F⋆: Dependent Types, Effects and the Future of Functional Programming

By Ahmad Salim Al-Sibahi

IT University of Copenhagen

About Me

This is my GitHub avatar

I am a humble PhD fellow

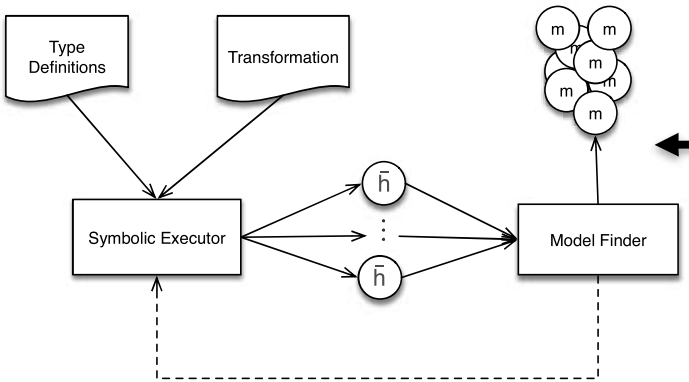
mostly

This is where I work

It's the ~~smallest~~ *best* university in Denmark

IT UNIVERSITY OF COPENHAGEN

About Me



This is what I do

Develop techniques for automated validation and verification of transformations

Another time

Also a bit of dependent types

With some contributions to the Idris programming language

Now a committer too!

So why Types?

Why Types: Documentation For Free

```
div : int → int → int

map : (α → β) → list α → list β
```

Why Types: Great Tool Support

Intelligent completion

Useful documentation

```
append : List a -> List a -> List a
append xs ys = ?append_rhs
```

```
append : List a -> List a -> List a
append [] ys = ?append_rhs_1
append (x :: xs) ys = ?append_rhs_2
```

Automated & correct case splitting

Why Types: Preventing Semantic Errors

I can't stop thinking of Jell-o

- You can't divide 10 by "hello"
  - I mean what would the result be?
- You can't map an integer function over a list of tuples

Why Types: Preventing Semantic Errors

- Seems like something so simple that we could take it for granted
- However, let us try executing the expression `10 / "hello"` in different dynamic languages

Why Types: Preventing Semantic Errors

```
racket> (/ 10 "Hello")

  contract violation
  expected: number?
  given: "Hello"
  argument position: 2nd
  other arguments...:
```

Why Types: Preventing Semantic Errors

```
node> 10 / "hello”
```

NaN

Why Types: Preventing Semantic Errors

```
php> 10 / "hello";

  Warning: Division by
  zero in php shell code
  on line 1
```

PHP: THE POPULAR WEB LANGUAGE YOU SHOULDN'T BE USING

BY AHMAD SALIM AL-SIBAHI

IT UNIVERSITY OF COPENHAGEN

Why Types: Using F#

Yay! A Compile-time error!

```
(* F# file *)
let test_div_string x = x / “hello”

(* F# compiler *)
error FS0001:
The type 'string' does not match the
type 'int'
```

Why Types: Trouble in Paradise

```
(* F# file *)
let test_div x = x / 0

(* F# compiler *)
«No errors»
```

