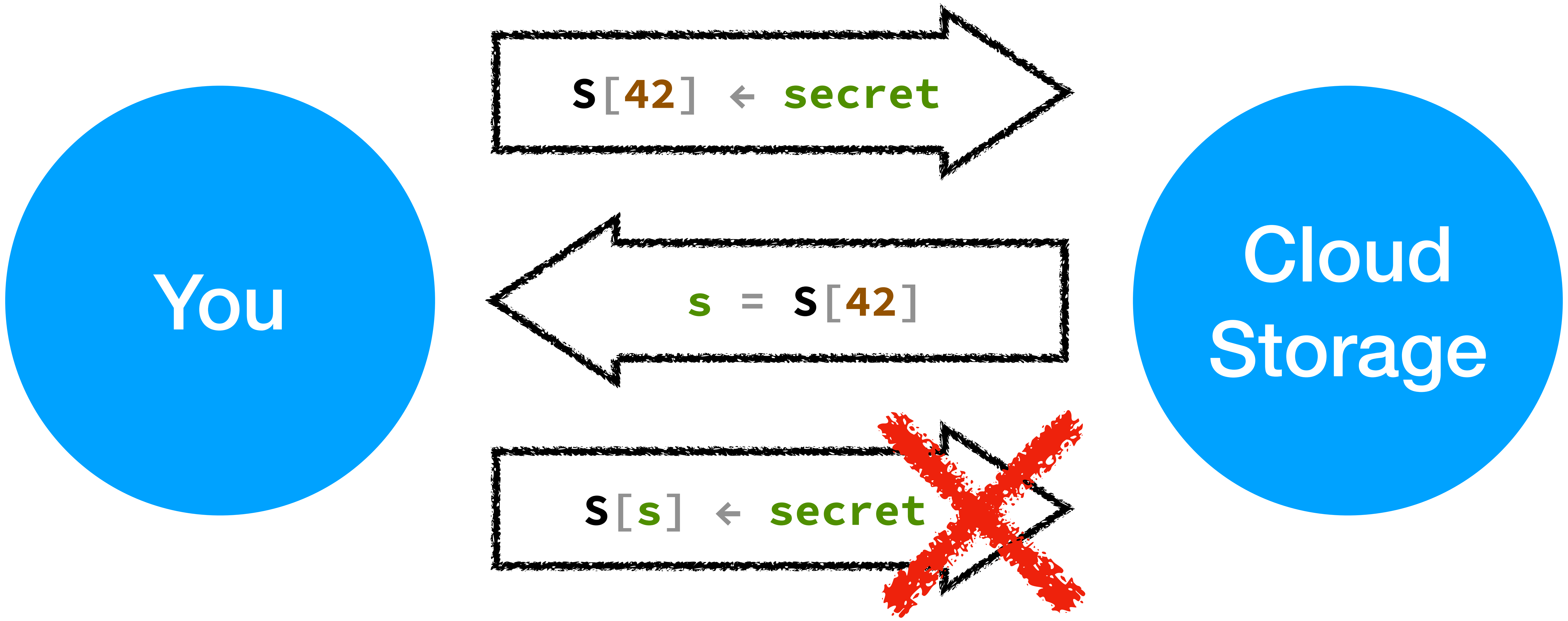
A Language for Probabilistically Oblivious Computation

