

# Idiomatic Inference for DSLs

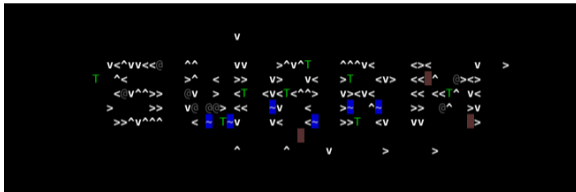
Brent Yorgey

Portland State PLV Seminar  
30 January 2025



# Motivation

# Swarm



<https://swarm-game.github.io/>

# World generation DSL

```
let
  pn0 = perlin seed 6 0.05 0.6,
  pn1 = perlin (seed + 1) 6 0.05 0.6,
  ...
in
overlay
[ mask (big && hard && artificial)
  (if (cl > 0.85) then {stone, copper ore} else {stone})
, mask (big && hard && natural)
  ( overlay
    [ {grass}
      , mask (cl > 0.0)
        (if (hash % 30 == 1)
          then {dirt, LaTeX}
          else {dirt, tree})
      , mask (hash % 30 == 0) {stone, boulder}
      , mask (cl > 0.5) {stone, mountain}
    ]
  )
, ...
]
```



# World generation DSL

*grass*

```
base 0 0 0
-----
Cependium
A atomic vector plotter 0
B big furnace 0
M big motor 0
b bit (0) 0
l bit (1) 0
board 0
boulder 0
box 0
branch 0

Burned-out remnants of
combustion.

0 0
-----
16 ticks / s
[C-p]pause [C-x/C-z]speed [M-]show REPL [M-h]hide robots Creative mode
[.]history [Enter]execute [M-p]pilot [PgUp/Pn]scroll
```

# World generation DSL

**if** ( $x > 3$ ) **then** *water* **else** *grass*

```
base 0 0 0 [F1]help [F2]Robots [F4]commands 0:00:47
Compendium
x00 0
A atomic vector plotter 0
b big furnace 0
M big motor 0
B bit (0) 0
l bit (1) 0
board 0
boulder 0
box 0
branch 0
Burned-out remnants of
combustion.
0:0 16 ticks / s -
[C-p]pause [C-x/C-z]speed [M-]show REPL [M-]hide robots Creative mode
[.]history [Enter]execute [M-p]pilot [PgUp/PgDn]scroll
```

# Types?

- $W a = (Int, Int) \rightarrow a$ .

# Types?

- $W a = (Int, Int) \rightarrow a$ .
- A top-level expression describing a world has type  $W Cell$ .

# Types?

- $W a = (Int, Int) \rightarrow a$ .
- A top-level expression describing a world has type  $W Cell$ .
- $grass, water : Cell$

# Types?

- $W\ a = (Int, Int) \rightarrow a$ .
- A top-level expression describing a world has type  $W\ Cell$ .
- $grass, water : Cell$
- **if** :  $Bool \rightarrow a \rightarrow a \rightarrow a$

# Types?

- $W a = (Int, Int) \rightarrow a$ .
- A top-level expression describing a world has type  $W Cell$ .
- $grass, water : Cell$
- **if** :  $Bool \rightarrow a \rightarrow a \rightarrow a$
- $x, y : W Int$

# Types?

- $W a = (Int, Int) \rightarrow a$ .
- A top-level expression describing a world has type  $W Cell$ .
- $grass, water : Cell$
- **if** :  $Bool \rightarrow a \rightarrow a \rightarrow a$
- $x, y : W Int$
- **if**  $(x > 3)$  **then**  $water$  **else**  $grass : W Cell ?$

## Another example

**if** *uniform* (0, 1) > 0.2 **then** "hello" **else** "world" : *Distribution String*

# FUNCTIONAL PEARL

## Applicative programming with effects

CUNOR MCBIRRIE

University of Nottingham

BRUNO PATRICKSON

City University, London

---

### Abstract

In this paper, we introduce *Applicative functors*—an abstract characterization of an application of effectful (side-effect) programming, weaker than *Monads* and hence more widespread. Indeed, it is the simplicity of this programming pattern that drew us to the abstraction. We extend our study in this paper, introducing the applicative pattern by diverse means: plots, monads, arrows,  $\lambda$ -calculus, the applicative type class and introducing a lambda calculus which interprets the normal applicative system in the terms of an Applicative Functor. Finally, we discuss the properties of applicative functors and the generic operators they support. We close by identifying the categorical structure of applicative functors and examining their relationship both with *Monads* and with *Arrows*.

---

### 1 Introduction

This is the story of a pattern that proved up there and again in our daily work, programming in Haskell (Freyton Jones, 2002), until our imagination to abstract it became irresistible. Let us illustrate with some examples.

