Verifying Probabilistic Programs with Separation Logic Part 3

Joseph Tassarotti
New York University

Based on joint work with: Alejandro Aguirre, Philipp G. Haselwarter, Kwing Hei-Li, Markus de Medeiros, Puming Liu, Alex Bai, Simon Oddershede Gregersen, and Lars Birkedal

Course page: clutch-project.org/epit2026.html
Rocq code: github.com/logsem/clutch/tree/epit2026

Previous Part

- **Eris**: error credits $↯(\epsilon)$ for "up-to-bad" error bounds reasoning.
- **Eris**$_t$: error-credit induction for almost-sure termination.

This Time

- **Continuous-Eris**: reasoning about exact samplers for continuous distributions using computable reals.
- **Clutch**: program logic for contextual equivalence of higher-order probabilistic programs
- Both use an idea called local presampling tapes that is a "hybrid" of sampling and distribution semantics.

A word on pseudorandomness

"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."

— John von Neumann (1951)

A word on floating-point

"Numerical data piles up and numerical programs grow ever more ambitious and complicated while their users become, on average, far less knowledgeable about numerical error-analysis, though no less clever than their predecessors about subjects they care to learn. Consequently numerical anomalies go mostly unobserved or, if observed, routinely misdiagnosed. Fortunately most of them don't matter. Most computations don't matter."

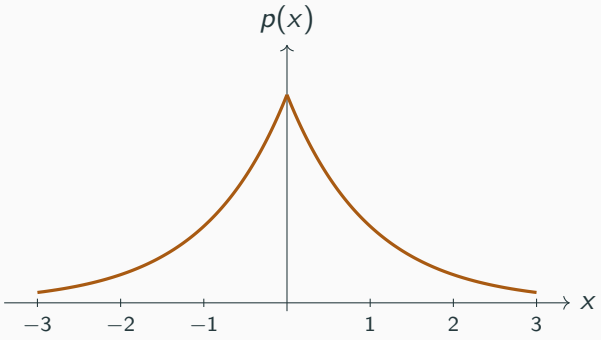
— W. Kahan, *How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?*

Some computations do matter!

Differential privacy: Given some private data D, compute $f(D)$ and output $f(D) + Y$, where $Y \sim \mathrm{Lap}(1/\epsilon)$. Output is ϵ-differentially private.

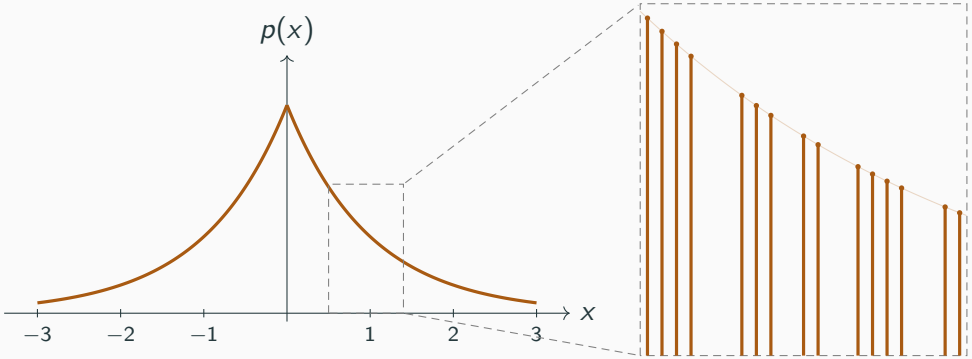
Some computations do matter!

Differential privacy: Given some private data D, compute $f(D)$ and output $f(D) + Y$, where $Y \sim \mathrm{Lap}(1/\epsilon)$. Output is ϵ-differentially private.



Some computations do matter!

Differential privacy: Given some private data D, compute $f(D)$ and output $f(D) + Y$, where $Y \sim \mathrm{Lap}(1/\epsilon)$. Output is ϵ-differentially private.



Mironov 2012: roundoff breaks ϵ-DP

With standard floating-point Laplace samplers, gaps in the support of $f(D) + Y$ depend on $f(D)$: neighboring inputs $f(D), f(D')$ produce different supports in the output distribution.

Mironov 2012: roundoff breaks ϵ-DP

With standard floating-point Laplace samplers, gaps in the support of $f(D) + Y$ depend on $f(D)$: neighboring inputs $f(D), f(D')$ produce different supports in the output distribution.

By inspecting the **low-order bits** of a single released value, an attacker recovers $f(D)$ exactly with high probability.

Mironov 2012: roundoff breaks ϵ-DP

With standard floating-point Laplace samplers, gaps in the support of $f(D) + Y$ depend on $f(D)$: neighboring inputs $f(D), f(D')$ produce different supports in the output distribution.

By inspecting the **low-order bits** of a single released value, an attacker recovers $f(D)$ exactly with high probability.

$\Rightarrow$ The textbook floating-point implementation provides no differential privacy.

Option 1

Forsake continuous distributions:
use discrete Gaussians, Laplacians, etc.

No roundoff.

Ironically: privacy math sometimes becomes much harder. (Fourier analysis needed for standard results.)

Two ways to respond

Option 1

Forsake continuous distributions:
use discrete Gaussians, Laplacians, etc.

No roundoff.

Ironically: privacy math sometimes becomes much harder. (Fourier analysis needed for standard results.)

Option 2

Use exact real arithmetic over computable reals.

Privacy math is as "on paper", but sampling is harder.

Two ways to respond

Option 1

Forsake continuous distributions:
use discrete Gaussians, Laplacians, etc.

No roundoff.

Ironically: privacy math sometimes becomes much harder. (Fourier analysis needed for standard results.)

Option 2

Use exact real arithmetic over computable reals.

Privacy math is as "on paper", but sampling is harder.

(One Form of) Computable Reals

Idea: represent a real number r by a function $s_r : \mathbb{Z} \to \mathbb{Z}$ such that, on input p,

$$|s_r(p) - r \cdot 2^{-p}| \leq 1 \quad \text{for every } p \in \mathbb{Z}.$$

(Smaller p = finer precision.)

(One Form of) Computable Reals

Idea: represent a real number r by a function $s_r : \mathbb{Z} \to \mathbb{Z}$ such that, on input p,

$$|s_r(p) - r \cdot 2^{-p}| \leq 1 \quad \text{for every } p \in \mathbb{Z}.$$

(Smaller p = finer precision.)

Example. For $r = 4.12345031\ldots$:

p	$r \cdot 2^{-p}$	$s_r(p)$
0	$4.12345\ldots$	4
-4	$65.9752\ldots$	66
-8	$1055.6032\ldots$	1056
-16	$270234.4197\ldots$	270234

Operations on Computable Reals

Arithmetic on reals = higher-order programs that consume and produce approximation functions. Addition:

$$\begin{aligned} \mathit{add}\ r_1\ r_2\ p \triangleq\ & \mathsf{let}\ z = r_1\ (p{-}2) + r_2\ (p{-}2)\ \mathsf{in} \\ & (z \operatorname{div} 4) + (z \bmod 4) \operatorname{div} 2. \end{aligned}$$

Operations on Computable Reals

Arithmetic on reals = higher-order programs that consume and produce approximation functions. Addition:

$$\begin{aligned} \mathit{add}\ r_1\ r_2\ p \triangleq\ & \mathsf{let}\ z = r_1\ (p{-}2) + r_2\ (p{-}2)\ \mathsf{in} \\ & (z \operatorname{div} 4) + (z \bmod 4) \operatorname{div} 2. \end{aligned}$$

Comparison: query at p, refine if too close to call:

$$\begin{aligned} \mathit{cmp}\ r_1\ r_2\ p \triangleq\ & \mathsf{let}\ n_1 = r_1\ p\ \mathsf{in}\ \mathsf{let}\ n_2 = r_2\ p\ \mathsf{in} \\ & \mathsf{if}\ n_1 + 2 < n_2\ \mathsf{then}\ -1\ \mathsf{else} \\ & \mathsf{if}\ n_2 + 2 < n_1\ \mathsf{then}\quad 1\ \mathsf{else} \\ & \mathit{cmp}\ r_1\ r_2\ (p{-}1). \end{aligned}$$

(Diverges when $r_1 = r_2$.)

