Adventures in Program Analysis

David Darais
University of Maryland

Adventures in Program Analysis

David Darais
University of Maryland
[Utah undergrad]

Let's Design an Analysis

Let's Design an Analysis

(in the paradigm of abstract interpretation)

Let's Design an Analysis

Program

```
0: int x y; // global state
1: void safe_fun(int N) {
2:   if (N≠0) {x := 0;}
3:   else       {x := 1;}
4:   if (N≠0) {y := 100/N;}
5:   else       {y := 100/x;}}
```

Let's Design an Analysis

Program

```
0: int x y; // global stat
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else
4:  if (N≠0)
5:  else
```

Analysis Property

x/0

Let's Design an Analysis

Program

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else
4:  if (N≠0)
5:  else
```

Analysis Property

$x \neq 0$

Abstract Values

$$\mathbb{Z} \sqsubseteq \{-, 0, +\}$$

Let's Design an Analysis

Program | **Analysis Property** | **Abstract Values**

Implement

```
analyze : exp → results
analyze(x := æ) :=
     .. x .. æ ..
analyze(IF(æ){e1}{e2}) :=
     .. æ .. e1 .. e2 ..
```

Let's Design an Analysis

Program | **Analysis Property** | **Abstract Values**

Get Results

```
N ∈ {-,0,+}
x ∈ {0,+}
y ∈ {-,0,+}

UNSAFE: {100/N}
UNSAFE: {100/x}
```

Let's Design an Analysis

Prove Correct

$$[\![e]\!] \in [\![analyze(e)]\!]$$

Let's Design an Analysis

Program

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

Analysis Property

$x/0$

Abstract Values

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

```
analyze : exp → results
analyze(x := æ) :=
    .. x .. æ ..
analyze(IF(æ){e1}{e2}) :=
    .. æ .. e1 .. e2 ..
```

Get Results

$N \in \{-,0,+\}$
$x \in \{0,+\}$
$y \in \{-,0,+\}$

UNSAFE: {100/N}
UNSAFE: {100/x}

Prove Correct

$[\![e]\!] \in [\![analyze(e)]\!]$

Let's Design an Analysis

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

Flow-insensitive

```
N ∈ {-,0,+}
x ∈ {0,+}
y ∈ {-,0,+}

UNSAFE: {100/N}
UNSAFE: {100/x}
```

```
results :=
var ↦ ℘({-,0,+})
```

Let's Design an Analysis

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

Flow-sensitive

```
results :=
loc ↦ (var ↦ ℘({-,0,+}))
```

Let's Design an Analysis

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

Flow-sensitive

```
4:    x ∈ {0,+}
4.T: N ∈ {-,+}
5.F: x ∈ {0,+}

N,y ∈ {-,0,+}

UNSAFE: {100/x}
```

```
results ≔
loc ↦ (var ↦ ℘({-,0,+}))
```

Let's Design an Analysis

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

Path-sensitive

```
results :=
loc ↦ ℘(var ↦ ℘({-,0,+}))
```

Let's Design an Analysis

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

Path-sensitive

```
4: N∈{-,+},x∈{0}
4: N∈{0}   ,x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0}   ,y∈{0,+}

SAFE
```

```
results :=
loc ↦ ℘(var ↦ ℘({-,0,+}))
```

Let's Design an Analysis

Program

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

Analysis Property

x/0

Abstract Values

$\mathbb{Z} \sqsubseteq \{-, 0, +\}$

Implement

```
analyze : exp → results
analyze(x := æ) :=
    .. x .. æ ..
analyze(IF(æ){e₁}{e₂}) :=
    .. æ .. e₁ .. e₂ ..
```

Get Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}

SAFE
```

Prove Correct

$[\![e]\!] \in [\![analyze(e)]\!]$

Let's Design an Analysis

Program

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

Analysis Property

x/0

Abstract Values

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

```
analyze : exp → results
analyze(x : ) :=
    .. x ..
analyze(IF(e) {e1}{e2}) :=
    .. æ .. e1 .. e2 ..
```

Get Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}

SAFE
```

Prove Correct

⟦e⟧ ∈ ⟦analyze(e)⟧

Let's Design an Analysis

Program

Analysis Property

x/0

Abstract Values

$\mathbb{Z} \sqsubseteq \{-, 0, +\}$

Implement

```
analyze : exp → results
analyze(x := æ) :=
    .. x .. æ ..
analyze(IF(æ){e1}{e2}) :=
    .. æ .. e1 .. e2 ..
```

Get Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}

SAFE
```

Prove Correct

$[\![e]\!] \in [\![analyze(e)]\!]$

Let's Design an Analysis

Program

safe_fun.js ?

Analysis Property

x/0

Abstract Values

$\mathbb{Z} \sqsubseteq \{-, 0, +\}$

Implement

```
analyze : exp → results
analyze(x := æ) :=
    .. x .. æ ..
analyze(IF(æ){e1}{e2}) :=
    .. æ .. e1 .. e2 ..
```

Get Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}

SAFE
```

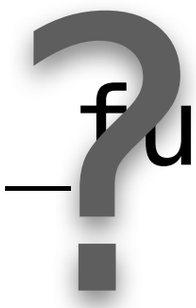
Prove Correct

$[\![e]\!] \in [\![analyze(e)]\!]$

Let's Design an Analysis

Program

safe_fun.js



Analysis Property

x/0

Abstract Values

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

```
analyze : exp → results
analyze(x : ℤ) :=
    .. x ..
analyze(IF(e){e1}{e2}) :=
    .. æ .. e1 .. e2 ..
```

Get Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}

SAFE
```

Prove Correct

⟦e⟧ ∈ ⟦analyze(e)⟧

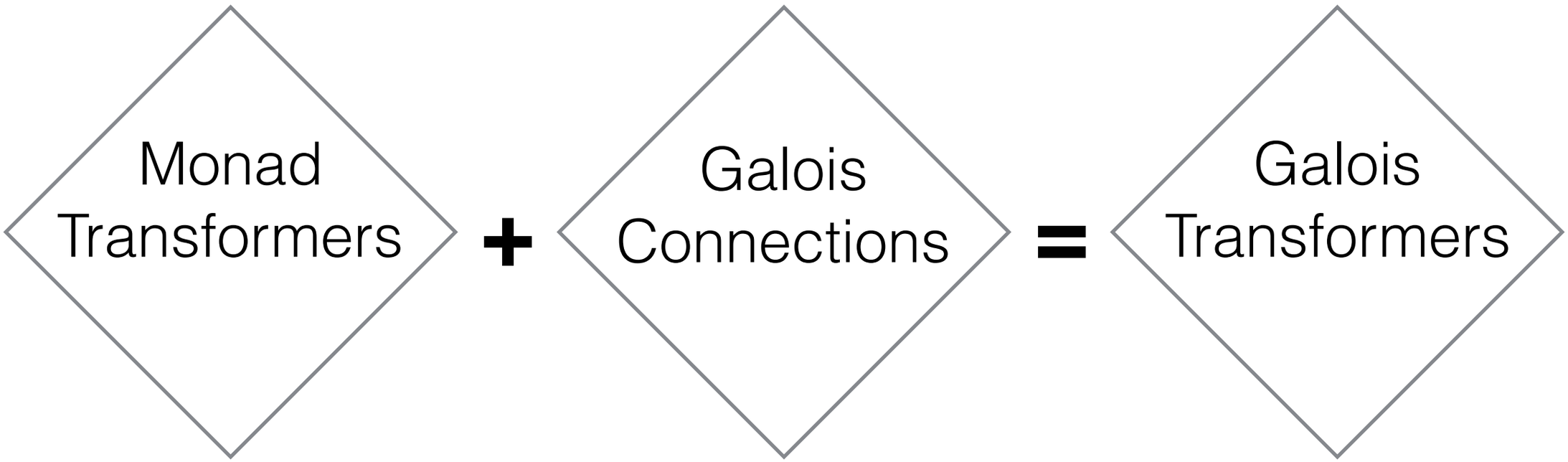
Problems Worth Solving

- How to change path/flow sensitivity without redesigning from scratch?

- How to reuse machinery between analyzers for different languages?

- How to translate proofs between different analysis designs?

Solution



Compositional interpreters

Compositional abstractions

Compositional abstract interpreters

Galois Transformers

- What's a Monad?
- What are Transformers?
- What are Galois Connections?

Galois Transformers

- What's a Monad?
- What are Transformers?
- What are Galois Connections?

A Monad

```
type M(t)

op x ← e1 ; e2
op return(e)

op get
op put(e)
op fail
op ...
```

- A module with:
  - a type operator M
  - a semicolon operator (bind)
  - effect operations
- M(t):
  - "A computation that performs some effects, then returns `t`"

A Monadic Interpreter

Program

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

Analysis Property

x/0

Abstract Domain

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

```
analyze : exp → results
analyze(x := æ) :=
    .. x .. æ ..
analyze(IF(æ){e₁}{e₂}) :=
    .. æ .. e₁ .. e₂ ..
```

Get Results

```
N ∈ {-,0,+}
x ∈ {0,+}
y ∈ {-,0,+}

UNSAFE: {100/N}
UNSAFE: {100/x}
```

Prove Correct

$[\![e]\!] \in [\![analyze(e)]\!]$

A Monadic Interpreter

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

A Monadic Interpreter

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

$$\mathtt{value} := \mathbb{Z} \cup \mathbb{B}$$

$$\rho \in \mathtt{env} := \mathtt{var} \mapsto \mathtt{value}$$

A Monadic Interpreter

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

$$\texttt{value} := \mathbb{Z} \cup \mathbb{B}$$
$$\rho \in \texttt{env} := \texttt{var} \mapsto \texttt{value}$$

```
type M(t)

op x ← e1 ; e2
op return(e)

op getEnv
op putEnv(e)

op fail
```

A Monadic Interpreter

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

$value := \mathbb{Z} \cup \mathbb{B}$

$\rho \in env := var \mapsto value$

```
step : exp → M(exp)
```

```
type M(t)

op x ← e1 ; e2
op return(e)

op getEnv
op putEnv(e)

op fail
```

A Monadic Interpreter

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

```
value := ℤ ∪ 𝔹
ρ ∈ env := var ↦ value

⟦_⟧ : atom → M(value)
```

```
step : exp → M(exp)
step(x := æ) := do
   v ← ⟦æ⟧
   ρ ← getEnv
   putEnv(ρ[x↦v])
   return(SKIP)
```

```
type M(t)

op x ← e1 ; e2
op return(e)

op getEnv
op putEnv(e)

op fail
```

A Monadic Interpreter

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

$$value := \mathbb{Z} \cup \mathbb{B}$$
$$\rho \in env := var \mapsto value$$

$$[\![\_]\!] : atom \to M(value)$$

```
step : exp → M(exp)
step(x := æ) := do
   v ← [[æ]]
   ρ ← getEnv
   putEnv(ρ[x↦v])
   return(SKIP)
step(IF(æ){e₁}{e₂}):= do
   v ← [[æ]]
   case v of
     True → return(e₁)
     False → return(e₂)
     _ → fail
```

```
type M(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail
```

Abstractify

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

$$value := \mathbb{Z} \cup \mathbb{B}$$
$$\rho \in env := var \mapsto value$$

$$[\![\_]\!] : atom \to M(value)$$

```
step : exp → M(exp)
step(x := æ) := do
  v ← [[æ]]
  ρ ← getEnv
  putEnv(ρ[x↦v])
  return(SKIP)
step(IF(æ){e₁}{e₂}):= do
  v ← [[æ]]
  case v of
    True → return(e₁)
    False → return(e₂)
    _ → fail
```

```
type M(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail
```

Abstractify

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

$value^\sharp := \wp(\{-, 0, +\}) \cup \wp(\mathbb{B})$

$\rho \in env^\sharp := var \mapsto value^\sharp$

$[\![\_]\!]^\sharp : atom \to M^\sharp(value^\sharp)$

```
step : exp → M♯(exp)
step(x := æ) := do
   v ← [[æ]]♯
   ρ ← getEnv
   putEnv(ρ[x↦v])
   return(SKIP)
step(IF(æ){e1}{e2}):= do
   v ← [[æ]]♯
   case v of
     True → return(e1)
     False → return(e2)
     _ → fail
```

```
type M♯(t)

op x ← e1 ; e2
op return(e)

op getEnv
op putEnv(e)

op fail
```

Abstractify

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

$$value^\sharp := \wp(\{-,0,+\}) \cup \wp(\mathbb{B})$$

$$\rho \in env^\sharp := var \mapsto value^\sharp$$

$$[\![\_]\!]^\sharp : atom \to M^\sharp(value^\sharp)$$

```
step : exp → M♯(exp)
step(x := æ) := do
   v ← [[æ]]♯
   ρ ← getEnv
   putEnv(ρ ⊔ [x↦v])
   return(SKIP)
step(IF(æ){e₁}{e₂}):= do
   v ← [[æ]]♯
   case v of
     True → return(e₁)
     False → return(e₂)
     _ → fail
```

```
type M♯(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail
```

Abstractify

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

```
step : exp → M♯(exp)
step(x := æ) := do
   v ← ⟦æ⟧♯
   ρ ← getEnv
   putEnv(ρ ⊔ [x↦v])
   return(SKIP)
step(IF(æ){e₁}{e₂}):= do
   v ← ⟦æ⟧♯
   b ← chooseBool(v)
   case b of
      True → return(e₁)
      False → return(e₂)
```

$value^\sharp := \wp(\{-,0,+\}) \cup \wp(\mathbb{B})$

$\rho \in env^\sharp := var \mapsto value^\sharp$

$[\![\_]\!]^\sharp : atom \to M^\sharp(value^\sharp)$

$chooseBool : value^\sharp \to M^\sharp(\mathbb{B})$

```
type M♯(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail
```

Abstractify

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

```
step : exp → M♯(exp)
step(x := æ) := do
   v ← ⟦æ⟧♯
   ρ ← getEnv
   putEnv(ρ ⊔ [x↦v])
   return(SKIP)
step(IF(æ){e₁}{e₂}):= do
   v ← ⟦æ⟧♯
   b ← chooseBool(v)
   case b of
     True → return(e₁)
     False → return(e₂)
```

$value^\sharp := \wp(\{-,0,+\}) \cup \wp(\mathbb{B})$

$\rho \in env^\sharp := var \mapsto value^\sharp$

$[\![\_]\!]^\sharp : atom \to M^\sharp(value^\sharp)$

$chooseBool : value^\sharp \to M^\sharp(\mathbb{B})$

```
type M♯(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail/e₁⊞e₂
```

Monadic Abs. Interpreters

- Start with a *concrete* monadic interpreter
- Abstract value space (`value♯`,`⟦_⟧♯`)
- Join results when updating `env♯` (⊔)
- Branch nondeterministically (`chooseBool`)

Why Monads

- A monadic interpreter can be simpler than a state machine or constraint system
- Two effects, `State[`s`]` and `Nondet`
  - Encode arbitrary small-step state machine relations
- Don't commit to a single implementation of $M^\sharp$
  - Different choices for $M^\sharp$ yield different analyses

Galois Transformers

- What's a Monad?
- What are Transformers?
- What are Galois Connections?

