Galois Transformers and Modular Abstract Interpreters

Reusable Metatheory for Program Analysis

David Darais
University of Maryland

Matthew Might
University of Utah

David Van Horn
University of Maryland

Program Analysis

Program Analysis

- Lots of choices when designing a program analysis

Program Analysis

- Lots of choices when designing a program analysis
- Choices make tradeoffs between *precision* and *performance*

Program Analysis

- Lots of choices when designing a program analysis
- Choices make tradeoffs between *precision* and *performance*
- Implementations are brittle and difficult to change

Program Analysis

- Lots of choices when designing a program analysis
- Choices make tradeoffs between *precision* and *performance*
- Implementations are brittle and difficult to change
- **Galois Transformers**:

  Reusable components for building program analyzers

Program Analysis

- Lots of choices when designing a program analysis
- Choices make tradeoffs between *precision* and *performance*
- Implementations are brittle and difficult to change
- **Galois Transformers**:

  Reusable components for building program analyzers
- **Bonus**:

  Variations in path/flow sensitivity of your analyzer for free

Let's Design an Analysis

(in the paradigm of abstract interpretation)

Let's Design an Analysis

Program

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

Let's Design an Analysis

Program

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

Analysis Property

$x/0$

Let's Design an Analysis

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

$$[\![e]\!] \in [\![\mathrm{analyze}(e)]\!]$$

Let's Design an Analysis

Program

```
int x y; // global state
void safe_fun(int N) {
 if (N≠0) {x := 0;}
 else     {x := 1;}
 if (N≠0) {y := 100/N;}
 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 ∈ {-,0,+}
x ∈ {0,+}
y ∈ {-,0,+}

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

Prove Correct

$[\![e]\!] \in [\![\text{analyze}(e)]\!]$

Let's Design an Analysis

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

```
N ∈ {-,0,+}
x ∈ {0,+}
y ∈ {-,0,+}

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

Flow-insensitive

results :
var ↦ $\mathcal{P}$({-,0,+})

Let's Design an Analysis

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

Flow-sensitive

results :
loc ↦ (var ↦ $\mathcal{P}$({-,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;}}
```

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

N,y ∈ {-,0,+}

UNSAFE: {100/x}
```

Flow-sensitive

results :
loc ↦ (var ↦ $\mathcal{P}$({-,0,+}))

Let's Design an Analysis

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

Path-sensitive

results :
$\mathrm{loc} \mapsto \mathcal{P}(\mathrm{var} \mapsto \mathcal{P}(\{-, 0, +\}))$

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;}}
```

```
4: N∈{-,+},x∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}
```

SAFE

Path-sensitive

results :
loc ↦ $\mathcal{P}$(var ↦ $\mathcal{P}$({-,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∈{-,+},y∈{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 : τ) :=
    .. x .. τ ..
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 : ...) :=
    .. x .. ..
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

$[\![e]\!] \in [\![\mathrm{analyze}(e)]\!]$

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

Monad Transformers + Galois Connections = Galois Transformers

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 operation
- *M*`(t)`:
  - "A computation that performs some effects, then returns `t`"

A Monadic Interpreter

Program

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

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

A Monadic Interpreter

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

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

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

A Monadic Interpreter

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

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

```
type M(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail
```

A Monadic Interpreter

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

```
value := ℤ ∪ 𝔹
ρ ∈ env := var ↦ 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

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

```
int x y; // global state
void safe_fun(int N) {
 if (N≠0) {x := 0;}
 else     {x := 1;}
 if (N≠0) {y := 100/N;}
 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⟧
  case v of
    True → return(e₁)
    False → return(e₂)
    _  → fail
```

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

⟦_⟧ : atom → M(value)
```

```
type M(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail
```

Abstractify

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

```
int x y; // global state
void safe_fun(int N) {
  if (N≠0) {x := 0;}
  else     {x := 1;}
  if (N≠0) {y := 100/N;}
  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⟧♯
  case v of
    True  → return(e₁)
    False → return(e₂)
    _     → fail
```

$\texttt{value}^\sharp := \mathcal{P}(\{-,0,+\}) \cup \mathcal{P}(\mathbb{B})$
$\rho \in \texttt{env}^\sharp := \texttt{var} \mapsto \texttt{value}^\sharp$

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

```
type M♯(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail
```

Abstractify

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

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

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

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

```
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)
```

$$\mathrm{value}^\sharp := \mathcal{P}(\{-,0,+\}) \cup \mathcal{P}(\mathbb{B})$$
$$\rho \in \mathrm{env}^\sharp := \mathrm{var} \mapsto \mathrm{value}^\sharp$$

$$[\![\_]\!]^\sharp : \mathrm{atom} \to M^\sharp(\mathrm{value}^\sharp)$$
$$\mathrm{chooseBool} : \mathrm{value}^\sharp \to M^\sharp(\mathbb{B})$$

```
type M♯(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail
```

Abstractify

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

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

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

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

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

```
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`$^\sharp$, $[\![\_]\!]^\sharp$)
- Join results when updating `env`$^\sharp$ ($\_\sqcup\_$)
- Branch nondeterministically (`chooseBool`)

Why Monads

- A monadic interpreter can be simpler than a state machine or constraint system
- Two effects, `State[`$\mathcal{s}$`]` 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[`$\mathcal{S}$`]` 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[`$\mathcal{S}$`]`

