Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, Pierre-Yves Strub

UPenn-Mines ParisTech, IMDEA Software Institute, Dundee

Gallium Seminar, Nov 17th, 2014
Motivation

Software Verification

- Reason *formally* about programs and their behavior.
- Increase trust in software, help programmers/designers.
- Has important practical and economical utility.
- Expressiveness? Automation?
Motivation

Software Verification

- Reason *formally* about programs and their behavior.
- Increase trust in software, help programmers/designers.
- Has important practical and economical utility.
- Expressiveness? Automation?

Today:

- Verification of probabilistic programs.
- *Truthfulness*: An agent gets best utility when telling the truth.
- *Privacy*: An agent’s information leak is bounded.
The Main Challenges

Relational Reasoning
Properties of interest are relational, that is, defined over \textit{two runs} of the \textit{same program}:

- \textit{Truthfulness}: agent telling the truth vs not.
- \textit{Privacy}: run including the agent vs not.
The Main Challenges

Relational Reasoning
Properties of interest are relational, that is, defined over two runs of the same program:
- Truthfulness: agent telling the truth vs not.
- Privacy: run including the agent vs not.

Probabilistic Reasoning
Interesting algorithms are randomized, properties rely on:
- Expected values.
- Distance on distributions.
Our Approach:

Related/Precursor Work:

- Relational logics.
- F*, RF*.
- CertiCrypt/CertiPriv.
- Fuzz/DFuzz.

Our Contributions:

- Extended type system: Support for Higher-Order refinements.
- Embedding of logical relations: DFuzz soundness proof.
- Probabilistic approximate types.
- New application domain and examples.
- Prototype implementation.
Our Approach:

Related/Precursor Work:

▶ Relational logics.
▶ F*, RF*.
▶ CertiCrypt/CertiPriv.
▶ Fuzz/DFuzz.

Our Contributions

▶ Extended type system:
  ▶ Support for Higher-Order refinements.
  ▶ Embedding of logical relations! DFuzz soundness proof.
  ▶ Probabilistic approximate types.
▶ New application domain and examples.
▶ Prototype implementation.
The System: Relational Refinement Types

Variables
Relational variables, \( x \in \mathcal{X}_R \); left/right instances \( x_\prec, x_\succ \in \mathcal{X}_R^\times \).

Expressions
\[
e^m ::= \begin{array}{l} C \mid x \in \mathcal{X}^m \mid e \mid \lambda x. e \mid \text{case } e \text{ with } [ε \Rightarrow e \mid x :: x \Rightarrow e] \mid \text{letrec}^\uparrow f x = e \mid \text{letrec}^\downarrow f x = e \mid e^\uparrow \mid \text{let}^\uparrow x = e \text{ in } e \mid \text{unit}_M e \mid \text{bind}_M x = e \text{ in } e \end{array}
\]

Regular Types
\[
\begin{array}{l}
\tilde{\tau}, \tilde{\sigma}, \ldots \in \text{CoreTy} ::= \begin{array}{l} \bullet \mid \mathbb{B} \mid \mathbb{N} \mid \overline{\mathbb{R}} \mid \overline{\mathbb{R}}^+ \mid L[\tilde{\tau}] \end{array} \\
\tau, \sigma, \ldots \in \text{Ty} ::= \tilde{\tau} \mid M[\tau] \mid C[\tau] \mid \tau \rightarrow \sigma
\end{array}
\]

Relational Refinement Types
\[
\begin{array}{l}
T, U \in \mathcal{T} ::= \tilde{\tau} \mid M_{\varepsilon, \delta}[T] \mid C[T] \mid \Pi(x :: T). T \mid \{x :: T \mid \phi\} \\
\phi, \psi \in \mathcal{A} ::= \begin{array}{l} Q(x : \tau). \phi \mid Q(x :: T). \phi \mid C(\phi_1, \ldots, \phi_n) \mid e^\bowtie = e^\bowtie \mid e^\bowtie \leq e^\bowtie
\end{array}
\end{array}
\]

\[
C = \{\top/0, \bot/0, \neg/1, \vee/2, \wedge/2, \Rightarrow/2\}
\]
Regular refinement types no enough to capture some properties.

$k$-sensitive function

∀x₁, x₂. |f(x₁) − f(x₂)| ≤ k · |x₁ − x₂|
Relational Refinement Types: Example

Regular refinement types no enough to capture some properties.

$k$-sensitive function

\[
\forall x_1, x_2. |f(x_1) - f(x_2)| \leq k |x_1 - x_2|
\]
Relational Refinement Types: Example

Regular refinement types no enough to capture some properties.