Galois Transformers

- What's a Monad?
- What are Transformers?
- What are Galois Connections?

```
type M(t)

op x ← e1 ; e2
op return(e)
```

Galois Transformers

- What's a Monad?
- What are Transformers?
- What are Galois Connections?

```
type M(t)

op x ← e1 ; e2
op return(e)
```

Why Monads

- A monadic interpreter can be simpler than a state machine or constraint system
- Two effects, `State[`$\mathscr{s}$`]` and `Nondet`
  - Encode arbitrary small-step state machine relations
- Don't commit to a single implementation of $M^\sharp$
  - Different choices for $M^\sharp$ yield different analyses

Monad Transformers

`State[`$\mathcal{S}$`]`

`get` : $M(\mathcal{S})$

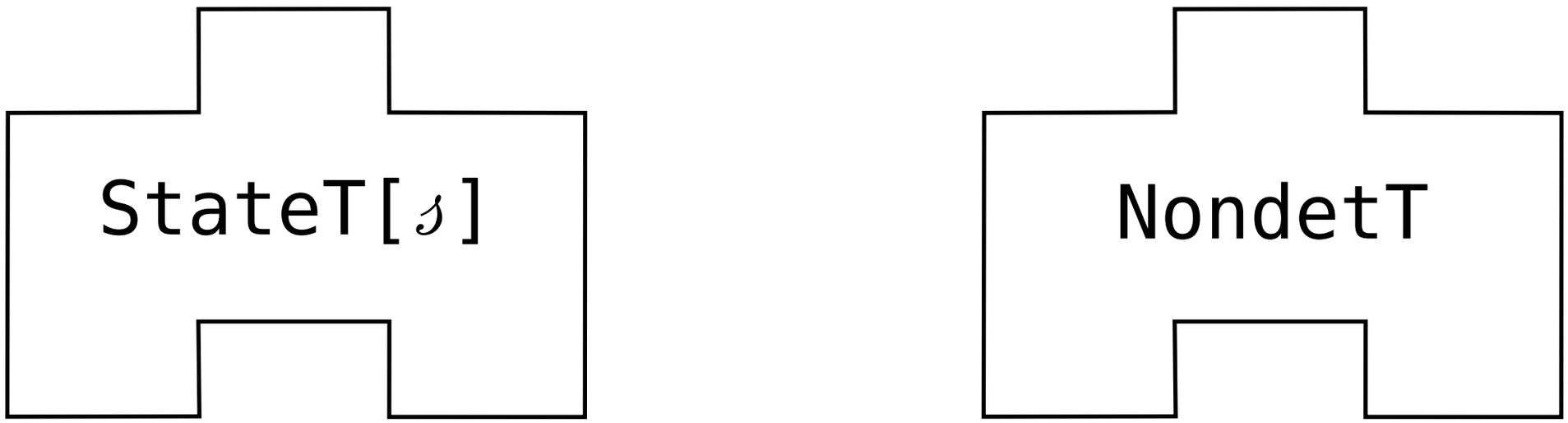
`put` : $\mathcal{S} \to M(1)$

`Nondet`

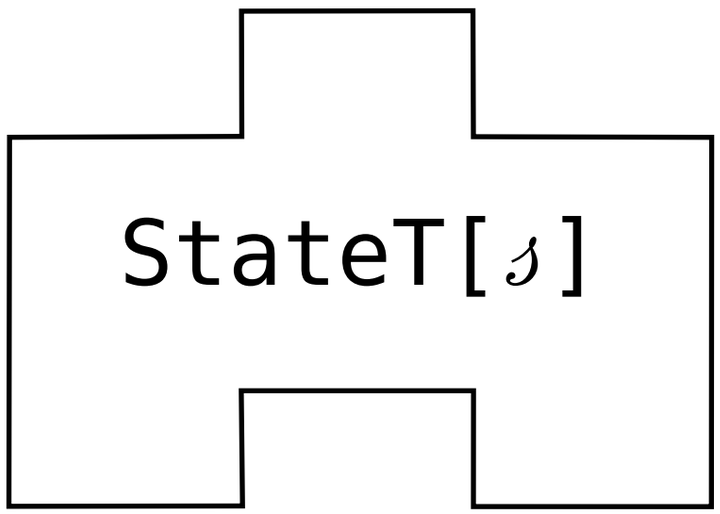
`fail` : `∀(A),`M`(A)`

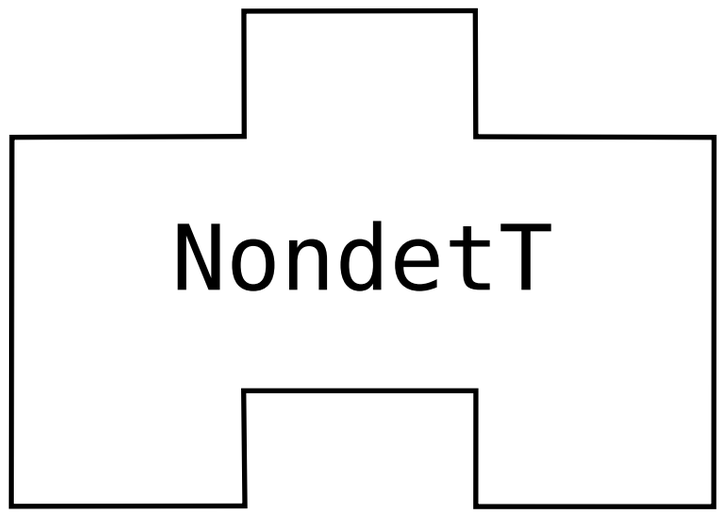
`_⊞_` : `∀(A),`$M(A) \times M(A) \to M(A)$