Representing commands Our often wants to create a sequence of commands and collect the outputs of their execution, and indeed there is such a function in the Haskell Prelude (here specialized to IO):

```
sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (c : cs) = do
  v <- c
  as <- sequence cs
  return (v : as)
```

In the  $(c : cs)$  case, we collect the values of some effectful computations, which we then use as the arguments to a pure function  $f$ . We could avoid the need for names to name these values through to their point of usage if we had a kind of “abstract applicative”. Fortunately exactly such a thing lives in the standard Haskell library:

# Applicative Functors

## Definition

An **applicative functor** is some  $\square : \text{Type} \rightarrow \text{Type}$  together with operations

$$\begin{aligned} \text{pure} &: a \rightarrow \square a \\ \text{ap} &: \square (a \rightarrow b) \rightarrow \square a \rightarrow \square b \end{aligned}$$

(satisfying certain laws).

# Convenience

For convenience, define

- $\langle * \rangle = ap$
- $\langle \$ \rangle = fmap = ap \circ pure : (a \rightarrow b) \rightarrow \square a \rightarrow \square b$

So **e.g.**  $ap (ap (pure (>)) x) (pure 3)$  can be written

$$\langle > \rangle \langle \$ \rangle x \langle * \rangle pure 3$$

## Example: *Maybe*

$$ap = \lambda \mathbf{cases} \left\{ \begin{array}{l} \text{pure} = \text{Just} \\ (\text{Just } f) (\text{Just } x) \rightarrow \text{Just } (f \ x); \_ \_ \rightarrow \text{Nothing} \end{array} \right\}$$

## Example: $W$

```
type  $W\ a = \text{Coords} \rightarrow a$   
 $\text{pure} : a \rightarrow W\ a = \text{const}$   
 $\text{ap} : W\ (a \rightarrow b) \rightarrow W\ a \rightarrow W\ b = \lambda f\ x\ c \rightarrow f\ c\ (x\ c)$ 
```

## Example: $W$

**if** ( $x > 3$ ) **then** *water* **else** *grass* :  $W$  Cell

↓

$ap (ap (pure \mathbf{if} ) (ap (ap (pure (>)) x) (pure 3)) (pure water)) (pure grass)$

=

**if**  $\langle \$ \rangle ((>) \langle \$ \rangle x \langle * \rangle pure\ 3) \langle * \rangle pure\ water \langle * \rangle pure\ grass$

## Question

How and when can we automatically insert *pure* and *ap*?

## Question

How and when can we automatically insert *pure* and *ap*?

Implemented in Swarm world DSL... but is it correct? And how does it generalize?

\*54.43.  $\vdash : \alpha, \beta \in 1 . \supset : \alpha \cap \beta = \Lambda . \equiv . \alpha \cup \beta \in 2$

*Dem.*

$\vdash . *54.26 . \supset \vdash : \alpha = t'x . \beta = t'y . \supset : \alpha \cup \beta \in 2 . \equiv . x \neq y .$

[\*51.231]  $\equiv . t'x \cap t'y = \Lambda .$

[\*13.12]  $\equiv . \alpha \cap \beta = \Lambda \quad (1)$

$\vdash . (1) . *11.11.35 . \supset$

$\vdash : (\exists x, y) . \alpha = t'x . \beta = t'y . \supset : \alpha \cup \beta \in 2 . \equiv . \alpha \cap \beta = \Lambda \quad (2)$

$\vdash . (2) . *11.54 . *52.1 . \supset \vdash . \text{Prop}$

From this proposition it will follow, when arithmetical addition has been defined, that  $1 + 1 = 2$ .

# Formalization

$term, t$	$::=$		term
		$x$	variable
		$\lambda x.t$	abstraction
		$t t'$	application
		$\mathbf{c}$	constant
		pure	pure
		ap	ap
$type, \sigma, \tau$	$::=$		type
		$B$	base type
		$\tau_1 \rightarrow \tau_2$	function type
		$\square \tau$	box type

$\boxed{\Gamma \vdash t : \tau}$  $t$  has type  $\tau$  in context  $\Gamma$ 

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{TY\_VAR}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x. t : \sigma \rightarrow \tau} \text{TY\_LAM}$$

$$\frac{\Gamma \vdash t_1 : \sigma \rightarrow \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 t_2 : \tau} \text{TY\_APP}$$

$$\frac{\mathbf{c} : \tau \in \Sigma}{\Gamma \vdash \mathbf{c} : \tau} \text{TY\_CONST}$$

$\boxed{\Gamma \vdash t : \tau}$   $t$  has type  $\tau$  in context  $\Gamma$

$$\frac{}{\Gamma \vdash \text{pure} : \sigma \rightarrow \Box \sigma} \text{TY\_PURE}$$

$$\frac{}{\Gamma \vdash \text{ap} : \Box(\sigma \rightarrow \tau) \rightarrow (\Box \sigma \rightarrow \Box \tau)} \text{TY\_AP}$$

So far, *pure* and *ap* must be explicitly included. . . how to model the possibility to infer them?