David Darais, Ian Sweet, Chang Liu, Michael Hicks

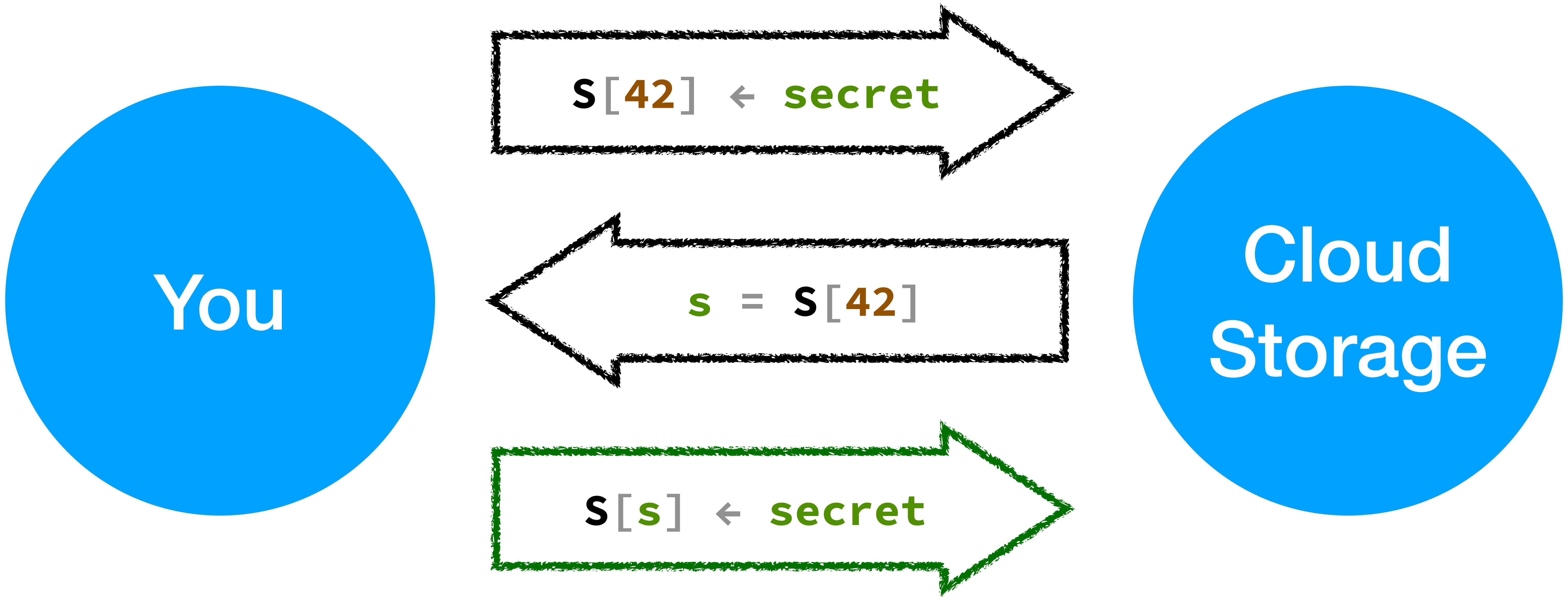
Secure Storage



Implementation = encrypt the data

*Read/write indices **in the clear, cannot** depend on secrets*

Oblivious RAM



Implementation = encrypt the data ***and garble indices***

Read/write indices ***can*** *depend on secrets*

λ-obliv

λ-obliv

…is for implementing **oblivious algorithm**

Oblivious RAM	`s[secret]`	(read)
	`s[secret] ← secret`	(write)

λ-obliv

…is for implementing **oblivious algorithm**

Secure databases and **secure multiparty computation**

Types, **semantics**, and **proofs** for **probabilistic programs**

Publicly available **implementation**

ORAM basics

λ-obliv design

λ-obliv proof

Memory Trace Obliviousness (MTO)

Adversary can see:

Public values

Program counter

Memory (and array) access patterns

Adversary can't see:

Secret values

MTO if you can't infer secret values from observations

Baby Not-secure ORAM

```
-- upload secrets
S[0] ← s_0 -- write secret 0
S[1] ← s_1 -- write secret 1
-- read secret index s
r = S[s] -- NOT OK
```

Adversary Observations

Baby Not-secure ORAM

```
-- upload secrets
S[0] ← s₀ -- write secret 0
S[1] ← s₁ -- write secret 1
-- read secret index s
r = S[s] -- NOT OK
```

Adversary Observations

0
1

Baby Not-secure ORAM

```
-- upload secrets
S[0] ← s_0 -- write secret 0
S[1] ← s_1 -- write secret 1
-- read secret index s
r = S[s] -- NOT OK
```

Adversary Observations
0
1
s

Violates ***Memory Trace Obliviousness (MTO)***

Baby Trivial ORAM

```
-- upload secrets
S[0] ← s₀ -- write secret 0
S[1] ← s₁ -- write secret 1
-- read secret index s
r₀ = S[0] -- read secret 0
r₁ = S[1] -- read secret 1
r, _ = mux(s, r₀, r₁) -- MTO
```

Adversary Observations

Baby Trivial ORAM

```
-- upload secrets
S[0] ← s0 -- write secret 0
S[1] ← s1 -- write secret 1
-- read secret index s
r0 = S[0] -- read secret 0
r1 = S[1] -- read secret 1
r, _ = mux(s, r0, r1) -- MTO
```

Adversary Observations
0
1

Baby Trivial ORAM

```
-- upload secrets
S[0] ← s0 -- write secret 0
S[1] ← s1 -- write secret 1
-- read secret index s
r0 = S[0] -- read secret 0
r1 = S[1] -- read secret 1
r, _ = mux(s, r0, r1) -- MTO
```