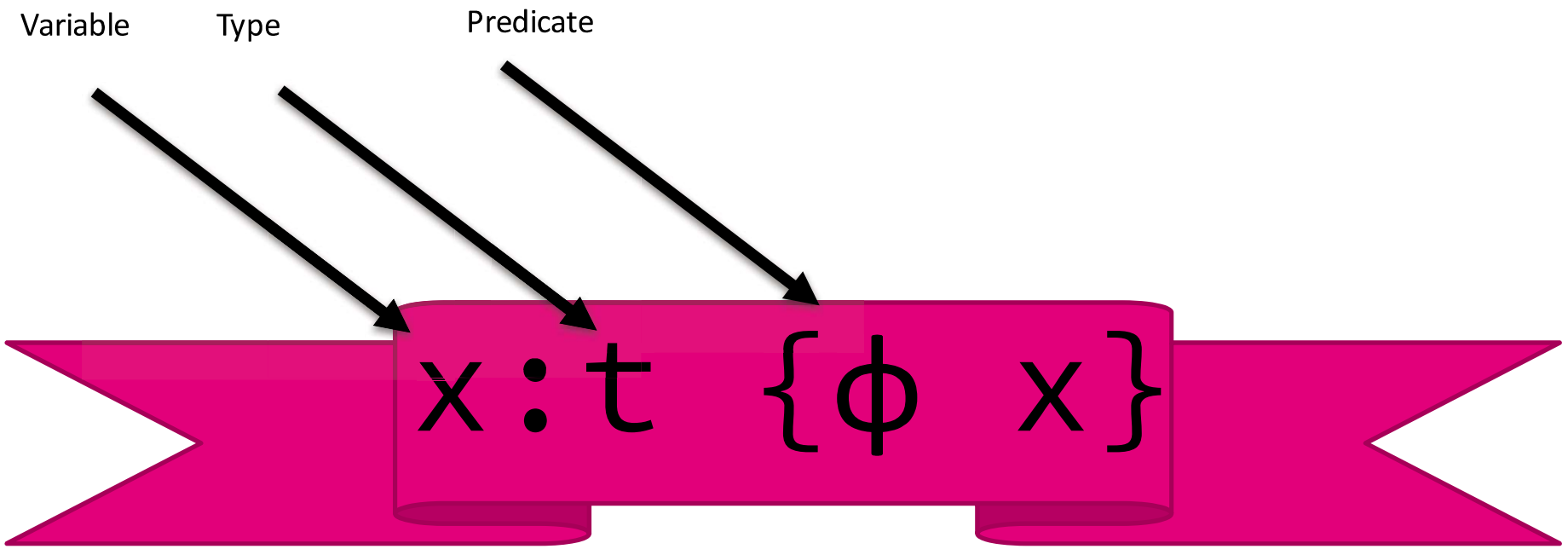
It otherwise went so well ☹

```
(* F# interactive runtime*)
fsharpi> test_div
System.DivideByZeroException: Attempted to
divide by zero… «Long stack trace»
```

We can do better

- `int` is the most precise type F# can give the second argument of `/`
- But, it still doesn't save us from runtime errors
- Standard solution of returning option types is cumbersome to work with in practice, and adds overhead to an otherwise simple operation

The Power of Refinements!



Typing Division in F⋆

```
(/) :(x:int{true})->(y:int{y <> 0})-> Tot int
```

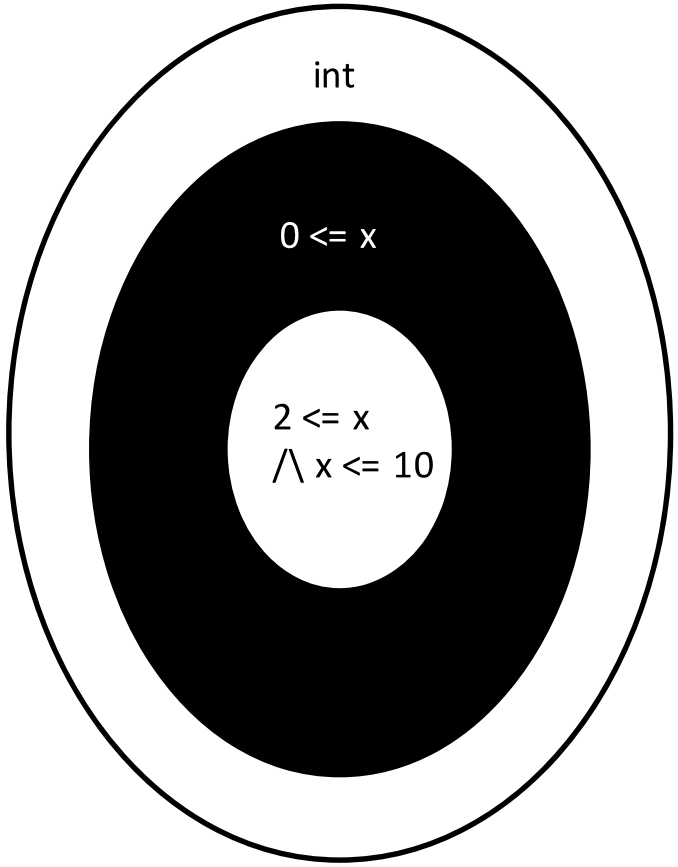
You can always forget the truth

Function is total: i.e. defined for all cases and is known to terminate.