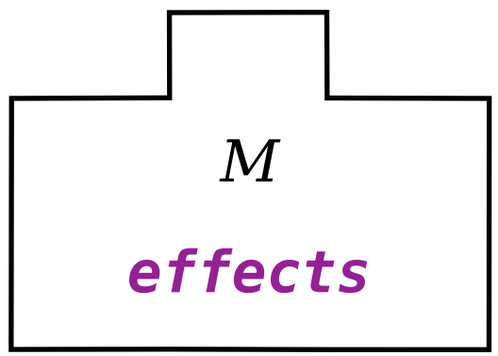
Monad Transformers



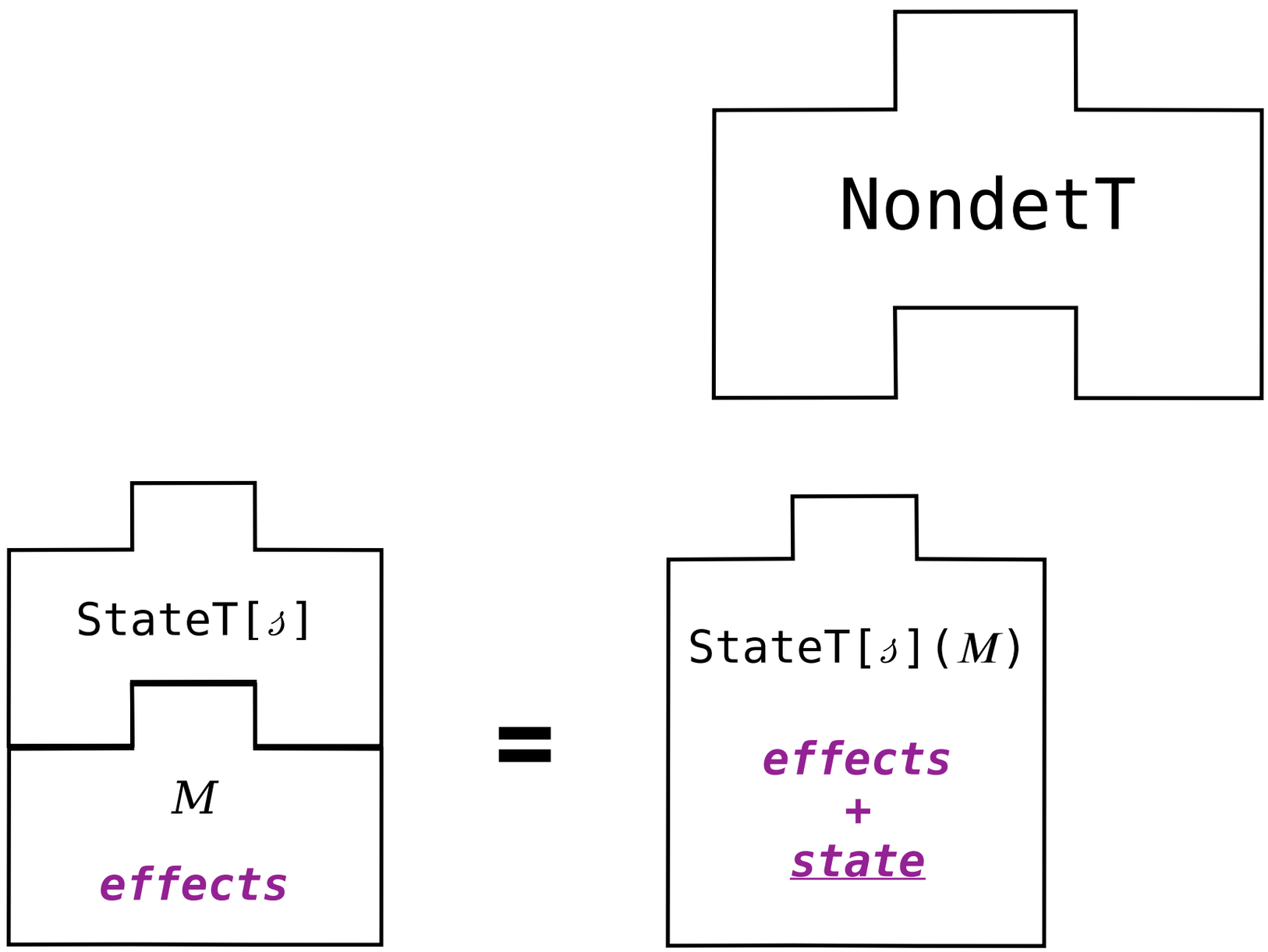
Monad Transformers



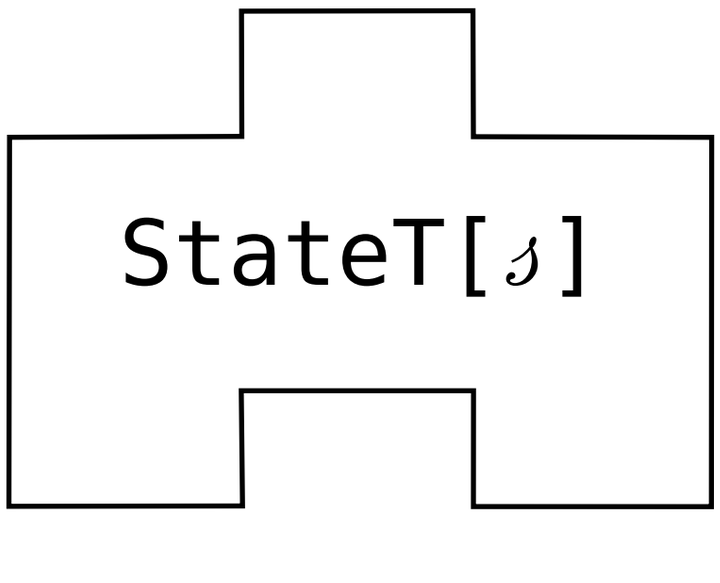


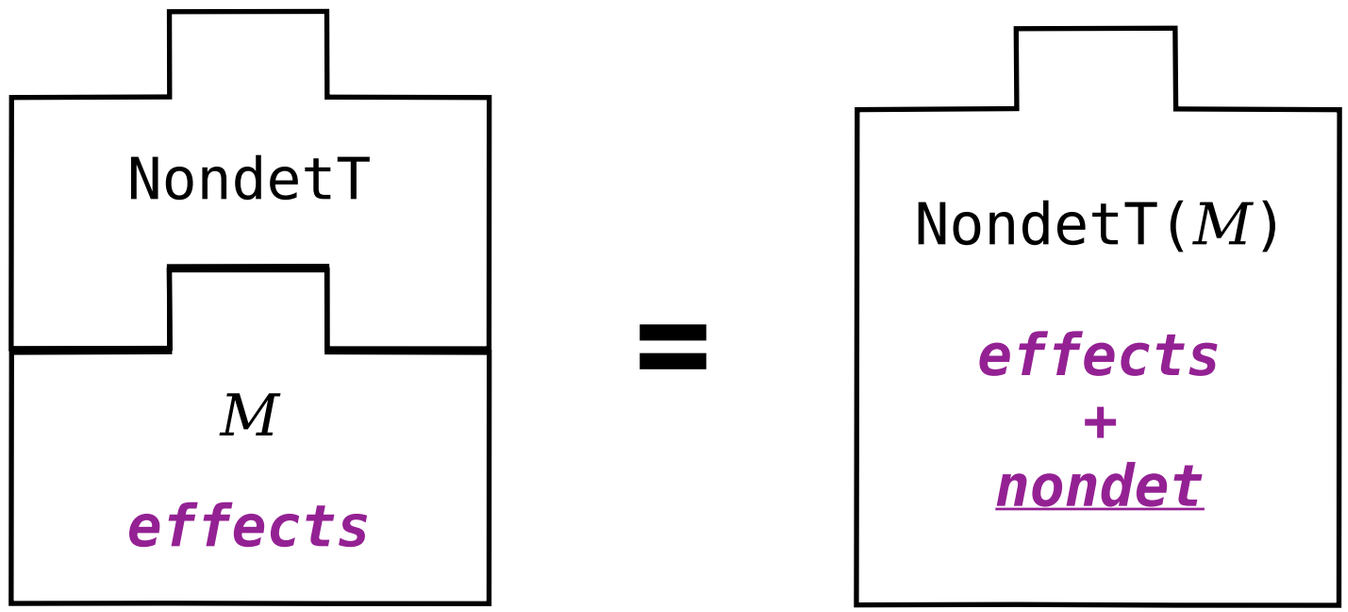


Monad Transformers

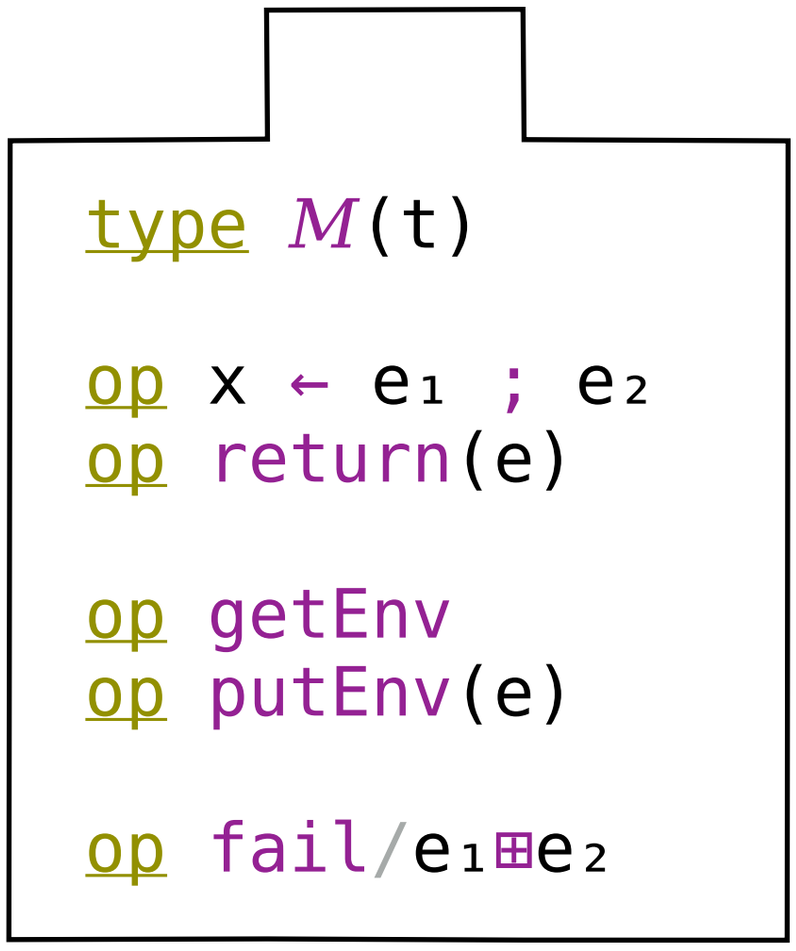


Monad Transformers





Monad Transformers

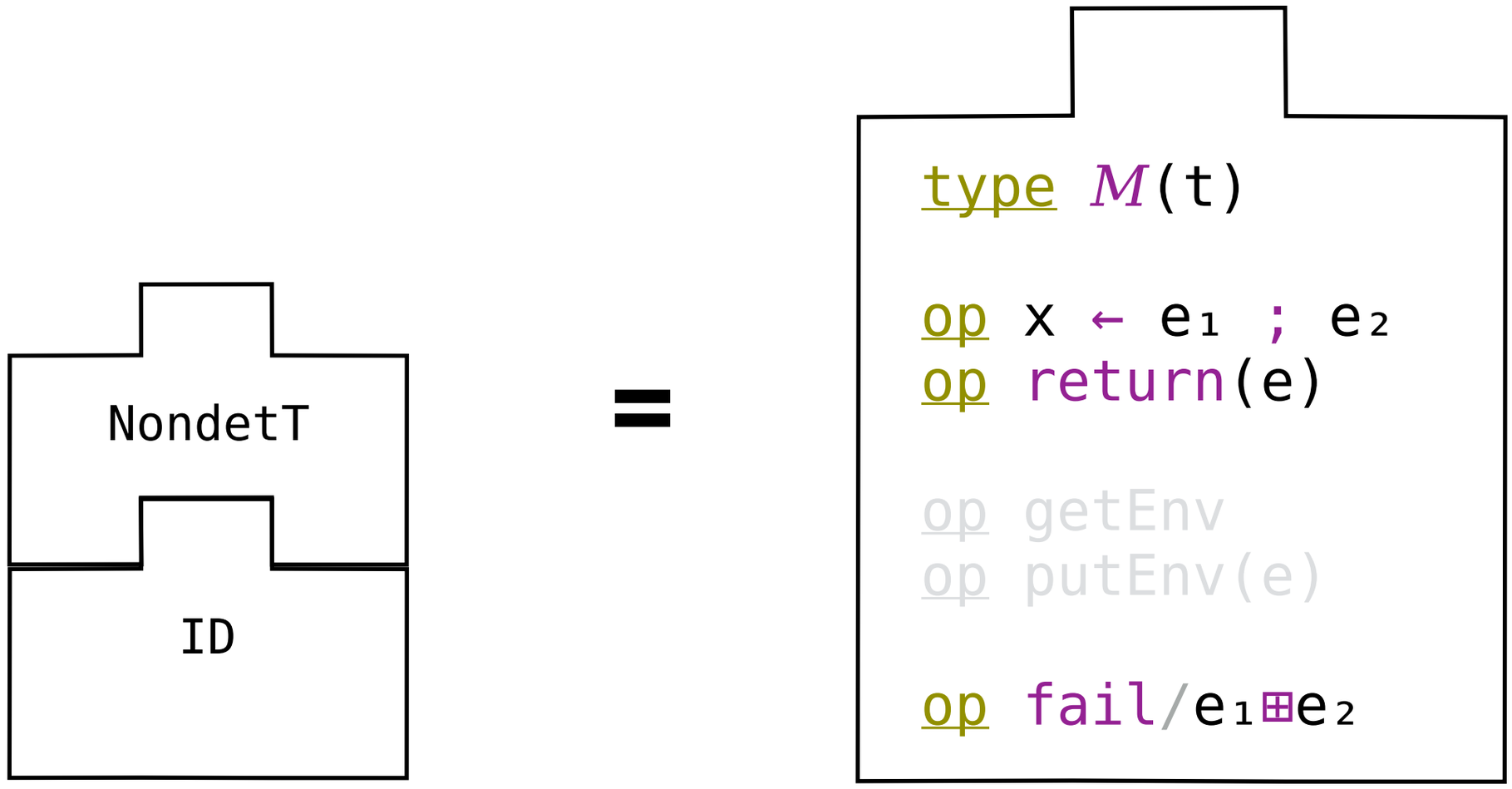


Monad Transformers

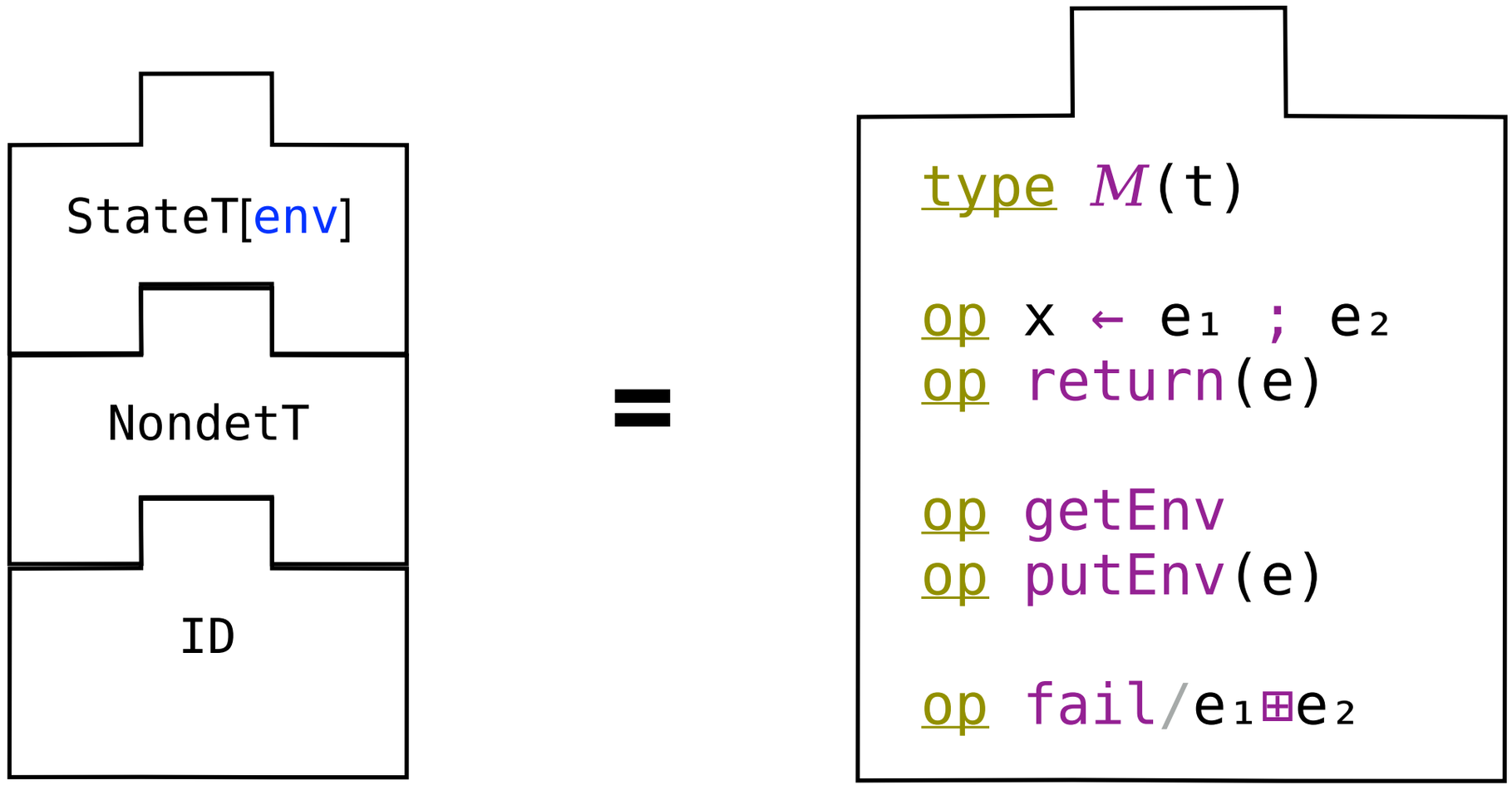
=

```
type M(t)

op x ← e1 ; e2
op return(e)

op getEnv
op putEnv(e)

op fail/e1⊞e2
```