Adversary Observations

0
1
0
1

Satisfies MTO, but ***inefficient***

Probabilistic Memory Trace Obliviousness (PMTO)

Adversary can see:

Public values

Program counter

Memory (and array) access patterns

Adversary can't see:

Secret values **AND random samples (coin flips)**

PMTO if you can't infer secret values from observations

Baby Tree ORAM

```
-- upload secrets
b = flip-coin() -- randomness
s₀', s₁' = mux(b, s₀, s₁)
S[0] ← s₀' -- write secret 0 or 1
S[1] ← s₁' -- write secret 1 or 0
-- read secret index s
r = S[b⊕s]
```

Violates secure data/information flow

*Satisfies **Probabilistic Memory Trace Obliviousness (PMTO)***

Baby Tree ORAM

```
-- upload secrets
b = flip-coin() -- randomness
s₀', s₁' = mux(b, s₀, s₁)
S[0] ← s₀' -- write secret 0 or 1
S[1] ← s₁' -- write secret 1 or 0
-- read secret index s
r = S[b⊕s]
```

Truth table for **b⊕s**

b	s	b⊕s
0	0	0
1	0	1
0	1	1
1	1	0

Baby Tree ORAM

```
-- upload secrets
b = flip-coin() -- randomness
s₀', s₁' = mux(b, s₀, s₁)
S[0] ← s₀' -- write secret 0 or 1
S[1] ← s₁' -- write secret 1 or 0
-- read secret index s
r = S[b⊕s]
```

Truth table for **b⊕s**

b	S	b⊕s
0	0	0
1	0	1
0	1	1
1	1	0

Observation: **b⊕s=1**

Baby Tree ORAM

```
-- upload secrets
b = flip-coin() -- randomness
s₀', s₁' = mux(b, s₀, s₁)
S[0] ← s₀' -- write secret 0 or 1
S[1] ← s₁' -- write secret 1 or 0
-- read secret index s
r = S[b⊕s]
```

Truth table for $b \oplus s$

b	s	b⊕s
0	0	0
1	0	1
0	1	1
1	1	0

Observation: $b \oplus s = 1$

Baby Tree ORAM

```
-- upload secrets
b = flip-coin() -- randomness
s₀', s₁' = mux(b, s₀, s₁)
S[0] ← s₀' -- write secret 0 or 1
S[1] ← s₁' -- write secret 1 or 0
-- read secret index s
r = S[b⊕s]
```

Truth table for **b⊕s**

b	s	b⊕s
0	0	0
1	0	1
0	1	1
1	1	0

Observation: **b⊕s=0**

output(b) *after* **S[b⊕s]** *would be problematic!*

ORAM basics

λ-obliv design

λ-obliv proof

λ-obliv design challenge

How to:

Allow **direct flows** from **uniform secrets** to **public values**

Prevent **revealing** any value **correlated with a secret**

λ-obliv features

```
τ ⩴ …
  | flip[R]        -- uniform secrets
```

Affine, uniformly distributed secret random values

R = probability region (elements in a join semilattice)

- Values in same region may be prob. dependent
- Values in strictly ordered regions guaranteed prob. independent

λ-obliv features

```
τ ⩴ …
  | flip[R]        -- uniform secrets
  | bit[R,ℓ]       -- bits
```

Non-affine, possibly random secret values

R = probability region, **ℓ** = information flow label

- Region tracks prob. dependence on random values

λ-obliv features

```
τ ⩴ …
  | flip[R]        -- uniform secrets
  | bit[R,ℓ]       -- bits
  | ref(τ)         -- references
  | τ → τ          -- functions
```

Standard features like references and functions

λ-obliv features

```
τ ⩴ …
  | flip[R]          -- uniform secrets
  | bit[R,ℓ]         -- bits
  | ref(τ)           -- references
  | τ → τ            -- functions

e ⩴ …
  | flip[R]()        -- create uniform secrets
```

New random values are allocated in static region

λ-obliv features

```
τ ⩴ …
  | flip[R]          -- uniform secrets
  | bit[R,ℓ]         -- bits
  | ref(τ)           -- references
  | τ → τ            -- functions

e ⩴ …
  | flip[R]()        -- create uniform secrets
  | castP(e)         -- reveal uniform secrets
  | castS(x)         -- non-affine use of x
```

Escape hatches needed to implement ORAM

```
castP : flip[R] → bit[⊥,P]  (consuming)

castS : flip[R] → bit[R,S]  (non-consuming)
```

