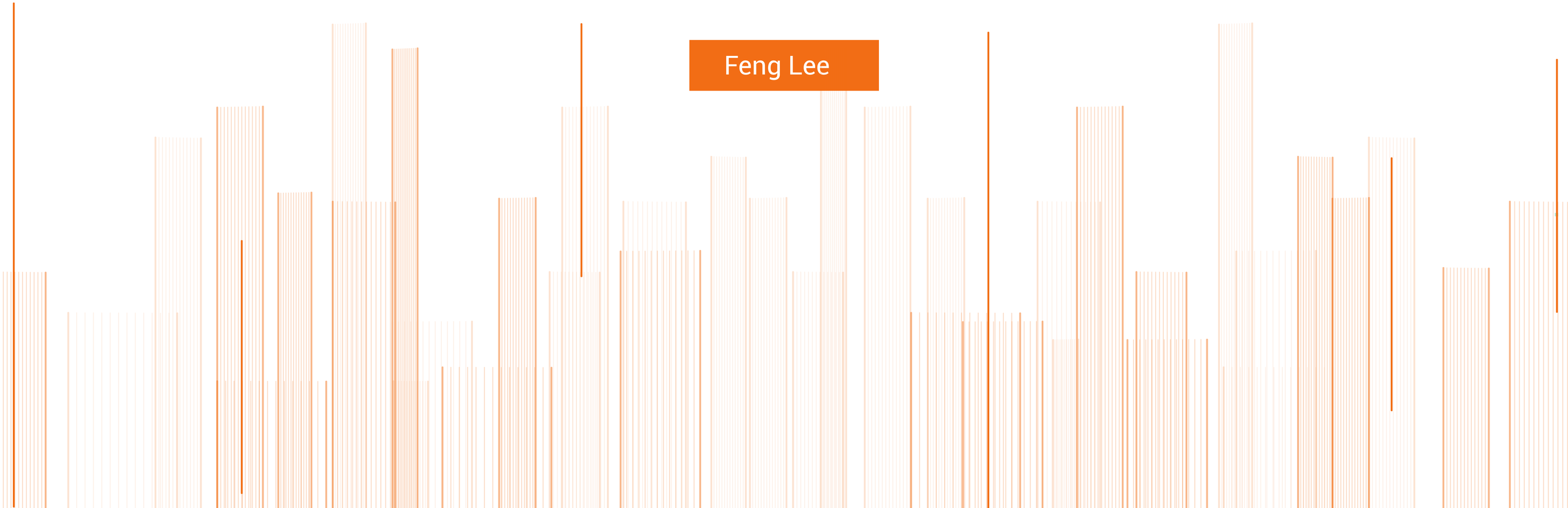