$k$-sensitive function

\[
\forall x_1, x_2. |f(x_1) - f(x_2)| \leq k \cdot |x_1 - x_2|
\]

What should the type for $f$ be?
Regular refinement types no enough to capture some properties.

\[ k\text{-sensitive function} \]

\[ \forall x_1, x_2. |f(x_1) - f(x_2)| \leq k \cdot |x_1 - x_2| \]
Relational Refinement Types: Example

Regular refinement types no enough to capture some properties.

$k$-sensitive function

\[ \forall x_1, x_2. |f(x_1) - f(x_2)| \leq k \cdot |x_1 - x_2| \]

What should the type for $f$ be?
Relational Refinement Types: Example

For the property:

\[ \forall x_1, x_2. |f(x_1) - f(x_2)| \leq k \cdot |x_1 - x_2| \]
Relational Refinement Types: Example

For the property:

\[ \forall x_1, x_2. |f(x_1) - f(x_2)| \leq k \cdot |x_1 - x_2| \]

we can do a refinement at a higher type:

\[ \{ f : \mathbb{R} \rightarrow \mathbb{R} \mid \forall x :: \mathbb{R}. |f(x_{\downarrow}) - f(x_{\uparrow})| \leq k \cdot |x_{\downarrow} - x_{\uparrow}| \} \]
For the property:

\[ \forall x_1, x_2. |f(x_1) - f(x_2)| \leq k \cdot |x_1 - x_2| \]

we can do a refinement at a higher type:

\[ \{ f : \mathbb{R} \rightarrow \mathbb{R} \mid \forall x :: \mathbb{R}. |f(x_{\triangleleft}) - f(x_{\triangleright})| \leq k \cdot |x_{\triangleleft} - x_{\triangleright}| \} \]

or we can refer to two copies of the input:

\[ f : \Pi(x :: \mathbb{R}). \{ r :: \mathbb{R} \mid k \cdot |r_{\triangleleft} - r_{\triangleright}| \leq |x_{\triangleleft} - x_{\triangleright}| \} \]

Both types are equivalent in our system, but the pre/post style more convenient for reasoning.
Semantic subtyping for non-relational types:

\[ \vdash e : T \quad \Gamma \models \phi[x/e] \]

\[ \vdash e : \{x : T \mid \phi\} \]
The System: Semantics

Semantic subtyping for non-relational types:

\[ \vdash e : T \quad \Gamma \models \phi[x/e] \quad \vdash e : \{x : T \mid \phi\} \quad \vdash e : T \Rightarrow e \in [T] \]
The System: Semantics

Semantic subtyping for non-relational types:

\[
\begin{align*}
\Gamma &\vdash e : T &\quad &\vdash \phi(x/e) \\
\vdash e : \{x : T \mid \phi\} &\quad &\vdash e : T \Rightarrow e \in [T] \\
\vdash \phi(v) &\quad &v \in [T] \Rightarrow v \in \{x : T \mid \phi(x)\} \\
\end{align*}
\]
The System: Semantics

Semantic subtyping for non-relational types:

\[
\begin{align*}
\frac{\vdash e : T \quad \Gamma \models \phi[x/e]}{\vdash e : \{x : T \mid \phi\}} & \quad \frac{\vdash e : T \Rightarrow e \in [T]}{\vdash \phi(v)}
\end{align*}
\]

Semantic subtyping for HO relational types:

\[
\begin{align*}
\langle T \rangle_\theta \subseteq [\| T \|] \times [\| T \|]
\end{align*}
\]

\[
\begin{align*}
(d_1, d_2) \in [\tau] \times [\tau] & \quad \frac{(d_1, d_2) \in \langle T \rangle_\theta \quad \phi_\theta(x_\downarrow \mapsto d_1, x_\uparrow \mapsto d_2)}{(f_1, f_2) \in [\| T \| \rightarrow \| U \|] \quad \forall (d_1, d_2) \in \langle T \rangle_\theta. (f_1(d_1), f_2(d_2)) \in \langle U \rangle_\theta \quad \langle \Pi(x :: T). U \rangle_\theta & \quad \frac{(f_1, f_2) \in [\| \Pi(x :: T). U \|]_\theta}{(f_1, f_2) \in [\Pi(x :: T). U \|]_\theta}
\end{align*}
\]
Subtyping

**Sub-Reflexivity**

\[
\frac{\Gamma \vdash T}{\Gamma \vdash T \leq T}
\]

**Sub-Transitivity**

\[
\frac{\Gamma \vdash T \leq U \quad \Gamma \vdash U \leq V}{\Gamma \vdash T \leq V}
\]

**Sub-Left**

\[
\frac{\Gamma \vdash \{x : T \mid \phi\}}{\Gamma \vdash \{x : T \mid \phi\} \leq T}
\]