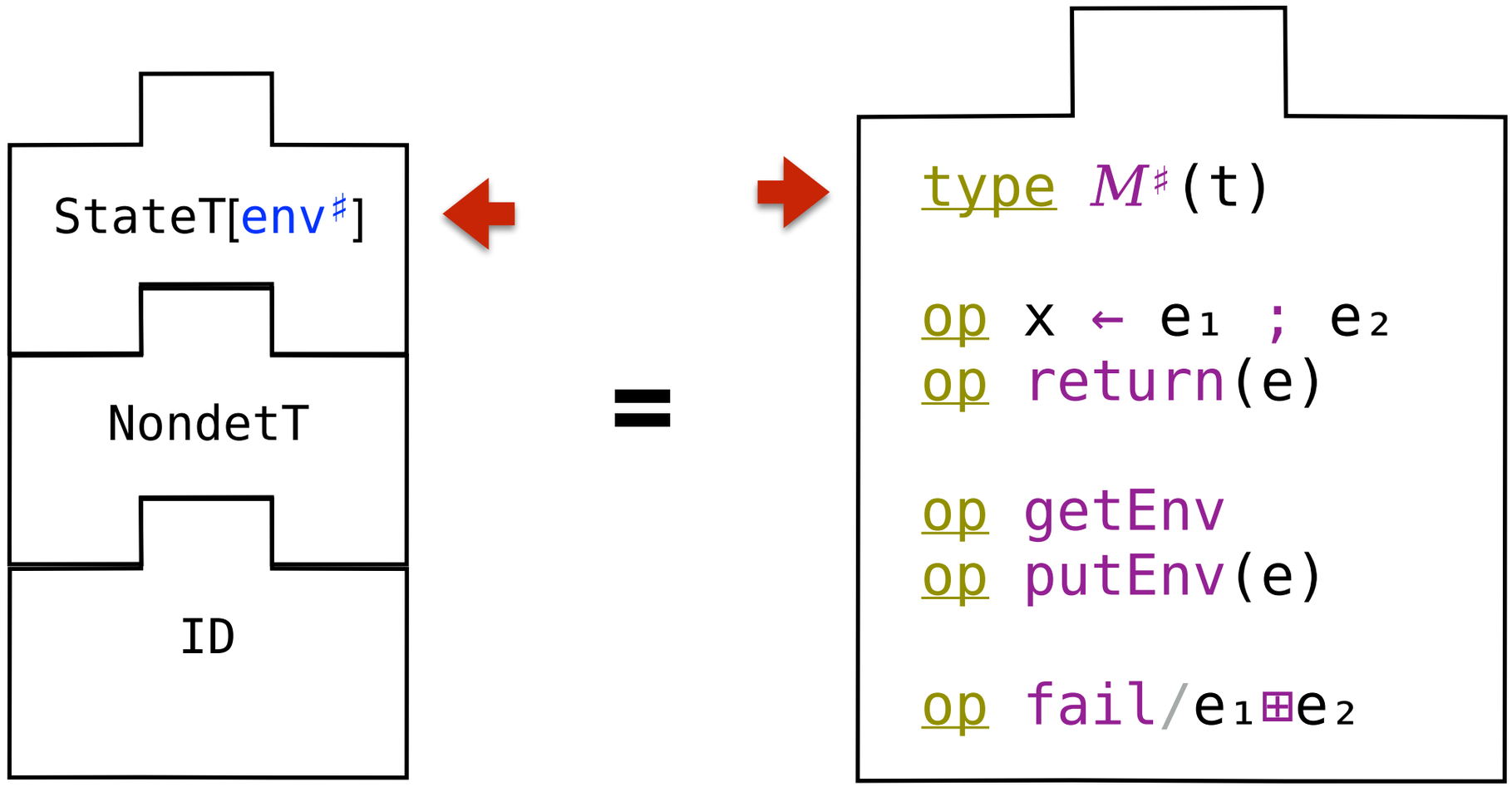
Monad Transformers



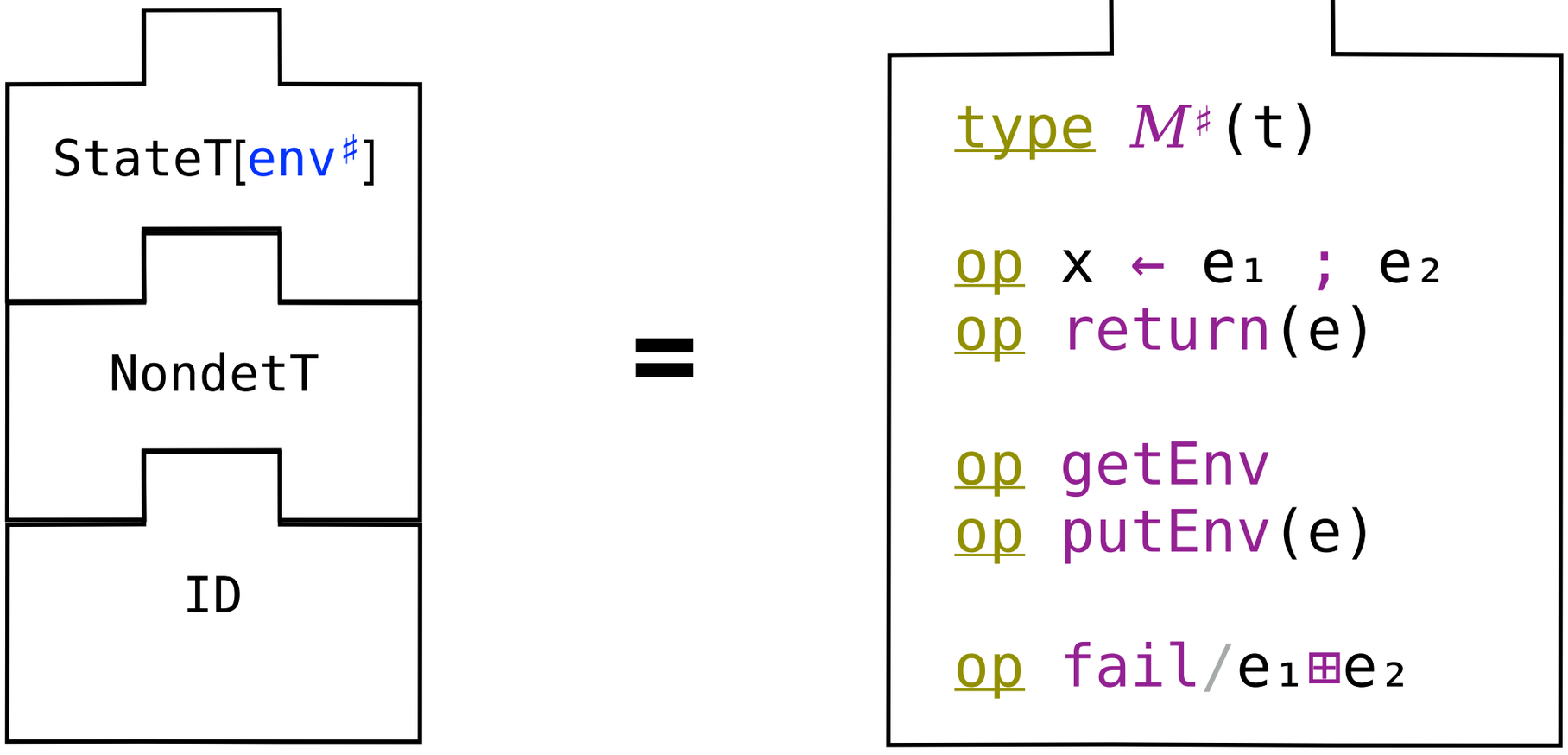
Monad Transformers



Monad Transformers

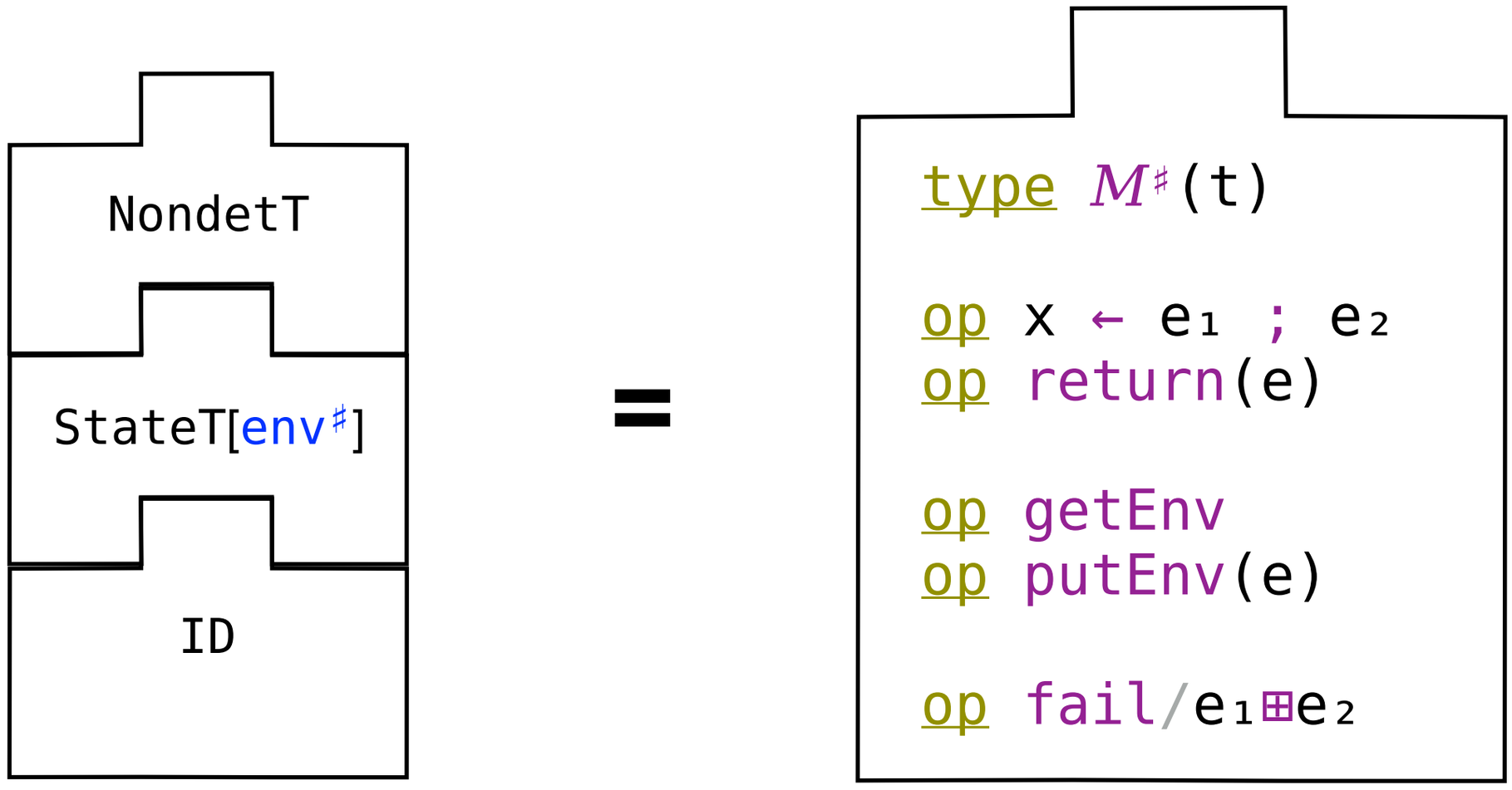


Monad Transformers



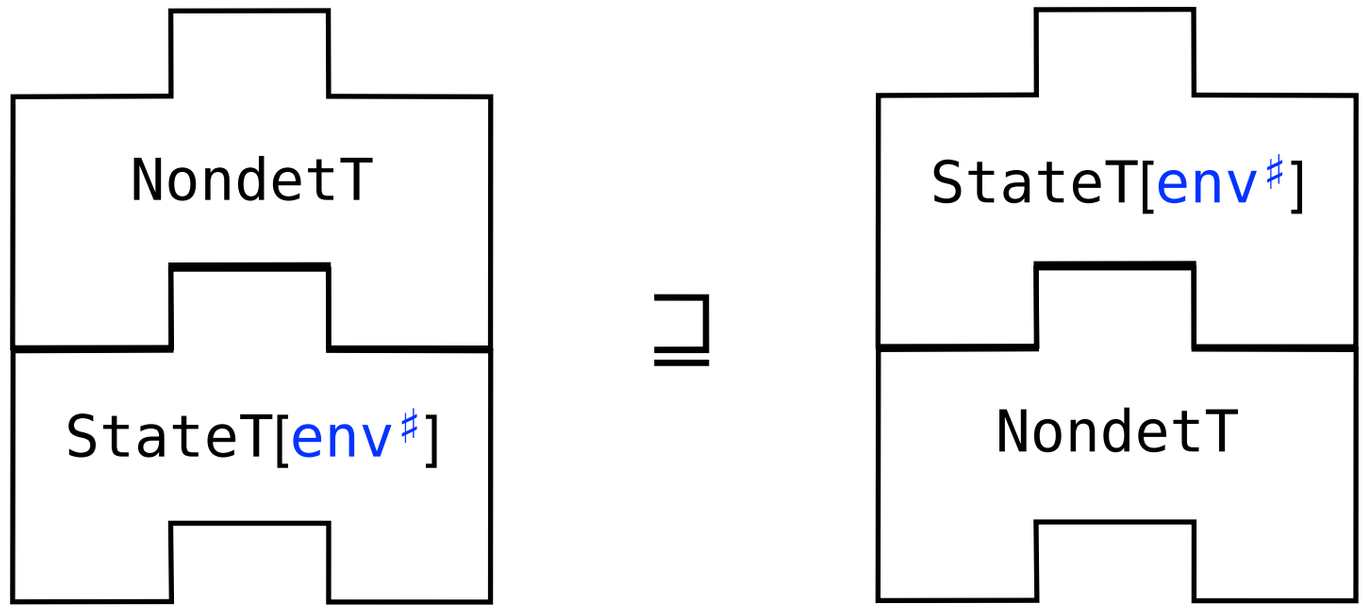
Path-sensitive

Monad Transformers



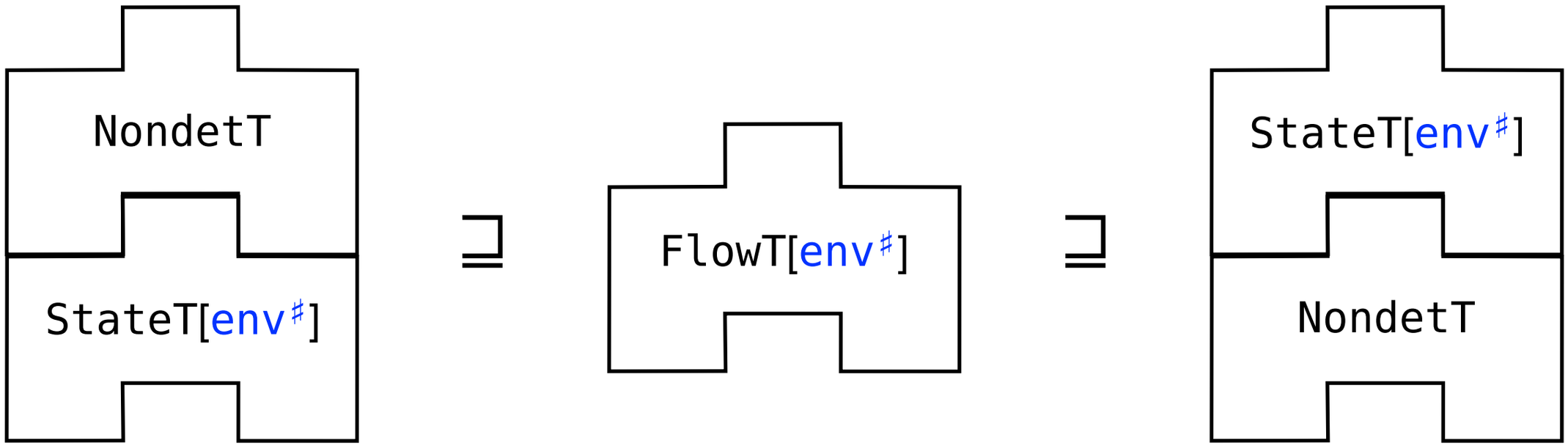
Flow-insensitive

Monad Transformers



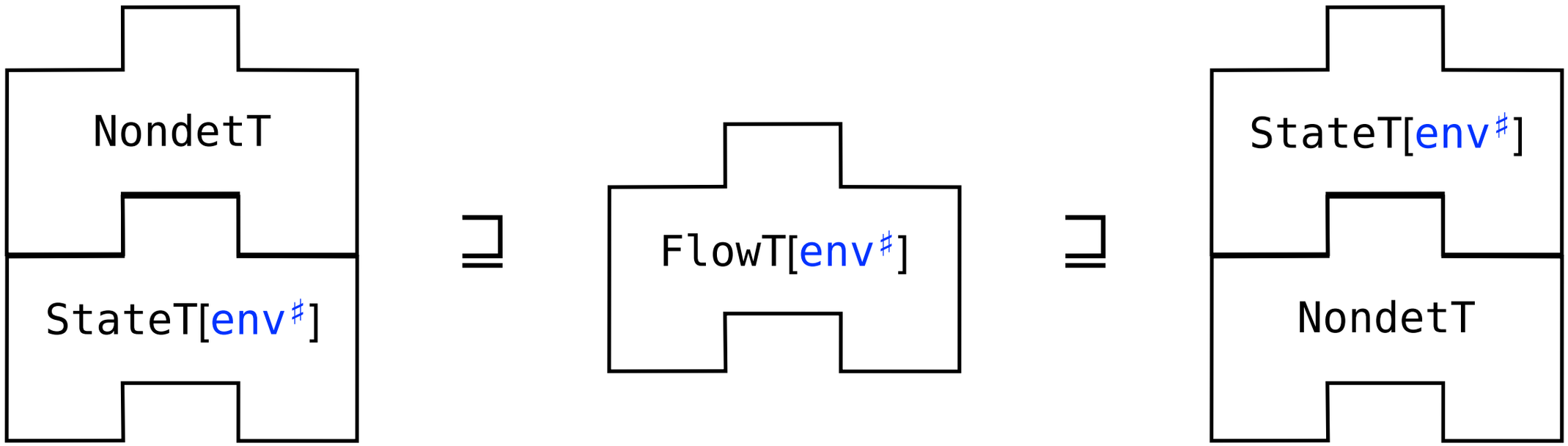
Flow-insensitive *Path-sensitive*

Monad Transformers



Flow-insensitive *Flow-sensitive* *Path-sensitive*

Monad Transformers



Flow-insensitive *Flow-sensitive* *Path-sensitive*

$\wp(\mathtt{exp}) \times \mathtt{env}^\sharp$ $\mathtt{exp} \mapsto \mathtt{env}^\sharp$ $\mathtt{exp} \mapsto \wp(\mathtt{env}^\sharp)$

Monad Transformers

Flow-insensitive	*Flow-sensitive*	*Path-sensitive*
$\wp(\mathsf{exp}) \times \mathsf{env}^\sharp$	$\mathsf{exp} \mapsto \mathsf{env}^\sharp$	$\mathsf{exp} \mapsto \wp(\mathsf{env}^\sharp)$

```
N ∈ {-,0,+}
x ∈ {0,+}
y ∈ {-,0,+}

UNSAFE: {100/N}
UNSAFE: {100/x}
```

```
4:    x ∈ {0,+}
4.T: N ∈ {-,+}
5.F: x ∈ {0,+}

N,y ∈ {-,0,+}

UNSAFE: {100/x}
```

```
4: N∈{-,+},x∈{0}
4: N∈{0}  ,x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0}  ,y∈{0,+}

SAFE
```

Building Monads

- Construct a monad using `StateT[𝑠]`, `FlowT[𝑠]` and `NondetT`
- Order matters, yielding different analyses
- Rapidly prototype precision performance tradeoffs

Why Transformers

- Semantics independent building blocks for writing interpreters—also apply to abstract interpreters!
- Reuse of analysis machinery
  - Different abs. interpreters use the same transformers
- Variations in analysis
  - Different transformer stacks fit into the same interpreter

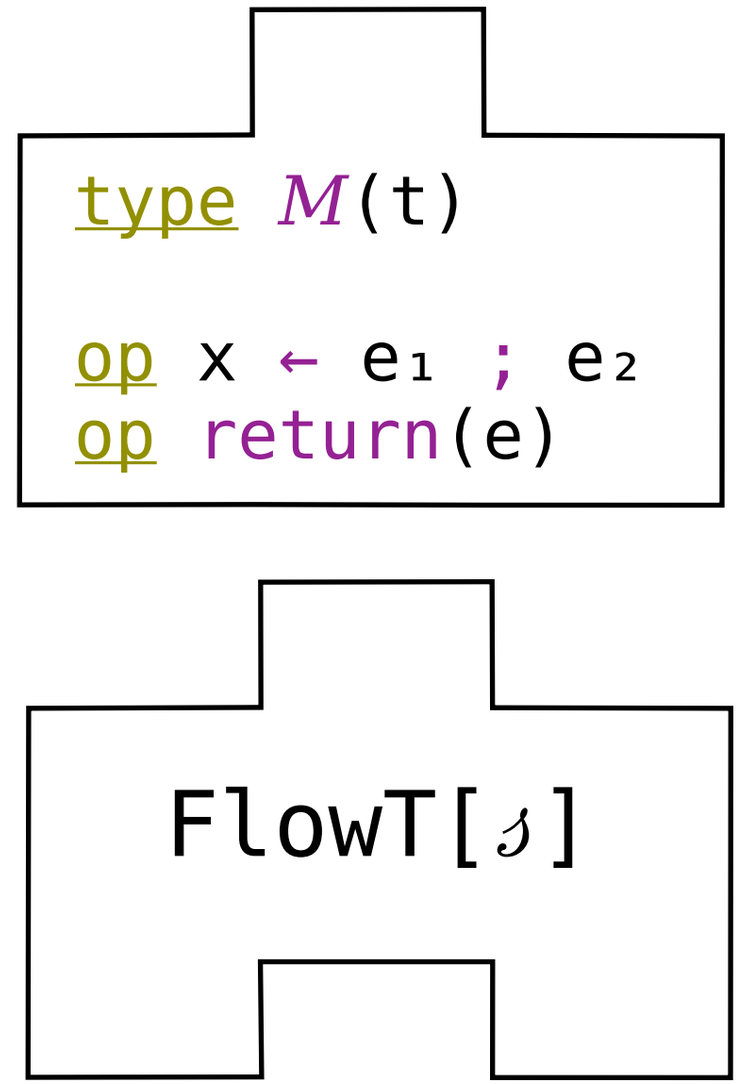
Galois Transformers

- What's a Monad?
- What are Transformers?
- What are Galois Connections?

```
type M(t)

op x ← e1 ; e2
op return(e)
```

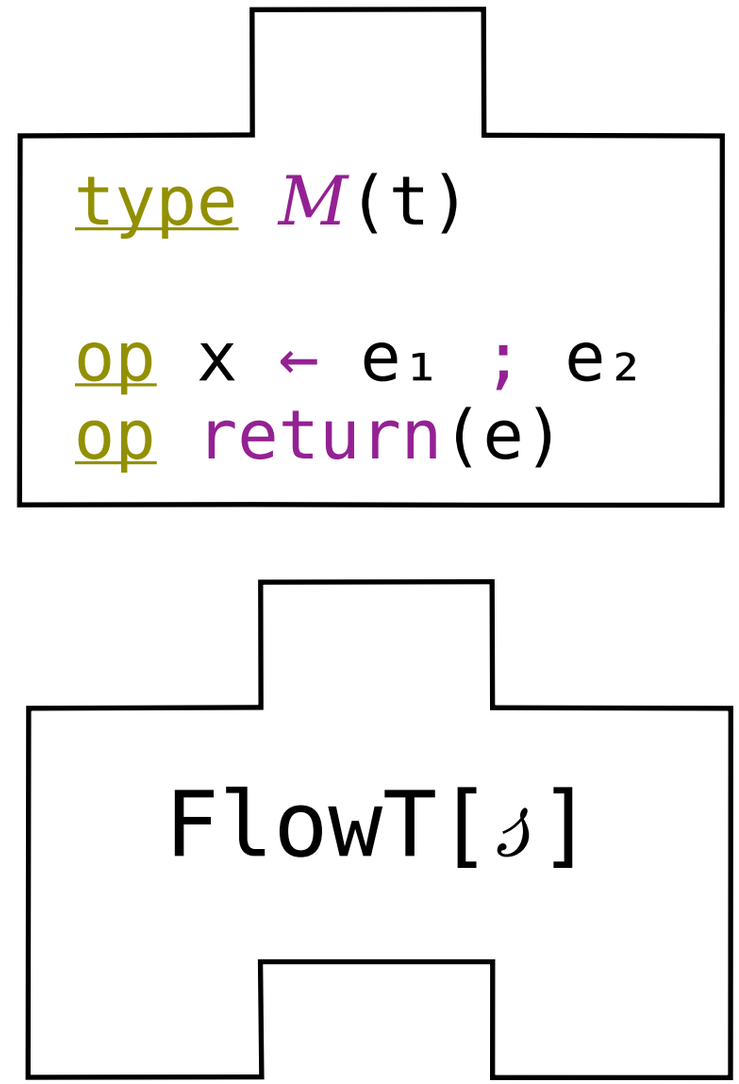
Galois Transformers

- What's a Monad?
- What are Transformers?
- What are Galois Connections?



Galois Transformers

- What's a Monad?
- What are Transformers?
- What are Galois Connections?