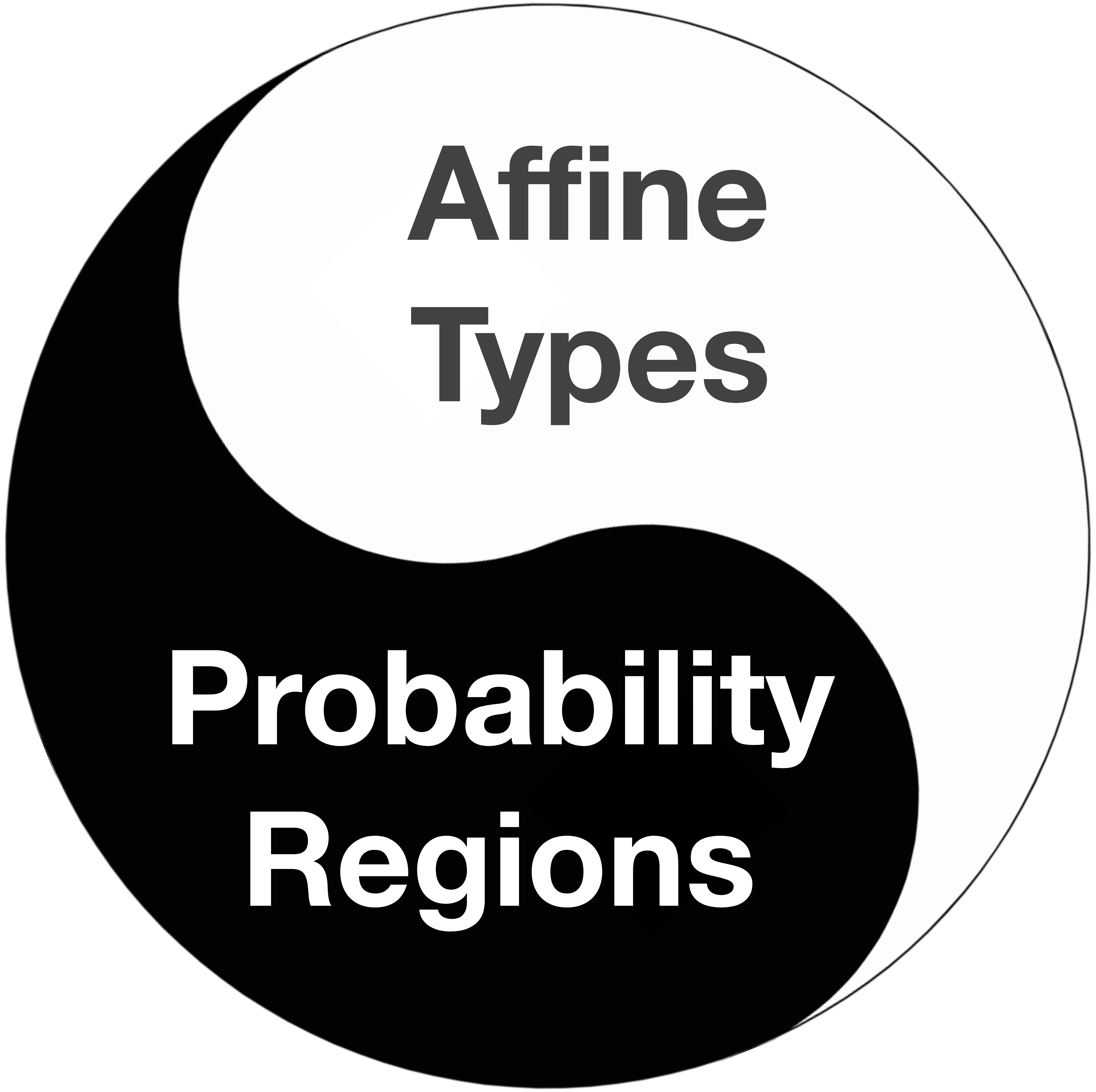
λ-obliv features

```
τ ⩴ …
  | flip[R]          -- uniform secrets
  | bit[R,ℓ]         -- bits
  | ref(τ)           -- references
  | τ → τ            -- functions

e ⩴ …
  | flip[R]()        -- create uniform secrets
  | castP(e)         -- reveal uniform secrets
  | castS(x)         -- non-affine use of x
  | e ⊕ e            -- xor
  | mux(e, e, e)     -- atomic mux
  | read(e)          -- reference read
  | write(e, e)      -- reference write
  | if(e){e}{e}      -- conditionals
  | λx.e | e(e)      -- functions
```

