TDD is Dead, Long Live TyDD

Rahul Goma Phulore (@missingfaktor)

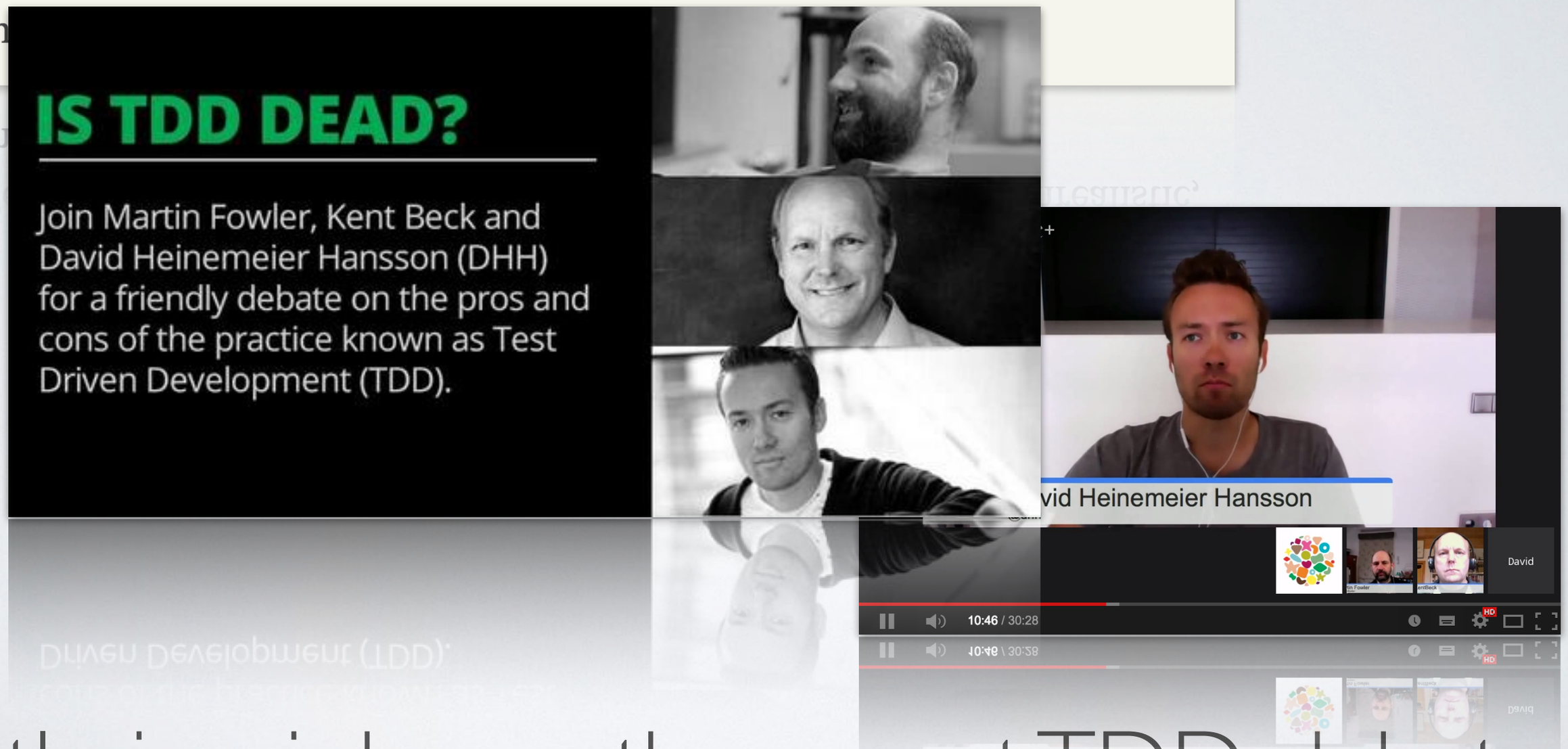
2014.09.06

Young Haskell Padawans, Pune

TDD is dead. Long live testing.

By David Heinemeier Hansson on April 23, 2014

Test-first fundamentalism is like abstinence-only sex ed: An unrealistic, in



Title is a joke on the recent TDD debate

But seriously…

We do actually believe that TDD is **NOT** a good idea

Why do people TDD anyway?

- Core tenets
  - Red, Green, Refactor
  - Never write any code unless it satisfies some test; write only as much required to get test passing

Why do people TDD anyway?

- Benefits* of TDD, as per the practitioners
  - Use cases become clearer
  - Component interactions become clearer; helps in achieving loose coupling
  - Harness always available

* Perceived ;)

** Half kidding

Alternative?

TyDD!

Don Stewart
@donsbot

Following

. @Functionalworks nope. We use types first, then API "discovery" via properties/QuickCheck. Types-first, algebra-driven development.

Reply Retweeted Favorited More

RETWEETS	FAVORITES
34	35

5:40 PM - 10 Aug 2014

o_O

I haven't copied all the functions that the `free` package provides. I mainly omitted the recursion schemes because there are a few other recursion schemes that I was also considering:

```
foldFree1 :: (a -> r) -> (f r -> r) -> Free f a -> r
-- or
foldFree2 :: (FreeF f a r -> r) -> Free f a -> r
```

... and their `FreeT` equivalents, which use `m r` as the carrier of the algebra instead:

```
foldFreeT1 :: (a -> m r) -> (f (m r) -> m r) -> FreeT f m a -> r
-- or
foldFreeT2 :: (FreeF f a (m r) -> m r) -> FreeT f m a -> r
```

which obeys the monad laws:

```
return x >>= f
  = (Just x) >>= f    -- definition of return for Maybe monad
  = f x               -- definition of >>= for Maybe monad
                      -- correct according to monad law 1

Just x >>= return
  = return x          -- definition of >>= for Maybe monad
  = Just x            -- definition of return for Maybe monad
                      -- correct according to monad law 2

Nothing >>= return
  = Nothing           -- definition of >>= for Maybe monad
                      -- correct according to monad law 2
```

What the hell are these people talking about?

DON'T PANIC

Take your towels*,
and come for a ride

* The hitchhiker's guide to the galaxy joke

Haskell syntax (basics)

```
x :: Int
x = 0

add :: Int -> Int -> Int
add a b = a + b

add 2 3

apply :: a -> (a -> b) -> b
apply v f = f v
```

TyDD
(Type-driven development)

- Aka
  - Type-directed programming
  - Denotational design (Peter Landin and Conal Eliott)
  - Types-first development (Tomas Petricek)
  - Algebra-driven development
  - Type-driven development (Don Stewart, during XMonad development)

The Haskell Way

Types

Algebra

Functions

Functions

- Not your grandma's functions!
- First-class functions
- Higher order functions, combinators
- Composition
- Partial application
- Seeing things as functions (bindings, patterns, state changes, instructions)

Types

- Types categorize terms, based on their properties and the kind of operations they can support
- Types encode invariants/knowledge about your program
- Intuition: sets

Types are **NOT** classes!