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

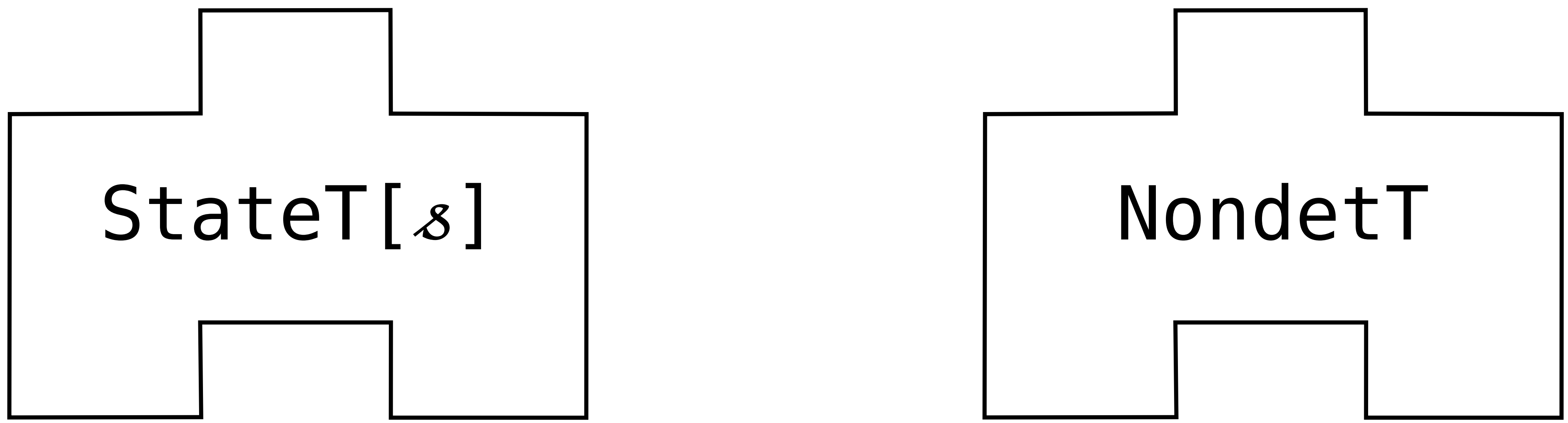
$\texttt{put} : \mathcal{S} \to M(1)$

`Nondet`

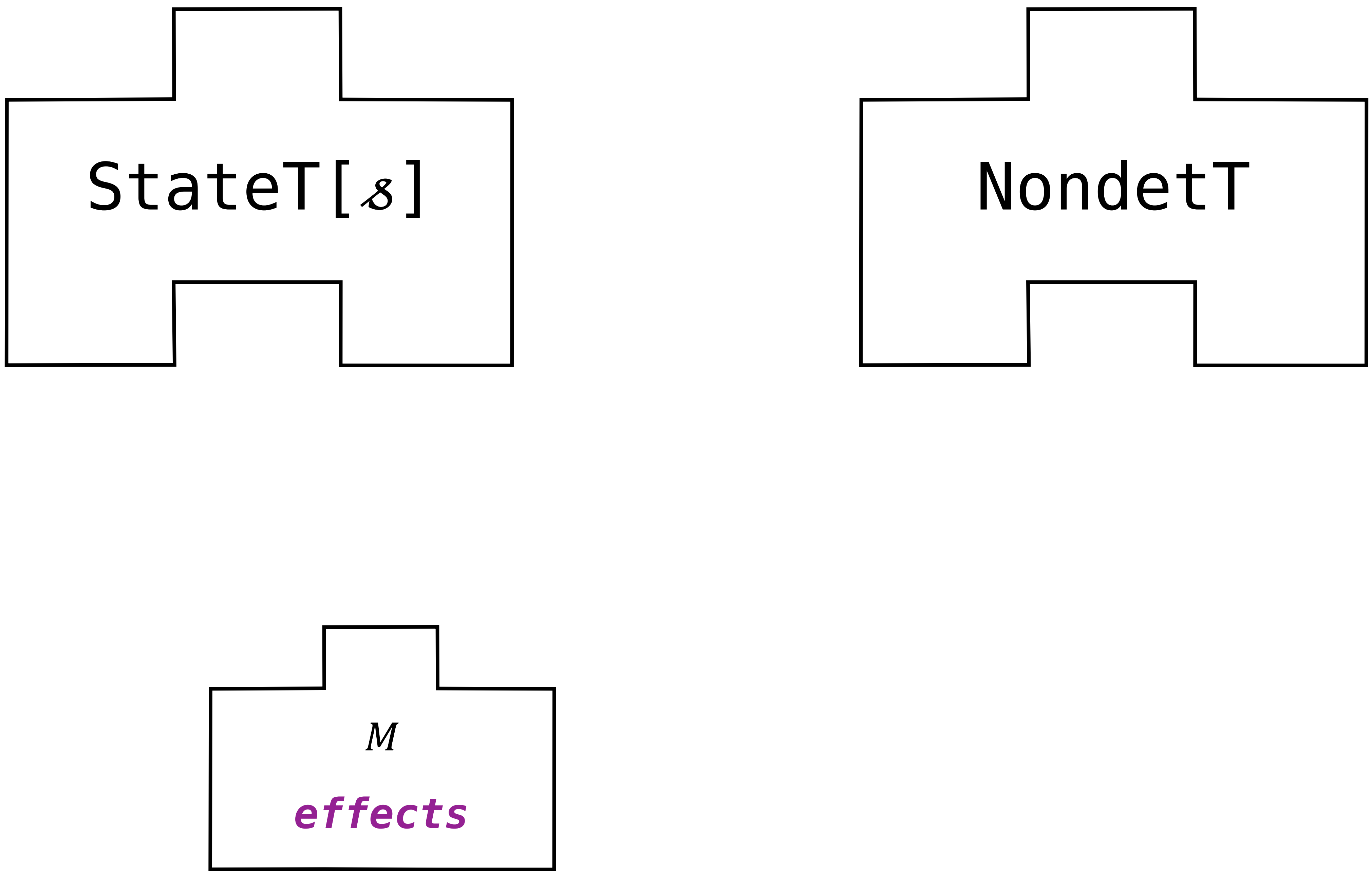
$\texttt{fail} : \forall\, \mathtt{A}.\ M(\mathtt{A})$

$\texttt{\_}\boxplus\texttt{\_} : \forall\, \mathtt{A}.\ M(\mathtt{A}) \times M(\mathtt{A}) \to M(\mathtt{A})$