**Sub-Right**

\[
\frac{\|\Gamma, x : U\| \vdash \phi \quad \forall \theta. \theta \vdash \Gamma, x : T \Rightarrow \llbracket \phi \rrbracket_{\theta}}{\Gamma \vdash T \leq \{x : U \mid \phi\}}
\]

**Sub-Product**

\[
\frac{\Gamma \vdash T_2 \leq T_1 \quad \Gamma, x : T_2 \vdash U_1 \leq U_2}{\Gamma \vdash \Pi(x : T_1). U_1 \leq \Pi(x : T_2). U_2}
\]
The System: Typing

The typing judgment relates two programs to a type:

\[ \mathcal{G} \vdash e_1 \sim e_2 :: T \]
The System: Typing

The typing judgment relates two programs to a type:

\[ \mathcal{G} \vdash e_1 \sim e_2 :: T \]

Soundness

\[ \mathcal{G} \vdash e_1 \sim e_2 :: T \Rightarrow \forall \mathcal{G} \vdash \theta, ([e_1]_\theta, [e_2]_\theta) \in (T)_\theta \]
The System: Typing

The typing judgment relates two programs to a type:

\[ \Gamma \vdash e_1 \sim e_2 :: T \]

Soundness

\[ \Gamma \vdash e_1 \sim e_2 :: T \Rightarrow \forall \Gamma \vdash \theta, ([e_1]_\theta, [e_2]_\theta) \in (T)_\theta \]

Synchronicity

In most cases programs are synchronous, so we use:

\[ \Gamma \vdash e :: T \equiv \Gamma \vdash e_\downarrow \sim e_\uparrow :: T \]

with \( e_\downarrow, e_\uparrow \) projecting the variables in \( e \).
Base Typing Rules

\[
\begin{align*}
\text{VAR} & \quad \frac{x :: T \in \text{dom}(\mathcal{G})}{\mathcal{G} \vdash x :: T} \\
\text{ABS} & \quad \frac{\mathcal{G}, x :: T \vdash e :: U}{\mathcal{G} \vdash \lambda x. e :: \Pi(x :: T). U} \\
\text{APP} & \quad \frac{\mathcal{G} \vdash e_f :: \Pi(x :: T). U \quad \mathcal{G} \vdash e_a :: T}{\mathcal{G} \vdash e_f \, e_a :: U\{x \mapsto e_a\}}
\end{align*}
\]
### Base Typing Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VAR</strong></td>
<td>$x :: T \in \text{dom}(G)$</td>
</tr>
<tr>
<td>$\vdash G \mid x :: T$</td>
<td></td>
</tr>
<tr>
<td><strong>ABS</strong></td>
<td>$g, x :: T \vdash e :: U$</td>
</tr>
<tr>
<td>$\vdash G \mid \lambda x. e :: \Pi(x :: T). U$</td>
<td></td>
</tr>
<tr>
<td><strong>APP</strong></td>
<td>$g \vdash e_f :: \Pi(x :: T). U$</td>
</tr>
<tr>
<td>$g \vdash e_a :: T$</td>
<td></td>
</tr>
<tr>
<td>$\vdash G \mid e_f e_a :: U{x \mapsto e_a}$</td>
<td></td>
</tr>
<tr>
<td><strong>CASE</strong></td>
<td>$g, x :: \tilde{\tau}, y :: L[\tilde{\tau}], {e_\downarrow = x_\downarrow :: y_\downarrow \land e_\uparrow = x_\uparrow :: y_\uparrow} \vdash e_2 :: T$</td>
</tr>
<tr>
<td>$\vdash G \mid \text{case } e \text{ with } [\epsilon \Rightarrow e_1 \mid x :: y \Rightarrow e_2] :: T$</td>
<td></td>
</tr>
</tbody>
</table>

Barthe et al. Verifying MD and DP
Typing Rules for Recursion

To ensure consistency at higher-types, we must embed non-terminating computations in the partiality monad:

\[
\begin{align*}
\text{LETREC} & \quad \text{SN-guard} \\
\frac{G, f :: \Pi(x :: T). \ U \vdash \lambda x. \ e :: \Pi(x :: T). \ U}{G \vdash \Pi(x :: T). \ U} & \quad \frac{G \vdash \Pi(x :: T). \ U}{G \vdash \text{letrec} \ f \ x = e :: \Pi(x :: T). \ U}
\end{align*}
\]
To ensure consistency at higher-types, we must embed non-terminating computations in the partiality monad:

\[
\begin{align*}
\text{\texttt{LETREC}} & \quad \frac{G, f :: \Pi(x :: T). U \vdash \lambda x. e :: \Pi(x :: T). U \quad G \vdash \Pi(x :: T). U}{\text{\texttt{SN}}\text{-guard}} \quad G \vdash \text{letrec} f x = e :: \Pi(x :: T). U \\
\text{\texttt{LETREC}} & \quad \frac{G \vdash \Pi(x :: T). \mathfrak{c}[U]}{G, f :: \Pi(x :: T). \mathfrak{c}[U] \vdash \lambda x. e :: \Pi(x :: T). \mathfrak{c}[U]} \quad G \vdash \text{letrec} f x = e :: \Pi(x :: T). \mathfrak{c}[U]
\end{align*}
\]
Typing Rules for Recursion

