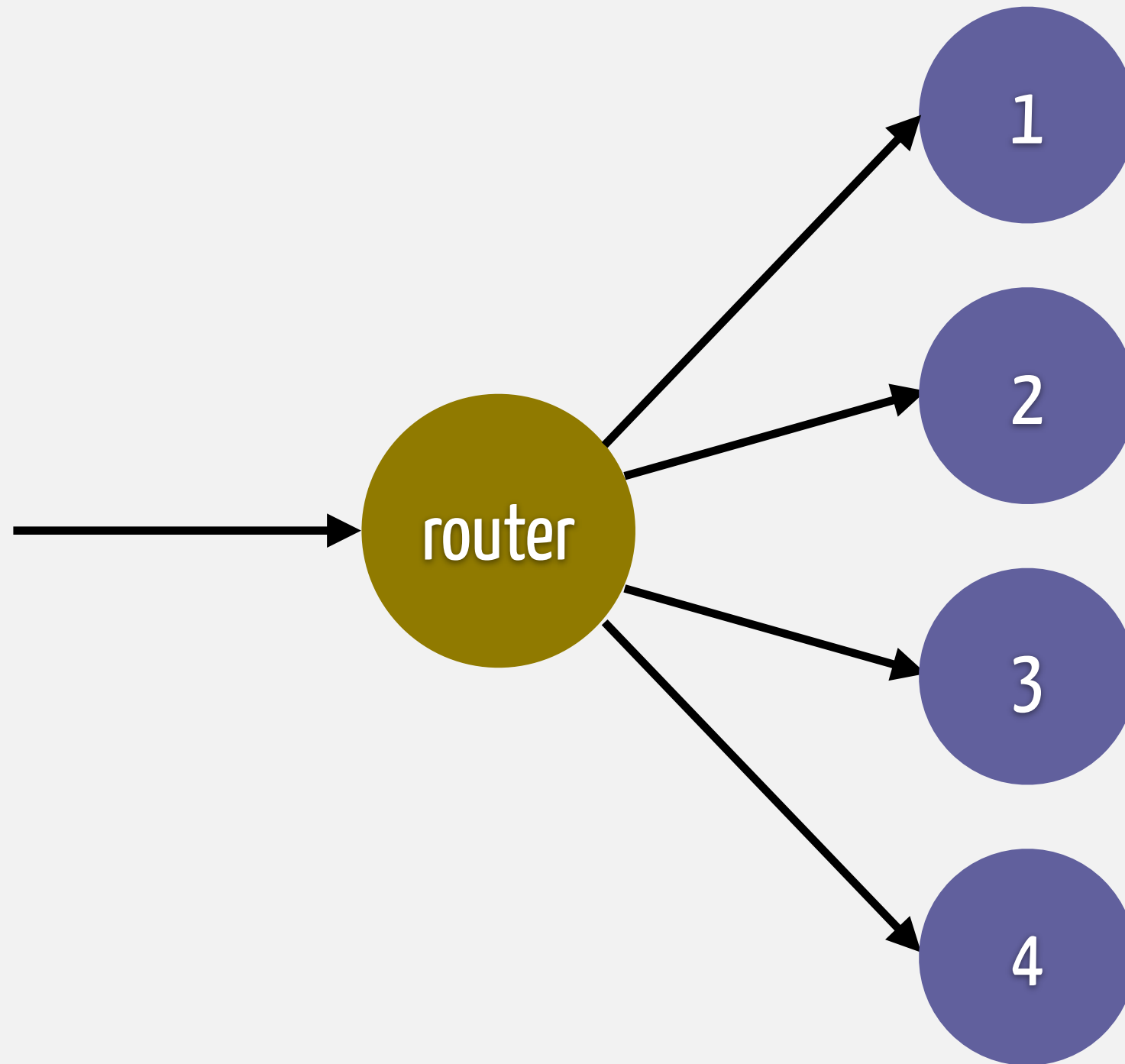

GenRouter