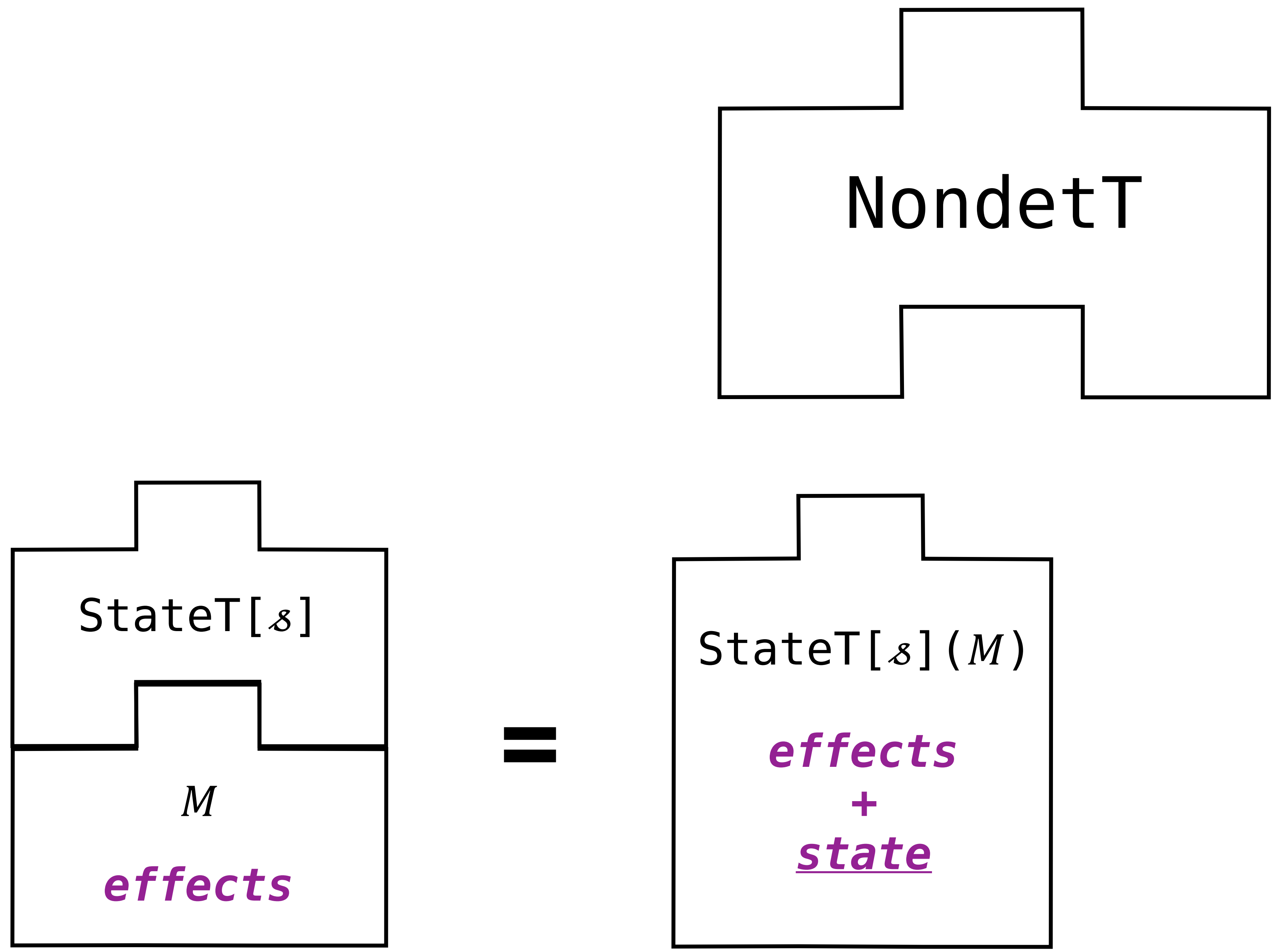
Monad Transformers



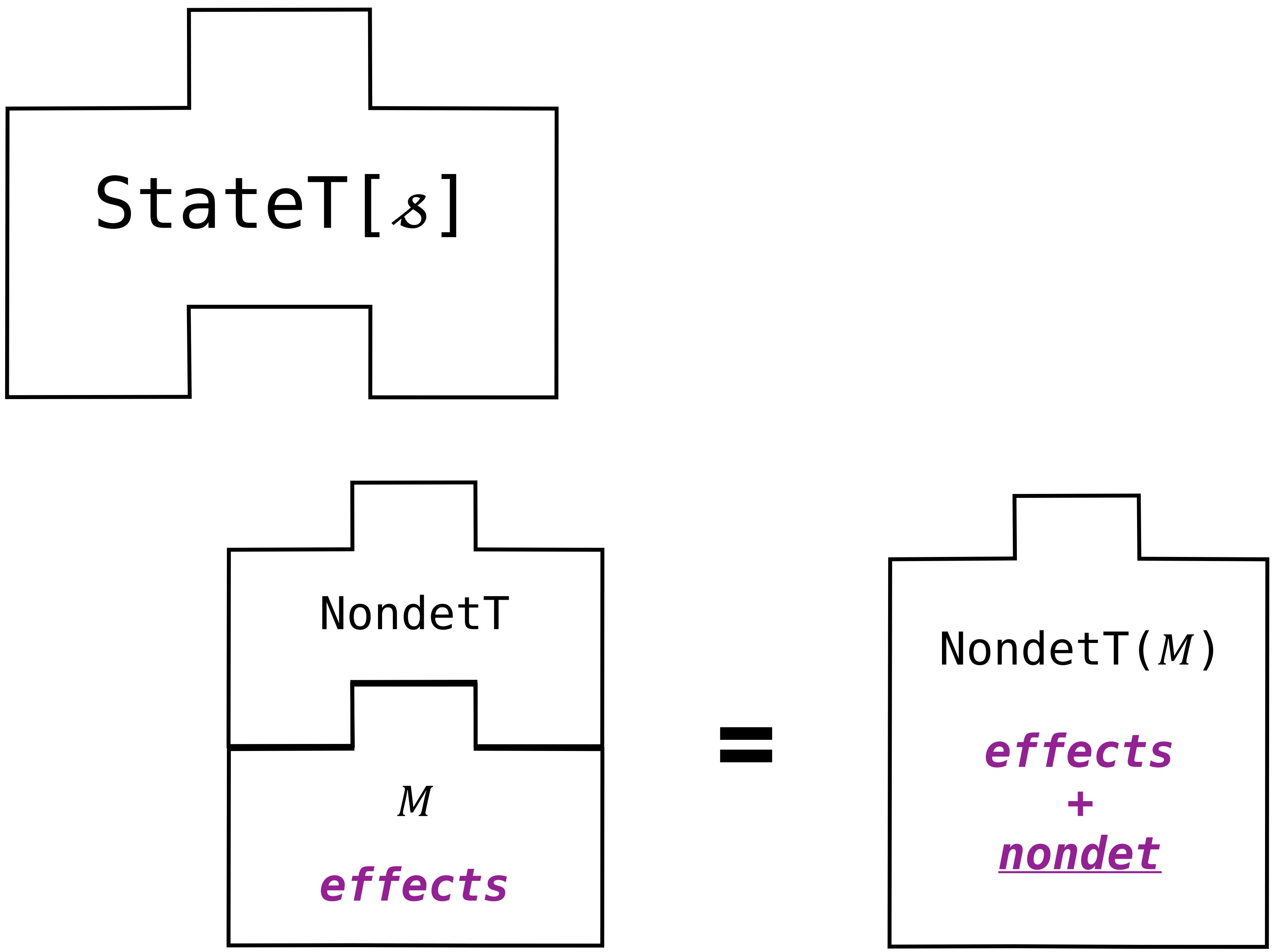
Monad Transformers



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
```

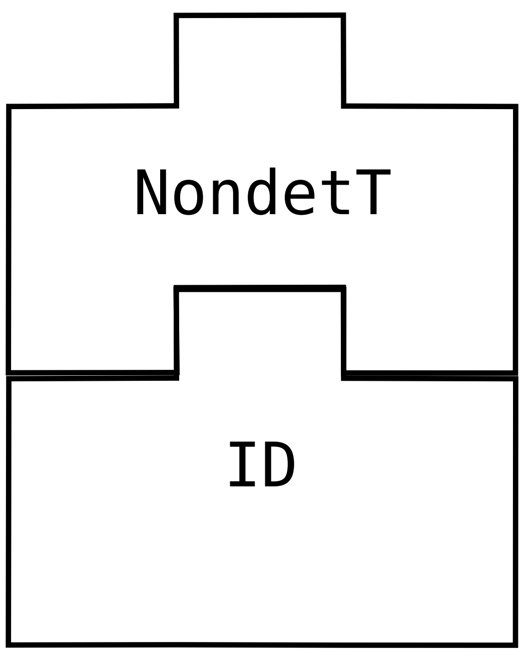
Monad Transformers

ID

=

```
type M(t)

op x ← e1 ; e2
op return(e)

op getEnv
op putEnv(e)