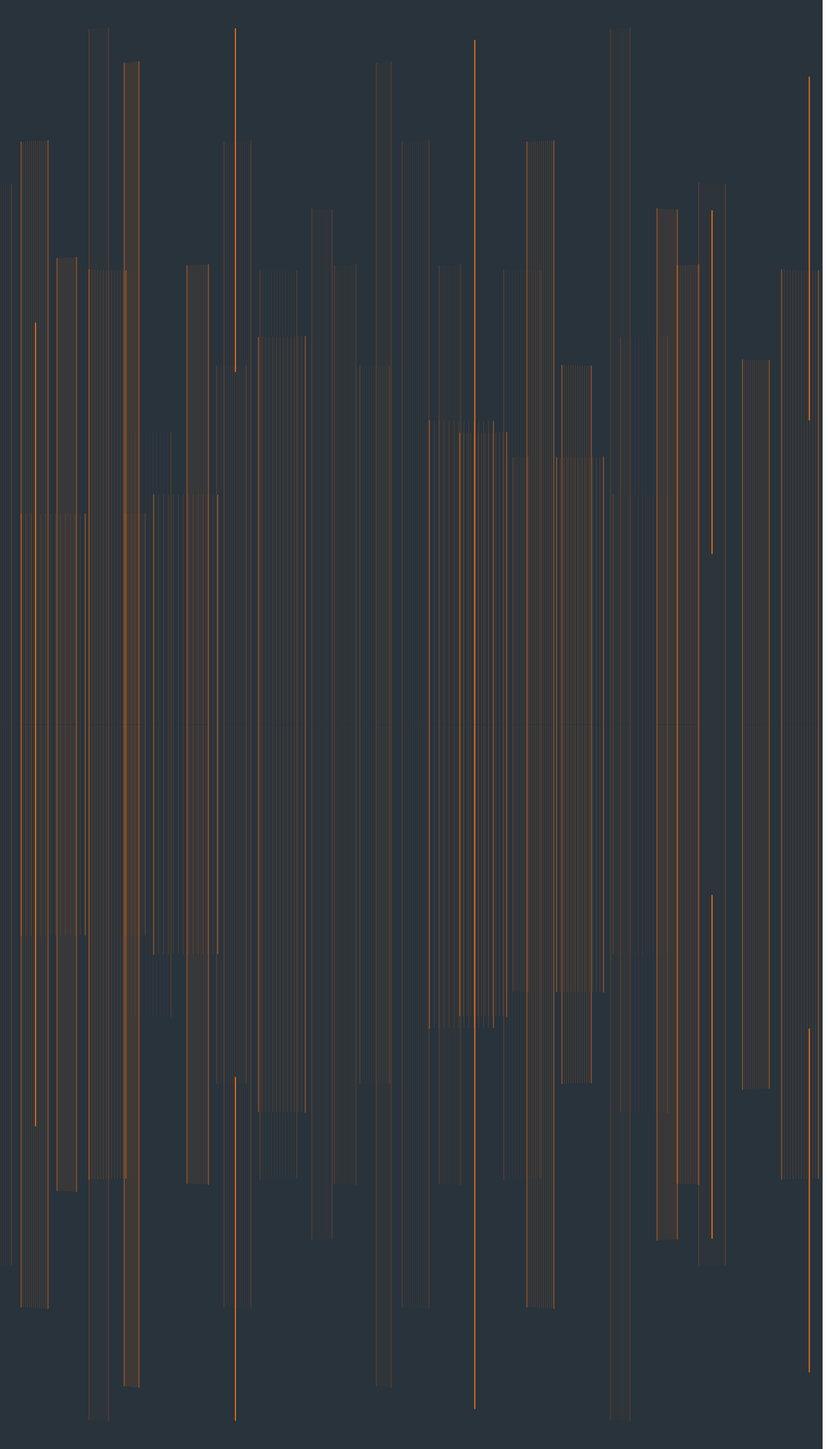
An Introduction to Hamler