Taming the escape hatches

```
e ::= ...
    | castP(e)
    | castS(x)
```



Affinity in Action

```
b₁       = flip[R1]()

output(castP(b₁)) -- OK
```

Affinity in Action

```
b₁, b₂ = flip[R1](), flip[R2]()
b₃, _ = mux(s, b₁, b₂)
-- each of b₁, b₂, b₃ uniform
output(castP(b₁)) -- OK
```

Affinity in Action

```
b₁, b₂ = flip[R1](), flip[R2]()
b₃, _ = mux(s, b₁, b₂)
-- each of b₁, b₂, b₃ uniform
output(castP(b₃)) -- OK
```

Affinity in Action

```
b₁, b₂ = flip[R1](), flip[R2]()
b₃, _ = mux(s, b₁, b₂)
-- each of b₁, b₂, b₃ uniform
output(castP(b₃)) -- OK
-- none of b₁, b₂, b₃ uniform
output(castP(b₁)) -- NOT OK
```

Affinity in Action

```
b₁, b₂ = flip[R1](), flip[R2]()
b₃, _ = mux(s, b₁, b₂)
-- each of b₁, b₂, b₃ uniform
output(castP(b₃)) -- OK
-- none of b₁, b₂, b₃ uniform
output(castP(b₁)) -- NOT OK
```

s	b_1	b_2	b_3
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	1
0	0	1	1
1	0	1	0
0	1	1	1
1	1	1	1

Affinity in Action

```
b₁, b₂ = flip[R1](), flip[R2]()
b₃, _ = mux(s, b₁, b₂)
-- each of b₁, b₂, b₃ uniform
output(castP(b₃)) -- OK
-- none of b₁, b₂, b₃ uniform
output(castP(b₁)) -- NOT OK
```

Observation: b_3=1

s	b_1	b_2	b_3
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	1
0	0	1	1
1	0	1	0
0	1	1	1
1	1	1	1

Affinity in Action

```
b₁, b₂ = flip[R1](), flip[R2]()
b₃, _ = mux(s, b₁, b₂)
-- each of b₁, b₂, b₃ uniform
output(castP(b₃)) -- OK
-- none of b₁, b₂, b₃ uniform
output(castP(b₁)) -- NOT OK
```

s	b_1	b_2	b_3
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	1
0	0	1	1
1	0	1	0
0	1	1	1
1	1	1	1

Observation: $b_3=1$

Observation: $b_1=0$

Learn: $s=0$

Affinity in Action

$$r_1, r_2 = \mathbf{mux}(s, \cancel{b_1}, \cancel{b_2})$$

Mux Rule: "consume" branch values

Affinity in Action

```
b₁, b₂ = flip[R1](), flip[R2]()
b₃, _ = mux(s, b₁, b₂)
-- each of b₁, b₂, b₃ uniform
output(castP(b₃)) -- OK
-- none of b₁, b₂, b₃ uniform
output(castP(b₁)) -- NOT OK
```