op fail/e1⊞e2
```

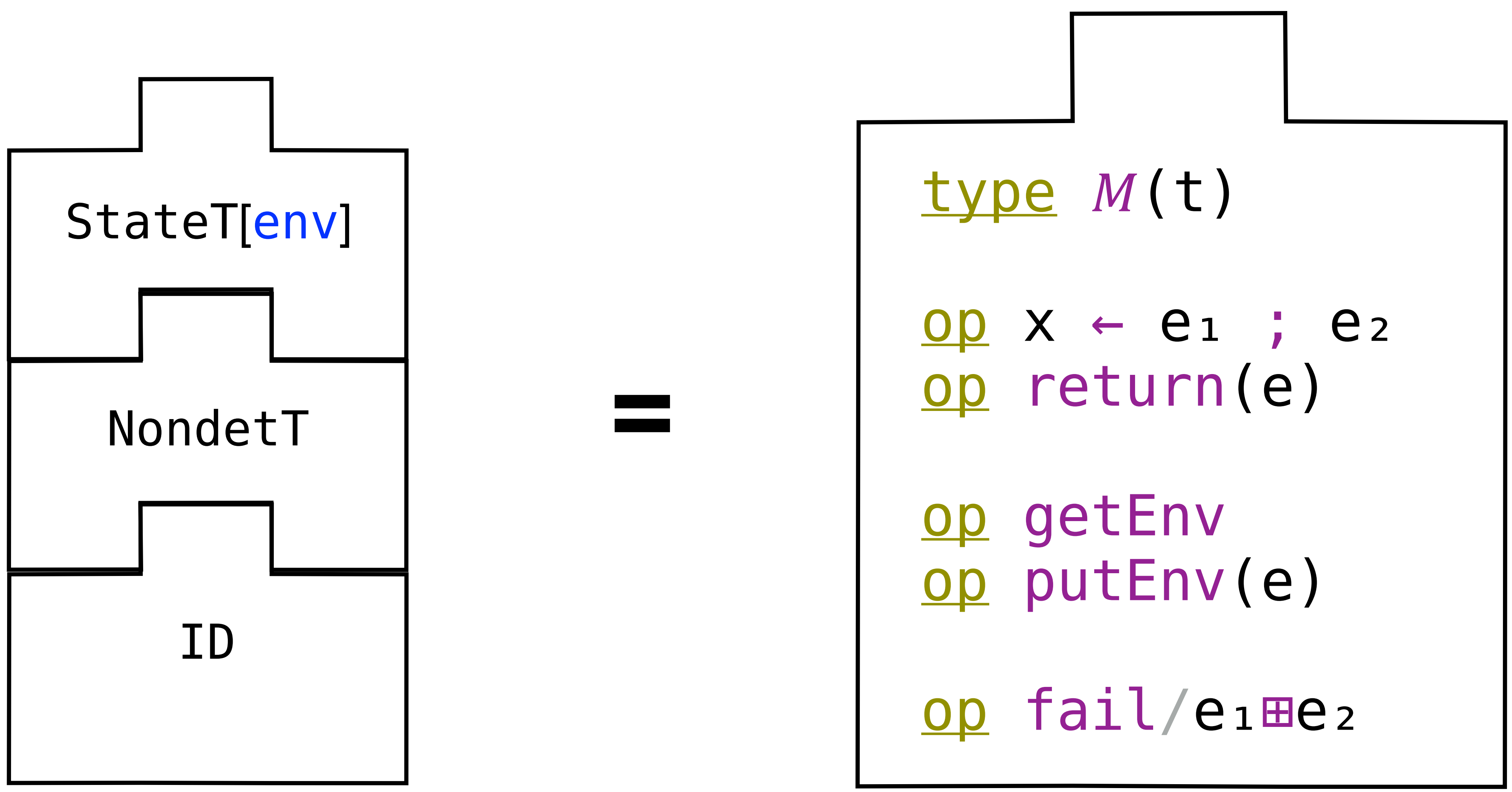
Monad Transformers



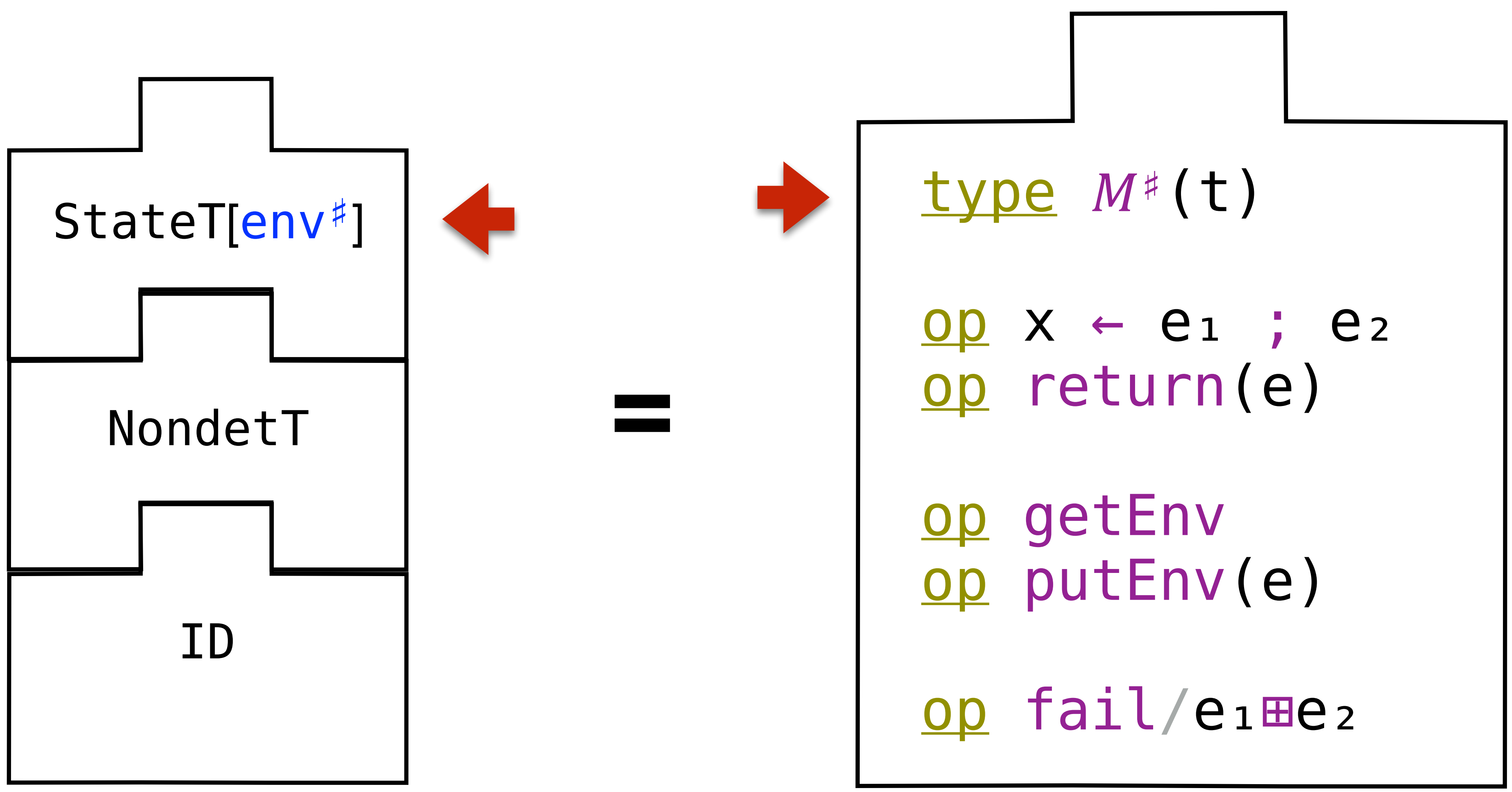
=

```
type M(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail/e₁⊞e₂