Haskell-style functional programming language running on Erlang VM



Feng Lee



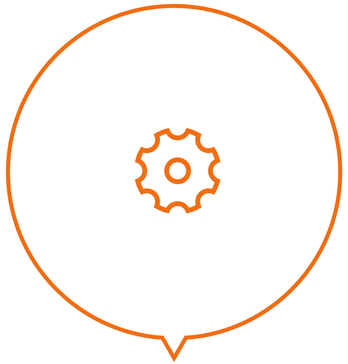
Content

The Erlang Programming Language

We have always believed that Erlang VM is a masterpiece of engineering.

EMQ X Broker and Erlang/OTP

For almost a decade, we have been developing software systems based on Erlang/OTP, especially our main product EMQ X - the scalable open-source MQTT broker.



Erlang VM is a masterpiece of engineering

We have always believed that Erlang is a masterpiece of engineering. With amazing concurrency, distribution and fault tolerance, it is one of the few general-purpose languages able to handle concurrency and soft realtime.

Key platform for IoT& 5G applications

Erlang VM is well-suited for the future potential of 5G, IoT, cloud and edge computing to build the next generation of highly concurrent, reliable, scalable and soft real-time applications.

About Actor Model

Professor Carl Hewitt published the famous paper Actor model of computation in 1974. In the thesis, he elaborates that an Actor is a computational entity that, in response to a message it receives, can concurrently:

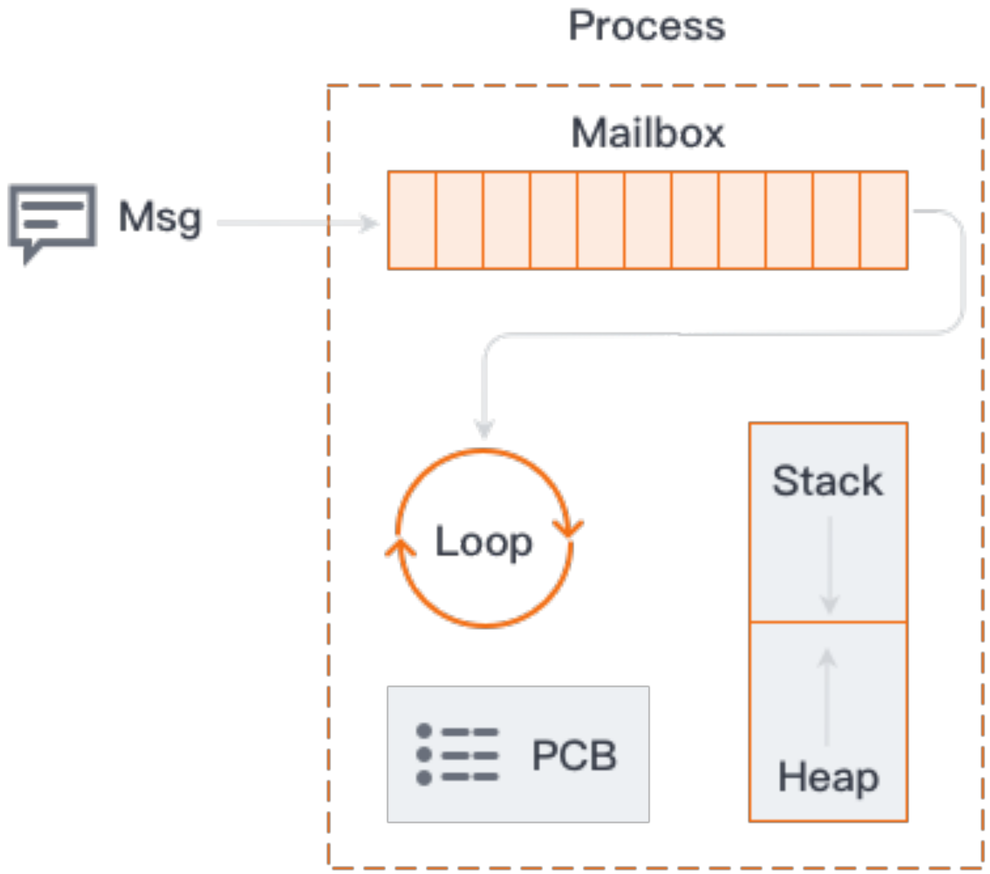
01 send a finite number of messages to other Actors;

02 create a finite number of new Actors;

03 designate the behaviour to be used for the next message it receives.

With the rise of multi-core computing and large-scale distributed systems, the Actor Model is becoming increasingly important because of its concurrent, parallel and distributed nature.

Erlang Process & Mailbox



An actor in Hamler/Erlang is defined as a process, which works similarly to an OS process.

Each process has its own memory, composed of a mailbox, a heap, a stack and a process control block(PCB) with information about the process.

Processes in Erlang are very lightweight. We can create millions of processes on a running Erlang virtual machine.