Galois Connections

- Compositional framework for proving correctness
- We build two sets of GCs alongside transformers
- **Code**: Enables execution of monadic analyzers
- **Proofs**: Large number of proofs built automatically
- (See the paper)

Proof Framework

Proof Framework

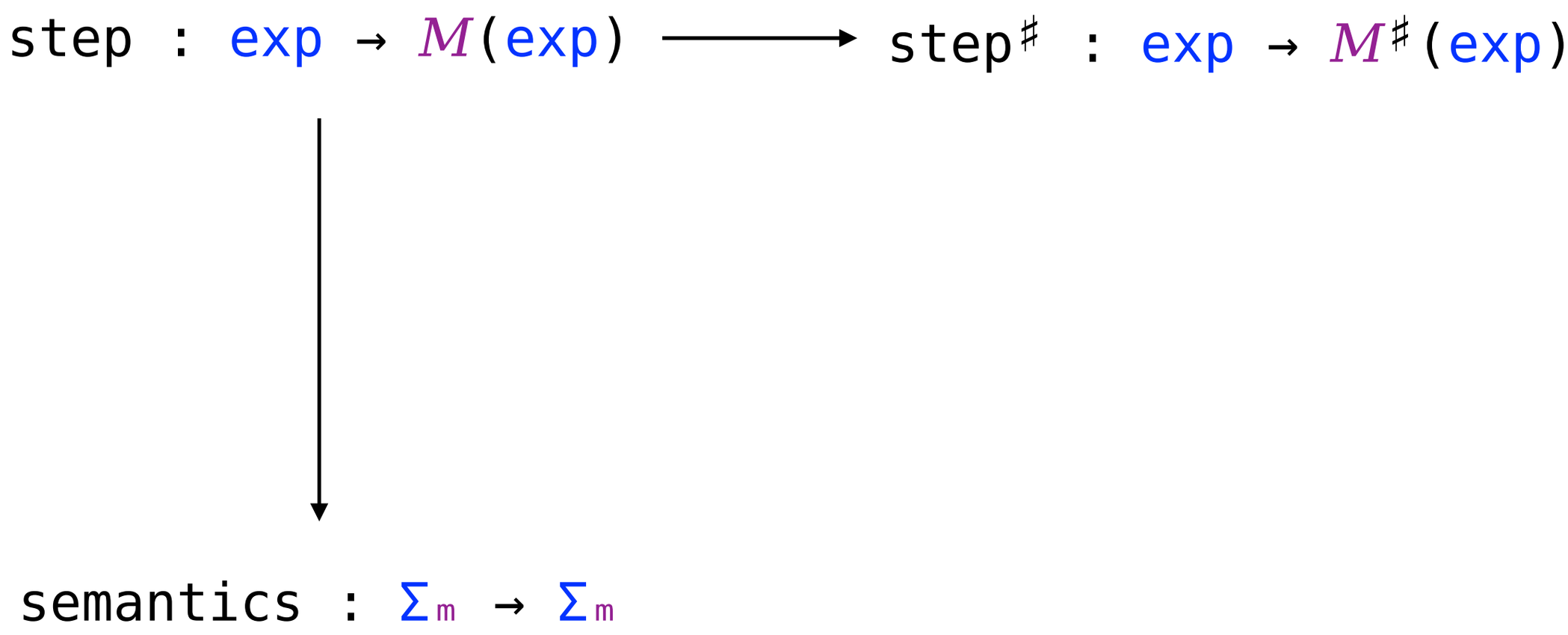
```
step : exp → M(exp)
```

Proof Framework

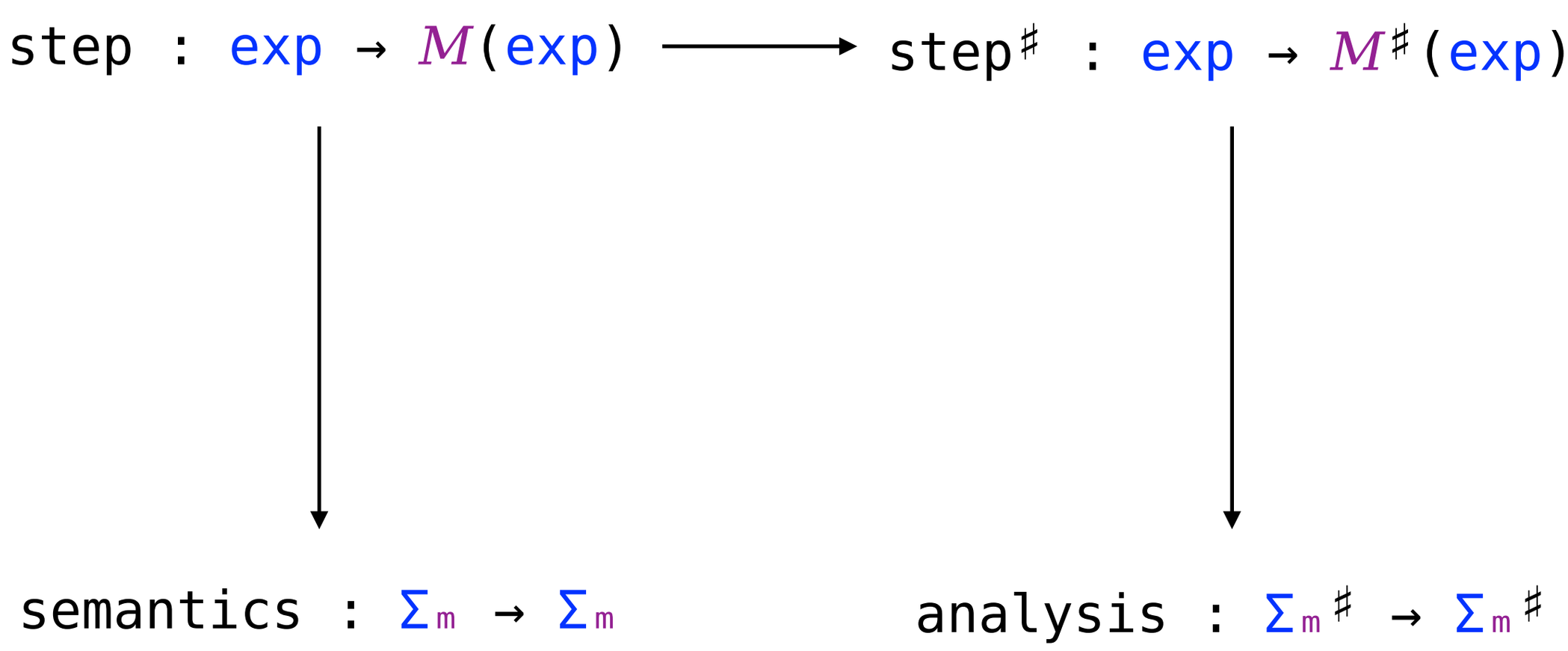
```
step : exp → M(exp)
```

```
semantics : Σm → Σm
```

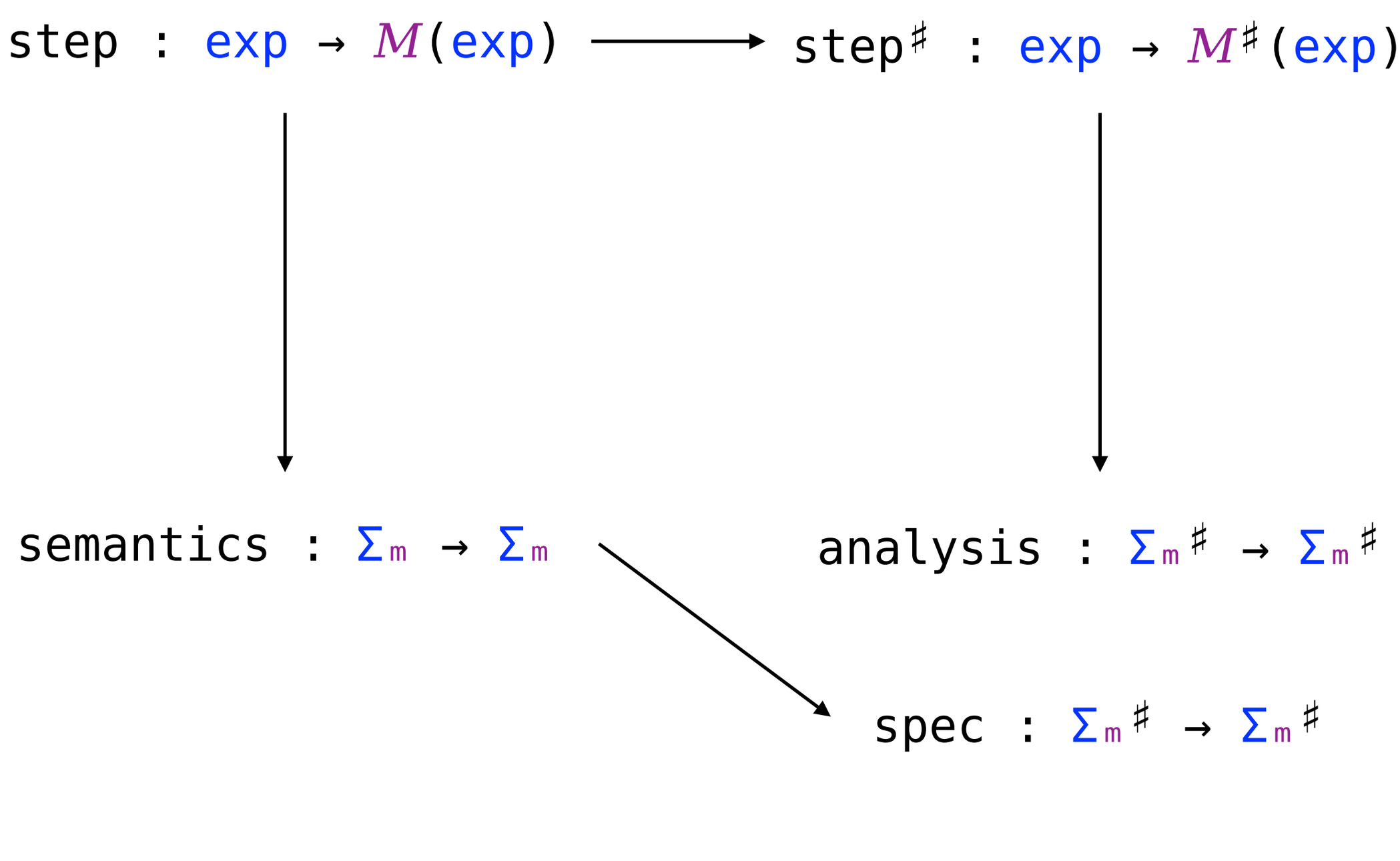
Proof Framework



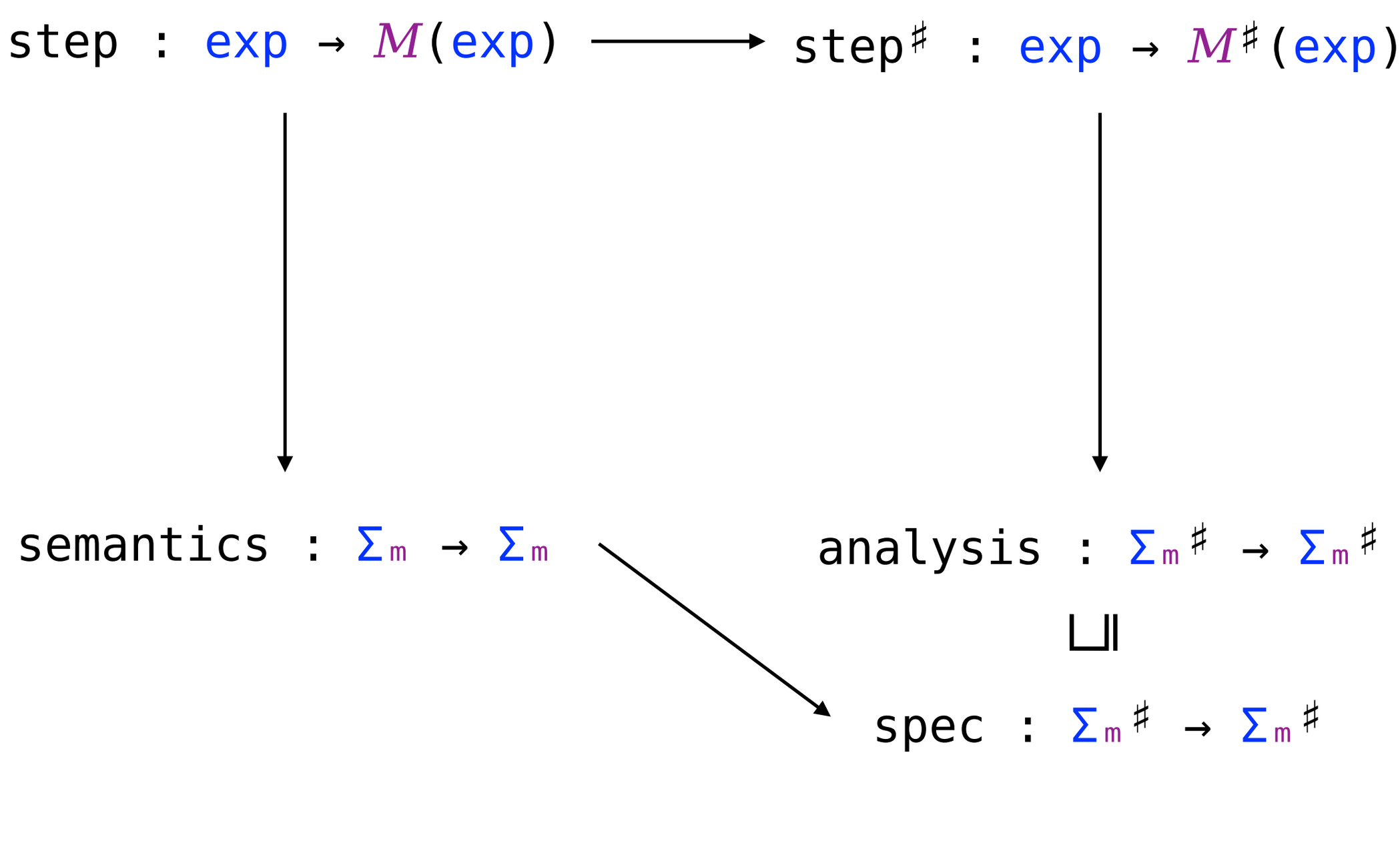
Proof Framework



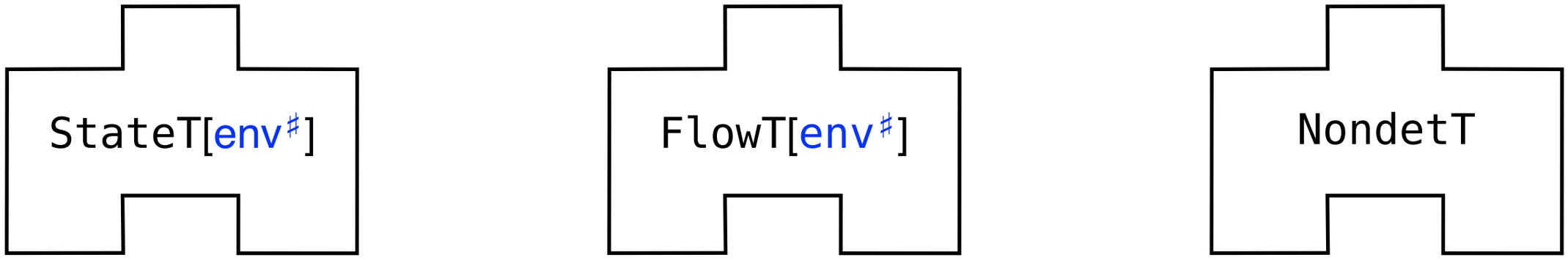
Proof Framework



Proof Framework



Galois Transformers



- GTs = Monad Transformers + Galois connections
- Galois connections are necessary for **execution** and central to **proof framework**

Putting it All Together

- You design a monadic abstract interpreter
- Instantiate with monad transformers
- Change underlying monad to change results
- Execution engine and proofs for free

Implementation

- Haskell package: `cabal install maam`

- Galois Transformers are implemented as a semantics independent library

- Haskell's support for monadic programming was helpful, but not necessary

Let's Design an Analysis

Program

```
0: int x y; // global state
1: void safe_fun(int N) {
2:  if (N≠0) {x := 0;}
3:  else     {x := 1;}
4:  if (N≠0) {y := 100/N;}
5:  else     {y := 100/x;}}
```

Analysis Property

x/0

Abstract Values

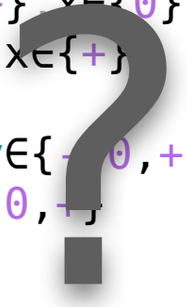
$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

```
analyze : exp → results
analyze(x := æ) :=
    .. x .. æ ..
analyze(if {e1}{e2}) :=
    .. æ .. e1 .. e2 ..
```

Get Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}

SAFE
```



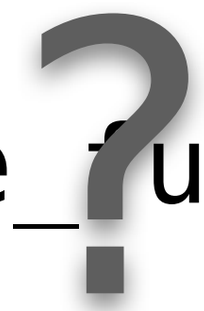
Prove Correct

⟦e⟧ ∈ ⟦analyze(e)⟧

Let's Design an Analysis

Program

safe_fun.js



Analysis Property

x/0

Abstract Values

$\mathbb{Z} \sqsubseteq \{-,0,+\}$

Implement