```

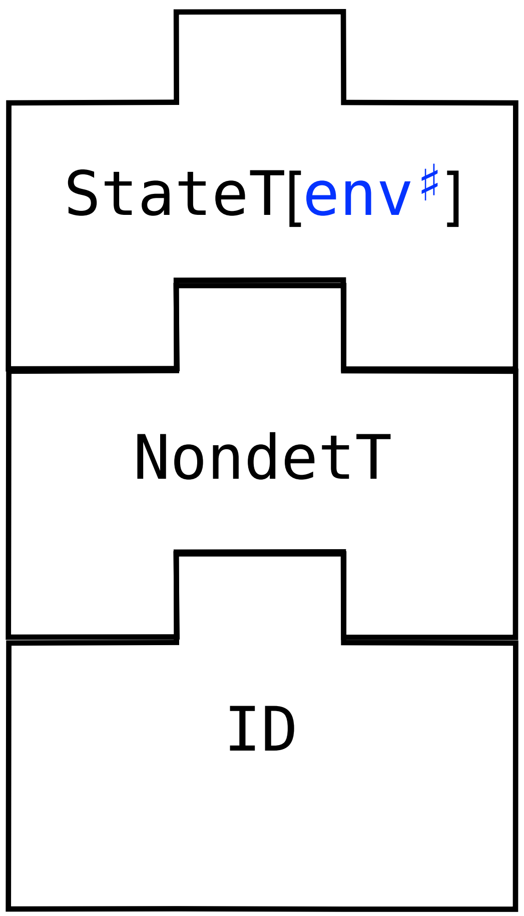
Monad Transformers



Monad Transformers



Monad Transformers

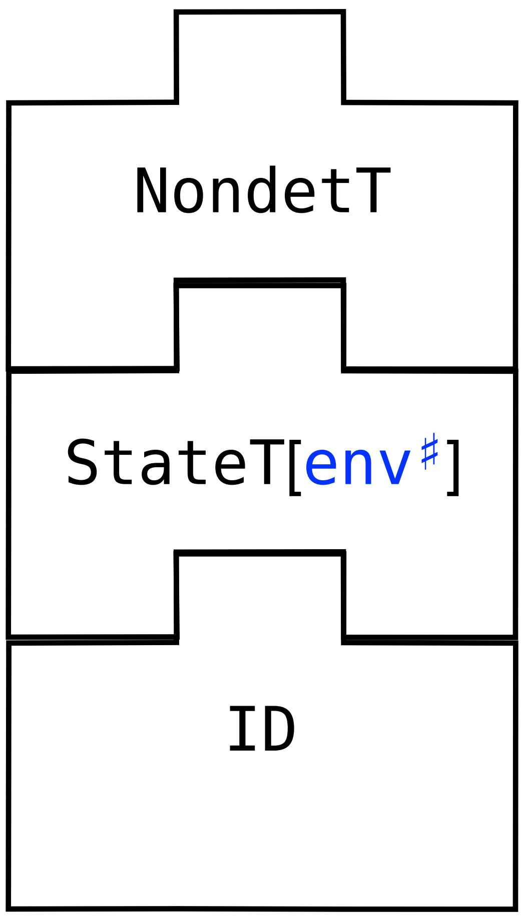


=