So far, *pure* and *ap* must be explicitly included. . . how to model the possibility to infer them?

Subtyping!

$\tau_1 <: \tau_2$  $\tau_1$  is a subtype of  $\tau_2$ 
$$\frac{}{\tau <: \tau} \text{ SUB\_REFL}$$
$$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \text{ SUB\_TRANS}$$
$$\frac{\tau_1 <: \sigma_1 \quad \sigma_2 <: \tau_2}{\sigma_1 \rightarrow \sigma_2 <: \tau_1 \rightarrow \tau_2} \text{ SUB\_ARR}$$

$\tau_1 <: \tau_2$  $\tau_1$  is a subtype of  $\tau_2$ 
$$\frac{}{\tau <: \Box \tau} \text{SUB\_PURE}$$
$$\frac{}{\Box(\sigma \rightarrow \tau) <: \Box \sigma \rightarrow \Box \tau} \text{SUB\_AP}$$

$\tau_1 <: \tau_2$  $\tau_1$  is a subtype of  $\tau_2$ 

$$\frac{\sigma <: \tau}{\Box \sigma <: \Box \tau} \text{ SUB\_BOX}$$

$\boxed{\Gamma \vdash_{<} t : \tau}$ 

$t$  has type  $\tau$  in context  $\Gamma$  with subtyping

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash_{<} t : \tau} \text{ TYS\_EMB}$$

$$\frac{\Gamma \vdash_{<} t : \sigma \quad \sigma <: \tau}{\Gamma \vdash_{<} t : \tau} \text{ TYS\_SUB}$$

# Elaboration

- $\sigma <: \tau$  elaborates to an explicit coercion term with type  $\sigma \rightarrow \tau$
- $\Gamma \vdash_{<} t : \tau$  elaborates to  $\Gamma \vdash t' : \tau$ , where all uses of subtyping have been replaced by an application of an elaborated coercion term

Question: is  $\sigma <: \tau$  decidable?



# Applicative Subtyping

## Transitivity is annoying

$$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \text{ SUB\_TRANS}$$

# Transitivity-free subtyping

$\tau_1 \triangleleft \tau_2$   $\tau_1$  is a subtype of  $\tau_2$

$$\frac{}{\tau \triangleleft \tau} \text{ SUBT\_REFL}$$

$$\frac{\tau_1 \triangleleft \sigma_1 \quad \sigma_2 \triangleleft \tau_2}{\sigma_1 \rightarrow \sigma_2 \triangleleft \tau_1 \rightarrow \tau_2} \text{ SUBT\_ARR}$$

# Transitivity-free subtyping

$\tau_1 \triangleleft \tau_2$   $\tau_1$  is a subtype of  $\tau_2$

$$\frac{\sigma \triangleleft \tau}{\Box \sigma \triangleleft \Box \tau} \text{ SUBT\_BOX}$$

$$\frac{\sigma \triangleleft \tau}{\sigma \triangleleft \Box \tau} \text{ SUBT\_PURE}$$

$$\frac{\sigma \triangleleft \sigma_1 \rightarrow \sigma_2 \quad \Box \sigma_1 \rightarrow \Box \sigma_2 \triangleleft \tau}{\Box \sigma \triangleleft \tau} \text{ SUBT\_AP}$$

# Transitivity-free subtyping

$\tau_1 \triangleleft \tau_2$   $\tau_1$  is a subtype of  $\tau_2$

$$\frac{\sigma \triangleleft \tau}{\Box \sigma \triangleleft \Box \tau} \text{ SUBT\_BOX}$$

$$\frac{\sigma \triangleleft \tau}{\sigma \triangleleft \Box \tau} \text{ SUBT\_PURE}$$

$$\frac{\sigma \triangleleft \sigma_1 \rightarrow \sigma_2 \quad \Box \sigma_1 \rightarrow \Box \sigma_2 \triangleleft \tau}{\Box \sigma \triangleleft \tau} \text{ SUBT\_AP}$$

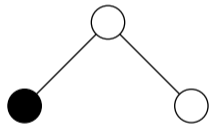
$$\frac{\sigma \triangleleft \sigma_1 \rightarrow \sigma_2 \quad \Box \sigma_1 \rightarrow \Box \sigma_2 \triangleleft \tau}{\sigma \triangleleft \tau} \text{ SUBT\_PUREAP}$$