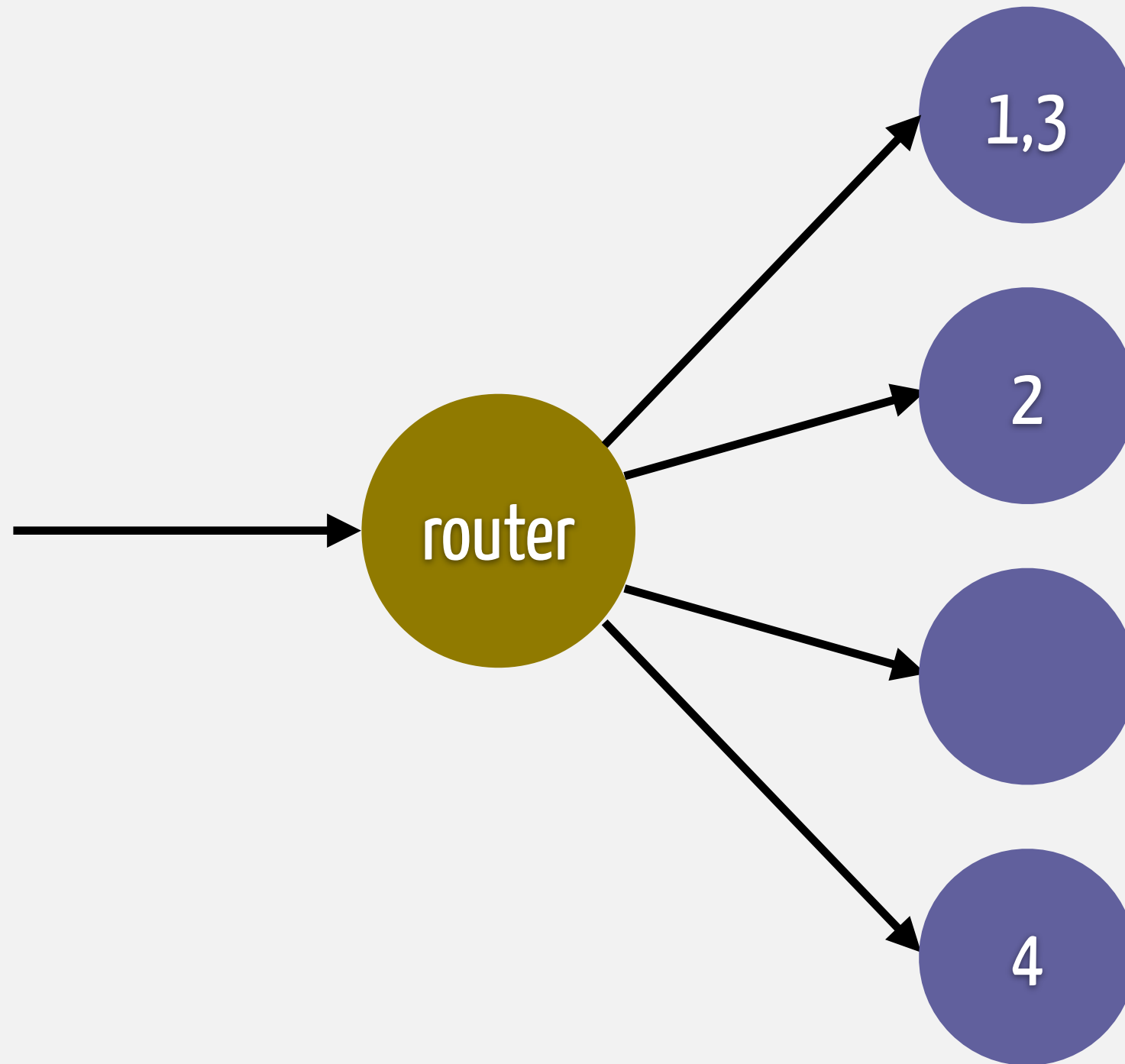
GenRouter (broadcast)