Problems with the Erlang programming language

20 years for academia and industry to improve Erlang

01 Type system for Erlang

Started with Prof. Philip Wadler and Simon Marlow in 2000, type annotation and **Dialyzer** a static analysis tool were introduced for Erlang:

Simon Marlow & Philip Wadler (1997): A practical subtyping system for Erlang

Philip Wadler (2002): The great type hope

02 New language on the Erlang VM

Since 2008, there have been about 20 projects in the industry trying to solve the problem. Elixir language project introduced Ruby syntax and attracted developers from the Ruby On Rails community!

here are others like lfe introduced Lisp syntax, alpaca、efene、elchemy、gleam etc. have attempted to introduce ML style syntax and static types, most of which are still in their early stage of development.

03 Erlang/OTP Arch on the JVM

The Akka project imitated the implementation of Erlang/OTP, but lost the soft real-time feature of Erlang/OTP.

Well-Typed's the Cloud Haskell project attempts to simulate the implementation of Erlang/ OTP in Haskell, the project is currently stalled.

Dreaming of a new language

Haskell-style functional programming language running on Erlang VM.

1. BEAM is a really awesome VM
2. Bring Haskell style language to BEAM
3. Introduce the Type System, ADT
4. Functional Programming & Concurrency

The Birth of Hamler

Hamler is a haskell-style strongly-typed programming language built on the Erlang virtual machine (VM) that uniquely combines compile-time type checking with support for runtime concurrency and soft real-time capabilities.