## Transitivity-free subtyping

$(\sigma <: \tau) \iff (\sigma \triangleleft \tau)$ : Proved in Agda! ✓

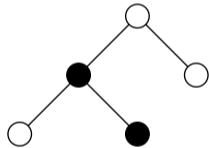
# Tree model

$B \rightarrow B$



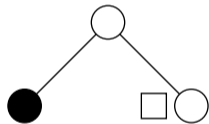
# Tree model

$(B \rightarrow B) \rightarrow B$



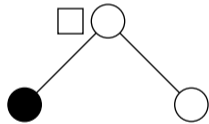
# Tree model

$$B \rightarrow \square B$$

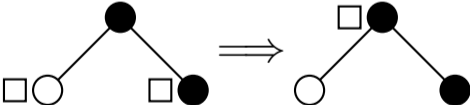
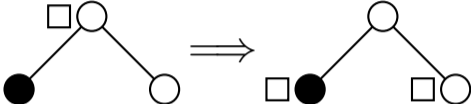


# Tree model

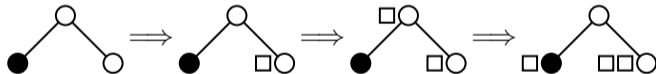
$\Box (B \rightarrow B)$



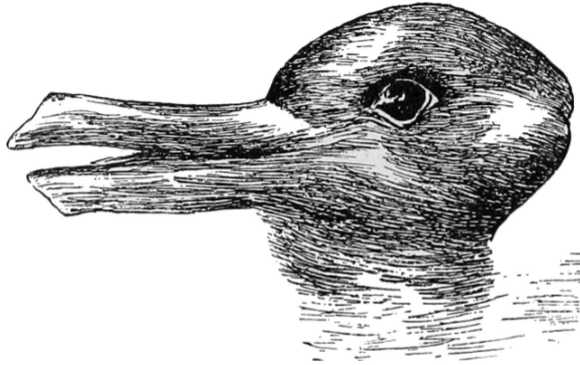
# Subtyping rules



# Subtyping







# Ambiguity and Coherence

# Coherence

## Theorem

*Any two derivations of  $\Gamma \vdash_{<} t : \tau$  elaborate to terms that are equivalent up to  $\beta$ ,  $\eta$ , and the applicative laws.*

# Coherence

## Not A Theorem ☹

*Any two derivations of  $\Gamma \vdash_{<} t : \tau$  elaborate to terms that are equivalent up to  $\beta$ ,  $\eta$ , and the applicative laws.*

# Counterexample

$$\lambda x \rightarrow x : \Box B \rightarrow \Box \Box B$$

$B \rightarrow B$   
 $< : B \rightarrow \Box B$   
 $< : \Box (B \rightarrow \Box B)$   
 $< : \Box B \rightarrow \Box \Box B$   
 $\rightsquigarrow \text{fmap pure}$

$B \rightarrow B$   
 $< : \Box (B \rightarrow B)$   
 $< : \Box \Box (B \rightarrow B)$   
 $< : \Box (\Box B \rightarrow \Box B)$   
 $< : \Box \Box B \rightarrow \Box \Box B$   
 $< : \Box B \rightarrow \Box \Box B$   
 $\rightsquigarrow \text{pure}$

## No nesting?

Intuitively, the problem has to do with nested boxes, like  $\square \square B$ .

But we often don't need or want that.

Call a type “boxbox-free” if it has no immediately nested boxes.

### Conjecture (Coherence)

*Any two derivations of  $\Gamma \vdash_{<} t : \tau$  elaborate to terms that are equivalent up to  $\beta$ ,  $\eta$ , and the applicative laws, as long as all embedded subtyping judgments are boxbox-free.*

## Conjecture

*If  $\tau$  and all the types in  $\Gamma$  are boxbox-free, then so are all the types that show up anywhere in a derivation  $\Gamma \vdash_{<} t : \tau$ .*

## Conjecture

*If  $\tau$  and all the types in  $\Gamma$  are boxbox-free, then so are all the types that show up anywhere in a derivation  $\Gamma \vdash_{<} t : \tau$ .*

In particular, if  $\Gamma = \emptyset$  and the final type we want has a single box, then there can never be any nested boxes anywhere.

## Conjecture

*If  $\tau$  and all the types in  $\Gamma$  are box-free, then so are all the types that show up anywhere in a derivation  $\Gamma \vdash_{<} t : \tau$ .*

In particular, if  $\Gamma = \emptyset$  and the final type we want has a single box, then there can never be any nested boxes anywhere.

Definitely not true for monads!  $join : \square \square a \rightarrow \square a$  lets us hide nested boxes anywhere.



# Future Work

# Future Work

- Prove conjectures / find the right things to prove
- Explore how to write effective + efficient inference procedures
- Incorporate product and sum types
- Extend to selective functors