- `Map<String, Integer>`
  `Map<Integer, Char>`

  Different type but same class!

- `Animal a = new Cat();`
  `Animal b = new Dog();`

  Same type but different class!

Well typed programs
never
go wrong!

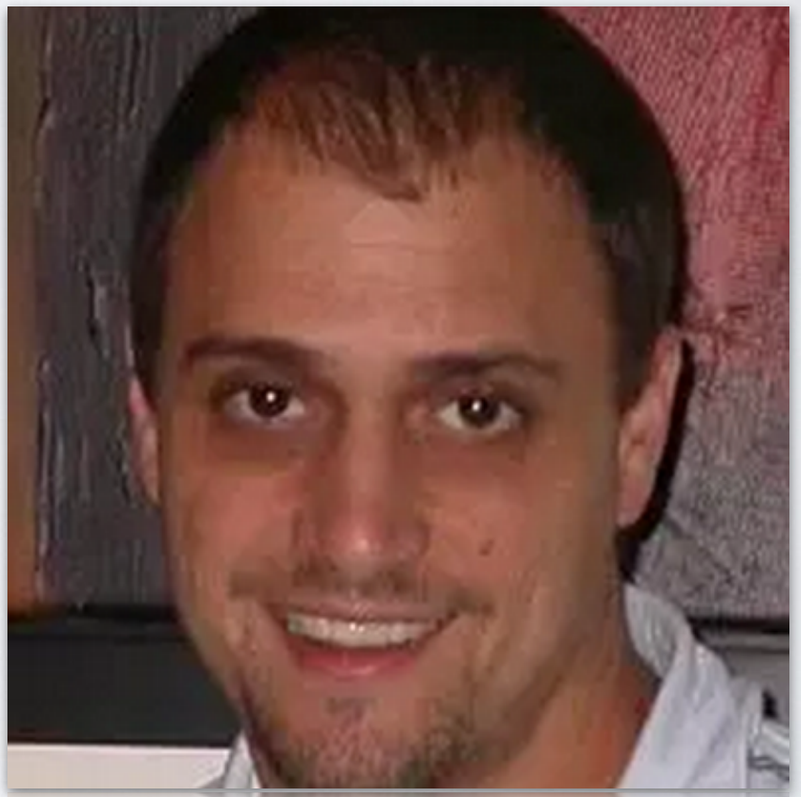
Dr Robin Milner

We can use *type systems* for:

- *Checking* a program has the intended properties
- *Guiding* a programmer towards a correct program
- Building *expressive* and *generic* libraries

Edwin Brady

> When writing code in a statically typed language sometimes types are considered as orthogonal to the logic of the code. We write them to appease the compiler, or get performance or IntelliSense & navigation, but all of these has no relation to the code itself.
>
> This is wrong.



Ittay Dror

Ken Scambler
@KenScambler

Following

"Is this a type error?" misses the point. Ask "How can I turn this into a type error?" instead.

Reply Retweeted Favorited More

RETWEETS	FAVORITES
17	7

12:52 PM - 2 Sep 2014

Make illegal states
unrepresentable.

Yaron Minsky

Let's take an example…

How many implementations are possible for this Java type?

```
public static <A> A func(A value) {
   // TODO
}
```

We could return the input value back.

```
public static <A> A func(A value) {
   return value;
}
```

We could return null.

```
public static <A> A func(A value) {
   return null;
}
```

We could throw an exception.

```
public static <A> A func(A value) {
   throw new
      RuntimeException("Good luck");
}
```

You could write to a file, and then return the value.

```
public static <A> A func(A value) {
  FileUtilities.write("stuff");
  return value;
}
```

You could log a message, and then return a value.

```
public static <A> A func(A value) {
   logger.debug("Strings suck.");
   return value;
}
```

You could change some state somewhere, and then return a value.

```
public static <A> A func(A value) {
   counter = counter + 1;
   return value;
}
```

You could even launch missiles, and Java type system won't bat an eye!

```
public static <A> A func(A value) {
```

```
    ;
    return value;
}
```

Let's see how Haskell fares here

```
-- Type to satisfy
func :: a -> a
```

You could return the input value

```
func :: a -> a
func x = x
```

And umm… that's it!

Actually I am cheating a bit…

```
func :: a -> a
func x = error “Good luck”
```

But let’s ignore this bit for now

In Haskell, all of those different things we did in Java will have different types

```
-- Returns value optionally
func :: a -> Maybe a

-- Could potentially fail
func :: a -> Either e a

-- Could write to a file
func :: a -> IO a

-- Could log a message
func :: a -> Writer l a
```

```
-- Could change some state
func :: a -> State s a

-- Launch missiles! (Explicit)
func :: a -> DangerOhMyGodNo a
```

Parametricity ⇒ Theorems for free!

Theorems for free!

Philip Wadler
University of Glasgow*

June 1989

Abstract

From the type of a polymorphic function we can derive a theorem that it satisfies. Every function of the same type satisfies the same theorem. This provides a free source of useful theorems, courtesy of Reynolds' abstraction theorem for the polymorphic lambda calculus.

1 Introduction

Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.

The purpose of this paper is to explain the trick. But first, let's look at an example.

Say that r is a function of type

$$r : \forall X.\ X^* \to X^*.$$

list of A yielding a list of A', and $r_A : A^* \to A^*$ is the instance of r at type A.

The intuitive explanation of this result is that r must work on lists of X for *any* type X. Since r is provided with no operations on values of type X, all it can do is rearrange such lists, independent of the values contained in them. Thus applying a to each element of a list and then rearranging yields the same result as rearranging and then applying a to each element.

For instance, r may be the function $reverse : \forall X.\ X^* \to X^*$ that reverses a list, and a may be the function $code : Char \to Int$ that converts a character to its ASCII code. Then we have

$$\begin{array}{rl} & code^*\ (reverse_{Char}\ [\text{'a'}, \text{'b'}, \text{'c'}]) \\ = & [99, 98, 97] \\ = & reverse_{Int}\ (code^*\ [\text{'a'}, \text{'b'}, \text{'c'}]) \end{array}$$

which satisfies the theorem. Or r may be the function $tail : \forall X.\ X^* \to X^*$ that returns all but the first element of a list, and a may be the function $inc : Int \to Int$ that adds one to an integer. Then we have

Algebraic Data Types

- Primary way of modelling data.
- Sum and product types.
- What makes them "algebraic"? We will talk about it some time later.