```
analyze : exp → results
analyze(x := æ) :=
    .. x .. æ ..
analyze(if {e1}{e2}) :=
    .. æ .. e1 .. e2 ..
```

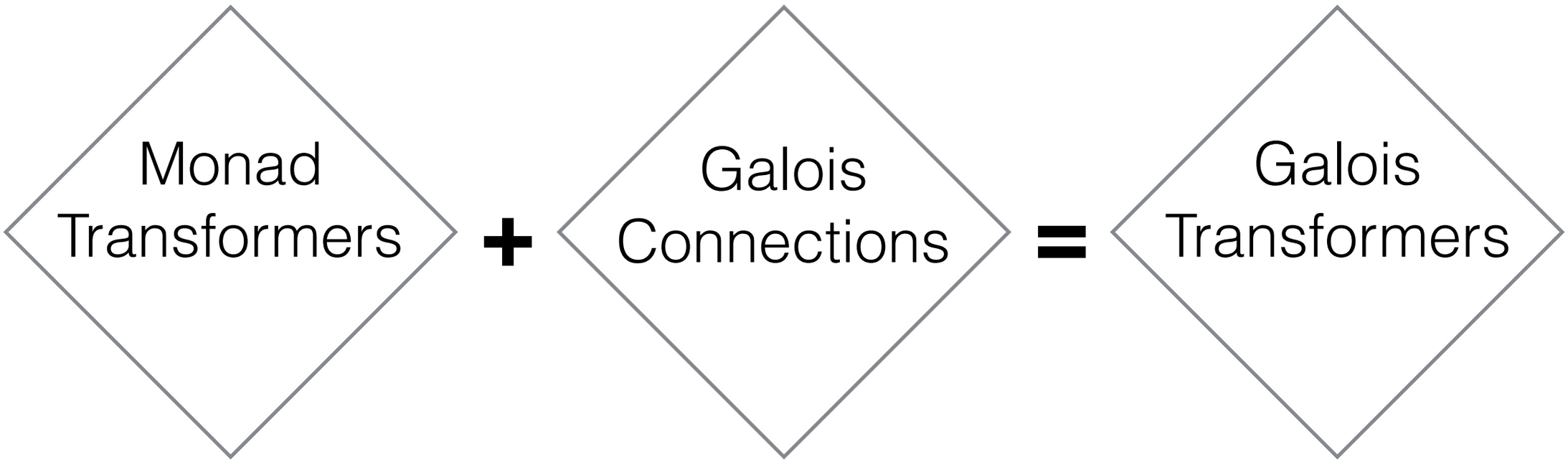
Get Results

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}

SAFE
```

Prove Correct

⟦e⟧ ∈ ⟦analyze(e)⟧

Galois Transformers



Compositional interpreters

Compositional abstractions

Compositional abstract interpreters

Let's Verify an Analysis

Let's Verify an Analysis

*(**NOT** in the paradigm of abstract interpretation)*

Specification

Specification

$$succ : \mathbb{N} \to \mathbb{N}$$

Specification

`succ : ℕ → ℕ`

“`succ(n)` flips the parity of `n`”

Specification

```
succ : ℕ → ℕ
```

“`succ(n)` flips the parity of `n`”

```
ℙ ≔ E | O
parity : ℕ → ℙ
flip   : ℙ → ℙ
```

Specification

```
succ : ℕ → ℕ
```

“`succ(n)` flips the parity of `n`”

```
ℙ ≔ E | O
parity : ℕ → ℙ
flip   : ℙ → ℙ

∀(n:ℕ), parity(succ(n)) = flip(parity(n))
```

Verification

Verification

```
flip : ℙ → ℙ
flip(E) ≔ O
flip(O) ≔ E
```

Verification

```
flip : ℙ → ℙ
flip(E) ≔ O
flip(O) ≔ E
```

```
parity : ℕ → ℙ
parity(0) ≔ E
parity(succ(n)) ≔ flip(parity(n))
```

Verification

```
flip : ℙ → ℙ
flip(E) ≔ O
flip(O) ≔ E
```

```
parity : ℕ → ℙ
parity(0) ≔ E
parity(succ(n)) ≔ flip(parity(n))

∀(n:ℕ), parity(succ(n)) = flip(parity(n))
```

Verification

```
flip : ℙ → ℙ
flip(E) ≔ O
flip(O) ≔ E
```

```
parity : ℕ → ℙ
parity(0) ≔ E
parity(succ(n)) ≔ flip(parity(n))
```

```
∀(n:ℕ), parity(succ(n)) = flip(parity(n))
```

Proof is trivial by definition. ∎

Let's Verify an Analysis

Let's Verify an Analysis

(in the paradigm of abstract interpretation)

Abstract Interpretation

Abstract Interpretation

1. Establish a connection between sets $\mathbb{N}$ and $\mathbb{P}$

Abstract Interpretation

1. Establish a connection between sets $\mathbb{N}$ and $\mathbb{P}$
2. Induce a specification for `succ`

Abstract Interpretation

1. Establish a connection between sets $\mathbb{N}$ and $\mathbb{P}$

2. Induce a specification for `succ`

3. Connect the specification of `succ` to `flip`

Connecting Sets

Connecting Sets

```
parity : ℕ → ℙ
parity(0) ≔ E
parity(succ(n)) ≔ flip(parity(n))
```

Connecting Sets

```
parity : ℕ → ℙ
parity(0) ≔ E
parity(succ(n)) ≔ flip(parity(n))


⟦_⟧ : ℙ → ℘(ℕ)
⟦E⟧ ≔ {n | n is even}
⟦O⟧ ≔ {n | n is odd}
```

Connecting Sets

```
parity : ℕ → ℙ
parity(0) ≔ E
parity(succ(n)) ≔ flip(parity(n))
```

```
⟦_⟧ : ℙ → ℘(ℕ)
⟦E⟧ ≔ {n | n is even}
⟦O⟧ ≔ {n | n is odd}
```

```
α : ℘(ℕ) → ℘(ℙ)
α(N) ≔ {parity(n) | n ∈ N}
```

Connecting Sets

```
parity : ℕ → ℙ
parity(0) ≔ E
parity(succ(n)) ≔ flip(parity(n))
```

```
⟦_⟧ : ℙ → ℘(ℕ)
⟦E⟧ ≔ {n | n is even}
⟦O⟧ ≔ {n | n is odd}
```

```
α : ℘(ℕ) → ℘(ℙ)
α(N) ≔ {parity(n) | n ∈ N}
```

```
γ : ℘(ℙ) → ℘(ℕ)
γ(P) ≔ {n | p ∈ P ∧ n ∈ ⟦p⟧}
```

Connecting Sets

$$\mathtt{sound} : \forall(N:\wp(\mathbb{N})),\ N \subseteq \gamma(\alpha(N))$$
$$\mathtt{tight} : \forall(P:\wp(\mathbb{P})),\ \alpha(\gamma(P)) \subseteq P$$

Connecting Sets

```
sound : ∀(N:℘(ℕ)), N ⊆ γ(α(N))
tight : ∀(P:℘(ℙ)), α(γ(P)) ⊆ P

γ(α({1,2}))
= γ({E,O})
= {n | n ∈ ℕ} ⊇ {1,2}
```

Connecting Sets

$$\mathit{sound} : \forall(N:\wp(\mathbb{N})),\ N \subseteq \gamma(\alpha(N))$$
$$\mathit{tight} : \forall(P:\wp(\mathbb{P})),\ \alpha(\gamma(P)) \subseteq P$$

$$\gamma(\alpha(\{1,2\}))$$
$$= \gamma(\{E,O\})$$
$$= \{n \mid n \in \mathbb{N}\} \supseteq \{1,2\}$$

$$\alpha(\gamma(\{E\}))$$
$$= \alpha(\{n \mid n \mathit{\ is\ even}\})$$
$$= \{E\} \subseteq \{E\}$$

Connecting Sets

$sound : \forall(N:\wp(\mathbb{N})),\ N \subseteq \gamma(\alpha(N))$
$tight : \forall(P:\wp(\mathbb{P})),\ \alpha(\gamma(P)) \subseteq P$

$\gamma(\alpha(\{1,2\}))$
$= \gamma(\{E,O\})$
$= \{n \mid n \in \mathbb{N}\} \supseteq \{1,2\}$

$\alpha(\gamma(\{E\}))$
$= \alpha(\{n \mid n$ *is even*$\})$
$= \{E\} \subseteq \{E\}$

[alternatively: $\alpha(N) \subseteq P$ *iff* $N \subseteq \gamma(P)$]

AI Specification (sound)

AI Specification (sound)

$$\uparrow succ : \wp(\mathbb{N}) \to \wp(\mathbb{N})$$
$$\uparrow succ(N) \coloneqq \{succ(n) \mid n \in N\}$$

AI Specification (sound)

```
↑succ : ℘(ℕ) → ℘(ℕ)
↑succ(N) ≔ {succ(n) | n ∈ N}
```

```
↑flip : ℘(ℙ) → ℘(ℙ)
↑flip(P) ≔ {flip(p) | p ∈ P}
```

AI Specification (sound)

```
↑succ : ℘(ℕ) → ℘(ℕ)
↑succ(N) ≔ {succ(n) | n ∈ N}

↑flip : ℘(ℙ) → ℘(ℙ)
↑flip(P) ≔ {flip(p) | p ∈ P}

sound :
∀(P:℘(ℙ)), α(↑succ(γ(P))) ⊆ ↑flip(P)
```

AI Specification (sound)

```
↑succ : ℘(ℕ) → ℘(ℕ)
↑succ(N) ≔ {succ(n) | n ∈ N}

↑flip : ℘(ℙ) → ℘(ℙ)
↑flip(P) ≔ {flip(p) | p ∈ P}

sound :
∀(P:℘(ℙ)), α(↑succ(γ(P))) ⊆ ↑flip(P)
```

specification

AI Verification (sound)

AI Verification (sound)

```
∀(P:℘(ℙ)), α(↑succ(γ(P))) ⊆ ↑flip(P)
```

AI Verification (sound)

```
∀(P:℘(ℙ)), α(↑succ(γ(P))) ⊆ ↑flip(P)

Proof by case analysis on P:
  Case [P = {E}]:
   α(↑succ(γ({E})))
   = α(↑succ({n | n is even}))
   = α({succ(n) | n is even})
   = α({n | n is odd})
   = {O}
   = ↑flip({E})
...
```

AI Verification (complete)

```
∀(P:℘(ℙ)), α(↑succ(γ(P))) = ↑flip(P)

Proof by case analysis on P:
  Case [P = {E}]:
   α(↑succ(γ({E})))
   = α(↑succ({n | n is even}))
   = α({succ(n) | n is even})
   = α({n | n is odd})
   = {0}
   = ↑flip({E})
...
```

Issues with Abstract Interpretation

Unwanted Complexity

Unwanted Complexity

```
∀(n:ℕ), parity(succ(n)) = flip(parity(n))
```

vs

```
∀(P:℘(ℙ)), α(↑succ(γ(P))) ⊆ ↑flip(P)
```

Unwanted Complexity

```
∀(n:ℕ), parity(succ(n)) = flip(parity(n))
```

vs

```
∀(P:℘(ℙ)), α(↑succ(γ(P))) ⊆ ↑flip(P)
```

Are these equivalent?

Mechanization Issues

Mechanization Issues

"α is non-constructive"

Mechanization Issues

Problem Verified abstract interpreter

Mechanization Issues

Problem Verified abstract interpreter

Solution Proof assistants and constructive logic

Mechanization Issues

Problem Representing `{n | n is even}`

Mechanization Issues

Problem Representing `{n | n is even}`

Solution $\wp(\mathbb{N}) \coloneqq \mathbb{N} \to$ `prop`

Mechanization Issues

Problem Computable representation for $\wp(\mathbb{P})$

Mechanization Issues

Problem Computable representation for $\wp(\mathbb{P})$

Solution $\wp(\mathbb{P}) \approx \mathbb{P}^+ \coloneqq \{\mathtt{E},\mathtt{O},\bot,\mathtt{T}\}$

Mechanization Issues

Problem α cannot be represented:

$$\alpha \ : \ \wp(\mathbb{N}) \ \rightarrow \ \mathbb{P}^{+}$$

Mechanization Issues

Problem α cannot be represented:

$$\alpha \ : \ \wp(\mathbb{N}) \ \rightarrow \ \mathbb{P}^+$$

Solution Only use γ:

$$\gamma \ : \ \mathbb{P}^+ \ \rightarrow \ \wp(\mathbb{N})$$

Constructive Galois Connections

Constructive GCs

Constructive GCs

- A “Parallel Universe” of Galois connections

Constructive GCs

- A “Parallel Universe” of Galois connections
- Simpler than classical GCs

Constructive GCs

- A “Parallel Universe” of Galois connections
- Simpler than classical GCs
- Interact seamlessly with classical GCs

Constructive GCs

- A “Parallel Universe” of Galois connections
- Simpler than classical GCs
- Interact seamlessly with classical GCs
- Support (constructive) mechanization

Constructive GCs

- A “Parallel Universe” of Galois connections
- Simpler than classical GCs
- Interact seamlessly with classical GCs
- Support (constructive) mechanization
- We’ve used them successfully in two case studies

Constructive GCs

```
parity : ℕ → ℙ
parity(n) ≔ …

⟦_⟧ : ℙ → ℘(ℕ)
⟦p⟧ ≔ …
```

Constructive GCs

```
parity : ℕ → ℙ
parity(n) ≔ …
```

```
α : ℘(ℕ) → ℘(ℙ)
α(N) ≔ {parity(n) | n ∈ N}
```

```
⟦_⟧ : ℙ → ℘(ℕ)
⟦p⟧ ≔ …
```

```
γ : ℘(ℙ) → ℘(ℕ)
γ(P) ≔ {n | p ∈ P ∧ n ∈ ⟦p⟧}
```

Constructive GCs

$$\eta : \mathbb{N} \to \mathbb{P}$$
$$\eta(n) \coloneqq \ldots$$

$$\alpha : \wp(\mathbb{N}) \to \wp(\mathbb{P})$$
$$\alpha(N) \coloneqq \{\eta(n) \mid n \in N\}$$

$$[\![\_]\!] : \mathbb{P} \to \wp(\mathbb{N})$$
$$[\![p]\!] \coloneqq \ldots$$

$$\gamma : \wp(\mathbb{P}) \to \wp(\mathbb{N})$$
$$\gamma(P) \coloneqq \{n \mid p \in P \wedge n \in [\![p]\!]\}$$

Constructive GCs

Constructive

```
η : ℕ → ℙ
η(n) ≔ …

⟦_⟧ : ℙ → ℘(ℕ)
⟦p⟧ ≔ …
```

Classical

```
α : ℘(ℕ) → ℘(ℙ)
α(N) ≔ {η(n) | n ∈ N}

γ : ℘(ℙ) → ℘(ℕ)
γ(P) ≔ {n | p ∈ P ∧ n ∈ ⟦p⟧}
```

