

PSBP

Program Specification Based Programming  
ITP 2025 Lean Workshop

Luc Duponcheel  
Programmer and Cyclist

September 30, 2025



Ceci n'est pas une pipe



# Programs and Computations



# Programs and Computations

- the words *program* and *computation* have a *specific technical meaning* in this talk



# Programs and Computations

- the words *program* and *computation* have a *specific technical meaning* in this talk
- a *program* is a, potentially *effectful*, *function* abstraction



# Programs and Computations

- the words *program* and *computation* have a *specific technical meaning* in this talk
- a *program* is a, potentially *effectful*, *function* abstraction
- a *computation* is a, potentially *effectful*, *expression* abstraction



# Open Components and Closed Components



## Open Components and Closed Components

- programs are *closed components*

$(apb \Rightarrow bpc) \Rightarrow cpd =$

$apb \Rightarrow (bpc \Rightarrow cpd)$

(*associativity* law)





## Open Components and Closed Components

- programs are *closed components*

```
(apb ==> bpc) ==> cpd =  
  apb ==> (bpc ==> cpd)
```

(*associativity* law)

- computations are *open components*

```
ca >>= afcb >>= bfcc =  
  ca >>= fun a => afcb a >>= bfcc
```

(*associativity* law)



## Complexity and Difficulty



## Complexity and Difficulty

- programs are *less complex* than computations



## Complexity and Difficulty

- programs are *less complex* than computations
- too much complexity becomes difficult for human beings



## Complexity and Difficulty

- programs are *less complex* than computations
- too much complexity becomes difficult for human beings
- many times being confronted with complexity is not fun



## Complexity and Difficulty

- programs are *less complex* than computations
- too much complexity becomes difficult for human beings
- many times being confronted with complexity is not fun
- programs are, perhaps, *more difficult* than computations



## Complexity and Difficulty

- programs are *less complex* than computations
- too much complexity becomes difficult for human beings
- many times being confronted with complexity is not fun
- programs are, perhaps, *more difficult* than computations
- once and for all understanding difficulty is fun



# Denotational versus Operational





# Denotational versus Operational

- programs are *denotational* artifacts



# Denotational versus Operational

- programs are *denotational* artifacts
- computations are *operational* artifacts



## Denotational versus Operational

- programs are *denotational* artifacts
- computations are *operational* artifacts
- thinking denotationally is more natural for human beings



PSBP



# PSBP

- is a *program specification based programming library*



# PSBP

- is a *program specification based programming library*
- can be seen as a *domain specific language* for the *theory of programming* domain



# PSBP

- is a *program specification based programming library*
- can be seen as a *domain specific language* for the *theory of programming* domain
- can be seen as a *programming course* for **Lean** itself



# PSBP

- is a *program specification based programming library*
- can be seen as a *domain specific language* for the *theory of programming* domain
- can be seen as a *programming course* for **Lean** itself
- comes with *documentation* that can be seen as a course for students interested in the theory of programming





# PSBP



# PSBP

- PSBP supports



# PSBP

- PSBP supports
  - *pointfree* programming



# PSBP

- PSBP supports
  - *pointfree* programming
  - *positional* programming



# PSBP

- PSBP supports
  - *pointfree* programming
  - *positional* programming
  - *effectful* programming



# PSBP



# PSBP

- PSBP is work in progress



# PSBP

- PSBP is work in progress
  - all contributions are welcome





## fibonacci

```
unsafe def fibonacci
  [Functional program]
  [Sequential program]
  [Creational program]
  [Conditional program] :
program Nat Nat :=
  if_ isZero one $
    if_ isOne one $
      (minusTwo ==> fibonacci) &&&
      (minusOne ==> fibonacci) ==>
      add
```



## factorial

```
unsafe def factorial
  [Functional program]
  [Sequential program]
  [Creational program]
  [Conditional program] :
program Nat Nat :=
  if_ isZero one $

    (identity) &&&
    (minusOne ==> factorial) ==>
    multiply
```



## factorial

```
unsafe def factorial
  [Functional program]
  [Sequential program]
  [Creational program]
  [Conditional program] :
program Nat Nat :=
  if_ isZero one $

    let_ (minusOne >=> factorial) $

      multiply
```



# Programs versus Program Specifications



## Programs versus Program Specifications

- `fibonacci` and `factorial` are not *programs*



## Programs versus Program Specifications

- `fibonacci` and `factorial` are not *programs*
- `fibonacci` and `factorial` are *program specifications*