Examples

```
data Option a = Some a
              | None
```

```
data ProcessState e a = Succeeded a
                      | Failed e
                      | Ongoing
```

```
data ColorChoice = Custom Color
                 | Default
```

Examples

```
data ParseResult a = ParseSuccess a
                   | ParseFailure Error
```

```
data Parser a =
   Parser (String -> ParseResult a)
```

Composition

- Types encode properties precisely, thereby facilitating composition.
- Help with iterative development like nothing else.

Example

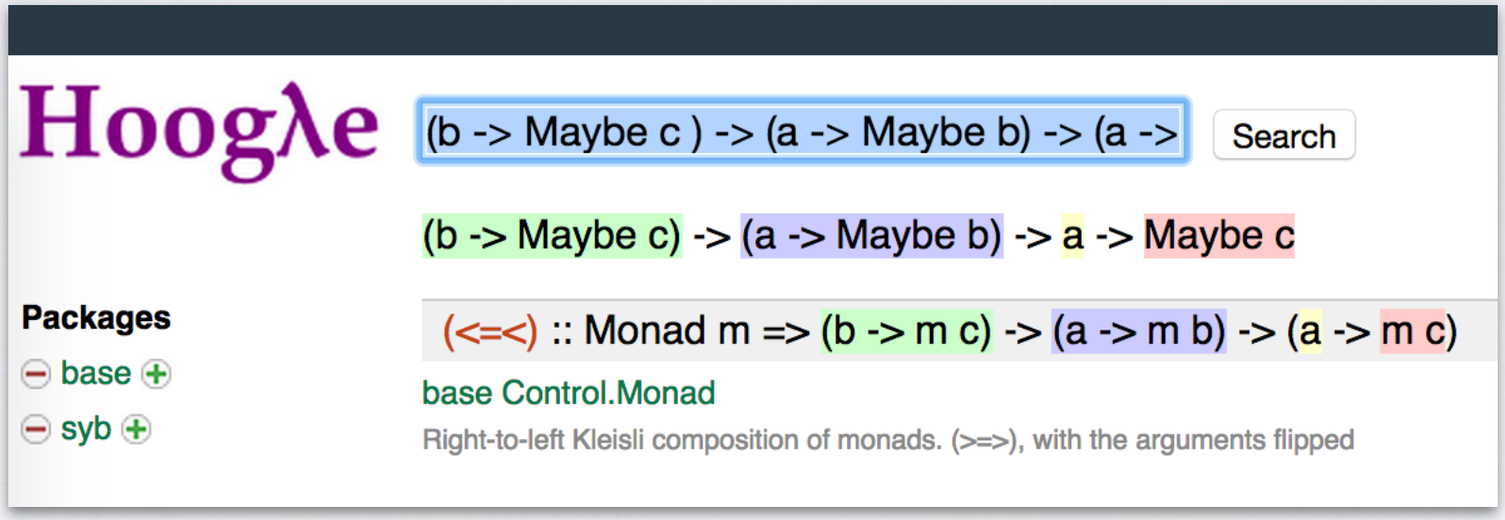
```
-- You had two functions as follows.
-- Both in separate modules.
foo :: X -> Y
bar :: Y -> Z

-- This is how they are composed in
-- some part of the system.
bar . foo
```

Example

```
-- Requirements changed, and the two
-- functions now return values
-- optionally.
foo :: X -> Maybe Y
bar :: Y -> Maybe Z

-- The following will now be a type
-- error!
bar . foo
```

Example

```
-- We will fire up Hoogle (Google for
-- Haskell) and search by wanted type.
```




```
-- We start using the suggested
-- function, and voila, everything
-- type-checks! And has the correct
-- semantics too.
bar <=< foo
```

TyDD workflow

- Meditate over your domain/system, and encode the knowledge into the type system.
- Write key function types.
- Come up with combinators and higher order functions.
- Observe what properties your types fulfil, and see if you can benefit from standard computational abstractions.

Tooling

Hoogle

Bool -> IO () -> IO ()

Search

Bool -> IO () -> IO ()

Packages

- base
- HTTP
- syb
- parallel

addFinalizer :: key -> IO () -> IO ()

base System.Mem.Weak

A specialised version of mkWeakPtr, object returned is simply thrown away (however the fi

unless :: Monad m => Bool -> m () -> m ()

base Control.Monad

The reverse of when.

when :: Monad m => Bool -> m () -> m ()

base Control.Monad

Conditional execution of monadic expressions. For example, > when debug (putStr "Debug

Hoogle

(a -> m b) -> (b -> m c) -> (a -> m c)

Search

(->-) :: (a -> m b) -> (b -> m c) -> a -> m c

semigroupoids - Data.Functor.Bind

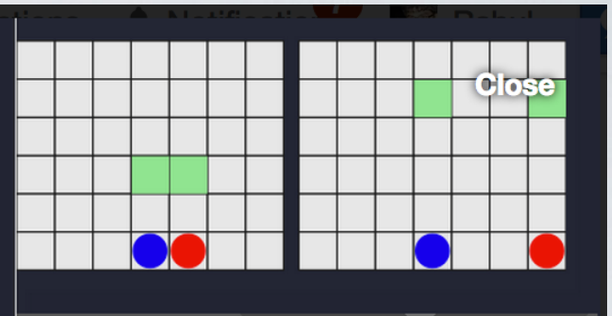
(>=>) :: (a -> m b) -> (b -> m c) -> a -> m c infixr 1

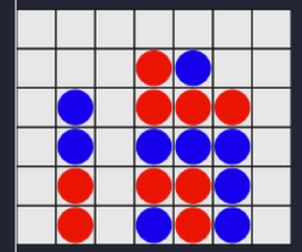
base - Control.Monad

Left-to-right Kleisli composition of monads.