To ensure consistency at higher-types, we must embed non-terminating computations in the partiality monad:

\[ \vdash \text{letrec } f \ x = e :: \Pi(x :: T).U \]

\[ \text{LETREC} \quad \vdash \Pi(x :: T).U \quad \text{SN-guard} \]

\[ \vdash \text{letrec } f \ x = e :: \Pi(x :: T).U \]

\[ \vdash \Pi(x :: T).c[U] \]

\[ \vdash \text{letrec } f \ x = e :: \Pi(x :: T).c[U] \]

\[ \vdash e :: T \]

\[ \vdash e :: c[T] \]

\[ \vdash \text{let } x = e_1 \text{ in } e_2 :: c[T_2] \]
Asynchronous Rules

\[ \text{ASYM} \quad \frac{g \vdash e_1 \sim e_2 :: T}{g \leftrightarrow \vdash e_2 \leftrightarrow \sim e_1 \leftrightarrow :: T} \]

\[ \text{AREDLEFT} \quad \frac{e_1 \rightarrow e'_1 \quad g \vdash e_1 \sim e_2 :: T}{g \vdash e'_1 \sim e_2 :: T} \]
Asynchronous Rules

\[
G \vdash e_1 \sim e_2 :: T
\]

\[
G \leftrightarrow \vdash e_2 \leftrightarrow \sim e_1 \leftrightarrow :: T\leftrightarrow
\]

\[
e_1 \rightarrow e_1' \quad G \vdash e_1 \sim e_2 :: T
\]

\[
\vdash e_1' \sim e_2 :: T
\]

\[
|G| \vdash e : L[\bar{\tau}] \quad |G| \vdash e' : |T|
\]

\[
G, \{e_\triangleleft = \epsilon\} \vdash e_1 \sim e' :: T
\]

\[
G, x :: \bar{\tau}, y :: L[\bar{\tau}], \{e_\triangleleft = x_\triangleleft :: y_\triangleleft\} \vdash e_2 \sim e' :: T
\]

\[
G \vdash \text{case } e \text{ with } [\epsilon \Rightarrow e_1 \mid x :: y \Rightarrow e_2] \sim e' :: T
\]
Mechanism design is the study of algorithm design where the inputs to the algorithm are controlled by strategic agents, who must be incentivized to faithfully report them.
More on Mechanism Design

Mechanism design is the study of algorithm design where the inputs to the algorithm are controlled by strategic agents, who must be incentivized to faithfully report them.

Formally

- $n$ agents, with type for actions $A_i$, $i \in \{1, \ldots, n\}$.
- A mechanism $M : A^n \to \mathcal{O}$.
- A payoff for every agent $P_i : \mathcal{O} \to R^+$.
- Probabilistic algorithms are common! Payoff becomes expected payoff.
Mechanism design is the study of algorithm design where the inputs to the algorithm are controlled by strategic agents, who must be incentivized to faithfully report them.

Formally

- $n$ agents, with type for actions $A_i$, $i \in \{1, \ldots, n\}$.
- A mechanism $M : A^n \rightarrow \mathcal{O}$.
- A payoff for every agent $P_i : \mathcal{O} \rightarrow \mathbb{R}^+$.
- Probabilistic algorithms are common! Payoff becomes expected payoff.

Verification

Incentives are not enough, the agents need to believe them. Verification is an attractive way to convince them.
Mechanism Examples

Auctions

- Buyers (agents), bids (actions), seller (mechanism).
- Outcome: price, goods assignation.
- An auction is *truthful* if the buyer gets maximal payoff when she reports her true valuation.

Nash Equilibrium Computation

- \( n \) players, action type \( A \).
- Payoff for \( i \), \( P_i : A^n \rightarrow R^+ \), depends on others actions.
- The mechanism suggests an action profile \((a_1, \ldots, a_n)\).
- If all the other players follow the suggestion, player \( i \) gets the best payoff by following too.
Mechanism Examples

Auctions

- Buyers (agents), bids (actions), seller (mechanism).
- Outcome: price, goods assignation.
- An auction is *truthful* if the buyer gets maximal payoff when she reports her true valuation.