Functional Programming

Haskell and ML-like friendly syntax

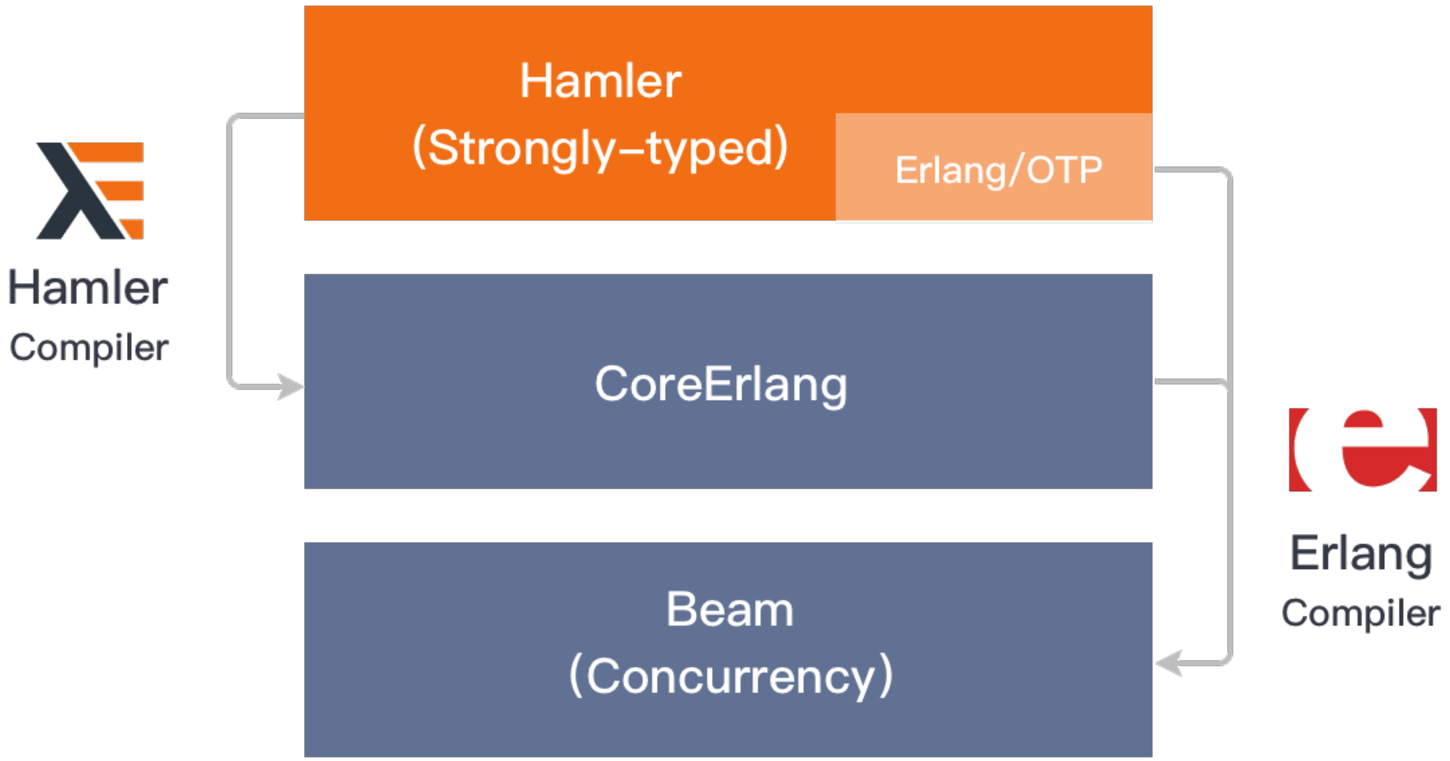
Strongly-typed Language

Type checking and inference at compile time

Concurrent Runtime

Concurrency, soft real-time support at runtime

The Hamler Compiler



The Hamler Compiler (cont.)

The Hamler 0.1 compiler was initially attempted to be implemented based on the GHC 8.10.1, but was later changed to adapt from Purescript Compiler 0.13.6's implementation.

Hamler Code

Haskell and ML-like friendly syntax

CST -> AST

Type checking, type inference

CoreFn

CoreFn of Purescript compiler

CoreErlang

CoreErlang IR is generated finally

Install Hamler

01 Homebrew (macOS)

```
brew tap hamler-lang/hamler
brew install hamler
```

02 Centos / Redhat

```
$ rpm -ivh hamler-$version-1.el7.x86_64.rpm
```

03 Debian / Ubuntu

```
$ dpkg -i hamler_$version_amd64.deb
```

Hello World!


```
module Main where

import Prelude

main :: IO ()
main = println "Hello, world!"
```

A Sexy QuickSort


```
quickSort :: forall a. Ord a => [a] -> [a]
quickSort [] = []
quickSort [x|xs] = quickSort [v|v <- xs, v < x] ++ [x] ++ quickSort [v|v <- xs, v >= x]
```

Hamler REPL


```
$ hamler repl
> -- List, range and enums
> [1,2,3]
> [1..10]
> ['a'..'z']

> -- erlang style maps
> import Data.Map as Map
> -- New map
> m = #{"foo" => "bar", "bar" => "foo"}
> -- Match Map
> #{"foo" := a, "bar" := b} = m
> -- get, put
> Map.get "foo" m -- a = "bar"
> Map.get "bar" m -- b = "foo"
> m1 = Map.put "key" "val"
> -- keys, values
> keys = Map.keys m
> values = Map.values m
```

Basic Types

Type	Values	Description
Atom	:ok, :error	Erlang Atom type
Boolean(Bool)	true | false	Boolean type
Char	'c', 'x'	UTF-8 character
String	"hello"	List of UTF-8 character
Integer(Int)	1, 2, -10	Integer type
Float(Double)	3.14	Float type
List	[1,2,3,4]	[Integer]
Tuple	(1, true)	
Map	#{"k" => "v"}	Erlang Map
Record		

Functional Programming

Functions, Recursive Functions, High-Order Functions, Closure, Lambda...