Hayoo

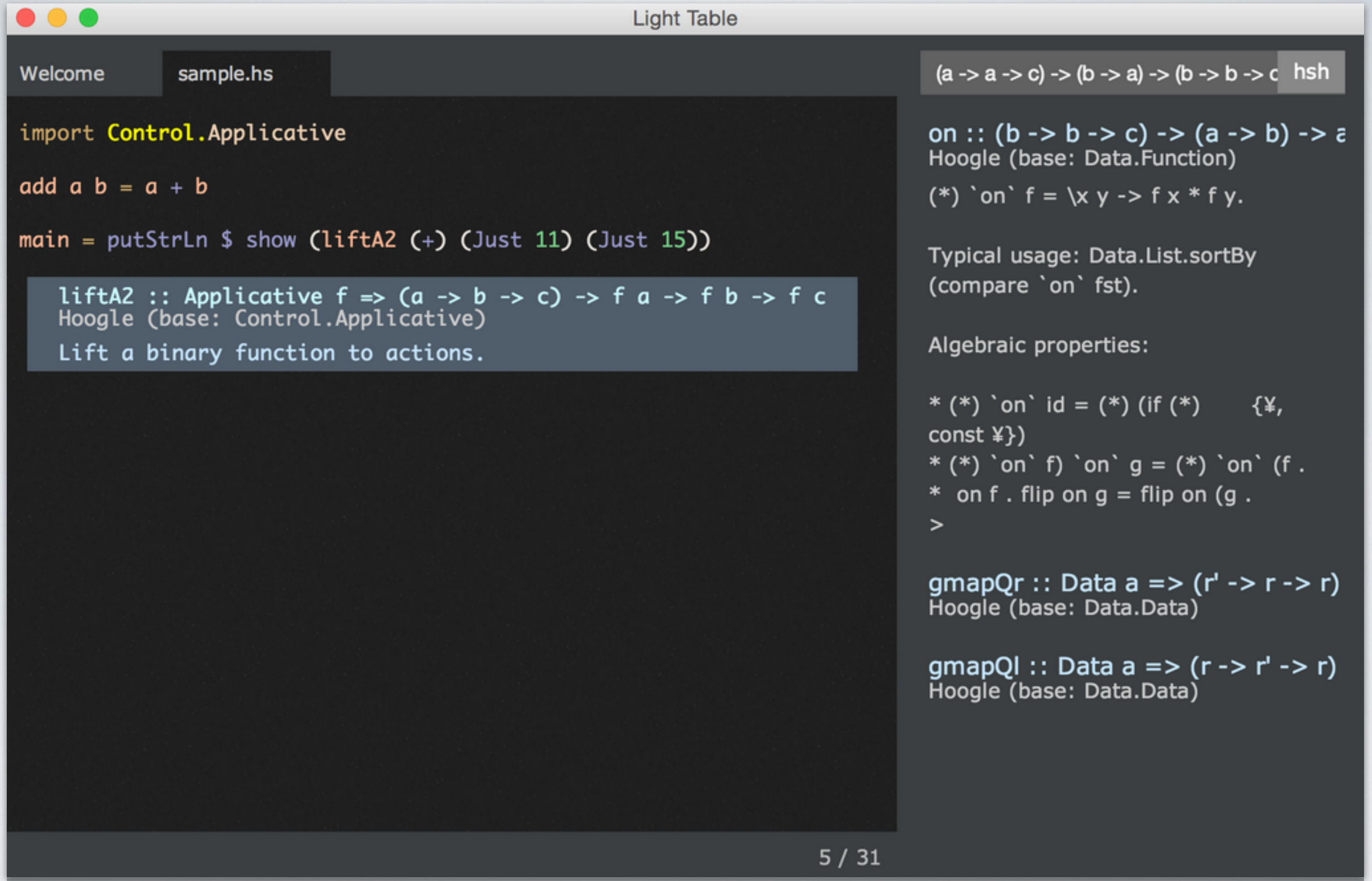
```
    renderSVG output (mkSizeSpec (Just 400) (Just 200)) boards

annotated :: Component
annotated = rows ||| board cells === cols
  where rows = vcat $ label ∘ show <$> [1..6]
        cols = strutX 100 ||| (hcat $ label ∘ return <$> "ABCDEFG")
        label str = (text str # scale 60) `atop` cell

background = square 700 # bg white

board :: Game → Component
board played = hcat $ vcat <$> cellList played

x = circle 46 # fc blue
o = circle 46 # fc red
g = square 100 # bg palegreen
r = square 100 # bg indianred

cell = square 100 # lc black # lw 4 # bg white

cellList cells = [[cells ! (row, col) | row :: Int ← [6,5..1]] | col :: Int ← [1..7]]

cells :: Game
cells = Map.fromList ∘ zip positions $ repeat cell

positions = [(row, col) | row ← [1..6], col ← [1..7]]
```





```
- ConnectFour.hs  Haskell Ind λP AC  42 : 0  40%  - odd.png  Image[png]  1 : 0  Al
Failed, modules loaded: none.
Prelude Data.List> :load "/home/tikhon/Documents/quora/connect-four/ConnectFour.hs"
[1 of 1] Compiling ConnectFour      ( /home/tikhon/Documents/quora/connect-four/ConnectFour.hs, interpreted )
Ok, modules loaded: ConnectFour.
*ConnectFour Data.List> run "aftereven"
*ConnectFour Data.List> :load "/home/tikhon/Documents/quora/connect-four/ConnectFour.hs"
[1 of 1] Compiling ConnectFour      ( /home/tikhon/Documents/quora/connect-four/ConnectFour.hs, interpreted )
Ok, modules loaded: ConnectFour.
*ConnectFour Data.List> run "aftereven"
*ConnectFour Data.List> run "aftereven"
*ConnectFour Data.List> run "aftereven"
*ConnectFour Data.List> :load "/home/tikhon/Documents/quora/connect-four/ConnectFour.hs"
[1 of 1] Compiling ConnectFour      ( /home/tikhon/Documents/quora/connect-four/ConnectFour.hs, interpreted )
Ok, modules loaded: ConnectFour.
*ConnectFour Data.List> run "aftereven"
*ConnectFour Data.List> :load "/home/tikhon/Documents/quora/connect-four/ConnectFour.hs"
[1 of 1] Compiling ConnectFour      ( /home/tikhon/Documents/quora/connect-four/ConnectFour.hs, interpreted )
Ok, modules loaded: ConnectFour.
*ConnectFour Data.List> run "aftereven"
*ConnectFour Data.List> run "vertical-rule"
*ConnectFour Data.List> run "baseinverse"
*ConnectFour Data.List> run "lowinverse"
*ConnectFour Data.List> run2 "lowinverse" "lowinverse2"
*ConnectFour Data.List> run2 "lowinverse" "lowinverse2"
*ConnectFour Data.List> run "dd"
*ConnectFour Data.List>
* *haskell — connect-four*  Inf-Haskell Compilation  7047 : 24  Bottom
```

ghc-mod (Emacs)