Sampling a uniform $r \in [0, 1]$

Represent the sample as a lazy linked list of random bits representing digits .

$$0, 1, 0, 1, 1, 0 \ldots$$

Sampling a uniform $r \in [0, 1]$

Represent the sample as a lazy linked list of random bits representing digits .

$$0, 1, 0, 1, 1, 0 \ldots$$

$U\ \_ \triangleq (\text{ref}\ \ \text{None})$

$\mathit{GetBits}\ \ell\ N\ A \triangleq$

$\quad \text{if}\ N = 0\ \text{then}\ A\ \text{else}$

$\quad \text{let}\ (b, \ell') = \mathit{ForceNext}\ \ell\ \text{in}$

$\quad \mathit{GetBits}\ \ell'\ (N-1)\ (2A + b)$

Sampling a uniform $r \in [0, 1]$

Represent the sample as a lazy linked list of random bits representing digits .

$$0, 1, 0, 1, 1, 0 \ldots$$

$U\ \_ \triangleq (\text{ref None})$

$GetBits\ \ell\ N\ A \triangleq$

  if $N = 0$ then A else

  let $(b, \ell') = ForceNext\ \ell$ in

  $GetBits\ \ell'\ (N-1)\ (2A + b)$

$ForceNext\ \ell \triangleq$ match ! ℓ with

  Some$(b, \ell') \Rightarrow (b, \ell')$

  | None $\Rightarrow$ let $b =$ rand 1 in

  let $\ell' =$ ref None in

  $\ell \leftarrow$ Some(b, ℓ');

  (b, ℓ')

  end

Exact Samplers for Other Continuous Distributions

Von Neumann (1952) – exponential distribution (Same 3 page paper where he describes rejection sampling based on densities)

Karney (2016) – Gaussian distribution

Correctness?

How can one prove the correctness of these samplers?

Not so bad in a "nice" lazy probabilistic language (like LazyPPL)?

But what about in an implementation in ML-like language, need:

- Mutable state
- Higher-order functions
- General recursion
- Probabilistic choice

Can we use Eris?

Recall: Marionneau et al. showed that to prove f samples from a distribution discrete D, it suffices to prove in Eris that:

$$\frac{\mathbb{E}_{x\sim D}[\epsilon(x)] = \epsilon_0}{\{\lightning(\epsilon_0)\}\ f\ ()\ \{x.\ \lightning(\epsilon(x))\}}$$

Can we use Eris?

Recall: Marionneau et al. showed that to prove f samples from a distribution discrete D, it suffices to prove in Eris that:

$$\frac{\sum_x D(x) \cdot \epsilon(x) = \epsilon_0}{\{\lightning(\epsilon_0)\}\ f\ ()\ \{x.\ \lightning(\epsilon(x))\}}$$

Can we use Eris?

Recall: Marionneau et al. showed that to prove f samples from a distribution discrete D, it suffices to prove in Eris that:

$$\frac{\sum_x D(x) \cdot \epsilon(x) = \epsilon_0}{\{\lightning(\epsilon_0)\}\ f\ ()\ \{x.\ \lightning(\epsilon(x))\}}$$

Idea: generalize this to the continuous case:

$$\frac{\int p(x) \cdot \epsilon(x) dx = \epsilon_0}{\{\lightning(\epsilon_0)\}\ f\ ()\ \{v.\ \exists x.\ \mathsf{IsReal(v,\ x)} * \lightning(\epsilon(x))\}}$$

Soundness for Continuous Samplers

If we prove that spec for f, what does that tell us about f's output distribution?

Soundness for Continuous Samplers

If we prove that spec for f, what does that tell us about f's output distribution? Idea: compare a sample to a dyadic rational $B/2^C$ via a small checker program:

- sample $< B/2^C \Rightarrow$ returns -1
- sample $> B/2^C \Rightarrow$ returns $\quad 1$
- sample $= B/2^C \Rightarrow$ diverges

Soundness for Continuous Samplers

If we prove that spec for f, what does that tell us about f's output distribution? Idea: compare a sample to a dyadic rational $B/2^C$ via a small checker program:

- sample $< B/2^C \Rightarrow$ returns -1
- sample $> B/2^C \Rightarrow$ returns $\quad 1$
- sample $= B/2^C \Rightarrow$ diverges

Adequacy theorem. If for every integrable ϵ we can derive

$$\left\{ \lightning \left(\int p(x) \cdot \epsilon(x)\, dx \right) \right\} \; f \; () \; \{ v.\, \exists x.\;\; \mathsf{IsReal}(\mathsf{v},\, \mathsf{x}) * \; \lightning(\epsilon(x)) \}$$

then

$$\Pr[\mathit{checker}\ f\ B\ C \neq 1] \;\leq\; \int_{-\infty}^{B/2^C} p(x)\, dx$$

Soundness for Continuous Samplers

If we prove that spec for f, what does that tell us about f's output distribution? Idea: compare a sample to a dyadic rational $B/2^C$ via a small checker program:

- sample $< B/2^C \Rightarrow$ returns -1
- sample $> B/2^C \Rightarrow$ returns $\quad 1$
- sample $= B/2^C \Rightarrow$ diverges

Adequacy theorem. If for every integrable ϵ we can derive

$$\left\{ \lightning \left(\int p(x) \cdot \epsilon(x)\, dx \right) \right\} f\ ()\ \{ v.\ \exists x.\ \mathsf{IsReal}(\mathsf{v},\ \mathsf{x}) * \lightning(\epsilon(x)) \}$$

then

$$\Pr[\mathit{checker}\ f\ B\ C \neq 1] \ \leq \ \int_{-\infty}^{B/2^C} p(x)\, dx$$

With almost-sure termination, equality holds. Dyadic rationals are dense, so this characterizes the CDF.

Challenge

If we try to apply this to our uniform sampler:

$$U\ \_ \triangleq (\mathsf{ref}\ \mathsf{None})$$

we would have to prove:

$$\frac{\int_0^1 1 \cdot \epsilon(x)dx = \epsilon_0}{\{\lightning(\epsilon_0)\}\ U\ ()\ \{v.\, \exists x.\, \mathsf{IsReal(v,\ x)} * \lightning(\epsilon(x))\}}$$

But this code hasn't even done anything randomized!

Presampling

We want to know "in advance" what random bits U will sample.

Handle this by introducing ghost presampling tapes.

Presampling

We want to know "in advance" what random bits U will sample.

Handle this by introducing ghost presampling tapes.

Extend the execution state with dynamically allocated tapes, which are lists of random values

$$\rho \in \mathit{Cfg} ::= \mathit{Expr} \times \mathit{State}$$
$$\sigma \in \mathit{State} ::= \mathit{Heap} \times \mathit{Tapes}$$

Presampling

We want to know "in advance" what random bits U will sample.

Handle this by introducing ghost presampling tapes.