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n - 1)

> fact 10
3628800
> fact 5
120

fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)

> fib 10
89
> fib 5
8
```

```
apply :: forall a b. (a -> b) -> a -> b
apply f x = f x

compose :: forall a b. (b -> c) -> (a -> b) -> a -> c
compose g f x = g (f x)
```

Lambda (Anonymous Function)

```
multBy :: Integer -> Integer -> Integer
multBy n = \m -> m * n

mean :: Integer -> Integer -> Integer
mean = \x y -> (x + y) `div` 2  -- f = (\x -> \y -> (x + y) `div` 2)
```

Currying and Partial application

Currying

```
-- uncurried
plus :: (Integer, Integer) -> Integer
plus (x, y) = x + y

-- sum is the curried version of plus
sum :: Integer -> Integer -> Integer
sum x y = x + y
```

Partial application

```
sum 1  2      :: Integer
sum 1 (2 + 3) :: Integer

add2 = sum 2  :: Integer -> Integer -- partially applied
x    = add2 3 :: Integer        -- x = 5
```

Pattern Matching, and Guards

Function Pattern Matching

```
(x, y) = (1, 2)

-- function declartion via pattern matching
allEmpty [] = True
allEmpty _ = False

-- pattern matching stops when it finds the first match
```

Guarded Equations

```
abs n | n > 0     = n
      | otherwise = -n
```

List Comprehension

A list comprehension consists of four types of elements: *generators*, *guards*, *local bindings*, and *targets*.

```
-- examples
[x*2 | x <- [1,2,3]]   -- [2,4,6]

[x * x | x <- [1..10]] -- [1,4,9,16,25,36,49,64,81,100]

-- multiple generators
[(x,y) | x <- [1,2,3], y <- [4,5]]

-- dependent generators
[(x,y) | x <- [1..3], y <- [x..3]]

-- conditions
even i = 0 == i % 2
[x | x <- [1..10], even x]
```

Algebraic Data Types(ADTs)


```
-- type synonym
type Name = String
"Miles" :: Name
"Miles" :: String

newtype UInt8 = UInt8 Integer
1 :: Integer
UInt8 1 :: UInt8

-- sum datatype
data Color = Red | Green | Blue
Blue :: Color

-- product datatype
data Pair = Pair Integer Integer
Pair 3 4 :: Pair

-- record product datatype
data Person = Person {
  name :: String
  age :: Integer
  address :: String
}
Person {name = "Miles", age = 50, address = "NY"} :: Person

-- generic datatype (maybe for example)
data Maybe a = Just a | None
data Result val err = Ok val | Error err

-- recursive datatype
data Tree = Leaf Integer | Node Tree Tree
```

Spawn a new process

In Hamler, a new process is created via the `spawn` functions, which are defined in `Control.Process.Spawn` module.

```
-- | Create a process
spawn :: forall a. IO a -> Process Pid

-- | Create and link a process
spawnLink :: forall a. IO a -> Process Pid

-- | Create and monitor a process
spawnMonitor :: forall a. IO a -> Process (Pid, Ref)
```

Send/Receive message


```
go :: Process ()
go = do
  pid <- spawn recv
  pid ! :msg

recv :: Process ()
recv = receive x -> printf "recv: %s" (showAny x)
```

```
go :: Process ()
go = do
  pid <- spawn selectiveRecv
  pid ! :bar
  pid ! :foo

selectiveRecv :: Process ()
selectiveRecv = do
  receive :foo -> println "foo"
  receive :bar -> println "bar"
```