Nash Equilibrium Computation

- $n$ players, action type $A$.
- Payoff for $i$, $P_i : A^n \rightarrow R^+$, depends on others actions.
- The mechanism suggests an *action profile* $(a_1, \ldots, a_n)$.
- If all the other players follow the suggestion, player $i$ gets the best payoff by following too.
Price a good with infinite supply. (i.e: Digital goods)
Digital Goods Auctions

- Price a good with infinite supply. (i.e: Digital goods)
- Bidders and seller.

- Bidders have a secret true value for the item $v_i$, and make a public bid $b_i$ before the price is known.
- The seller knows the bids, but not the real values. Sets the price $p$ after the bids.
- If $b_i \geq p$, the bidder $i$ gets the item, with utility $v_i - p$. Otherwise she doesn't get it, and utility is 0.

The auction is truthful if buyers have optimal utility when they report the true value $v_i$ as their bids $b_i$.

In general, an auction cannot be truthful if it depends on the bidder’s price!
Digital Goods Auctions

- Price a good with infinite supply. (i.e: Digital goods)
- Bidders and seller.
- Bidders have a secret *true* value for the item $v_i$, and make a public *bid* $b_i$ before the price is known.
Digital Goods Auctions

- Price a good with infinite supply. (i.e: Digital goods)
- Bidders and seller.
- Bidders have a secret *true* value for the item $v_i$, and make a public *bid* $b_i$ before the price is known.
- The seller knows the bids, but not the real values. Sets the price $p$ after the bids.

Barthe el al. Verifying MD and DP 16
Price a good with infinite supply. (i.e: Digital goods)

Bidders and seller.

Bidders have a secret true value for the item \( v_i \), and make a public bid \( b_i \) before the price is known.

The seller knows the bids, but not the real values. Sets the price \( p \) after the bids.

If \( b_i \geq p \), the bidder \( i \) gets the item, with utility \( v_i - p \). Otherwise she doesn’t get it, and utility is 0.
Digital Goods Auctions

- Price a good with infinite supply. (i.e: Digital goods)
- Bidders and seller.
- Bidders have a secret true value for the item $v_i$, and make a public bid $b_i$ before the price is known.
- The seller knows the bids, but not the real values. Sets the price $p$ after the bids.
- If $b_i \geq p$, the bidder $i$ gets the item, with utility $v_i - p$. Otherwise she doesn’t get it, and utility is 0.

The auction is truthful if buyers have optimal utility when they report the true value $v_i$ as their bids $b_i$.

In general, an auction cannot be truthful if it depends on the bidder’s price!
Fixed Price Auctions

The simplest truthful auction is the *fixed price auction*. The seller will set $p$ independently of the bid $b$ for a seller with true value $v$. If $b \geq p$, then utility $v - p$, else 0. Note the bad revenue properties.
Fixed Price Auctions
The simplest truthful auction is the fixed price auction. The seller will set \( p \) independently of the bid \( b \) for a seller with true value \( v \). If \( b \geq p \), then utility \( v - p \), else 0. Note the bad revenue properties.

Informal proof of truthfulness
The price \( p \) is fixed, we compare \( b_{\downarrow} = v \) vs \( b_{\uparrow} \neq v \). The interesting cases are when the bidder gets the item in one run and doesn’t in the other:

- If \( b_{\uparrow} \) got the item, utility is negative, thus less than 0 for the \( b_{\downarrow} \) case (remember \( b_{\downarrow} \) didn’t get the item).
- If \( b_{\downarrow} \) got the item, utility will be greater or equal than 0, thus better or equal than \( b_{\uparrow} \)’s utility (0).
The Fixed Price Auction

We model the utility as a program:

```haskell
let fp_utility (v : R) {b :: R ▷} (p : R)
    : { u :: R ▷ ▷ } =
  if b >= p then v - p
  else 0.0
```

Barthe el al. Verifying MD and DP
The Fixed Price Auction

We model the utility as a program:

```
let fp_utility (v : R) {b :: R | b ▷= v} (p : R) :
  { u :: R | u ▷= u ⊿ } =
  if b >= p then v - p
  else 0.0
```

We use asynchronous reasoning. The interesting case is:

\[
\{ b ▷= v, b ▷= p, b ⊿ < p \} ⊢ v - p ≈ 0.0 \]

substituting \[v - p/ u ▷, 0.0/ u ⊿ \] we get the proof obligation:

\[ v ≥ p ⇒ v - p ≥ 0.0 \]

Barthe el al. Verifying MD and DP 18
We model the utility as a program:

```ml
let fp_utility (v : R) {b :: R | b ▽ = v} (p : R) :
  { u :: R | u ▽ >= u ▷ } =
  if b >= p then v - p
  else 0.0
```