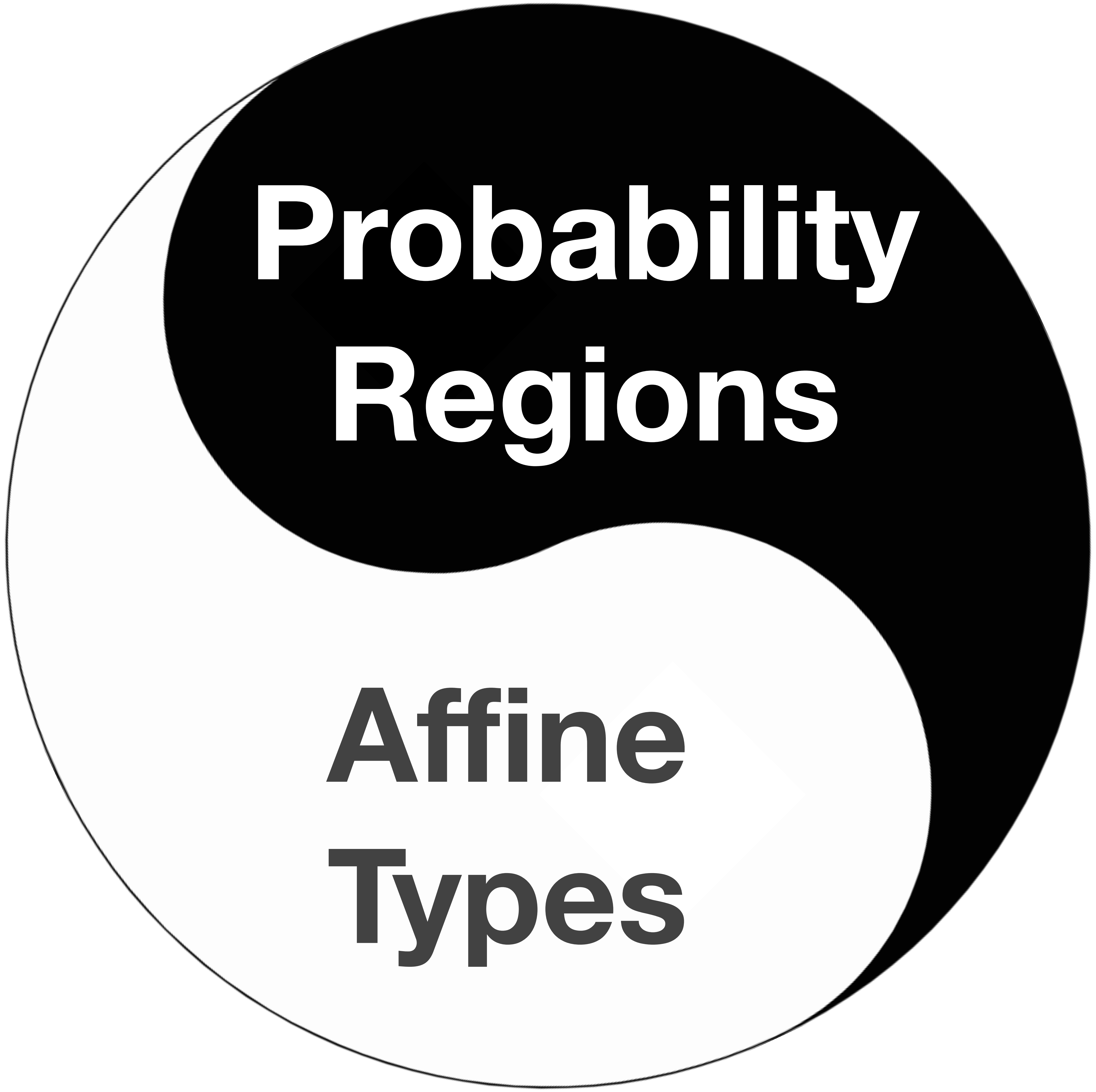
Rejected by λ-obliv type system

Taming the escape hatches

```
e ::= …
    | castP(e)
    | castS(x)
```

Taming the escape hatches

```
e ::= ...
    | castP(e)
    | castS(x)
```



Probability Regions in Action

```
b₁, b₂ = flip[R1](), flip[R2]()
-- each of b₁, b₂ uniform
b₃, _ = mux(castS(b₁), b₁, b₂)
-- b₃ not uniform
b₄, _ = mux(s, b₃, flip[R3]())
-- b₄ not uniform because b₃ isn't
output(castP(b₄)) -- NOT OK
```

Probability Regions in Action

```
b₁, b₂ = flip[R1](), flip[R2]()
                                   b₁ ⊥̸ b₂
b₃, _ = mux(castS(b₁), b₁, b₂)

b₄, _ = mux(s, b₃, flip[R3]())

output(castP(b₄)) -- NOT OK
```

Probability Regions in Action


```
r₁, r₂ = mux(s, b₁, b₂)
```

Rule: probabilistic independence from guard

Probability Regions in Action

```
b1, b2 = flip[R1](), flip[R2]()
```

$R_1 \not\subseteq R_1$

```
b3, _ = mux(castS(b1), b1, b2)

b4, _ = mux(s, b3, flip[R3]())

output(castP(b4)) -- NOT OK
```

Rejected by λ-obliv type system

Probability Regions

Property	Values	Types
Noninterference	$b_1 \triangleright b_2$	$R_1 \sqsubseteq R_2$

Probability Regions

Property	Values	Types
Noninterference	$b_1 \triangleright b_2$	$R_1 \sqsubseteq R_2$
Probabilistic Independence	$b_1 \perp\!\!\!\perp b_2$	

Probability Regions

Property	Values	Types
Noninterference	$b_1 \triangleright b_2$	$R_1 \sqsubseteq R_2$
Probabilistic Independence	$b_1 \perp\!\!\!\perp b_2$	$R_1 \sqcap R_2 = \bot$

Probability Regions

Property	Values	Types
Noninterference	$b_1 \triangleright b_2$	$R_1 \sqsubseteq R_2$
Probabilistic Independence	$b_1 \perp\!\!\!\perp b_2$	$R_1 \sqcap R_2 = \bot$
Robust w.r.t. Revelations	$b_1 \perp\!\!\!\perp b_2 \mid \phi$	