Automatic Subtyping by refinement

```
(x:int {0 <= x}) <: (x: int)

(x:int {2 <= x /\ x <= 10}) <:
(x:int {0 <= x} )
```



I Know What You Subtyped Last Summer

$$x{:}t\ \{\phi\ x\} <: x{:}t\ \{\phi'\ x\}$$

$$\Leftrightarrow$$

$$\phi\ x \supset \phi'\ x$$

LET US TRY AGAIN

Success!

```
(* F* file *)
let test_div x = x / 0

(* Console output *)
```

Subtyping check failed; expected type y:int{(y <> 0)}; got type int

LET US TRY AGAIN

Success!

```
(* F* file *)
let test_div x = x / 1

(* Console output *)
```

All verification conditions discharged successfully

Not Only Constants

```
(* F* file *)
val is_odd : (x: int) -> Tot bool
let is_odd x = abs x % 2 = 1

val oddify: (x: int) -> Tot (x:int{is_odd x})
let oddify x = 2 * x + 1

let test_div1 x y = x / oddify y

(* Console output *)
```

All verification conditions discharged successfully

Note: Refined return type!

Even with stateful functions!

```
(* F* file *)
type nat = x:int{x >= 0}

val bounded_rand: (seed: ref nat)
  -> (lower: nat) -> (upper: nat{upper > lower})
  -> r:nat{lower <= r /\ r < upper}

let test_div3 = x / (bounded_rand (ref 1337) 10 100)

(* Console output *)
```

Type alias

Takes a reference

Note: Not Tot, so can have unlimited side-effects

All verification conditions discharged successfully

Defining Datatypes in F⋆

```
type list 'a =
     | Nil
     | Cons of hd:'a * tl:list 'a
```

Note: Similar to ordinary F#

Note: We can also use `[]` and hd `::` `tl` as syntactic sugar

Déjà Vu

F★ GIVES US SOME ADDITIONAL DEFINITIONS FOR FREE!

Discrimininator (similarly with `is_Nil`)

```
val is_Cons : list 'a -> Tot bool
let is_Cons = function
  | Cons _ -> true
  | Nil -> false
```

Type-Safe accessor(similarly with `Cons.tl`)

```
val Cons.hd : xs:list 'a{is_Cons xs} -> Tot 'a
let Cons.hd = function
  | Cons(x, xs) -> x
```

Isn't that just cool!?

We can defined some ordinary functions

```
val length : xs:list 'a -> Tot nat
let rec length = function
  | [] -> 0
  | _ :: xs -> 1 + length xs
```

But more precise: just eliminated 50% of potential illegal values by using nat instead of int*

** Practically, but not theoretically since ℤ is isomorphic to ℕ, infinites you know…*

We can enforce some interesting relations intrinstically

```
val append : xs:list 'a -> ys:list 'a
    -> Tot (zs:list 'a{length zs = length xs + length ys})
let rec append xs ys = match xs with
  | [] -> ys
  | x :: xs -> x :: (append xs ys)
```

Checked at compile time!

Using append ys ys, gives a compile-time error!

Even between different data types

```
val take : n:nat -> xs:list 'a{length xs >= n}
    -> Tot(zs:list 'a{length zs = n})
let rec take n xs = match n, xs with
  | 0, _ -> []
  | n, x :: xs -> x :: take (n - 1) xs

val drop : n:nat -> xs:list 'a{length xs >= n}
    -> Tot(zs:list 'a{length zs = length xs - n})
let rec drop n xs = match n, xs with
  | 0, xs -> xs
  | n, x :: xs -> drop (n - 1) xs
```

We can even show interesting relations!

Lemmas have no computation meaning, so they return `unit` and are erased at runtime!

```
val take_drop_identity : n:nat -> xs:list 'a{length xs >= n}
     -> Lemma (append (take n xs) (drop n xs) = xs)
let rec take_drop_identity n xs = match n, xs with
  | 0, _ -> ()
  | n, x :: xs -> take_drop_identity (n - 1) xs
```

Proof by induction!