Lighttable plugin

The djinn package

[Tags: bsd3, program]

Djinn uses an theorem prover for intuitionistic propositional logic to generate a Haskell expression when given a type.

```
> pl \x y -> x y
id

> unpl id
(\ a -> a)

> pl \x y -> x + 1
const . (1 +)

> unpl const . (1 +)
(\ e _ -> 1 + e)

> pl \v1 v2 -> sum (zipWith (*) v1 v2)
(sum .) . zipWith (*)

> unpl (sum .) . zipWith (*)
(\ d g -> sum (zipWith (*) d g))

> pl \x y z -> f (g x y z)
((f .) .) . g

> unpl ((f .) .) . g
(\ e j m -> f (g e j m))
```

Djinn, pl, and more

Code editing, IDEs, and type systems

Code editing will be done in structural editors, which will look nothing like the existing batch of IDEs that are little more than glorified text editors (and they are actually rather poor text editors). In a structural editor, the programmer will construct expressions which may have holes in them not yet filled with terms. Importantly, these structural editors will be type-directed, so for any given hole the programmer can be presented with set of values matching the expected type, ordered in some sensible way. The editor will perform local program search to enable autocompleting of multiple expressions. If you've ever seen someone live-code Agda, you'll know how powerful and productive this idea could be. Yeah, the actual interface for programming Agda is still kind of 1970s (a custom Emacs mode), but the idea of type-directed editors is powerful. It makes it clear that types are effective not just at preventing many software errors, but also in *guiding development*. They are a powerful tool to augment our puny programming brains.

What future holds

Benefits

- TyDD makes you think about the problem, and pin it down precisely
- It emphasizes and simplifies communication.

Benefits

- Type system constantly ensures the constraints hold throughout.

- Tests = There exists. (∃)
  Types = For all. (∀)

- Better harness means more confidence. More confidence means more peaceful sleep!

Benefits

- When evolving software / incorporating change, type direction helps a lot. (As we just saw with an example. Except the impact is much larger in real codebases.)

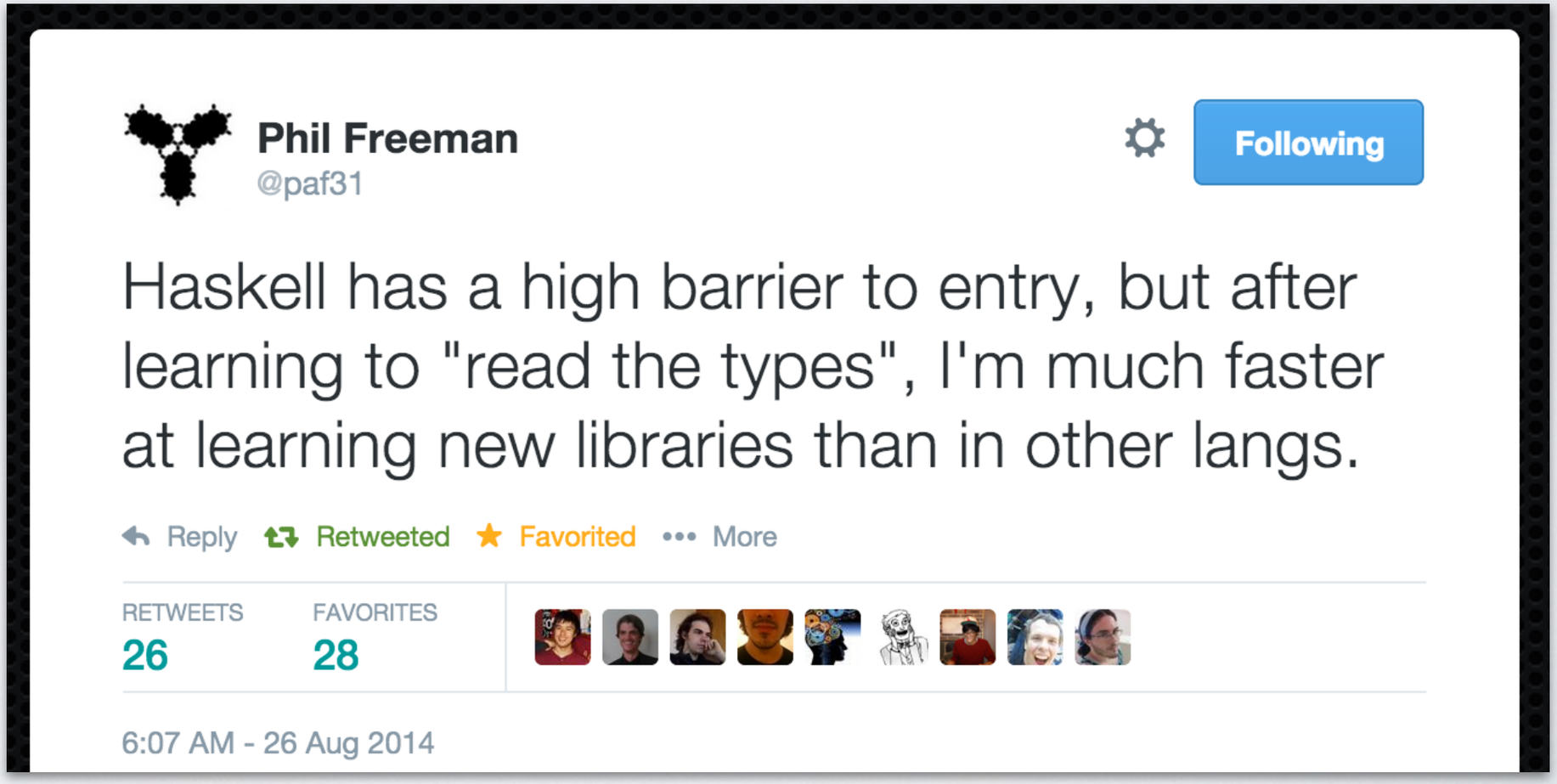
Benefits

- More principled, compositional, reusable, extensible abstractions.
- There is a reason you don't see abstractions like applicative or monad in wide use in languages with weaker type system or untyped languages.