## Programs versus Program Specifications

- `fibonacci` and `factorial` are not *programs*
- `fibonacci` and `factorial` are *program specifications*
- compare this with the painting that is a *pipe description*  
think of a specification as a (special kind of) description



## Programs versus Program Specifications

- `fibonacci` and `factorial` are not *programs*
- `fibonacci` and `factorial` are *program specifications*
- compare this with the painting that is a *pipe description*  
think of a specification as a (special kind of) description
- by abuse of language  
*program* is used instead of *program specification*





# Abbreviation

```
abbrev function a b := a → b
```



# Functional

```
class Functional
  (program : Type → Type → Type) where
  asProgram {a b : Type} :
    function a b → program a b
```



# Functional

```
class Functional
  (program : Type → Type → Type) where
asProgram {a b : Type} :
  function a b → program a b
```

- a *function* (from a to b) can be used as an *effectfree program* (from a to b)



# Functional

```
class Functional
  (program : Type → Type → Type) where
  asProgram {a b : Type} :
    function a b → program a b
```

- a *function* (from a to b) can be used as an *effectfree program* (from a to b)
- by abuse of language (from a to b) is used instead of (from *type* a to *type* b)



## Sequential

```
class Sequential
  (program : Type → Type → Type) where
  andThenProgram {a b c : Type} :
    program a b → program b c → program a c

infixl:50 " >=> " => andThenProgram
```



## Sequential

```
class Sequential
  (program : Type → Type → Type) where
  andThenProgram {a b c : Type} :
    program a b → program b c → program a c
```

```
infixl:50 " >=> " => andThenProgram
```

- a *program* (from a to b) and a *program* (from b to c)  
can be *sequentially combined*  
obtaining a *program* (from a to c)



# Functorial

```
class Functorial
  (program : Type → Type → Type) where
  functionAction {a b c : Type} :
    function b c → (program a b → program a c)
```



# Functorial

```
class Functorial
```

```
  (program : Type → Type → Type) where  
  functionAction {a b c : Type} :  
    function b c → (program a b → program a c)
```

- a *function* (from b to c) can *act*, *effectfree*,  
upon a *program* (from a to b)  
obtaining a *program* (from a to c)





## Creational

```
class Creational
  (program : Type → Type → Type) where
  sequentialProduct {a b c : Type} :
    program a b → program a c → program a (b x c)

infixl:60 " &&& " => sequentialProduct
```



## Creational

```
class Creational
  (program : Type → Type → Type) where
  sequentialProduct {a b c : Type} :
    program a b → program a c → program a (b x c)

infixl:60 " &&& " => sequentialProduct
```

- a *program* (from a to b) and a *program* (from a to c)  
can be *sequentially combined*  
obtaining program (from a to b x c)  
transforming *to* a final *product value*



## Conditional

```
class Conditional
  (program : Type → Type → Type) where
  sum {a b c : Type} :
    program c a → program b a → program (c + b) a

infixl:55 " ||| " => sum
```



## Conditional

```
class Conditional
  (program : Type → Type → Type) where
  sum {a b c : Type} :
    program c a → program b a → program (c + b) a

infixl:55 " ||| " => sum
```

- a *program* (from c to a) and a *program* (from b to a) can be combined obtaining a *program* (from c + b to a) transforming *from* an initial *sum value*



## Parallel

```
class Parallel (program : Type → Type → Type) where
  bothPar {a b c d : Type} :
    program a c → program b d → program (a x b) (c x d)

infixl:60 " |&| " => bothPar
```



## Parallel

```
class Parallel (program : Type → Type → Type) where
  bothPar {a b c d : Type} :
    program a c → program b d → program (a x b) (c x d)

infixl:60 " |&| " => bothPar
```

- programs can be combined in *parallel*



## parallelFibonacci

```
unsafe def fibonacci
  [Functional program]
  [Sequential program]
  [Creational program]
  [Conditional program] :
program Nat Nat :=
  if_ isZero one $
    if_ isOne one $
      (minusTwo ==> fibonacci) &|&
      (minusOne ==> fibonacci) ==>
      add
```



# Implementations





# Implementations

- *implementations* of *program specifications* are *computation valued function* based



# Implementations

- *implementations* of *program specifications* are *computation valued function* based
- *definitions* of *functions* are *expression* based



# Implementations

- *implementations* of *program specifications* are *computation valued function* based
- *definitions* of *functions* are *expression* based
- Sync



# Implementations

- *implementations* of *program specifications* are *computation valued function* based
- *definitions* of *functions* are *expression* based
- Sync
- Async



# Implementations

- *implementations* of *program specifications* are *computation valued function* based
- *definitions* of *functions* are *expression* based
- Sync
- Async
- ...