```
type M♯(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail/e₁⊞e₂
```

Path-sensitive

Monad Transformers

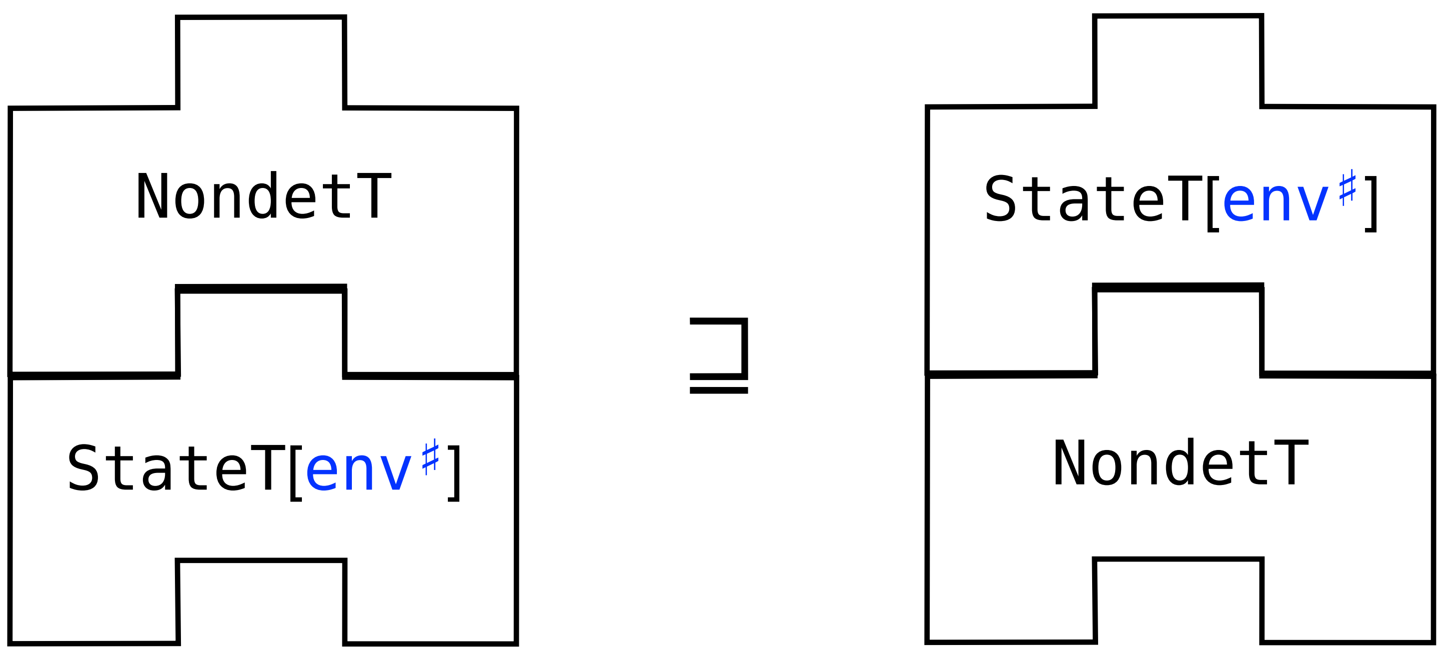


=

```
type M♯(t)

op x ← e₁ ; e₂
op return(e)

op getEnv
op putEnv(e)