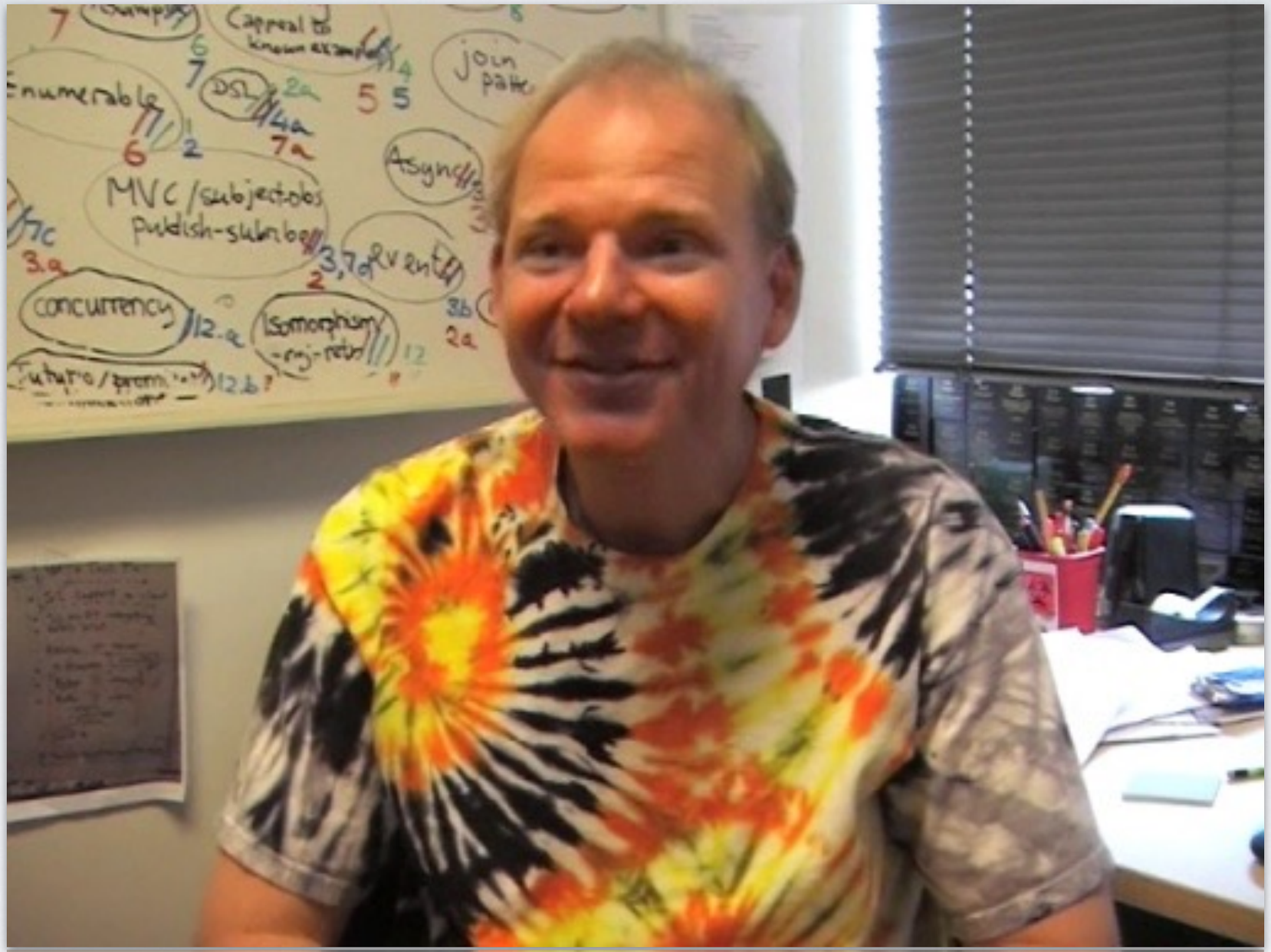
Benefits

- Easy discoverability. We have just seen what Hoogle (and the likes) are capable of!

- Also:



But is any of this useful in my “real world”?

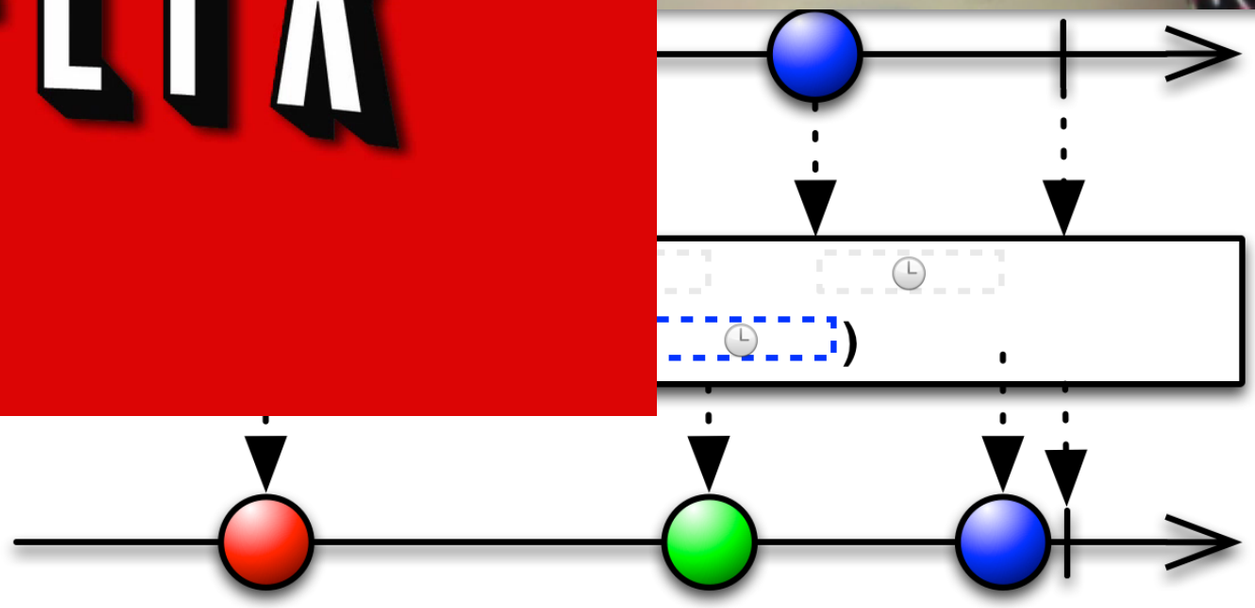

Enumerable
MVC / subject.obs
publish-subscribe
concurrency
Isomorphism
join pattern
Async