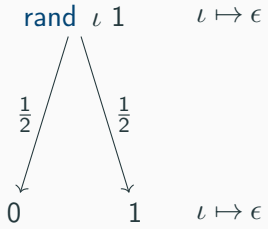
Extend the execution state with dynamically allocated tapes, which are lists of random values

$$\rho \in \mathit{Cfg} ::= \mathit{Expr} \times \mathit{State}$$
$$\sigma \in \mathit{State} ::= \mathit{Heap} \times \mathit{Tapes}$$
$$e \in \mathit{Expr} ::= \ldots \mid \mathsf{rand}\ N \mid \mathsf{rand}\ e\ N \mid \mathsf{tape}$$

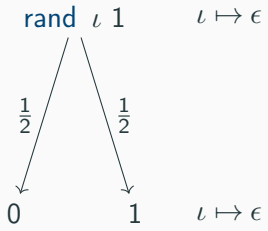
Presampling tapes

$\mathsf{rand}\ \iota\ 1 \qquad \iota \mapsto \epsilon$

Presampling tapes

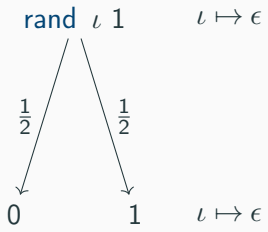


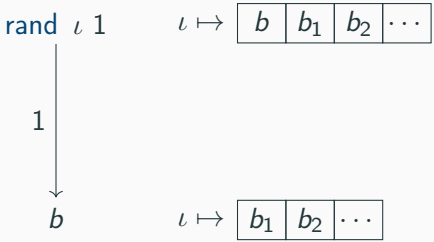
Presampling tapes



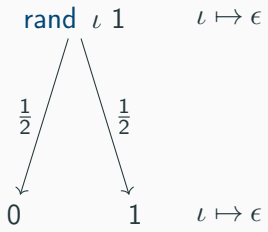
rand ι 1 $\quad \iota \mapsto$ | b | b_1 | b_2 | $\cdots$ |

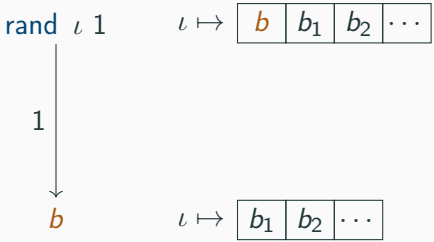
Presampling tapes





Presampling tapes





... but, operationally, no language primitives add values to the tapes!

$$\iota : \mathsf{tape} \vdash \mathsf{rand}\ \ 1 \simeq_{\mathsf{ctx}} \mathsf{rand}\ \ \iota\ 1 : \mathsf{nat}$$

... but, operationally, no language primitives add values to the tapes!

$$\iota : \mathsf{tape} \vdash \mathsf{rand}\ \ 1 \simeq_{\mathsf{ctx}} \mathsf{rand}\ \ \iota\ 1 : \mathsf{nat}$$

Instead, tapes will non-deterministically be populated with fresh samples.

... but, operationally, no language primitives add values to the tapes!

$$\iota : \mathsf{tape} \vdash \mathsf{rand}\ \ 1 \simeq_{\mathsf{ctx}} \mathsf{rand}\ \ \iota\ 1 : \mathsf{nat}$$

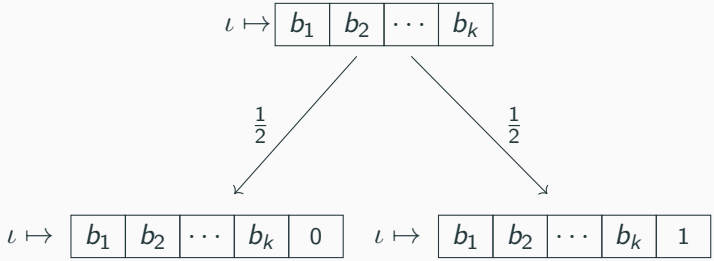
Instead, tapes will non-deterministically be populated with fresh samples.

$$\iota \mapsto \boxed{b_1 \mid b_2 \mid \cdots \mid b_k}$$

... but, operationally, no language primitives add values to the tapes!

$$\iota : \mathsf{tape} \vdash \mathsf{rand}\ \ 1 \simeq_{\mathsf{ctx}} \mathsf{rand}\ \ \iota\ 1 : \mathsf{nat}$$

Instead, tapes will non-deterministically be populated with fresh samples.



Tape Points-to

Introduce a separation logic resource

$$\alpha \hookrightarrow (N, \vec{n})$$

that denotes ownership of a tape labeled α with bound N and contents $\vec{n}$.

Tape Points-to

Introduce a separation logic resource

$$\alpha \hookrightarrow (N, \vec{n})$$

that denotes ownership of a tape labeled α with bound N and contents $\vec{n}$.

$$\{\alpha \hookrightarrow (N, n \cdot \vec{n})\} \text{ rand } \alpha\ N \{v.\ v = n * \alpha \hookrightarrow (N, \vec{n})\}$$

Tape Points-to

Introduce a separation logic resource

$$\alpha \hookrightarrow (N, \vec{n})$$

that denotes ownership of a tape labeled α with bound N and contents $\vec{n}$.

$$\{\alpha \hookrightarrow (N, n \cdot \vec{n})\} \text{ rand } \alpha\ N \{v.\ v = n * \alpha \hookrightarrow (N, \vec{n})\}$$

$$\frac{\{\exists n.\ \alpha \hookrightarrow (N, \vec{n} \cdot n) * \lightning(\epsilon(n)) * P\}\ e\ \{Q\}}{\{\alpha \hookrightarrow (N, \vec{n}) * \lightning(\mathbb{E}_{\mathsf{Unif}(N)}[\epsilon]) * P\}\ e\ \{Q\}}$$

Tape Points-to

Introduce a separation logic resource

$$\alpha \hookrightarrow (N, \vec{n})$$

that denotes ownership of a tape labeled α with bound N and contents $\vec{n}$.

$$\{\alpha \hookrightarrow (N, n \cdot \vec{n})\} \text{ rand } \alpha\ N \{v.\ v = n * \alpha \hookrightarrow (N, \vec{n})\}$$

$$\frac{\{\exists n.\ \alpha \hookrightarrow (N, \vec{n} \cdot n) * \lightning(\epsilon(n)) * P\}\ e\ \{Q\}}{\{\alpha \hookrightarrow (N, \vec{n}) * \lightning(\mathbb{E}_{\mathsf{Unif}(N)}[\epsilon]) * P\}\ e\ \{Q\}}$$

It turns reasoning about probabilistic choice into reasoning about state.

Finite Tapes are a Problem

Now to reason about U we can try presample all the samples it needs when it is allocated.

But to stay in the discrete operational world tapes are finite lists of samples. (Why?)

Finite Tapes are a Problem

Now to reason about U we can try presample all the samples it needs when it is allocated.

But to stay in the discrete operational world tapes are finite lists of samples. (Why?)

Problem: uniform sampler U could access an unbounded number of samples.

From Finite to Infinite: Internalize Partiality

Key idea: we work in the **partial** variant of Eris.

We can internalize this partiality using an assertion $\mathsf{stepsLeft}(k)$ saying "in the execution we're reasoning about, the program will run for at most k more steps":

$$\frac{\forall k.\ \{\mathsf{stepsLeft}(k) * P\}\ e\ \{Q\}}{\{P\}\ e\ \{Q\}}$$

Since we must prove the triple for *every* k, every finite prefix of an execution is covered.

From Finite to Infinite: Internalize Partiality

Key idea: we work in the **partial** variant of Eris.

We can internalize this partiality using an assertion $\mathsf{stepsLeft}(k)$ saying "in the execution we're reasoning about, the program will run for at most k more steps":

$$\frac{\forall k.\ \{\mathsf{stepsLeft}(k) * P\}\ e\ \{Q\}}{\{P\}\ e\ \{Q\}}$$

Since we must prove the triple for *every* k, every finite prefix of an execution is covered.

$$k < 0\ \Rightarrow\ \mathsf{stepsLeft}(k) \vdash \bot.$$

From Finite to Infinite: Internalize Partiality

Key idea: we work in the **partial** variant of Eris.