op fail/e₁⊞e₂
```

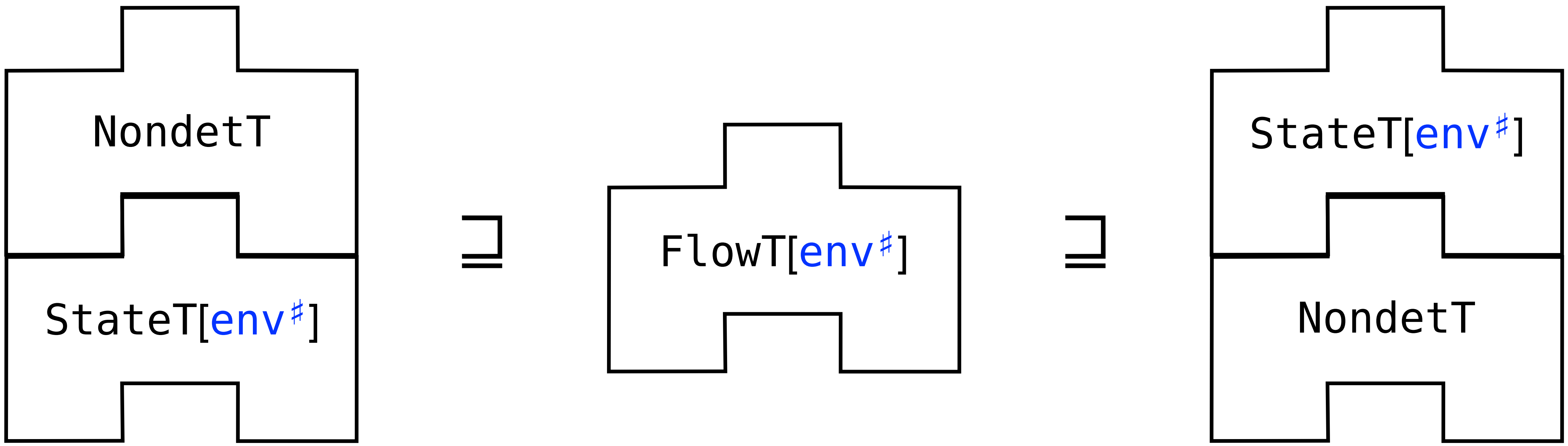
Flow-insensitive

Monad Transformers



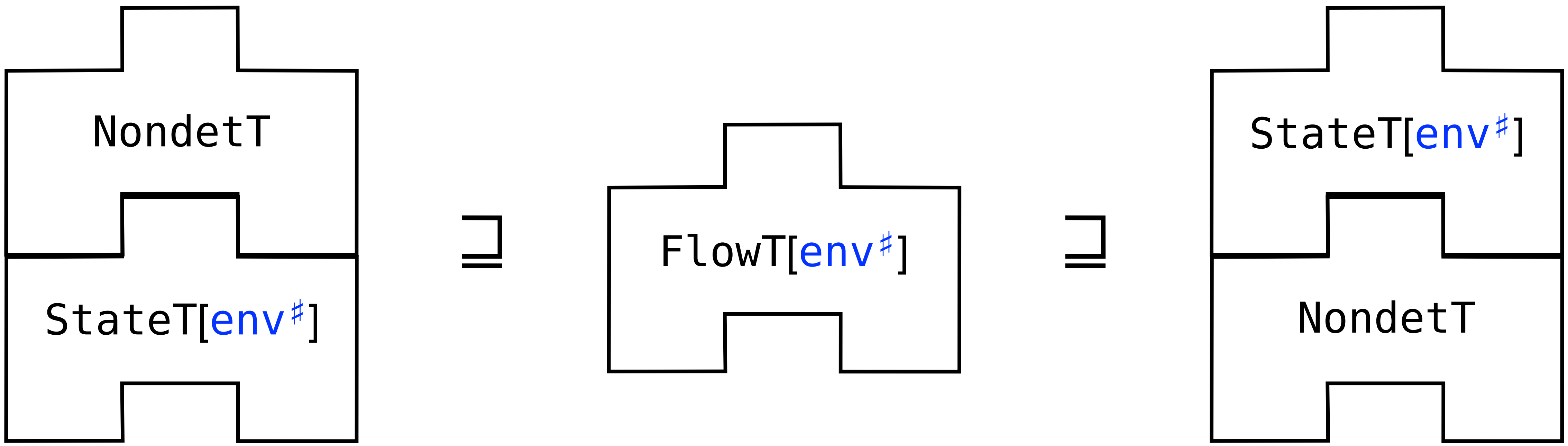
Flow-insensitive *Path-sensitive*

Monad Transformers



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

Monad Transformers



Flow-insensitive	*Flow-sensitive*	*Path-sensitive*
$\mathcal{P}(\mathtt{exp}) \times \mathtt{env}^{\sharp}$	$\mathtt{exp} \mapsto \mathtt{env}^{\sharp}$	$\mathtt{exp} \mapsto \mathcal{P}(\mathtt{env}^{\sharp})$

Monad Transformers

Flow-insensitive	*Flow-sensitive*	*Path-sensitive*
$\mathcal{P}(\text{exp}) \times \text{env}^\sharp$	$\text{exp} \mapsto \text{env}^\sharp$	$\text{exp} \mapsto \mathcal{P}(\text{env}^\sharp)$

```
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[`$\mathcal{S}$`]`, `FlowT[`$\mathcal{S}$`]` 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

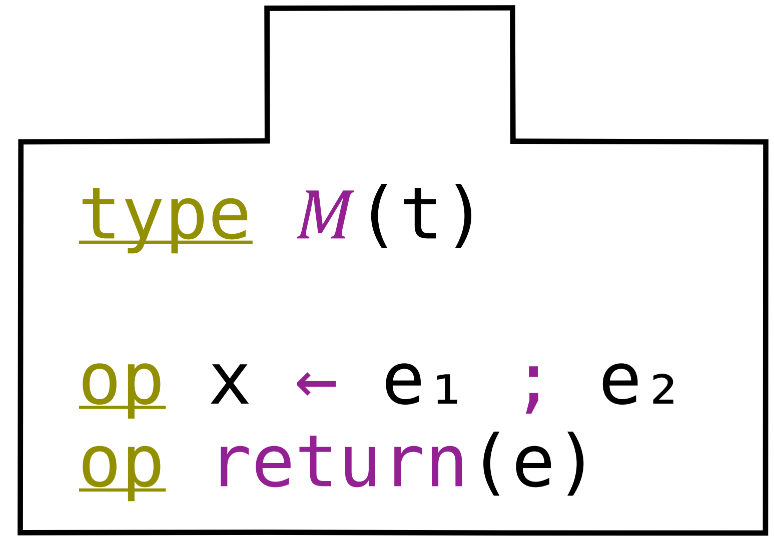
Galois Transformers

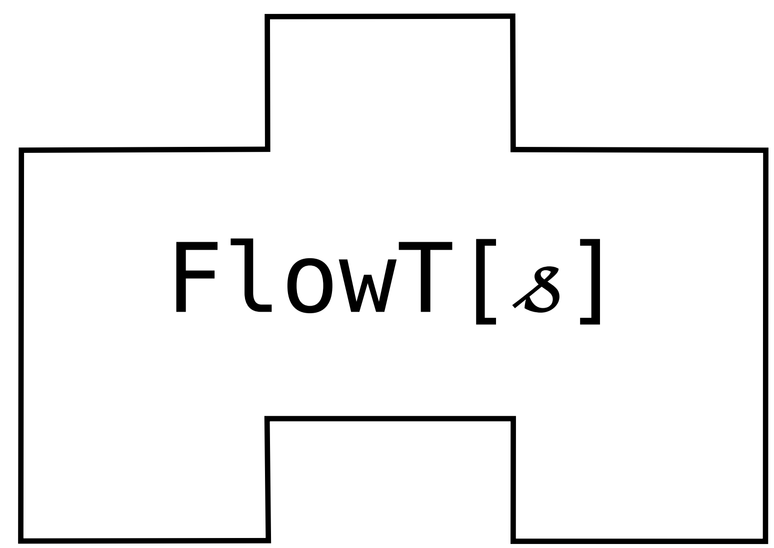
- What's a Monad?
- What are Transformers?
- What are Galois Connections?

```
type M(t)

op x ← e₁ ; e₂
op return(e)
```