```
go :: Process ()
go = do
  pid <- spawn recvAfter
  pid ! :foo

recvAfter :: Process ()
recvAfter =
  receive
    :bar -> println "recv bar"
  after
    1000 -> println "timeout"
```

A Ping/Pong Example

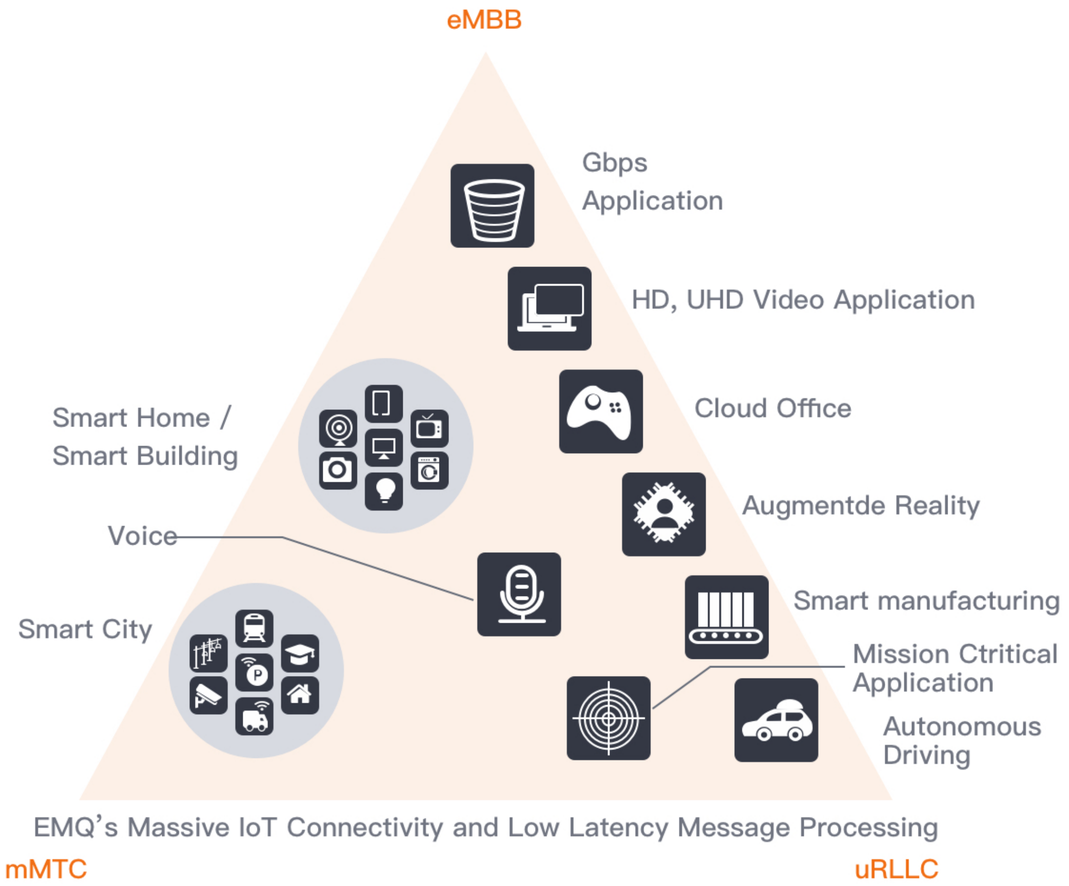

```
import Prelude

go :: Process ()
go = do
  self <- getSelf
  pid <- spawn loop
  pid ! (self, :ping)
  receive
    :pong -> println "Pong!"
  pid ! :stop