Constructive GCs

η α

⟦_⟧ γ

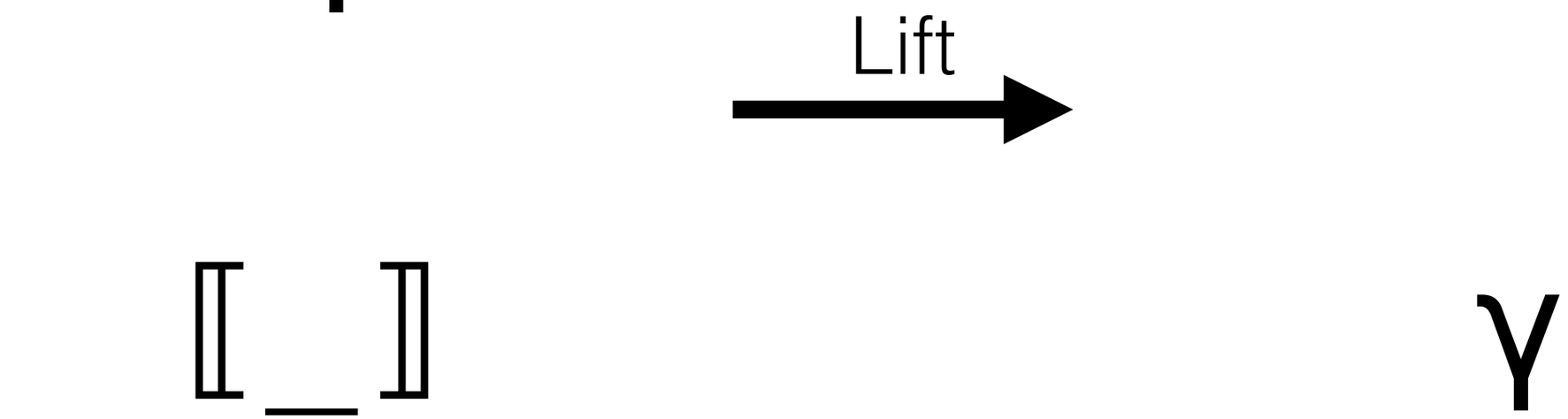
Constructive GCs

η α

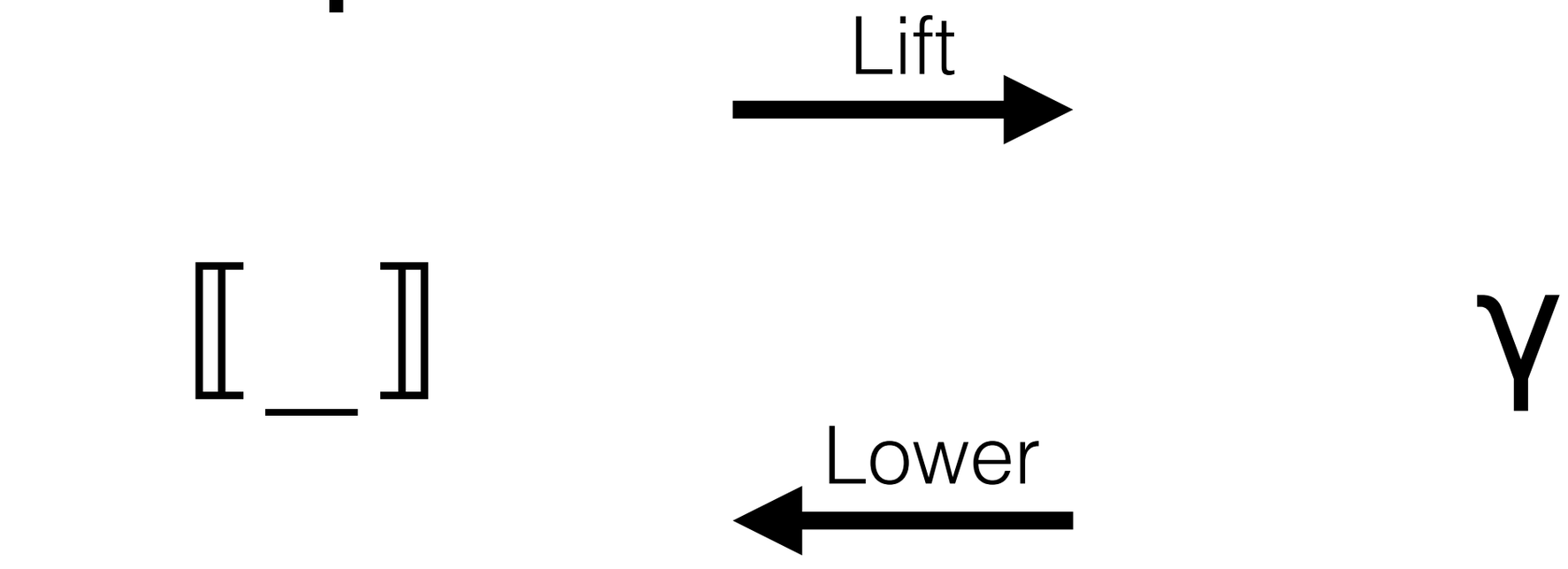
⟦ _ ⟧ γ

…laws… …laws…

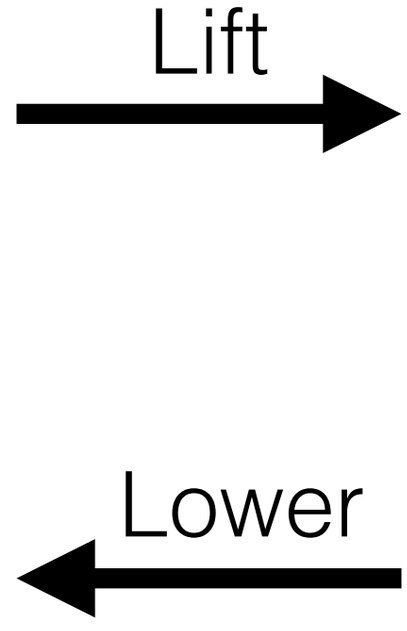
Constructive GCs



Constructive GCs



Constructive GCs



Constructive GCs

Lift

$\models$ `succ` ⇝ `flip` $\models$ `↑succ` ⇝ `↑flip`

Lower

Intuitions

Constructive GCs

$\eta : \mathbb{N} \to \mathbb{P}$

$[\![\_]\!] : \mathbb{P} \to \wp(\mathbb{N})$

Classical GCs

$\alpha : \wp(\mathbb{N}) \to \wp(\mathbb{P})$

$\gamma : \wp(\mathbb{P}) \to \wp(\mathbb{N})$

Intuitions

Constructive GCs

Classical GCs

functorial **`map`**

$\eta : \mathbb{N} \to \mathbb{P}$

$\alpha : \wp(\mathbb{N}) \to \wp(\mathbb{P})$

$[\![\_]\!] : \mathbb{P} \to \wp(\mathbb{N})$

$\gamma : \wp(\mathbb{P}) \to \wp(\mathbb{N})$

Intuitions

Constructive GCs

Classical GCs

functorial **`map`**

$\eta : \mathbb{N} \to \mathbb{P}$ ⟶ $\alpha : \wp(\mathbb{N}) \to \wp(\mathbb{P})$

$[\![\_]\!] : \mathbb{P} \to \wp(\mathbb{N})$ ⟶ $\gamma : \wp(\mathbb{P}) \to \wp(\mathbb{N})$

monadic **`bind`**

Constructive GCs

"Galois connections in the powerset monad"

A ⇄ B

A ⇄ B

Classical

```
α : A ↗ B
γ : B ↗ A


sound: id^A ⊑ γ ∘ α
tight: α ∘ γ ⊑ id^B


sound: ∀(x : A), x ⊑ γ(α(x))
tight: ∀(z : B), α(γ(z)) ⊑ z
```

$A \rightleftarrows B$

Classical

$\alpha : A \nearrow B$
$\gamma : B \nearrow A$

$\mathrm{sound}: \mathrm{id}^A \sqsubseteq \gamma \circ \alpha$
$\mathrm{tight}: \alpha \circ \gamma \sqsubseteq \mathrm{id}^B$

$\mathrm{sound}: \forall (x : A),\ x \sqsubseteq \gamma(\alpha(x))$
$\mathrm{tight}: \forall (z : B),\ \alpha(\gamma(z)) \sqsubseteq z$

Kleisli

$\alpha : A \nearrow \wp(B)$
$\gamma : B \nearrow \wp(A)$

$\mathrm{sound}: \mathrm{return}^A \sqsubseteq \gamma \diamond \alpha$
$\mathrm{tight}: \alpha \diamond \gamma \sqsubseteq \mathrm{return}^B$

$\mathrm{sound}: \forall (x : A),\ \{x\} \subseteq \gamma^*(\alpha(x))$
$\mathrm{tight}: \forall (z : B),\ \alpha^*(\gamma(z)) \subseteq \{z\}$

$A \rightleftarrows B$

Classical

$\alpha : A \nearrow B$
$\gamma : B \nearrow A$

$\text{sound: } \mathrm{id}^A \sqsubseteq \gamma \circ \alpha$
$\text{tight: } \alpha \circ \gamma \sqsubseteq \mathrm{id}^B$

$\text{sound: } \forall (x : A),\ x \sqsubseteq \gamma(\alpha(x))$
$\text{tight: } \forall (z : B),\ \alpha(\gamma(z)) \sqsubseteq z$

Kleisli

$\alpha : A \nearrow \wp(B)$
$\gamma : B \nearrow \wp(A)$

$\text{sound: } \mathrm{return}^A \sqsubseteq \gamma \diamond \alpha$
$\text{tight: } \alpha \diamond \gamma \sqsubseteq \mathrm{return}^B$

$\text{sound: } \forall (x : A),\ \{x\} \subseteq \gamma^*(\alpha(x))$
$\text{tight: } \forall (z : B),\ \alpha^*(\gamma(z)) \subseteq \{z\}$

For Classical, A is typically instantiated to $\wp(A)$

$A \rightleftarrows B$

Classical

$\alpha : \wp(A) \nearrow B$
$\gamma : B \nearrow \wp(A)$

$\mathrm{sound}: \mathrm{id}^A \subseteq \gamma \circ \alpha$
$\mathrm{tight}: \alpha \circ \gamma \sqsubseteq \mathrm{id}^B$

$\mathrm{sound}: \forall(X : \wp(A)),\ X \subseteq \gamma(\alpha(X))$
$\mathrm{tight}: \forall(z : B),\ \alpha(\gamma(z)) \sqsubseteq z$

Kleisli

$\alpha : A \nearrow \wp(B)$
$\gamma : B \nearrow \wp(A)$

$\mathrm{sound}: \mathrm{return}^A \sqsubseteq \gamma \diamond \alpha$
$\mathrm{tight}: \alpha \diamond \gamma \sqsubseteq \mathrm{return}^B$

$\mathrm{sound}: \forall(x : A),\ \{x\} \subseteq \gamma^*(\alpha(x))$
$\mathrm{tight}: \forall(z : B),\ \alpha^*(\gamma(z)) \subseteq \{z\}$

For Classical, A is typically instantiated to $\wp(A)$

A ⇄ B

Kleisli

```
α : A ↗ ℘(B)
γ : B ↗ ℘(A)


sound: return^A ⊑ γ ⋄ α
tight: α ⋄ γ ⊑ return^B


sound: ∀(x : A), {x} ⊆ γ*(α(x))
tight: ∀(z : B), α*(γ(z)) ⊆ {z}
```

A ⇄ B

"α has no monadic effect"

Kleisli

```
α : A ↗ ℘(B)
γ : B ↗ ℘(A)


sound: return^A ⊑ γ ⋄ α
tight: α ⋄ γ ⊑ return^B


sound: ∀(x : A), {x} ⊆ γ*(α(x))
tight: ∀(z : B), α*(γ(z)) ⊆ {z}
```

A ⇄ B

"α has no monadic effect"

$$\exists(\eta : A \nearrow B),$$

$$\alpha = \lambda x.\{\eta(x)\}$$

Kleisli

$$\alpha : A \nearrow \wp(B)$$
$$\gamma : B \nearrow \wp(A)$$

$$\text{sound: } \text{return}^A \sqsubseteq \gamma \diamond \alpha$$
$$\text{tight: } \alpha \diamond \gamma \sqsubseteq \text{return}^B$$

$$\text{sound: } \forall(x : A),\ \{x\} \subseteq \gamma^*(\alpha(x))$$
$$\text{tight: } \forall(z : B),\ \alpha^*(\gamma(z)) \subseteq \{z\}$$

A ⇄ B

"α has no monadic effect"

Kleisli

```
α : A ↗ ℘(B)
γ : B ↗ ℘(A)

sound: return^A ⊑ γ ⋄ α
tight: α ⋄ γ ⊑ return^B

sound: ∀(x : A), {x} ⊆ γ*(α(x))
tight: ∀(z : B), α*(γ(z)) ⊆ {z}
```

```
∃(η : A ↗ B),

α = λx.{η(x)}
```

```
sound: ∀(x : A), ∃(z : B), z ∈ α(x) ∧ x ∈ γ(z)
```

$A \rightleftarrows B$

Kleisli

$\alpha : A \nearrow \wp(B)$
$\gamma : B \nearrow \wp(A)$

$\text{sound: } \text{return}^A \sqsubseteq \gamma \diamond \alpha$
$\text{tight: } \alpha \diamond \gamma \sqsubseteq \text{return}^B$

$\text{sound: } \forall(x : A),\ \{x\} \subseteq \gamma^*(\alpha(x))$
$\text{tight: } \forall(z : B),\ \alpha^*(\gamma(z)) \subseteq \{z\}$

Constructive

$\eta : A \nearrow B$
$\gamma : B \nearrow \wp(A)$

$\text{sound: } \text{return}^A \sqsubseteq \gamma \diamond \alpha$
$\text{tight: } \alpha \diamond \gamma \sqsubseteq \text{return}^B$

$\text{sound: } \forall(x : A),\ \{x\} \subseteq \gamma^*(\alpha(x))$
$\text{tight: } \forall(z : B),\ \alpha^*(\gamma(z)) \subseteq \{z\}$

For Constructive GC, $\alpha \coloneqq \lambda x.\{\eta(x)\}$

$A \rightleftarrows B$

Constructive

$\eta : A \nearrow B$
$\gamma : B \nearrow \wp(A)$

$\text{sound: } \text{return}^A \sqsubseteq \gamma \diamond \alpha$
$\text{tight: } \alpha \diamond \gamma \sqsubseteq \text{return}^B$

$\text{sound: } \forall(x : A),\ \{x\} \subseteq \gamma^*(\alpha(x))$
$\text{tight: } \forall(z : B),\ \alpha^*(\gamma(z)) \subseteq \{z\}$

For Constructive GC, $\alpha \coloneqq \lambda x.\{\eta(x)\}$

A ⇄ B

sound:
x ∈ γ(η(x))

Constructive

```
η : A ↗ B
γ : B ↗ ℘(A)


sound: return^A ⊑ γ ⊙ α
tight: α ⊙ γ ⊑ return^B


sound: ∀(x : A), {x} ⊆ γ*(α(x))
tight: ∀(z : B), α*(γ(z)) ⊆ {z}
```

For Constructive GC, α ≔ λx.{η(x)}

$A \rightleftarrows B$

sound:
$x \in \gamma(\eta(x))$

tight:
$x \in \gamma(z) \Rightarrow \eta(x) \sqsubseteq z$

Constructive

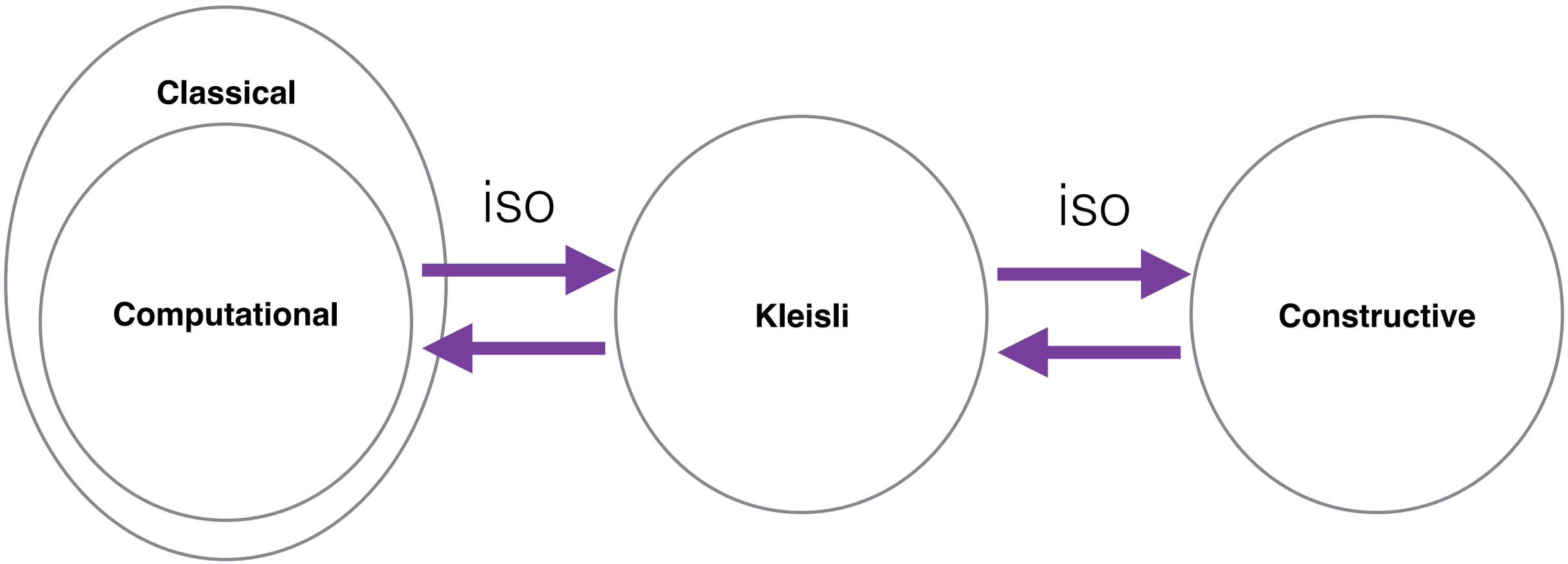
$\eta : A \nearrow B$
$\gamma : B \nearrow \wp(A)$

sound: $\text{return}^A \sqsubseteq \gamma \diamond \alpha$
tight: $\alpha \diamond \gamma \sqsubseteq \text{return}^B$

sound: $\forall(x : A), \{x\} \subseteq \gamma^*(\alpha(x))$
tight: $\forall(z : B), \alpha^*(\gamma(z)) \subseteq \{z\}$