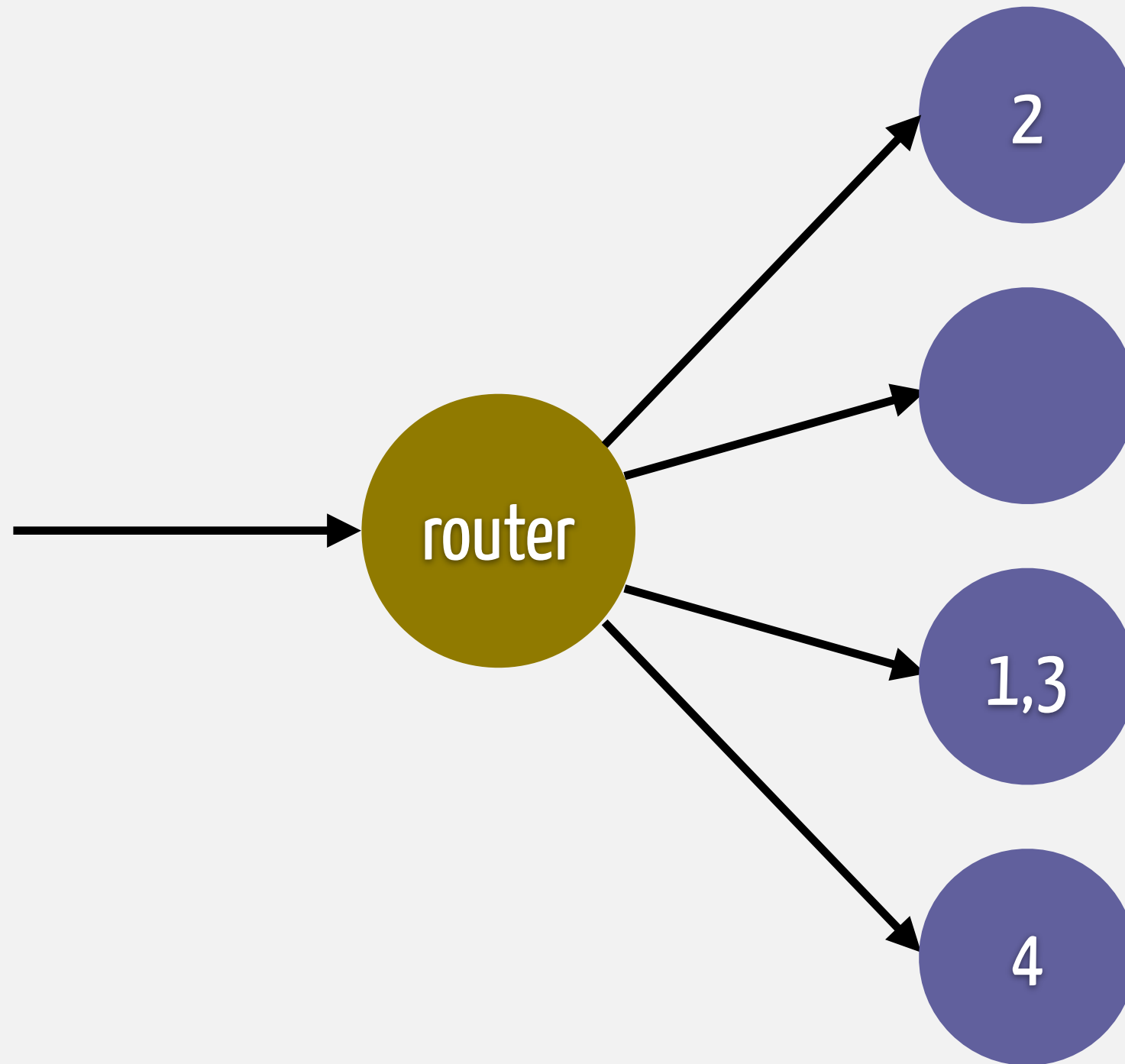
GenRouter (round-robin)



GenRouter (sharding)



GenRouter (random)



Data Parallelism

- `Stream.farm(data, ...)`
- `Stream.pmap(data, ...)`
- `Stream.chunked_pmap(data, ...)`

VM + OTP

Many interesting challenges

- What is the most efficient way of doing polymorphic dispatch?
- The most effective technique for implementing inline caches?

Many interesting challenges

- How to provide pipeline parallelism with back pressure efficiently?
- Which strategies are relevant for data parallelism?



```
defprotocol String.Inspect
  only: [BitString, List,

defimpl String.Inspect, for
  def inspect(false), do:
  def inspect(true), do:
  def inspect(nil), do:
  def inspect(""), do:
  def inspect(atom) do
```

Elixir is a functional, meta-programming aware language built on top of the Erlang VM. It is a dynamic language that focuses on tooling to leverage Erlang's abilities to build concurrent, distributed and fault-tolerant applications with hot code upgrades.

To install Elixir or learn more about it, check our [getting started guide](#). We also have [online documentation available](#) and a [Crash Course for Erlang developers](#). Or you can just keep on reading for a few code samples!

News: [Elixir v0.13.0 released](#),
[hex.pm](#) and [ElixirConf](#)
[announced](#)

JOIN THE COMMUNITY

- [#elixir-lang](#) on freenode IRC
- [elixir-talk](#) mailing list (questions)
- [elixir-core](#) mailing list (development)
- [Issues tracker](#)
- [@elixirlang](#) on Twitter

IMPORTANT LINKS

- [Wiki](#) with articles, projects and talks done by the community
- [Crash course](#) for Erlang developers

LEARNING RESOURCES

Language highlights

Everything is an expression

```
defmodule Hello do
  IO.puts "Defining the function world"

  def world do
    IO.puts "Hello World"
  end

  IO.puts "Function world defined"
end
```

elixir-lang.org





plataformatec

consulting and software engineering

Elixir Radar

The weekly Elixir newsletter, by Plataformatec

Elixir has a thriving community that is growing fast! Our community needs a way to keep up with all the interesting stuff that Elixirists are building and sharing. That's why we created Elixir Radar.

[Subscribe now](#)

By subscribing to Elixir Radar, you'll receive one email every week, with fresh news from the Elixir Community.

You will receive stuff like: top blog posts, interesting libraries, great talks, events and job opportunities.

Elixir Radar is published by [Plataformatec](#).

plataformatec.com.br/elixir-radar



elixir

@elixirlang / elixir-lang.org