Galois Transformers

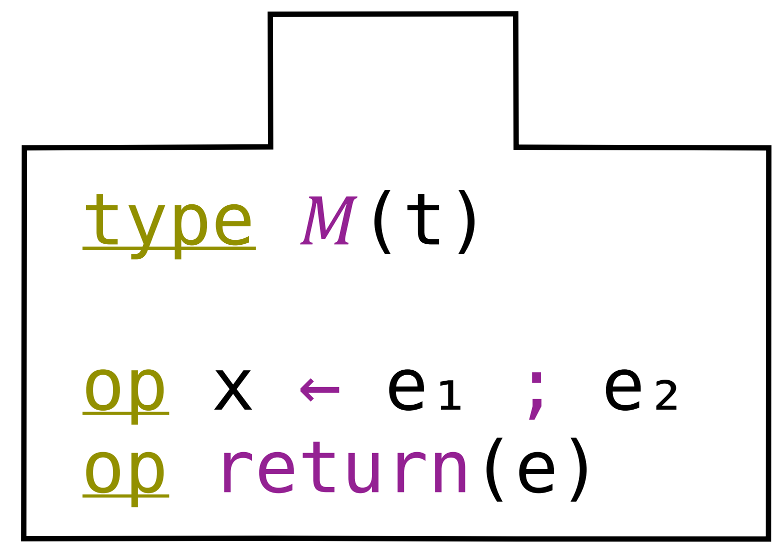
- What's a Monad?
- What are Transformers?
- What are Galois Connections?

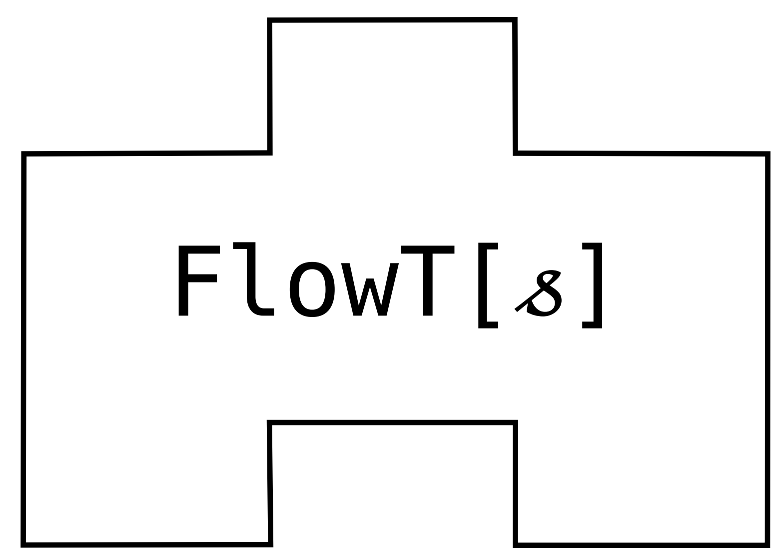




Galois Transformers

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

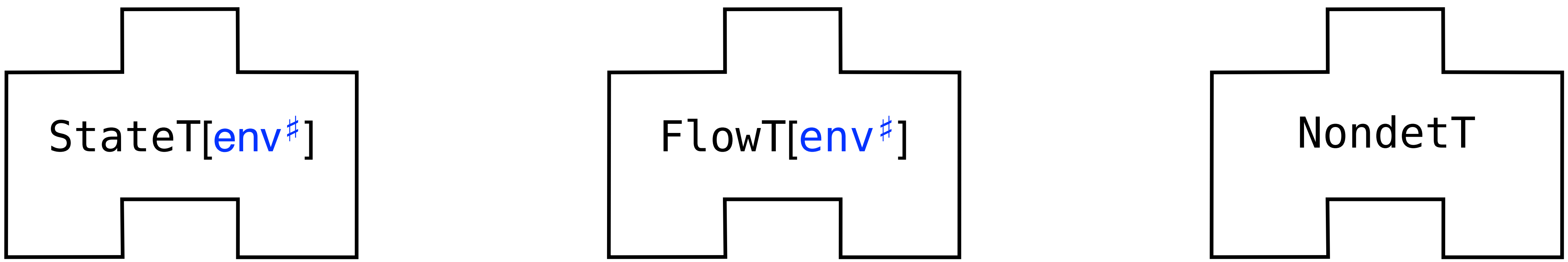




Galois Connections

- 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)

Galois Transformers



- GTs = Monad Transformers + Galois connections
- Galois connections are necessary for execution and proof of correctness for abstract interpreter

Putting it All Together

- You design a monadic abstract interpreter
- Instantiate with monad transformers
- Change underlying monad to change results
- Galois connections synthesized for free:
    - **Code**: Execution engine for running the analysis
    - **Proofs**: Large part of correctness argument

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(I {e₁}{e₂}) :=
    .. æ .. e₁ .. e₂ ..
```

Get Results

```
4: N∈{-,+},y∈{0}
4: N∈{0},x∈{+}

N∈{-,+},y∈{-,0,+}
N∈{0},y∈{0,+}
```

SAFE

Prove Correct

$[\![e]\!] \in [\![\mathrm{analyze}(e)]\!]$

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(I... {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

$[\![e]\!] \in [\![\mathrm{analyze}(e)]\!]$

Future Work

- Benchmark interaction between flow sensitivity and other design choices, like context or object sensitivity
- Explore uses of `NondetT` and `FlowT[`$\mathcal{s}$`]` outside analysis
- Other methods for executing monadic abstract interpreters; might relate to pushdown analysis
- Steps toward modular *verified* abstract interpreters in Coq or Agda using Galois Transformer proof framework
  - First step, mechanizing Galois connections
  - Draft: *Mechanically Verified Calculational Abstract Interpretation* (w/Van Horn)