Probability Regions

Property	Values	Types
Noninterference	$b_1 \triangleright b_2$	$R_1 \sqsubseteq R_2$
Probabilistic Independence	$b_1 \perp\!\!\!\perp b_2$	$R_1 \sqcap R_2 = \bot$
Robust w.r.t. Revelations	$b_1 \perp\!\!\!\perp b_2 \mid \Phi$	$R_1 \sqsubset R_2$

λ-obliv Typing

$$\frac{s : \mathtt{secret} \quad b_1 : \mathtt{flip} \quad b_2 : \mathtt{flip}}{\mathtt{mux}(s, b_1, b_2)}$$

λ-obliv Typing

$$\frac{\begin{array}{lll} s & : \texttt{secret} & @\ R_1 \\ b_1 & : \texttt{flip} & @\ R_2 \\ b_2 & : \texttt{flip} & @\ R_3 \end{array}}{\texttt{mux}(s,\ b_1,\ b_2)}$$

λ-obliv Typing

$$\frac{\begin{array}{llll} s & : \texttt{secret} & @\ R_1 & \qquad R_1 \sqsubseteq R_2 \\ b_1 & : \texttt{flip} & @\ R_2 & \qquad R_1 \sqsubseteq R_3 \\ b_2 & : \texttt{flip} & @\ R_3 & \end{array}}{\texttt{mux}(s,\ b_1,\ b_2)}$$

λ-obliv Typing

$$\frac{\begin{array}{llll} s & : & \mathtt{secret} @ R_1 & \qquad R_1 \sqsubseteq R_2 \\ b_1 & : & \mathtt{flip} @ R_2 & \qquad R_1 \sqsubseteq R_3 \\ b_2 & : & \mathtt{flip} @ R_3 & \end{array}}{\mathtt{mux}(s, b_1, b_2) : (\mathtt{flip} @ R) \times (\mathtt{flip} @ R)}$$

λ-obliv Typing

$$\frac{\begin{array}{llll} s & : \texttt{secret} & @\ R_1 & \quad R_1 \sqsubseteq R_2 \\ b_1 & : \texttt{flip} & @\ R_2 & \quad R_1 \sqsubseteq R_3 \\ b_2 & : \texttt{flip} & @\ R_3 & \quad R = R_1 \sqcup R_2 \sqcup R_3 \end{array}}{\texttt{mux}(s,\ b_1,\ b_2) : (\texttt{flip}\ @\ R) \times (\texttt{flip}\ @\ R)}$$