QuickCheck no more!
QuickProve

Beyond Pure Programming

& Evil

IT University of Copenhagen

A Quick Introduction to Monads

- A Monad is an endofunctor M equiped with two natural transformations:
  - $\eta : 1 \rightarrow T$
  - $\mu : T^2 \rightarrow T$
- Essentially these operations make monads monoids in the category of endofunctors
- We also use η and μ in languages like Haskell where they are called `return` and `join` respectively.
- For some reason, some people also like to have a function called (equivalent to $\mu \circ M$)
  - `(>>=) :: m a -> (a -> m b) -> m b`

A Quick Introduction to Monads

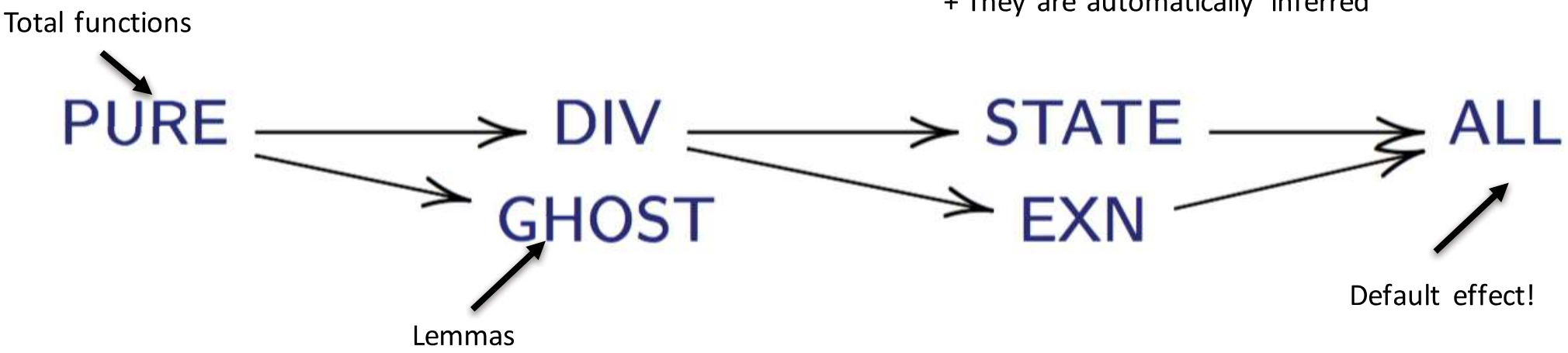
- A Monad is an endofunctor M equiped with two natural transformations:
  - $\eta : 1 \rightarrow T$
  - $\mu : T^2 \rightarrow T$
- Essentially these operations make monads monoids in the category of endofunctors
- We also use η and μ in languages like Haskell where they are called `return` and `join` respectively.
- For some reason, some people also like to have a function called (equivalent to $\mu \circ M$)
  - `(>>=) :: m a -> (a -> m b) -> m b`

Enough with the Category Theory!

Give Me Some Effects!

Allows us to write code that is easy to reason about without having to know category theory!

+ They are automatically inferred



An impure implementation of random number generation

```
val rand: (seed: ref nat)
  -> ST nat (requires (fun h -> true))
            (ensures (fun h x h' -> modifies !{seed} h h'))
let rand seed =
  let m = 2147483647 in
  let a = 16807 in  let c = 0 in
  seed := ((a * !seed + c) % m);
  !seed
```

Working with exceptions

```
exception DivisionByZero
val unsafeDiv : (x: int) -> (y: int) ->
  Exn int (requires true)
          (ensures (fun r -> if y = 0 then r = E DivisionByZero
                             else r = V (x / y)))
let unsafeDiv x y =
  if y = 0 then raise DivisionByZero
  else x / y
```

Can be caught as well

We are falling a bit short on time, but F⋆ can so much more!

- F#/OCaml interaction
- Build your own effects
- Hyper-heaps
- Totality proof metrics
- SMT activation patterns
- GADTs
- Rely-Guarantee reasoning via modules