LINQ on .NET


```
-- UPDATE
velStep d obj =
    let f n = if d.x == 0 || d.y == 0 then toFloat n els
    in  { obj | vx <- f d.x, vy <- f d.y }

dirStep {x,y} obj =
    { obj | dir <- if | x > 0 -> "east"
                      | x < 0 -> "west"
                      | y < 0 -> "south"
                      | y > 0 -> "north"
                      | otherwise -> obj.dir }

runStep running obj =
    let scale = if running then 2 else 1
    in  { obj | vx <- obj.vx * scale, vy <- obj.vy * sca

timeStep t ({x,y,vx,vy} as obj) =
    { obj | x <- clamp (-areaW/2) (areaW/2) (x + t * vx)
          y <- clamp (-areaH/2) (areaH/2) (y + t * vy)

step (time,arrows,run) hero =
    timeStep time . dirStep arrows . runStep run . velSt


-- HERO
```

Functional reactive programming

NETFLIX



sets on each

Information on debounce vs throttle:

- http://drupalmotion.com/article/debounce-and-throttle-visual-explanation
- http://unscriptable.com/2009/03/20/debouncing-javascript-methods/
- http://www.illyriad.co.uk/blog/index.php/2011/09/javascript-dont-spam-your-server-debounce-and-throttle/

timeout	The time each value has to be 'the most recent' of the rx.lang.scala.Observable to ensure that it's not dropped.
returns	An rx.lang.scala.Observable which filters out values which are too quickly followed up with newer values.

See also `Observable.throttleWithTimeout`

Reactive extensions, RxJava, Rx.js

```haskell
newtype Haxl a = Haxl { unHaxl :: Result a }

data Result a = Done a
              | Blocked (Haxl a)

instance Monad Haxl where
  return a = Haxl (Done a)
  m >>= k = Haxl $
    case unHaxl m of
      Done a    -> unHaxl (k a)
      Blocked r -> Blocked (r >>= k)
```

Haxl from Facebook

```haskell
main :: IO ()
main = hakyllWith config $ do
    -- Static files
    match ("images/*.jpg" .||. "images/*.png" .||. "images/*.gif" .||.
           "favicon.ico" .||. "files/**") $ do
        route   idRoute
        compile copyFileCompiler

    -- Formula images
    match "images/*.tex" $ do
        route   $ setExtension "
        compile $ getResourceBoc
            >>= loadAndApplyTemp
            >>= pdflatex >>= pdf

    -- Compress CSS
    match "css/*" $ do
        route idRoute
        compile compressCssCompi
```


About me

Hi! I am Jasper Van der Jeugt. I was born in 1990, and spent most of my youth in Lokeren & Ghent, Belgium. Currently, I am a professional Haskell programmer at Better in Zürich. Before that, I studied Computer Science at Ghent University.

Did I tell you I like to make stuff? I love to make stuff! That is why I am somewhat active in open source development.

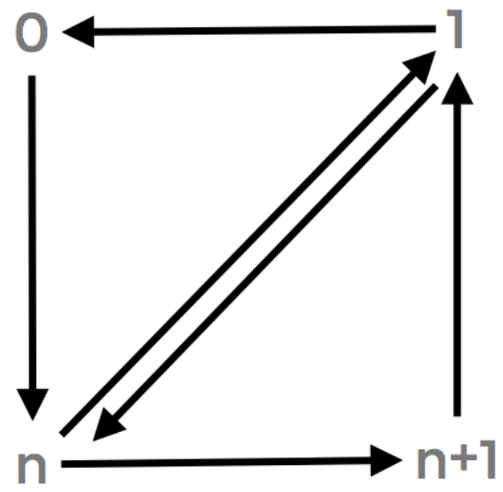
Apart from that, I have a very broad taste in music and movies, and I also enjoy skateboarding and taking pictures.

Recent blogposts

- "Profiteur: a visualiser for Haskell GHC .prof files" - *February 25, 2014*
  A nicer way to browse big profile files
- "I do not have a lot of strong opinions" - *January 5, 2014*
  An introspective blogpost on my apparent lack of strong views

Hakyll

```haskell
type AppInfo = PlainRec '[ "proj"  ::: M44 GLfloat ]
type Pos = "vertexPos" ::: V4 GLfloat

pos :: Pos
pos = Field

-- | Uniform grid of vertices with @n*2@ vertices on a side.
grid :: Integral a => a -> [V2 GLfloat]
grid n = map (fmap ((*s) . fromIntegral)) [V2 x y | y <- [-n..n], x <- [-n..n]]
  where s = 1 / fromIntegral n

-- | Indices of an @n@ x @n@ grid in row-major order.
inds :: Word32 -> [Word32]
inds n = take (numQuads*6) $ concat $ iterate (map (n+)) row
  where numQuads = fromIntegral $ (n - 1) * (n - 1)
        row = concatMap (\c -> map (c+) [0,n,1,1,n,n+1]) [0..n - 2]
```



Robotics

Even in our current project!

(which is in Scala, another advanced typed functional language)

So are you saying Haskellers don't write tests?

Nope.

- Tests are important. Not all invariants can be encoded into types. (Not in Haskell anyway.)

- Most of us don't like the "driven" or "first" part. We like tests for our software.

- We believe types and algebras serve as much better design language, and test-first methodology doesn't seem to reconcile well with that.

Testing in Haskell world

- QuickCheck, property based (aka generative) testing

```
*A> deepCheck (\s -> length (take5 s) < 5)
Falsifiable, after 125 tests:
";:iD^*NNi~Y\\RegMob\DEL@krsx/=dcf7kub|EQi\DELD*"
```

- Has been applied to things such as concurrency too!

Testing in Haskell world

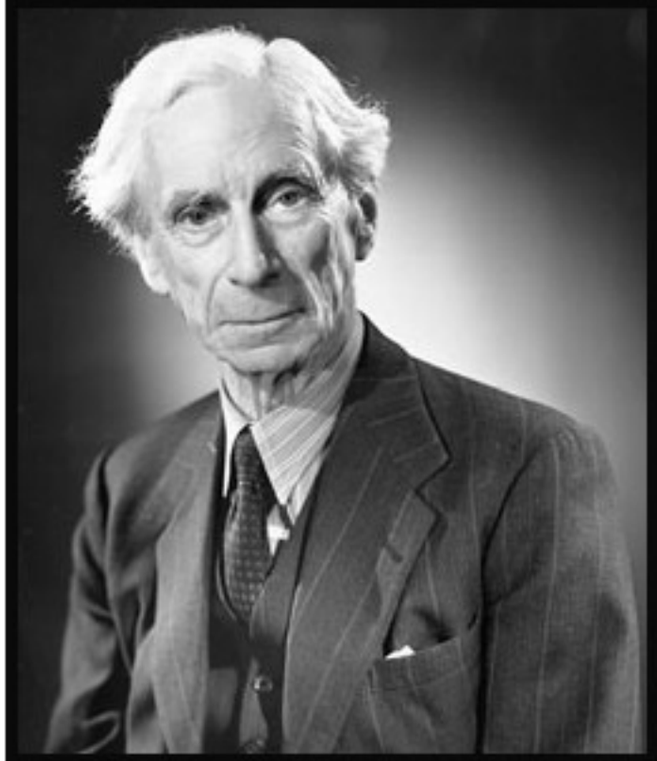
- HUnit

```
test1 = TestCase (assertEqual "for (foo 3)," (1,2) (foo 3))
test2 = TestCase (do (x,y) <- partA 3
                     assertEqual "for the first result of partA," 5 x
                     b <- partB y
                     assertBool ("(partB " ++ show y ++ ") failed") b)
```

Testing in Haskell world

- Some even practice TDD in conjunction

However…



Everything is vague to a degree you do not realize till you have tried to make it precise.