We use asynchronous reasoning. The interesting case is:

\[ \{ b ▽ = v, b ▽ ≥ p, b ▷ < p \} \vdash v - p \sim 0.0 :: \{ u :: R | u ▽ ≥ u ▷ \} \]

substituting \([v - p/u▽, 0.0/u▽]\) we get the proof obligation:
The Fixed Price Auction

We model the utility as a program:

```hljs
let fp_utility (v : R) {b :: R | b ≦ = v} (p : R) :
   { u :: R | u ≦ ≧ ≥ u⊥ } =
   if b ≧ = p then v - p
   else 0.0
```

We use asynchronous reasoning. The interesting case is:

\[
\{ b ≦ = v, b ≦ ≧ ≥ p, b⊥ < p \} \vdash v - p \sim 0.0 :: \{ u :: \mathbb{R} | u ≦ ≥ u⊥ \}
\]

substituting \([v - p/uₜ, 0.0/u⊥]\) we get the proof obligation:

\[
v ≥ p \Rightarrow v - p ≥ 0.0
\]
The Distribution Type

We didn’t specify the semantics of relational distribution types.

A first approach to lifting

\[
(\mu_1, \mu_2) \in \mathcal{M}[|T|] \times \mathcal{M}[|T|] \\
(\mu_1, \mu_2) \in (\mathcal{M}[T])_\theta
\]
We didn’t specify the semantics of relational distribution types.

A first approach to lifting

\[(d_1, d_2) \in (\mathcal{T})_\theta \quad (\mu_1, \mu_2) \in \mathcal{M}[|T|] \times \mathcal{M}[|T|] \]

\[\Rightarrow \quad (\mu_1, \mu_2) \in (\mathcal{M}[|T|])_\theta\]
We didn’t specify the semantics of relational distribution types.

A first approach to lifting

\[ (d_1, d_2) \in (T)_\theta \quad (\mu_1, \mu_2) \in \mathcal{M}[|T|] \times \mathcal{M}[|T|] \]

\[ (\mu_1, \mu_2) \in (\mathcal{M}[|T|])_\theta \]
We didn’t specify the semantics of relational distribution types.

A first approach to lifting

\[ (d_1, d_2) \in \langle T \rangle_\theta \quad (\mu_1, \mu_2) \in \mathcal{M}[|T|] \times \mathcal{M}[|T|] \]

\[ (\mu_1, \mu_2) \in (\mathcal{M}[T])_\theta \]

We need to relate \((d_1, d_2)\) to \((\mu_1, \mu_2)\)!
We didn’t specify the semantics of relational distribution types.

A first approach to lifting

\[ (d_1, d_2) \in \langle T \rangle_\theta \quad (\mu_1, \mu_2) \in \mathcal{M}[|T|] \times \mathcal{M}[|T|] \]

\[ (\mu_1, \mu_2) \in \langle \mathcal{M}[T] \rangle_\theta \]

We need to relate \((d_1, d_2)\) to \((\mu_1, \mu_2)\)!

Solution: define a lifting of the relation \(\langle T \rangle_\theta\) through a witness distribution \(\mu = \mathcal{M}[|T| \times |T|]\), such that:

\[ \Pr_{x \leftarrow \mu_1} x \in \llbracket T \rrbracket = \sum_{y \in T} \Pr_{(x, y) \leftarrow \mu} (x, y) \in \langle T \rangle_\theta \]
More formally, for a relation $\Phi : T_1 \times T_2$, the predicate $L(\Phi) \mu_1 \mu_2$ holds iff there exists a distribution $\mu \in \mathcal{M}[T_1 \times T_2]$ such that for every $H \subseteq T_1$, we have

$$\Pr_{x \leftarrow \mu_1}[H(x)] = \sum_{y \in T_2} \Pr_{(x,y) \leftarrow \mu}[H(x) \land \Phi(x,y)]$$

and symmetrically for $T_2$.

“Probability of events in $\mu_1 \mu_2$ must respect the relation”.
Examples of Lifting

As an example, for $\Phi \equiv \{(F, F), (F, T), (T, T)\}$ we have liftings:

\[
\begin{align*}
\mu_1(F) &= 2/3 & \mu(F, F) &= 1/3 \\
\mu_1(T) &= 1/3 & \mu(F, T) &= 1/3 \\
\mu_2(F) &= 1/3 & \mu(T, F) &= 0 \\
\mu_2(T) &= 2/3 & \mu(T, T) &= 1/3 \\
\end{align*}
\]

\[
\begin{align*}
\mu_1(F) &= 1 & \mu(F, F) &= 1 \\
\mu_1(T) &= 0 & \mu(F, T) &= 0 \\
\mu_2(F) &= 1 & \mu(T, F) &= 0 \\
\mu_2(T) &= 0 & \mu(T, T) &= 0 \\
\end{align*}
\]
We can now interpret the relational distribution type as all the distributions satisfying the lifting:

$$\mu_1, \mu_2 \in M[|T|] \quad \mathcal{L}((|T|)_{\theta}) \mu_1 \mu_2$$

$$(\mu_1, \mu_2) \in (M[T])_{\theta}$$

In particular, the type $M[\{x :: T \mid x_\triangleleft = x_\triangleright\}]$ forces equal distributions.
Expectation