We can internalize this partiality using an assertion $\mathsf{stepsLeft}(k)$ saying "in the execution we're reasoning about, the program will run for at most k more steps":

$$\frac{\forall k.\ \{\mathsf{stepsLeft}(k) * P\}\ e\ \{Q\}}{\{P\}\ e\ \{Q\}}$$

Since we must prove the triple for *every* k, every finite prefix of an execution is covered.

$$k < 0\ \Rightarrow\ \mathsf{stepsLeft}(k) \vdash \bot.$$

Each program step decrements stepsLeft.

Proving the Spec for U

$$\frac{\int_0^1 \cdot \epsilon(x)dx = \epsilon_0}{\{\lightning(\epsilon_0)\}\ U\ ()\ \{v.\ \exists x.\ \mathsf{IsReal(v,\ x)} * \lightning(\epsilon(x))\}}$$

Proving the Spec for U

$$\frac{\int_0^1 \cdot \epsilon(x)dx = \epsilon_0}{\{\lightning(\epsilon_0)\}\ U\ ()\ \{v.\ \exists x.\ \mathsf{IsReal(v,\ x)} * \lightning(\epsilon(x))\}}$$

1. Get $\mathsf{stepsLeft}(\mathsf{k}_1)$ for some k_1.

Proving the Spec for U

$$\frac{\int_0^1 \cdot \epsilon(x)dx = \epsilon_0}{\{\lightning(\epsilon_0)\}\ U\ ()\ \{v.\ \exists x.\ \mathsf{IsReal(v,\ x)} * \lightning(\epsilon(x))\}}$$

1. Get $\mathsf{stepsLeft}(\mathsf{k}_1)$ for some k_1.
2. Get thin-air $\lightning(\epsilon')$ error credit for some $\epsilon' > 0$.

Proving the Spec for U

$$\frac{\int_0^1 \cdot \epsilon(x)dx = \epsilon_0}{\{\lightning(\epsilon_0)\}\ U\ ()\ \{v.\ \exists x.\ \mathsf{IsReal(v,\ x)} * \lightning(\epsilon(x))\}}$$

1. Get $\mathsf{stepsLeft(k_1)}$ for some k_1.
2. Get thin-air $\lightning(\epsilon')$ error credit for some $\epsilon' > 0$.
3. By Riemann integrability, $\exists k_2 > 0$ such that the integral is within ϵ' of any Riemann sum using partitions of width $< 1/2^{k_2}$.

Proving the Spec for U

$$\frac{\int_0^1 \cdot \epsilon(x)dx = \epsilon_0}{\{\lightning(\epsilon_0)\}\ U\ ()\ \{v.\, \exists x.\, \mathsf{IsReal(v,\ x)} * \lightning(\epsilon(x))\}}$$

1. Get $\mathsf{stepsLeft}(\mathsf{k}_1)$ for some k_1.
2. Get thin-air $\lightning(\epsilon')$ error credit for some $\epsilon' > 0$.
3. By Riemann integrability, $\exists k_2 > 0$ such that the integral is within ϵ' of any Riemann sum using partitions of width $< 1/2^{k_2}$.
4. Set $N = \max(k_1, k_2)$ and pre-sample N bits, using credit averaging so that when we draw $b_1, \ldots, b_N$, we end up with

   $$\lightning\left(b_1 \cdot 2^{-1} + b_2 \cdot 2^{-2} + \cdots + b_N \cdot 2^{-N}\right)$$

   The expected value sum is a Riemann sum that is within ϵ' of ϵ_0.

From U to Beyond

Once you have exact uniform samples and verify the computable real operations, can verify other samplers "just" by doing some integration.

Examples in paper:

- Beta(2,1)
- Exponential
- Gaussian

Clutch: Relational Reasoning

Motivating example

$$\text{let } b = \text{flip in}$$

$$\lambda\_.\, b$$

Motivating example

$$\begin{array}{l}\mathsf{let}\ b = \mathsf{flip}\ \mathsf{in}\\ \lambda\_.\ b\end{array}$$

$$\begin{array}{l}\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in}\\ \lambda\_.\ \mathsf{match}\ !\,r\ \mathsf{with}\\ \qquad \mathsf{Some}(b) \Rightarrow b\\ \qquad \mid \mathsf{None} \quad \Rightarrow \mathsf{let}\ b = \mathsf{flip}\ \mathsf{in}\\ \qquad\qquad\qquad\quad\ \ r \leftarrow \mathsf{Some}(b);\\ \qquad\qquad\qquad\quad\ \ b\\ \qquad \mathsf{end}\end{array}$$

Motivating example

$$
\begin{array}{l}
\mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\lambda\_.\ b
\end{array}
\quad \simeq \quad
\begin{array}{l}
\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\
\lambda\_.\ \mathsf{match}\ !\,r\ \mathsf{with} \\
\qquad \mathsf{Some}(b) \Rightarrow b \\
\qquad \mid \mathsf{None} \quad \Rightarrow \mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\qquad\qquad\qquad\quad\ \ r \leftarrow \mathsf{Some}(b); \\
\qquad\qquad\qquad\quad\ \ b \\
\qquad \mathsf{end}
\end{array}
$$

Motivating example

$$
\begin{array}{l}
\mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\lambda\_.\ b
\end{array}
\quad \simeq \quad
\begin{array}{l}
\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\
\lambda\_.\ \mathsf{match}\ !\,r\ \mathsf{with} \\
\qquad \mathsf{Some}(b) \Rightarrow b \\
\qquad \mid \mathsf{None} \Rightarrow \mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\qquad\qquad\qquad\quad r \leftarrow \mathsf{Some}(b); \\
\qquad\qquad\qquad\quad b \\
\qquad \mathsf{end}
\end{array}
$$

While this example seems esoteric, the pattern shows up in many places: cryptographic security, hash functions, lazily-sampled big integers, ...

What does equivalence mean formally?

Want to say that a "well-behaved client program" can't "distinguish" these two programs.

To define this, we formalized **well-typed contexts**:

1. Define contexts $\mathcal{C}$ – "program with a hole in it"

What does equivalence mean formally?

Want to say that a "well-behaved client program" can't "distinguish" these two programs.

To define this, we formalized **well-typed contexts**:

1. Define contexts $\mathcal{C}$ – "program with a hole in it"
2. Define filling operation $\mathcal{C}[e]$ – "plug e in hole in $\mathcal{C}$"
3. Extend type system to types on contexts:

$$\mathcal{C} : (\Gamma \vdash \tau) \Rightarrow (\emptyset \vdash \tau')$$

"if e has type $\Gamma \vdash e : \tau$, then $\emptyset \vdash \mathcal{C}[e] : \tau$"

Contextual equivalence and contextual refinement: classical case

Next, define contextual refinement.

Classical, **non-probabilistic case**:

$$\Gamma \vdash e_1 \precsim_{\text{ctx}} e_2 : \tau \quad \triangleq \quad \forall \tau', (\mathcal{C} : (\Gamma \vdash \tau) \Rightarrow (\emptyset \vdash \text{bool})), \sigma, b.$$
$$(\mathcal{C}[e_1], \sigma) \Downarrow b \Rightarrow \mathcal{C}[e_2], \sigma) \Downarrow b$$

Contextual equivalence and contextual refinement: classical case

Next, define contextual refinement.

Classical, **non-probabilistic case**:

$$\Gamma \vdash e_1 \precsim_{\text{ctx}} e_2 : \tau \quad \triangleq \quad \forall \tau', (\mathcal{C} : (\Gamma \vdash \tau) \Rightarrow (\emptyset \vdash \text{bool})), \sigma, b. \\ (\mathcal{C}[e_1], \sigma) \Downarrow b \Rightarrow \mathcal{C}[e_2], \sigma) \Downarrow b$$

or equivalently:

$$\Gamma \vdash e_1 \precsim_{\text{ctx}} e_2 : \tau \quad \triangleq \quad \forall \tau', (\mathcal{C} : (\Gamma \vdash \tau) \Rightarrow (\emptyset \vdash \tau')), \sigma. \\ (\Downarrow \mathcal{C}[e_1], \sigma) \Rightarrow (\Downarrow \mathcal{C}[e_2], \sigma)$$

Contextual equivalence $\Gamma \vdash e_1 \simeq_{\text{ctx}} e_2 : \tau$ defined as refinement in both directions.