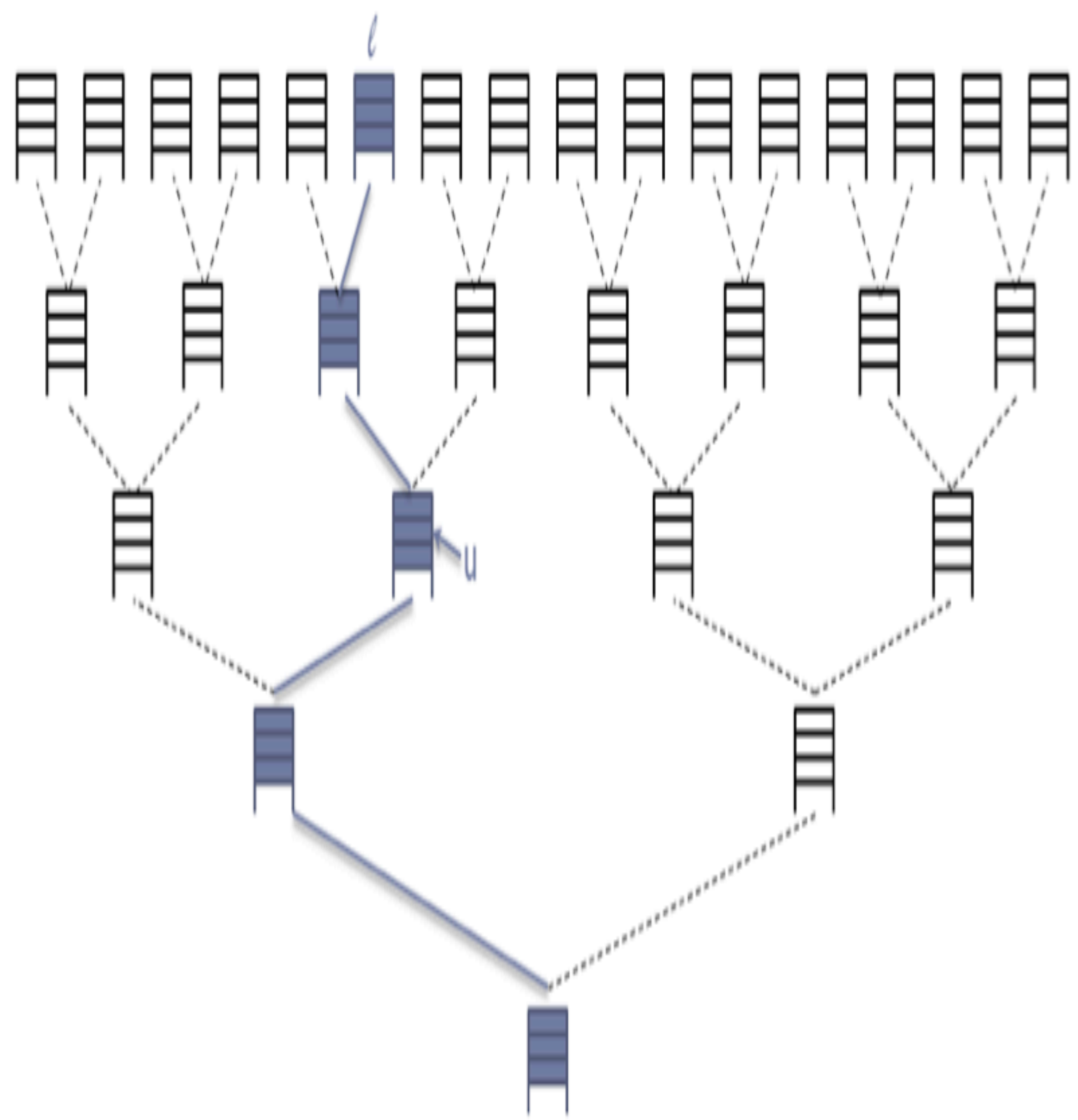
Case Study: Tree ORAM

λ-obliv is expressive enough to implement full ORAM

ORAM security verified entirely via type checking

Implemented in OCaml and publicly available

(+ other case studies)



ORAM basics

λ-obliv design

λ-obliv proof

λ-obliv Enjoys PMTO

THEOREM: typing implies PMTO for small-step sampling semantics

PROOF: via alternative “mixed” semantics which:

Mixes operational and denotational methods

Uses a new probability monad for reasoning about conditional (in)dependence

PROOF INVARIANT: flip values are:

Uniformly distributed

Independent from all other flip values, conditioned on any subset of secrets typed at smaller regions

Related Work

Prior work [1] verifies deterministic MTO by typing.
We push this to probabilistic (PMTO).

Prior work [2] claims to solve PMTO by typing but unsound.
(fix = probability regions; proof much more involved)

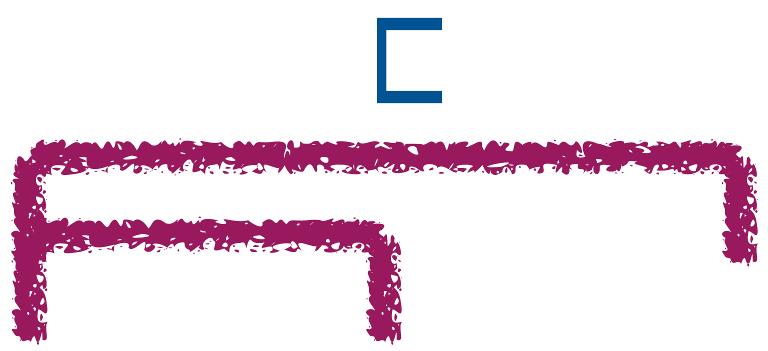
Related work this POPL [3] (tomorrow 14:43) solves PMTO for ORAM via a program logic.

[1]: Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. ASPLOS 2015.
[2]: Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A Programming Framework for Secure Computation. IEEE S&P 2015.
[3]: Gilles Barthe, Justin Hsu, Mingsheng Ying, Nengkun Yu, Li Zhou. Relational Proofs for Quantum Programs. POPL 2020.

λ-obliv

```
-- upload secrets
b = flip[R]() -- randomness
s₀', s₁' = mux(b, s₀, s₁)
S[0] ← s₀' -- write secret 0 or 1
S[1] ← s₁' -- write secret 1 or 0
-- read secret index s
r = S[b⊕s] – PMTO
```

```
e ::= …
    | castP(e)
    | castS(x)
```



mux(s, ~~b₁~~, ~~b₂~~) + mux(s, b₁, b₂) = PMTO

Mux Rule: affine branches

Mux Rule: independence from guard