Expectation of a function $f$ over $\mu$ is:

$$E_{\mu} f := \sum_{x \in D} (f(x) \cdot (\mu x))$$
Expectation

Expectation of a function $f$ over $\mu$ is:

$$E_{\mu} f := \sum_{x \in D} (f x) \cdot (\mu x)$$

We capture monotonicity of expectation as:

$$I := [0, 1]$$

$$IBF := \{ f :: D \rightarrow I \mid \forall d : D. f_\triangleleft d \geq f_\triangleright d \}$$

$$E : \Pi(\mu :: \mathcal{M}[\{ x :: D \mid x_\triangleleft = x_\triangleright \}]). \Pi(f :: IBF). \{ e :: I \mid e_\triangleleft \geq e_\triangleright \}$$

Sound as a primitive; other types are possible.
Randomized Auctions

- Using the probabilistic primitives, we can now define and verify randomized auctions, which have much better revenue properties than the fixed price one.
- The price a bidder gets won’t still depend on her bid, however:
  - we *randomly* split the bidders in two groups, $g_a, g_b$, we compute the revenue-maximizing price for each group, $p_a, p_b$, and sell to $g_a$ using $p_b$ and conversely.
- This auction is truthful on the *expected* utility.

**Universal truthfulness:**
A bidder will be never able to gain from lying, even knowing the random coins of the mechanism.
The Competitive Auction

```ocaml
let utility (v : real) (bid :: { b :: R | b < v }) (otherbids : L[R]) (g, groups) : (B * L[B]) : { u :: real | u < u ⊳ } =
match split g bid others otherbids with
| (g1, g2) →
  if g then fixedprice v bid (prices g2)
  else fixedprice v bid (prices g1)

let auction (n : N) (v : R) (bid :: { b :: R | b < v }) (otherbids : L[R]) : { u :: real | u < u ⊳ } =
let grouping :: M{ r :: (B * B list) | r < r ⊳ } =
  mlet mycoin = flip in
  mlet coins = flipN n in
  munit (mycoin, coins)
in E grouping (utility v bid otherbids)
```
The Competitive Auction

\[ \text{let } E (\mu : M[ r : \alpha \mid r_\triangleleft = r_\triangleright ]) \]
\[ (f : \alpha \to \text{real} \mid \forall x : \alpha, f_\triangleleft x \geq f_\triangleright x) \]
\[ : \{ r :: \text{real} \mid r_\triangleleft \geq r_\triangleright \} = ... \]

\[ \text{let } \text{utility} (v : \text{real}) \]
\[ (\text{bid} :: \{ b :: \text{R} \mid b_\triangleleft = v \}) \]
\[ (\text{otherbids} : \text{L}[\text{R}]) \]
\[ (g, \text{groups}) : (\text{B} \times \text{L}[\text{B}]) \]
\[ : \{ u :: \text{real} \mid u_\triangleleft \geq u_\triangleright \} = ... \]

\[ \text{let } \text{auction} (n : \text{N}) (v : \text{R}) \]
\[ (\text{bid} :: \{ b :: \text{R} \mid b_\triangleleft = v \}) \]
\[ (\text{otherbids} : \text{L}[\text{R}]) \]
\[ : \{ u :: \text{real} \mid u_\triangleleft \geq u_\triangleright \} = \]
\[ \text{let } \text{grouping} :: M\{ r :: (\text{B} \times \text{B} \text{list}) \mid r_\triangleleft = r_\triangleright \} = ... \]
\[ \text{in } E \text{ grouping (utility v bid otherbids)} \]
Differential Privacy

Contribution of a single individual to the output of a mechanism cannot be effectively distinguished by an attacker under worst-case assumptions.
Formal Definition
A probabilistic function $F : T \rightarrow S$ is $(\epsilon, \delta)$-Differentially Private if for all pairs of adjacent $t_1, t_2 \in T$ and for every $E \subseteq S$:

$$\Pr_{x \leftarrow F t_1} [x \in E] \leq \exp(\epsilon) \Pr_{x \leftarrow F t_2} [x \in E] + \delta$$

Example: The Laplace Mechanism:
▶ Compute the sensitivity $k$ of $f$.
▶ For input $t$, release $f(t) + \text{random noise}$, scaled by $k$. 