Contextual equivalence and contextual refinement: probabilistic case

For probabilistic programs, generalize the second definition to talk about lower bound of termination probability:

$$\Gamma \vdash e_1 \precsim_{\text{ctx}} e_2 : \tau \quad \triangleq \quad \forall \tau', (\mathcal{C} : (\Gamma \vdash \tau) \Rightarrow (\emptyset \vdash \tau')), \sigma. \\ (\Downarrow \mathcal{C}[e_1], \sigma) \leq (\Downarrow \mathcal{C}[e_2], \sigma)$$

$\Gamma \vdash e_1 \simeq_{\text{ctx}} e_2 : \tau$ is again defined as refinement in both directions.

Proving contextual equivalence is hard

Contextual equivalence quantifies over **all** well-typed contexts $\mathcal{C}$:

$$\Gamma \vdash e_1 \precsim_{\text{ctx}} e_2 : \tau \quad \triangleq \quad \forall \tau', (\mathcal{C} : (\Gamma \vdash \tau) \Rightarrow (\emptyset \vdash \tau')), \sigma. \\ (\Downarrow \mathcal{C}[e_1], \sigma) \leq (\Downarrow \mathcal{C}[e_2], \sigma)$$

- How can we reason about all such $\mathcal{C}$?
- How do we exploit the typing assumption?

Logical Relations

Logical relations are a standard technique for proving contextual equivalences.

Define a **semantic model** of a typing system:

- Unary case: "what does it mean for a program e to have type τ"
- Binary case: "what does it mean for two programs e_1 and e_2 to be related/equivalent at type τ"

How to build a logical relation?

Defining a logical relation "with your bare hands" is not easy for a language with:

- higher-order state
- recursive types
- probabilistic choice

Program Logic Approach to Logical Relations

A general recipe for logical relations for "hard" languages.

For **unary** logical relations:

1. Start with an expressive Hoare logic $\{P\}\ e\ \{Q\}$ for this language.

Program Logic Approach to Logical Relations

A general recipe for logical relations for "hard" languages.

For **unary** logical relations:

1. Start with an expressive Hoare logic $\{P\}\ e\ \{Q\}$ for this language.
2. Define interpretation of typed values $[\![v : \tau]\!]$ from types to assertions.

Program Logic Approach to Logical Relations

A general recipe for logical relations for "hard" languages.

For **unary** logical relations:

1. Start with an expressive Hoare logic $\{P\}\ e\ \{Q\}$ for this language.
2. Define interpretation of typed values $[\![v : \tau]\!]$ from types to assertions.
3. Lift to interpretation $[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau]\!]$ in terms of Hoare triple:

$$\forall v_1, \ldots, v_n.\ \{[\![v_1 : \tau_1]\!] * \cdots * [\![v_n : \tau_n]\!]\}\ e[v_1/x_1] \cdots [v_n/x_n]\ \{x.[\![x : \tau]\!]\}$$

Program Logic Approach to Logical Relations

A general recipe for logical relations for "hard" languages.

For **unary** logical relations:

1. Start with an expressive Hoare logic $\{P\}\ e\ \{Q\}$ for this language.
2. Define interpretation of typed values $[\![v : \tau]\!]$ from types to assertions.
3. Lift to interpretation $[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau]\!]$ in terms of Hoare triple:

$$\forall v_1, \ldots, v_n.\ \{[\![v_1 : \tau_1]\!] * \cdots * [\![v_n : \tau_n]\!]\}\ e[v_1/x_1] \cdots [v_n/x_n]\ \{x.[\![x : \tau]\!]\}$$

4. Prove fundamental lemma: if $\Gamma \vdash e : \tau$ then $[\![\Gamma \vdash e : \tau]\!]$

Program Logic Approach to Logical Relations

A general recipe for logical relations for "hard" languages.

For **unary** logical relations:

1. Start with an expressive Hoare logic $\{P\}\ e\ \{Q\}$ for this language.
2. Define interpretation of typed values $[\![v : \tau]\!]$ from types to assertions.
3. Lift to interpretation $[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau]\!]$ in terms of Hoare triple:

$$\forall v_1, \ldots, v_n.\ \{[\![v_1 : \tau_1]\!] * \cdots * [\![v_n : \tau_n]\!]\}\ e[v_1/x_1] \cdots [v_n/x_n]\ \{x.[\![x : \tau]\!]\}$$

4. Prove fundamental lemma: if $\Gamma \vdash e : \tau$ then $[\![\Gamma \vdash e : \tau]\!]$
5. Deduce property of interest from soundness of Hoare logic.

Relational Hoare Logic

To apply this recipe for binary logical relations, we need a **relational program logic**.

For non-probabilistic programs, Benton proposed relational hoare logic (RHL):

$$\{P\}\ e_1 \sim e_2\ \{x_1, x_2.Q\}$$

Now P, Q are **relations** on program states.

Read: if we execute e_1 and e_2 in states related by P, and they terminate with values v_1, v_2, then their terminal states will be related by $Q[v_1/x_1][v_2/x_2]$

Probabilistic Relational Hoare Logic

Barthe et al. introduced probabilistic Relational Hoare Logic (pRHL)

Extends RHL with so-called coupling rule:

$$\frac{f : \mathsf{Bool} \to \mathsf{Bool} \text{ is a bijection}}{\{\mathsf{True}\}\ \mathsf{flip} \sim \mathsf{flip}\ \{v_1, v_2.\ v_1 = f(v_2)\}}\ \textsc{pRHL-couple}$$

Why is it called the coupling rule?

A coupling of distributions $\mu_1 \in \mathcal{D}(A)$ and $\mu_2 \in \mathcal{D}(B)$ is a distribution $\mu \in \mathcal{D}(A \times B)$ such that:

- $\sum_{b \in B} \mu(a, b) = \mu_1(a)$ for all $a \in A$
- $\sum_{a \in A} \mu(a, b) = \mu_2(b)$ for all $b \in B$

i.e. the marginals of μ are μ_1 and μ_2.

Why is it called the coupling rule?

A coupling of distributions $\mu_1 \in \mathcal{D}(A)$ and $\mu_2 \in \mathcal{D}(B)$ is a distribution $\mu \in \mathcal{D}(A \times B)$ such that:

- $\sum_{b \in B} \mu(a, b) = \mu_1(a)$ for all $a \in A$
- $\sum_{a \in A} \mu(a, b) = \mu_2(b)$ for all $b \in B$

i.e. the marginals of μ are μ_1 and μ_2.

Given a relation $R \subseteq A \times B$, μ is an R-coupling of μ_1 and μ_2 if additionally $\mathrm{supp}(\mu) \subseteq R$.

Why is it called the coupling rule?

A coupling of distributions $\mu_1 \in \mathcal{D}(A)$ and $\mu_2 \in \mathcal{D}(B)$ is a distribution $\mu \in \mathcal{D}(A \times B)$ such that:

- $\sum_{b \in B} \mu(a, b) = \mu_1(a)$ for all $a \in A$
- $\sum_{a \in A} \mu(a, b) = \mu_2(b)$ for all $b \in B$

i.e. the marginals of μ are μ_1 and μ_2.

Given a relation $R \subseteq A \times B$, μ is an R-coupling of μ_1 and μ_2 if additionally $\text{supp}(\mu) \subseteq R$.

The pRHL coupling rule is constructing a coupling between the two flip.

Soundness theorem of pRHL

Recall from Part 1 the evaluation distribution: $(e, \sigma) \Downarrow \mu$ with $\mu \in \mathcal{D}(\mathit{Cfg})$.