(Bertrand Russell)

izquotes.com

- There are languages with more advanced type systems than Haskell.
  - Dependent types
  - Extensible algebraic effects
  - Row polymorphism
- Better invariants, better program direction.
- According to some of them, types could make tests redundant.

Sorting a list

Haskell:

```
sort :: Ord a ⇒ [a] → [a]
```

Idris (my example, > 150 LOC):

```
quickSort : {a : Type} → {less : a → a → Type}
  → {eq : a → a → Type}
  → TotalOrder less eq
  → (xs : List a)
  → Exists (List a) (\ys ⇒ (IsSorted less ys, Permutation xs ys))
```

Idris!

By the way…

- This style of development can also be used in untyped (aka, dynamically typed) languages.

- Tomas Petricek:

> And I believe this is the case for many other F# programmers. In fact, this use of types is not limited to *statically-typed languages*. I think it applies similarly to other functional langauges, although Scheme or LISP programmers would probably talk about *data structures* instead of *types*.

- Mattias Felleisen:

> Here is an expanded explanation for your readers. As you know from How to Design Programs, (good) DT programmers use something like types to design and organize their programs. The big difference is that there is no explicit language of types as in other explicitly, statically and soundly typed languages such as ML, Haskell (I know about exceptions), all the way to Agda. This is both a curse and a blessing.

Where do we go from here?

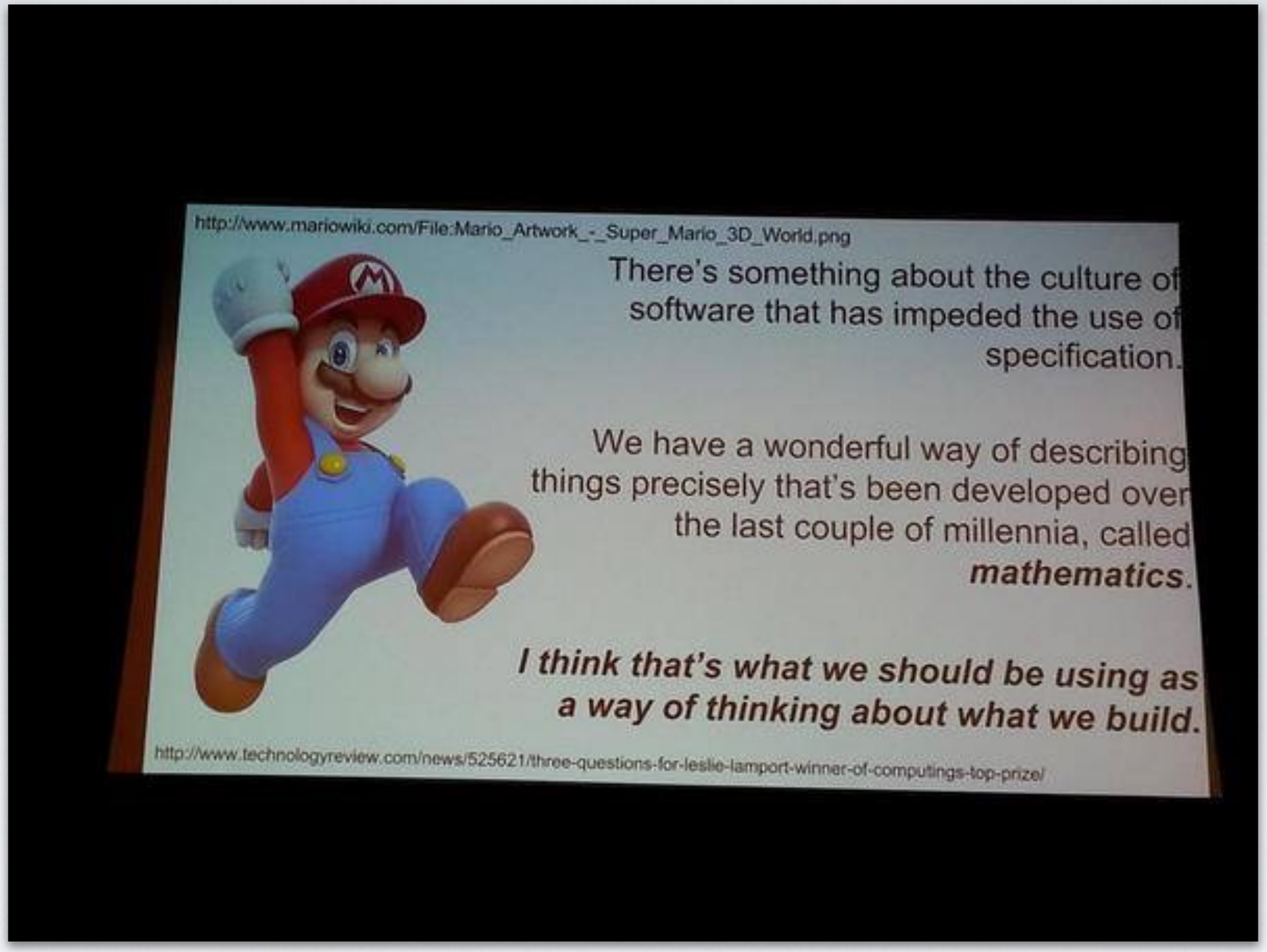
- Learn Haskell, try it on some project.
- Try Scala or F#, easier to adopt for practical reasons.

Where do we go from here?

- Keep an open mind. Keep learning. Read.

Where do we go from here?

- Let's meet up twice a month or so, and keep the momentum going.
- Workshops planned for near future.



Thank You!

Rahul Goma Phulore (@missingfaktor)