For Constructive GC, $\alpha \coloneqq \lambda x.\{\eta(x)\}$

Relationships



Using CGCs

Using CGCs

Define $\gamma : B \nearrow \wp(A)$ just as before

Using CGCs

Define $\gamma : B \nearrow \wp(A)$ just as before

Define $\eta : A \nearrow B$ instead of $\alpha : \wp(A) \nearrow B$

Using CGCs

Define $\gamma : B \nearrow \wp(A)$ just as before

Define $\eta : A \nearrow B$ instead of $\alpha : \wp(A) \nearrow B$

Lift proofs of soundness for free (through isomorphisms)

Using CGCs

Define $\gamma : B \nearrow \wp(A)$ just as before

Define $\eta : A \nearrow B$ instead of $\alpha : \wp(A) \nearrow B$

Lift proofs of soundness for free (through isomorphisms)

Interact with classical GCs (through isomorphisms)

Using CGCs

Define $\gamma : B \nearrow \wp(A)$ just as before

Define $\eta : A \nearrow B$ instead of $\alpha : \wp(A) \nearrow B$

Lift proofs of soundness for free (through isomorphisms)

Interact with classical GCs (through isomorphisms)

Supports synthesis of correct-by-construction static analyzers via *calculational abstract interpretation*

Using CGCs

Define $\gamma : B \nearrow \wp(A)$ just as before

Define $\eta : A \nearrow B$ instead of $\alpha : \wp(A) \nearrow B$

Lift proofs of soundness for free (through isomorphisms)

Interact with classical GCs (through isomorphisms)

Supports synthesis of correct-by-construction static analyzers via *calculational abstract interpretation*

η and γ are constructive, and amenable to mechanization and verified program extraction

Using CGCs

Using CGCs

```
∀(n:ℕ), parity(succ(n)) = flip(parity(n))
```

vs

```
∀(P:℘(ℙ)), α(↑succ(γ(P))) ⊆ ↑flip(P)
```

Using CGCs

```
∀(n:ℕ), parity(succ(n)) = flip(parity(n))
```

vs

```
∀(P:℘(ℙ)), α(↑succ(γ(P))) ⊆ ↑flip(P)
```

Are these equivalent?

Using CGCs

$$\forall(n:\mathbb{N}),\ \mathrm{parity}(\mathrm{succ}(n)) = \mathrm{flip}(\mathrm{parity}(n))$$

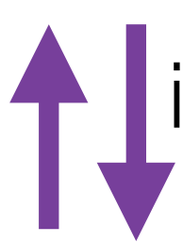
vs

$$\forall(P:\wp(\mathbb{P})),\ \alpha(\uparrow\mathrm{succ}(\gamma(P))) \subseteq \uparrow\mathrm{flip}(P)$$

Are these equivalent? **Yes!**

Using CGCs

```
∀(n:ℕ), parity(succ(n)) = flip(parity(n))
```

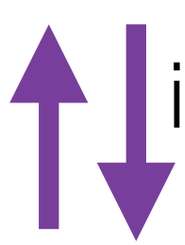
vs  iso

```
∀(P:℘(ℙ)), α(↑succ(γ(P))) ⊆ ↑flip(P)
```

Are these equivalent? **Yes!**

Using CGCs

```
∀(n:ℕ), parity(succ(n)) = flip(parity(n))
```

vs  iso

```
∀(P:℘(ℙ)), α(↑succ(γ(P))) ⊆ ↑flip(P)
```

Are these equivalent? **Yes!**
Also **simpler** and **mechanizable**

Two Case Studies

- (1) A **synthesized abstract interpreter** for an imperative programming language
- (2) A **gradual type system** built on abstract interpretation
- Mechanized in Agda; would not have been possible using classical GCs in both cases
- Simpler proofs and calculations in both cases

My Toolbox

AAM

AAM

"The hard part of static analysis is
defining the right *concrete* interpreter"
-Me *[wisdom of MM+DVH]*

A(AAM)

A(AAM)

"Want a language with feature X?
Write an interpreter in a meta-language with X."
-Me *[wisdom of MM+DVH+MH+WB]*

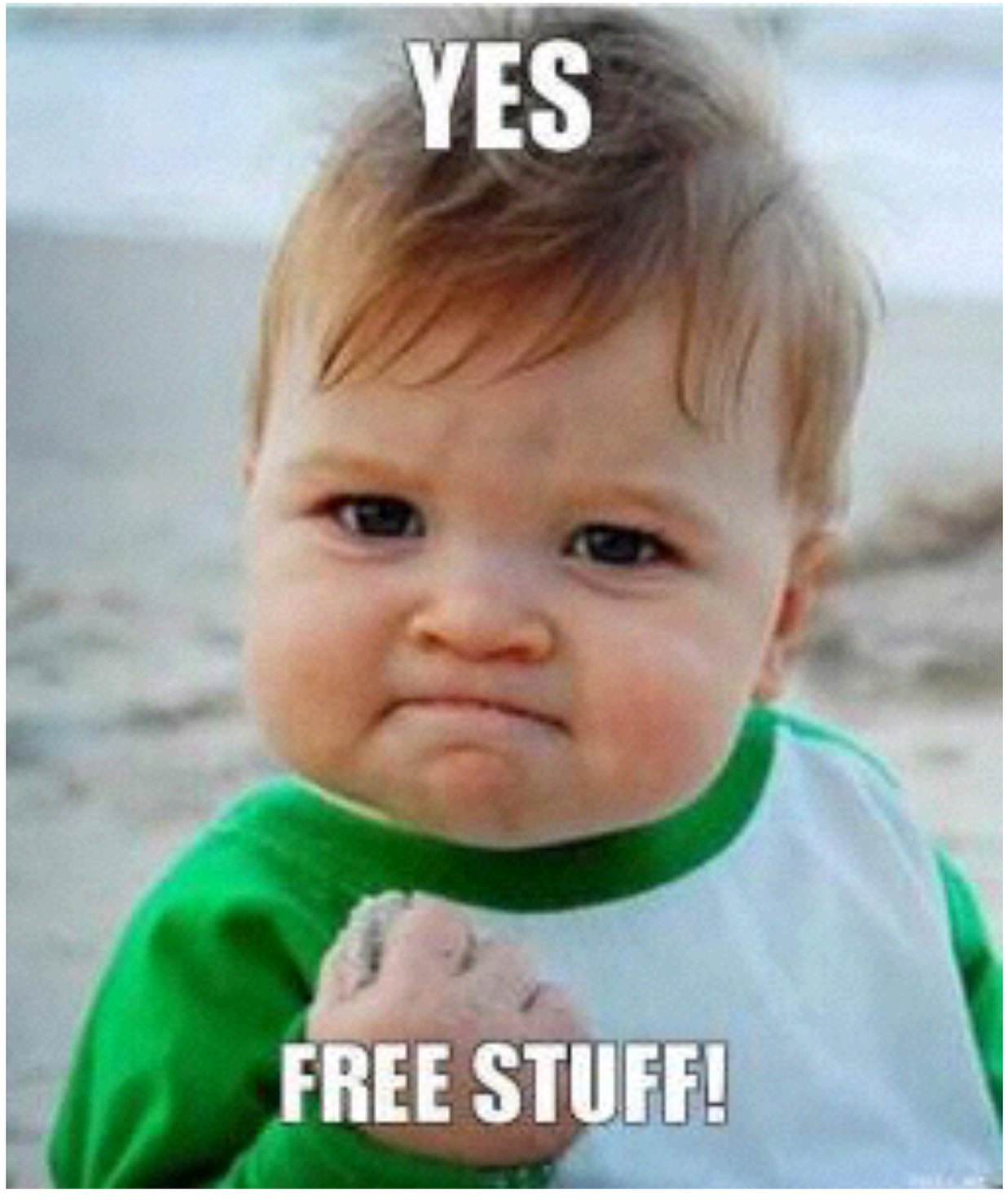
Proof Assistants

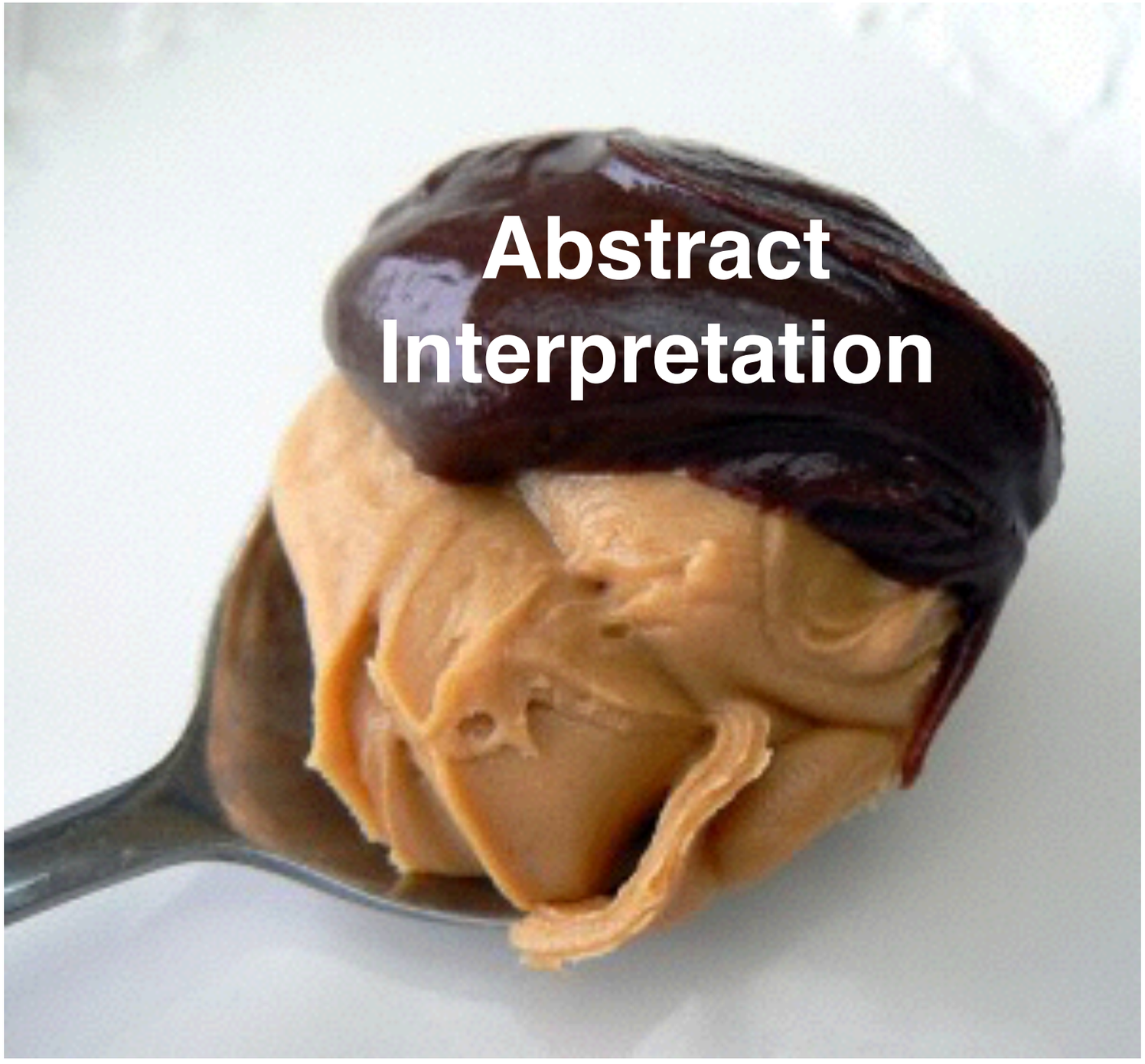
Proof Assistants

“Mechanize tiny and huge formalisms.
LaTeX anything in-between.”
-Me


MONADS
MONADS EVERYWHERE
imgflip.com

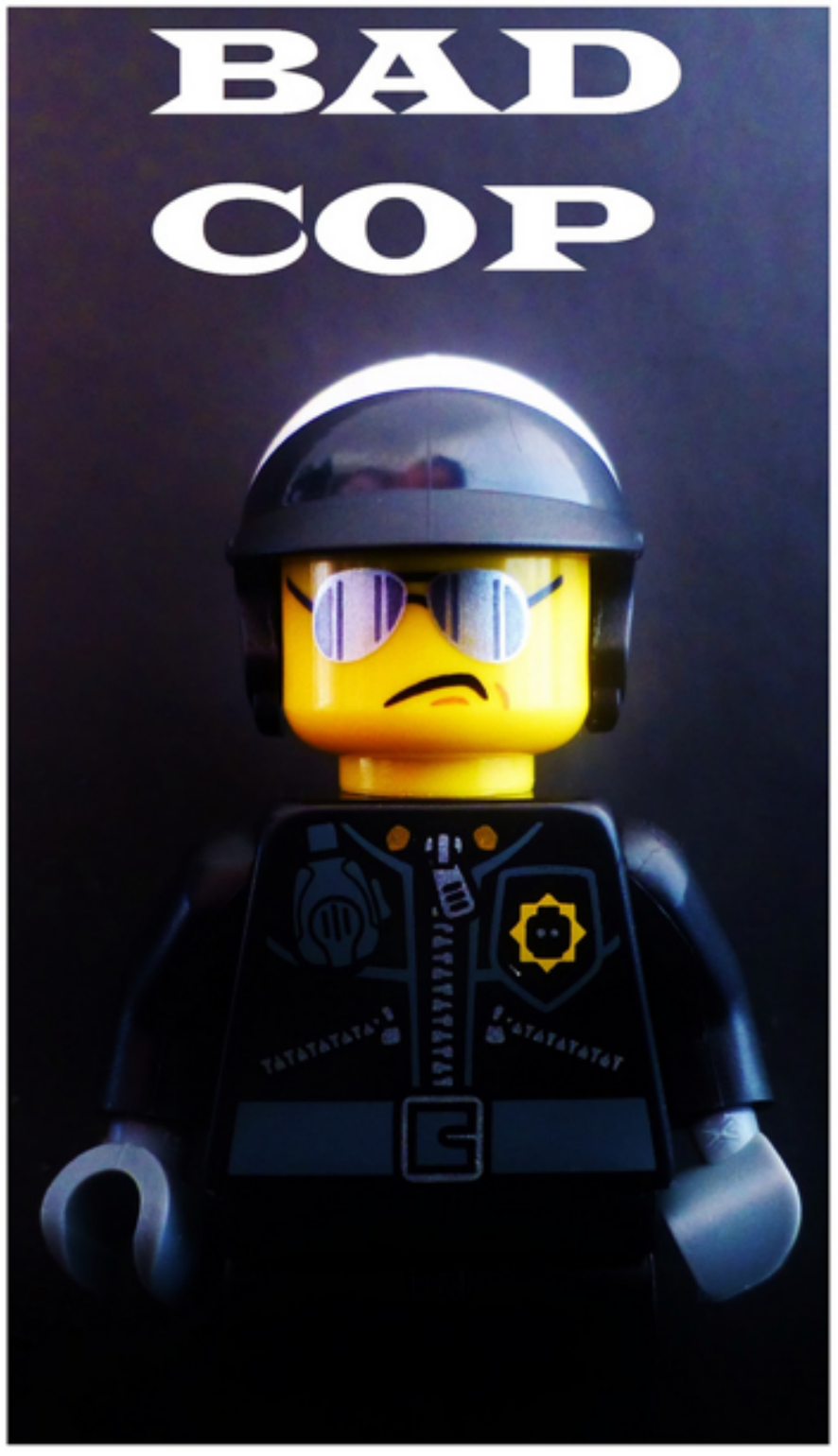
Galois Connections
+
Monotonicity


YES
FREE STUFF!


Abstract
Interpretation


BANG HEAD HERE


GOOD
COP


BAD
COP


HATERS
GONNA HATE
memegenerator.net