Soundness theorem of pRHL

Recall from Part 1 the evaluation distribution: $(e, \sigma) \Downarrow \mu$ with $\mu \in \mathcal{D}(Cfg)$.

Soundness: if $\{P\}\ e_1 \sim e_2\ \{v_1, v_2.\ Q\}$ holds, then for all $(\sigma_1, \sigma_2) \in P$, whenever $(e_1, \sigma_1) \Downarrow \mu_1$ and $(e_2, \sigma_2) \Downarrow \mu_2$:

$$\mu_1 \sim \mu_2 : R_Q$$

where $R_Q = \{((v_1, \sigma_1'), (v_2, \sigma_2')) \mid (\sigma_1', \sigma_2') \in Q[v_1/x_1][v_2/x_2]\}$.

Soundness theorem of pRHL

Recall from Part 1 the evaluation distribution: $(e, \sigma) \Downarrow \mu$ with $\mu \in \mathcal{D}(Cfg)$.

Soundness: if $\{P\}\ e_1 \sim e_2\ \{v_1, v_2.\ Q\}$ holds, then for all $(\sigma_1, \sigma_2) \in P$, whenever $(e_1, \sigma_1) \Downarrow \mu_1$ and $(e_2, \sigma_2) \Downarrow \mu_2$:

$$\mu_1 \sim \mu_2 : R_Q$$

where $R_Q = \{((v_1, \sigma_1'), (v_2, \sigma_2')) \mid (\sigma_1', \sigma_2') \in Q[v_1/x_1][v_2/x_2]\}$.

i.e. a derivable pRHL quadruple yields an R_Q-coupling of the two evaluation distributions.

Why do we care about constructing a coupling?

An R-coupling lets us transfer a relational property from paired samples to the full distributions.

Why do we care about constructing a coupling?

An R-coupling lets us transfer a relational property from paired samples to the full distributions.

Most importantly, take R to be equality:

$$\text{if } \mu_1 \sim \mu_2 : (=), \text{ then } \mu_1 = \mu_2.$$

Why do we care about constructing a coupling?

An R-coupling lets us transfer a relational property from paired samples to the full distributions.

Most importantly, take R to be equality:

$$\text{if } \mu_1 \sim \mu_2 : (=), \text{ then } \mu_1 = \mu_2.$$

So if pRHL gives us $\{P\}\ e_1 \sim e_2\ \{v_1, v_2.\ v_1 = v_2\}$, we can conclude that e_1 and e_2 produce the same output distribution.

Limitation of the Coupling Rule

The pRHL coupling rule requires us to "synchronize" the probabilistic choices:

$$\frac{f : \mathsf{Bool} \to \mathsf{Bool} \text{ is a bijection}}{\{\mathsf{True}\}\ \mathsf{flip} \sim \mathsf{flip}\ \{v_1, v_2.\ v_1 = f(v_2)\}}\ \textsc{pRHL-couple}$$

Motivating example – Synchronous Couplings Insufficient

$$\texttt{let } b = \texttt{flip in}\quad \lambda\_.\, b \qquad \simeq$$

$$\begin{array}{l}
\texttt{let } r = \texttt{ref}(\texttt{None}) \texttt{ in} \\
\lambda_.\ \texttt{match } !\, r \texttt{ with} \\
\qquad \texttt{Some}(b) \Rightarrow b \\
\qquad \mid \texttt{None} \quad \Rightarrow \texttt{let } b = \texttt{flip in} \\
\qquad\qquad\qquad\quad\ \ r \leftarrow \texttt{Some}(b); \\
\qquad\qquad\qquad\quad\ \ b \\
\qquad \texttt{end}
\end{array}$$

Motivating example – Synchronous Couplings Insufficient

$$\text{let } b = \text{flip in}$$
$$\lambda\_.\ b$$

$$\simeq$$

$$\begin{array}{l}
\text{let } r = \text{ref}(\text{None}) \text{ in} \\
\lambda_.\ \text{match } !\, r \text{ with} \\
\qquad \text{Some}(b) \Rightarrow b \\
\qquad \mid \text{None} \quad \Rightarrow \text{let } b = \text{flip in} \\
\qquad\qquad\qquad\quad\ \ r \leftarrow \text{Some}(b); \\
\qquad\qquad\qquad\quad\ \ b \\
\qquad \text{end}
\end{array}$$

Motivating example – Synchronous Couplings Insufficient

$$\lambda\_.\, b \quad \simeq$$

```
let r = ref(None) in
λ_. match !r with
      Some(b) ⇒ b
    | None    ⇒ let b = flip in
                r ← Some(b);
                b
    end
```

Clutch – Relational Reasoning with Asynchronous Couplings

Clutch is a logic that allows proving contextual equivalence of

... probabilistic programs written in an expressive programming language

... using higher-order separation logic,

... and asynchronous probabilistic couplings

More specifically:

Clutch – Relational Reasoning with Asynchronous Couplings

Clutch is a logic that allows proving contextual equivalence of

... probabilistic programs written in an expressive programming language

... using higher-order separation logic,

... and asynchronous probabilistic couplings

More specifically:

1. A probabilistic relational separation logic on top of Iris

Clutch – Relational Reasoning with Asynchronous Couplings

Clutch is a logic that allows proving contextual equivalence of

... probabilistic programs written in an expressive programming language
... using higher-order separation logic,
... and asynchronous probabilistic couplings

More specifically:

1. A probabilistic relational separation logic on top of Iris
2. A logical refinement judgment (a "logical" logical relation)

$$\Gamma \vDash e_1 \precsim e_2 : \tau$$

that implies contextual refinement.

Refinement judgment

The judgment

$$\Gamma \vDash e_1 \precsim e_2 : \tau$$

should be read as "in env. Γ, expression e_1 logically refines expression e_2 at type τ".

Refinement judgment

The judgment

$$\Gamma \vDash e_1 \precsim e_2 : \tau$$

should be read as "in env. Γ, expression e_1 logically refines expression e_2 at type τ".

Theorem (Fundamental)
If $\Gamma \vdash e : \tau$ *then* $\Gamma \vDash e \precsim e : \tau$.

Refinement judgment

The judgment

$$\Gamma \vDash e_1 \precsim e_2 : \tau$$

should be read as "in env. Γ, expression e_1 logically refines expression e_2 at type τ".

Theorem (Fundamental)
If $\Gamma \vdash e : \tau$ *then* $\Gamma \vDash e \precsim e : \tau$.

Theorem (Soundness)
If $\Gamma \vDash e_1 \precsim e_2 : \tau$ *then* $\Gamma \vdash e_1 \precsim_{ctx} e_2 : \tau$.

Symbolic execution rules

Symbolic execution rules

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Gamma \vDash K[e_1'] \precsim e_2 : \tau}{\Gamma \vDash K[e_1] \precsim e_2 : \tau}$$

Symbolic execution rules

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Gamma \vDash K[e_1'] \precsim e_2 : \tau}{\Gamma \vDash K[e_1] \precsim e_2 : \tau}$$

Symbolic execution rules

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Gamma \vDash K[e_1'] \precsim e_2 : \tau}{\Gamma \vDash K[e_1] \precsim e_2 : \tau}$$

Symbolic execution rules

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Gamma \vDash K[e_1'] \precsim e_2 : \tau}{\Gamma \vDash K[e_1] \precsim e_2 : \tau}$$

Symbolic execution rules

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Gamma \vDash K[e_1'] \precsim e_2 : \tau}{\Gamma \vDash K[e_1] \precsim e_2 : \tau}$$

$$\frac{\ell \mapsto v \qquad \ell \mapsto v \mathrel{-\!\!*} \Gamma \vDash K[v] \precsim e_2 : \tau}{\Gamma \vDash K[!\ell] \precsim e_2 : \tau}$$