loop :: Process ()
loop =
  receive
    (from, :ping) -> do
      println "Ping!"
      from ! :pong
      loop
    :stop -> return ()
```

Hamler for next decade

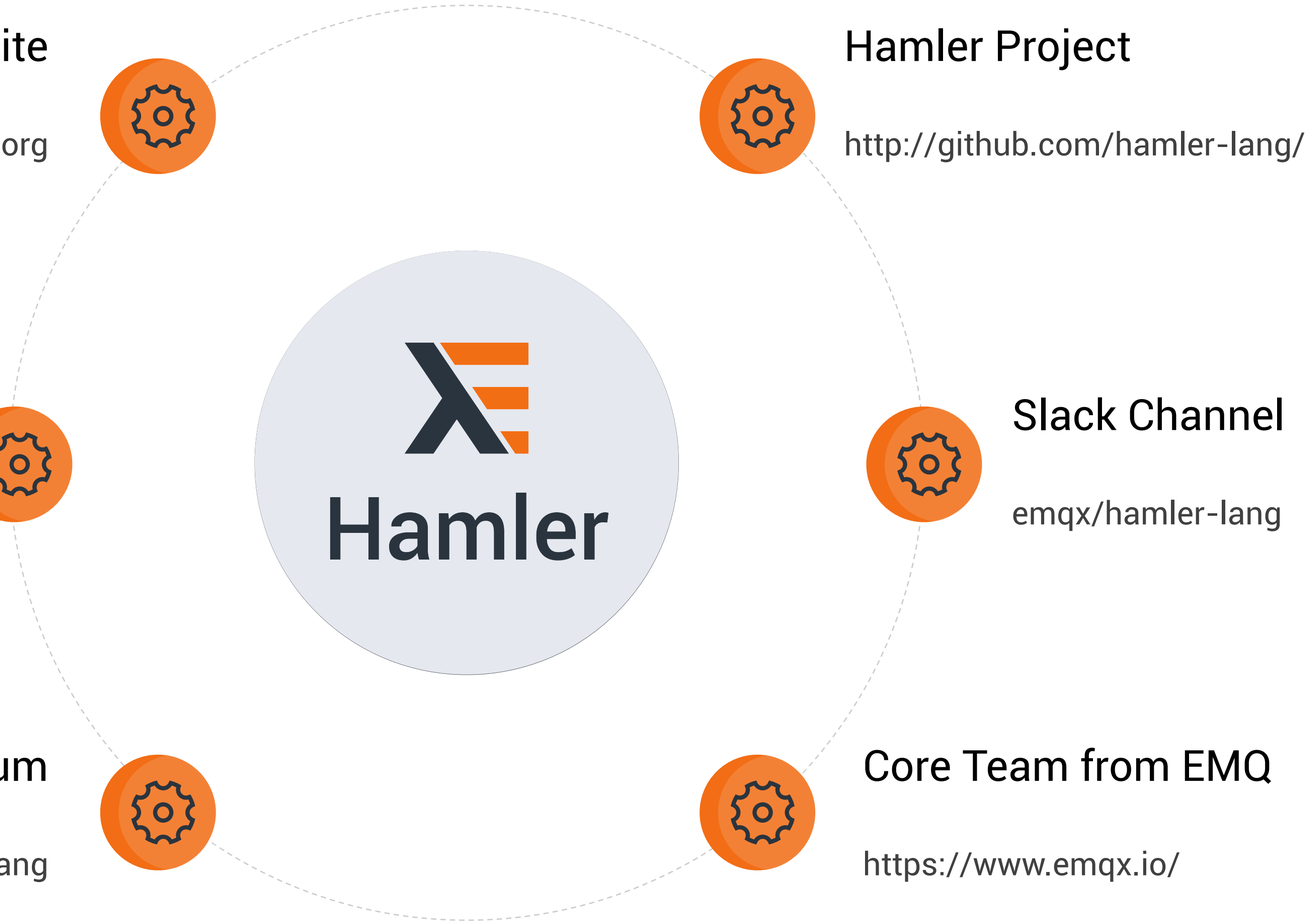
Hamler empowers industries to build the next generation of scalable, reliable, realtime applications, especially for 5G, IoT and edge computing.



- eMBB - enhanced Mobile Broadband
- mMTC - massive Machine Type Communications
- uRLLC - ultra Reliable Low Latency Communications

Communities, discussion and supports

The hamler programming language is an open-source project, licensed under BSD3



Hamler-lang website

https://www.hamler-lang.org

Hamler Project

http://github.com/hamler-lang/

Hamler Documentation

https://github.com/hamler-lang/documentation/

Slack Channel

emqx/hamler-lang

Twitter, Reddit, Medium

@hamlerlang /r/HamlerLang @hamlerlang

Core Team from EMQ

https://www.emqx.io/

Thanks