# Reactive Implementations



## Reactive Implementations

- abbrev ReactiveT  
    (r : Type)  
    (computation: Type → Type)  
    (a : Type) :=  
    (a → computation r) → computation r



# Parallel Implementations





# Parallel Implementations

- *async Task*



## Parallel Implementations

- *async* Task
- *actor* ???



# Parallel Implementations

- *async* Task
- *actor* ???
- ...



## WithState

```
class WithState
  (s : outParam Type)
  (program : Type → Type → Type) where
  readState {a : Type} : program a s
  writeState : program s Unit
```



## WithState

```
class WithState
  (s : outParam Type)
  (program : Type → Type → Type) where
  readState {a : Type} : program a s
  writeState : program s Unit
```

- Programs can handle *state*.



## WithState Implementations



## WithState Implementations

- `StateT`  
(from `Lean` standard library)



## WithFailure

```
class WithFailure
  (e : outParam Type)
  (program : Type → Type → Type) where
  failWith {a b : Type} : function a e → program a b
```





## WithFailure

```
class WithFailure
  (e : outParam Type)
  (program : Type → Type → Type) where
  failWith {a b : Type} : function a e → program a b
```

- programs can handle *failure*



## WithFailure Implementations



## WithFailure Implementations

- ```
def FailureT
  (e : Type)
  (computation : Type → Type)
  (b : Type) : Type :=
  computation (e + b)
```



## WithFailure Implementations

- `def FailureT`  
    `(e : Type)`  
    `(computation : Type → Type)`  
    `(b : Type) : Type :=`  
    `computation (e + b)`
- `Monad`  
  for *first failure*



## WithFailure Implementations

- `def FailureT`  
    `(e : Type)`  
    `(computation : Type → Type)`  
    `(b : Type) : Type :=`  
    `computation (e + b)`
- `Monad`  
  for *first failure*
- `Applicative` and `Monoid`  
  for *accumulating failure*



## GitHub repository



## GitHub repository

- [\*https://github.com/LucDuponcheelAtGitHub/PSBP\*](https://github.com/LucDuponcheelAtGitHub/PSBP)



## GitHub repository

- [\*https://github.com/LucDuponcheelAtGitHub/PSBP\*](https://github.com/LucDuponcheelAtGitHub/PSBP)
- Program Specification Bases Programming in Lean





## More GitHub repositories



## More GitHub repositories

- [\*https://github.com/LucDuponcheelAtGitHub/yoneda\*](https://github.com/LucDuponcheelAtGitHub/yoneda)



## More GitHub repositories

- [\*https://github.com/LucDuponcheelAtGitHub/yoneda\*](https://github.com/LucDuponcheelAtGitHub/yoneda)
- Pointfree Yoneda Lemma for Endofunctors of Functional Categories



## More GitHub repositories

- [\*https://github.com/LucDuponcheelAtGitHub/yoneda\*](https://github.com/LucDuponcheelAtGitHub/yoneda)
- Pointfree Yoneda Lemma for Endofunctors of Functional Categories
- [\*https://github.com/LucDuponcheelAtGitHub/timeHybrids\*](https://github.com/LucDuponcheelAtGitHub/timeHybrids)



## More GitHub repositories

- [\*https://github.com/LucDuponcheelAtGitHub/yoneda\*](https://github.com/LucDuponcheelAtGitHub/yoneda)
- Pointfree Yoneda Lemma for  
Endofunctors of Functional Categories
- [\*https://github.com/LucDuponcheelAtGitHub/timeHybrids\*](https://github.com/LucDuponcheelAtGitHub/timeHybrids)
- Time Hybrids  
Unifying Framework for a Theory of Reality



## More GitHub repositories

- [\*https://github.com/LucDuponcheelAtGitHub/yoneda\*](https://github.com/LucDuponcheelAtGitHub/yoneda)
- Pointfree Yoneda Lemma for Endofunctors of Functional Categories
- [\*https://github.com/LucDuponcheelAtGitHub/timeHybrids\*](https://github.com/LucDuponcheelAtGitHub/timeHybrids)
- Time Hybrids  
Unifying Framework for a Theory of Reality
- all collaboration to convert to **MathLib**-like **Lean** is welcome



Thanks for attending

*[luc.duponcheel@gmail.com](mailto:luc.duponcheel@gmail.com)*