Symbolic execution rules

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Gamma \vDash K[e_1'] \precsim e_2 : \tau}{\Gamma \vDash K[e_1] \precsim e_2 : \tau}$$

$$\frac{\ell \mapsto v \qquad \ell \mapsto v \mathrel{-\!\!*} \Gamma \vDash K[v] \precsim e_2 : \tau}{\Gamma \vDash K[!\ell] \precsim e_2 : \tau}$$

Symbolic execution rules

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Gamma \vDash K[e_1'] \precsim e_2 : \tau}{\Gamma \vDash K[e_1] \precsim e_2 : \tau}$$

$$\frac{\ell \mapsto v \qquad \ell \mapsto v \mathrel{-\!\!*} \Gamma \vDash K[v] \precsim e_2 : \tau}{\Gamma \vDash K[!\ell] \precsim e_2 : \tau}$$

Symbolic execution rules

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Gamma \vDash K[e_1'] \precsim e_2 : \tau}{\Gamma \vDash K[e_1] \precsim e_2 : \tau}$$

$$\frac{\ell \mapsto v \qquad \ell \mapsto v \mathrel{-\!\!*} \Gamma \vDash K[v] \precsim e_2 : \tau}{\Gamma \vDash K[!\ell] \precsim e_2 : \tau}$$

Symbolic execution rules

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Gamma \vDash K[e_1'] \precsim e_2 : \tau}{\Gamma \vDash K[e_1] \precsim e_2 : \tau}$$

$$\frac{\ell \mapsto v \qquad \ell \mapsto v \mathrel{-\!\!*} \Gamma \vDash K[v] \precsim e_2 : \tau}{\Gamma \vDash K[!\ell] \precsim e_2 : \tau}$$

$$\frac{\forall b.\, \Gamma \vDash K[b] \precsim K'[b] : \tau}{\Gamma \vDash K[\,\mathsf{flip}\,] \precsim K'[\,\mathsf{flip}\,] : \tau}$$

Symbolic execution rules

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Gamma \vDash K[e_1'] \precsim e_2 : \tau}{\Gamma \vDash K[e_1] \precsim e_2 : \tau} \qquad \frac{\ell \mapsto v \qquad \ell \mapsto v \mathrel{-\!\!*} \Gamma \vDash K[v] \precsim e_2 : \tau}{\Gamma \vDash K[!\ell] \precsim e_2 : \tau}$$

$$\frac{\forall b.\, \Gamma \vDash K[b] \precsim K'[b] : \tau}{\Gamma \vDash K[\,\mathsf{flip}\,] \precsim K'[\,\mathsf{flip}\,] : \tau}$$

Symbolic execution rules

$$\frac{e_1 \overset{\text{pure}}{\rightsquigarrow} e_1' \qquad \Gamma \vDash K[e_1'] \precsim e_2 : \tau}{\Gamma \vDash K[e_1] \precsim e_2 : \tau} \qquad \frac{\ell \mapsto v \qquad \ell \mapsto v \mathrel{-\!\!*} \Gamma \vDash K[v] \precsim e_2 : \tau}{\Gamma \vDash K[!\ell] \precsim e_2 : \tau}$$

$$\frac{\forall b.\, \Gamma \vDash K[b] \precsim K'[b] : \tau}{\Gamma \vDash K[\,\mathsf{flip}\,] \precsim K'[\,\mathsf{flip}\,] : \tau}$$

Asynchronous couplings

With presampling tapes, we can synchronously couple tape samplings with program samplings

$$\frac{\iota \hookrightarrow \vec{b} \qquad \forall b.\, \iota \hookrightarrow \vec{b} \cdot b \twoheadrightarrow \Gamma \vDash e \precsim K[b] : \tau}{\Gamma \vDash e \precsim K[\,\mathsf{flip}\,] : \tau}$$

to couple program samplings asynchronously!

Asynchronous couplings

With presampling tapes, we can synchronously couple tape samplings with program samplings

$$\frac{\iota \hookrightarrow \vec{b} \qquad \forall b.\, \iota \hookrightarrow \vec{b} \cdot b \twoheadrightarrow \Gamma \vDash e \precsim K[b] : \tau}{\Gamma \vDash e \precsim K[\mathsf{flip}] : \tau}$$

to couple program samplings asynchronously!

Asynchronous couplings

With presampling tapes, we can synchronously couple tape samplings with program samplings

$$\frac{\iota \hookrightarrow \vec{b} \qquad \forall b.\, \iota \hookrightarrow \vec{b} \cdot b \twoheadrightarrow \Gamma \vDash e \precsim K[b] : \tau}{\Gamma \vDash e \precsim K[\mathsf{flip}] : \tau}$$

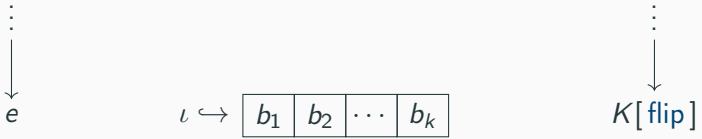
to couple program samplings asynchronously!

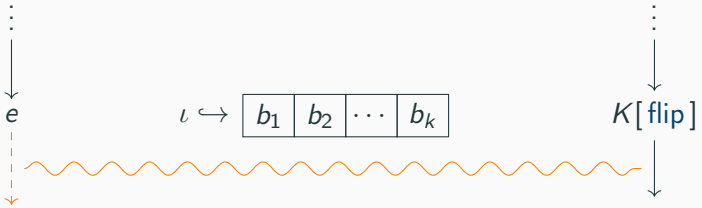
Asynchronous couplings

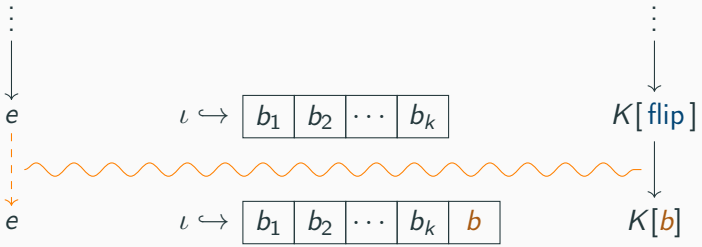
With presampling tapes, we can synchronously couple tape samplings with program samplings

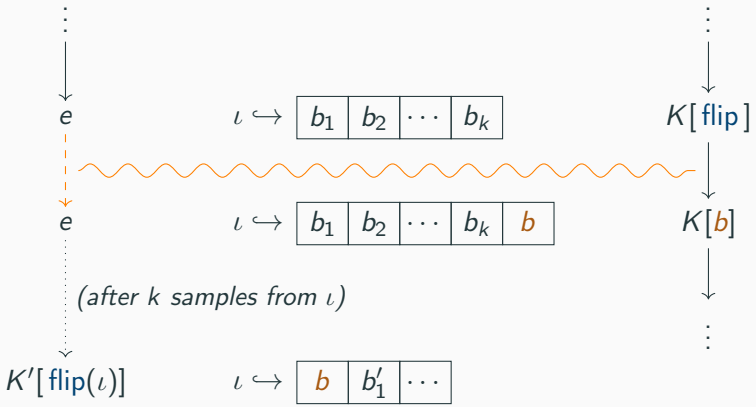
$$\frac{\iota \hookrightarrow \vec{b} \qquad \forall b.\, \iota \hookrightarrow \vec{b} \cdot b \twoheadrightarrow \Gamma \vDash e \precsim K[b] : \tau}{\Gamma \vDash e \precsim K[\mathsf{flip}] : \tau}$$

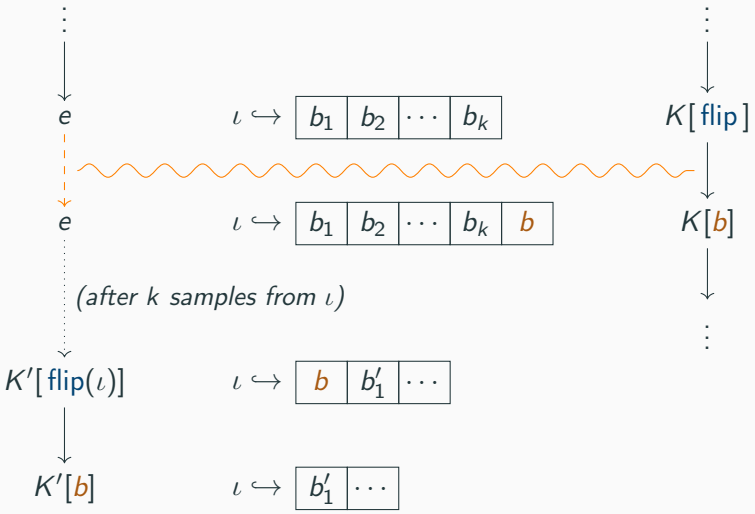
to couple program samplings asynchronously!


e
ι ↪
b1
b2
· · ·
bk
K [flip]


e
$\iota \hookrightarrow$
b_1
b_2
$\cdots$
b_k
$K[\,\mathsf{flip}\,]$


e
e
$\iota \hookrightarrow$
b_1 b_2 $\cdots$ b_k
$\iota \hookrightarrow$
b_1 b_2 $\cdots$ b_k b
$K[\mathsf{flip}]$
$K[b]$


e
e
(after k samples from ι)
K′[flip(ι)]
ι ↪
b1 b2 · · · bk
ι ↪
b1 b2 · · · bk b
ι ↪
b b′1 · · ·
K[flip]
K[b]


e
e
(after k samples from ι)
K′[flip(ι)]
K′[b]
ι ↪ b1 b2 · · · bk
ι ↪ b1 b2 · · · bk b
ι ↪ b b′1 · · ·
ι ↪ b′1 · · ·
K[flip]
K[b]

Motivating example cont'd

$$
\begin{array}{l}
\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\
\lambda\_.\ \mathsf{match}\ !\, r\ \mathsf{with} \\
\qquad \mathsf{Some}(b) \Rightarrow b \\
\qquad \mid \mathsf{None} \Rightarrow \mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\qquad\qquad\qquad\quad r \leftarrow \mathsf{Some}(b); \\
\qquad\qquad\qquad\quad b \\
\qquad \mathsf{end}
\end{array}
\qquad \precsim_{\mathsf{ctx}} \qquad
\begin{array}{l}
\mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\lambda\_.\ b
\end{array}
$$

Motivating example cont'd

$$
\begin{array}{l}
\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\
\lambda\_.\ \mathsf{match}\ !\, r\ \mathsf{with} \\
\quad \mathsf{Some}(b) \Rightarrow b \\
\quad \mid \mathsf{None} \Rightarrow \mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\qquad\qquad\quad r \leftarrow \mathsf{Some}(b); \\
\qquad\qquad\quad b \\
\quad \mathsf{end}
\end{array}
\precsim_{\mathsf{ctx}}
\begin{array}{l}
\mathsf{let}\ \iota = \mathsf{tape}\ \mathsf{in} \\
\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\
\lambda\_.\ \mathsf{match}\ !\, r\ \mathsf{with} \\
\quad \mathsf{Some}(b) \Rightarrow b \\
\quad \mid \mathsf{None} \Rightarrow \mathsf{let}\ b = \mathsf{flip}(\iota)\ \mathsf{in} \\
\qquad\qquad\quad r \leftarrow \mathsf{Some}(b); \\
\qquad\qquad\quad b \\
\quad \mathsf{end}
\end{array}
\precsim_{\mathsf{ctx}}
\begin{array}{l}
\mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\lambda\_.\ b
\end{array}
$$

Motivating example cont'd

$$
\begin{array}{l}
\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\
\lambda\_.\ \mathsf{match}\ !\,r\ \mathsf{with} \\
\quad \mathsf{Some}(b) \Rightarrow b \\
\quad \mid \mathsf{None} \Rightarrow \mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\qquad\qquad\quad r \leftarrow \mathsf{Some}(b); \\
\qquad\qquad\quad b \\
\quad \mathsf{end}
\end{array}
\precsim_{\mathsf{ctx}}
\begin{array}{l}
\mathsf{let}\ \iota = \mathsf{tape}\ \mathsf{in} \\
\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\
\lambda\_.\ \mathsf{match}\ !\,r\ \mathsf{with} \\
\quad \mathsf{Some}(b) \Rightarrow b \\
\quad \mid \mathsf{None} \Rightarrow \mathsf{let}\ b = \mathsf{flip}(\iota)\ \mathsf{in} \\
\qquad\qquad\quad r \leftarrow \mathsf{Some}(b); \\
\qquad\qquad\quad b \\
\quad \mathsf{end}
\end{array}
\precsim_{\mathsf{ctx}}
\begin{array}{l}
\mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\lambda\_.\ b
\end{array}
$$

Motivating example cont'd

$$
\begin{array}{l}
\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\
\lambda\_.\ \mathsf{match}\ !\, r\ \mathsf{with} \\
\quad \mathsf{Some}(b) \Rightarrow b \\
\quad \mid \mathsf{None} \Rightarrow \mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\qquad\qquad\quad r \leftarrow \mathsf{Some}(b); \\
\qquad\qquad\quad b \\
\quad \mathsf{end}
\end{array}
\precsim_{\mathsf{ctx}}
\begin{array}{l}
\mathsf{let}\ \iota = \mathsf{tape}\ \mathsf{in} \\
\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\
\lambda\_.\ \mathsf{match}\ !\, r\ \mathsf{with} \\
\quad \mathsf{Some}(b) \Rightarrow b \\
\quad \mid \mathsf{None} \Rightarrow \mathsf{let}\ b = \mathsf{flip}(\iota)\ \mathsf{in} \\
\qquad\qquad\quad r \leftarrow \mathsf{Some}(b); \\
\qquad\qquad\quad b \\
\quad \mathsf{end}
\end{array}
\precsim_{\mathsf{ctx}}
\begin{array}{l}
\mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\lambda\_.\ b
\end{array}
$$

Motivating example cont'd

$$
\begin{array}{l}
\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\
\lambda\_.\ \mathsf{match}\ !\, r\ \mathsf{with} \\
\quad \mathsf{Some}(b) \Rightarrow b \\
\quad \mid \mathsf{None} \Rightarrow \mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\qquad\qquad\quad r \leftarrow \mathsf{Some}(b); \\
\qquad\qquad\quad b \\
\quad \mathsf{end}
\end{array}
\precsim_{\mathsf{ctx}}
\begin{array}{l}
\mathsf{let}\ \iota = \mathsf{tape}\ \mathsf{in} \\
\mathsf{let}\ r = \mathsf{ref}(\mathsf{None})\ \mathsf{in} \\
\lambda\_.\ \mathsf{match}\ !\, r\ \mathsf{with} \\
\quad \mathsf{Some}(b) \Rightarrow b \\
\quad \mid \mathsf{None} \Rightarrow \mathsf{let}\ b = \mathsf{flip}(\iota)\ \mathsf{in} \\
\qquad\qquad\quad r \leftarrow \mathsf{Some}(b); \\
\qquad\qquad\quad b \\
\quad \mathsf{end}
\end{array}
\precsim_{\mathsf{ctx}}
\begin{array}{l}
\mathsf{let}\ b = \mathsf{flip}\ \mathsf{in} \\
\lambda\_.\ b
\end{array}
$$

Further examples:

- Cryptographic security (ElGamal)
- Lazy vs. Eager Hashing
- Lazily Sampled Big Integers
- Example from Sangiorgi and Vignudelli
- ...

Approximate Equivalences

By extending Clutch with error credits, we can prove approximate equivalences:

- Approxis: error credits to prove total variation bounds
- Clutch-DP: two forms of error credits to prove (ϵ, δ) differential privacy