Many algorithms are DP: private database release, counters, analytics, strong connection to Mechanism Design!
Differential Privacy

Formal Definition
A probabilistic function $F : T \rightarrow S$ is $(\epsilon, \delta)$-Differentially Private if for all pairs of adjacent $t_1, t_2 \in T$ and for every $E \subseteq S$:

$$\Pr_{x \leftarrow F t_1} [x \in E] \leq \exp(\epsilon) \Pr_{x \leftarrow F t_2} [x \in E] + \delta$$

Example: The Laplace Mechanism:
- Compute the sensitivity $k$ of $f$.
- For input $t$, release $f(t) + \text{random noise}$, scaled by $k$. 
Differential Privacy

**Formal Definition**

A probabilistic function $F : T \rightarrow S$ is $(\epsilon, \delta)$-**Differentially Private** if for all pairs of adjacent $t_1, t_2 \in T$ and for every $E \subseteq S$:

$$\Pr_{x \leftarrow F t_1} [x \in E] \leq \exp(\epsilon) \Pr_{x \leftarrow F t_2} [x \in E] + \delta$$

**Example: The Laplace Mechanism:**

- Compute the *sensitivity* $k$ of $f$.
- For input $t$, release $f(t) + random\ noise$, scaled by $k$.

Many algorithms are DP: private database release, counters, analytics, **strong connection to Mechanism Design**!
We can capture DP with a refinement over the type of probability distributions using the definition of $\Delta$-distance:

$$\Delta_\epsilon(\mu_1, \mu_2) = \max_{E \subseteq U} \left( \Pr_{x \leftarrow \mu_2} [x \in E] - \exp(\epsilon) \Pr_{x \leftarrow \mu_1} [x \in E] \right)$$
We can capture DP with a refinement over the type of probability distributions using the definition of $\Delta$-distance:

$$\Delta_\epsilon(\mu_1, \mu_2) = \max_{E \subseteq U} \left( \Pr_{x \leftarrow \mu_2} [x \in E] - \exp(\epsilon) \Pr_{x \leftarrow \mu_1} [x \in E] \right)$$

Then, $f$ is $(\epsilon, \delta)$ differentially private if it has type:

$$\{ d :: T \mid \text{Adj}(d_\triangleleft, d_\triangleright) \} \rightarrow \{ r :: \mathcal{M}[\mathbb{R}] \mid \Delta_\epsilon(r_\triangleleft, r_\triangleright) \leq \delta \}$$

However, verification conditions involving $\Delta$ are quite hard.
Our solution: Internalize distribution distance in the types:

\[
\frac{\mu_1, \mu_2 \in \mathcal{M}[|T|] \quad \mathcal{L}_{\epsilon,\delta}(|T|\theta) \mu_1 \mu_2}{(\mu_1, \mu_2) \in (\mathcal{M}_{\epsilon,\delta}[T])\theta}
\]

Lifting is extended from \( p = p_1 \) to \( p \leq p_1 \leq \exp(p) + \delta \).
The Relational Distribution Type

Our solution: Internalize distribution distance in the types:

\[
\mu_1, \mu_2 \in \mathcal{M}[|T|] \quad \mathcal{L}_{\epsilon, \delta}(|T|) \mu_1 \mu_2 \\
(\mu_1, \mu_2) \in (\mathcal{M}_{\epsilon, \delta}[T])_\theta
\]

Lifting is extended from \( p = p_1 \) to \( p \leq p_1 \leq \exp(p) + \delta \).

Capturing DP

The interpretation of \( \mathcal{M}_{\epsilon, \delta}[\{r :: \mathbb{R} \mid r_{\downarrow} = r_{\uparrow}\}] \) is the set of pairs of probability distributions that are \((\epsilon, \delta)\)-apart, capturing DP.
Our solution: Internalize distribution distance in the types:

\[
\begin{aligned}
\mu_1, \mu_2 &\in \mathcal{M}[|T|] \\
L_{\epsilon, \delta}(|T|_\theta)(\mu_1, \mu_2) &\quad (\mu_1, \mu_2) \in (\mathcal{M}_{\epsilon, \delta}[T])_\theta
\end{aligned}
\]

Lifting is extended from \( p = p_1 \) to \( p \leq p_1 \leq \exp(p) + \delta \).

Capturing DP

The interpretation of \( \mathcal{M}_{\epsilon, \delta}[\{r :: \mathbb{R} \mid r_\downarrow = r_\uparrow\}] \) is the set of pairs of probability distributions that are \((\epsilon, \delta)\)-apart, capturing DP.

DP algorithms are typed as:

\[
f : \{d :: T \mid \text{Adj}(d_\downarrow, d_\uparrow)\} \rightarrow \mathcal{M}_{\epsilon, \delta}[\{r :: \mathbb{R} \mid r_\downarrow = r_\uparrow\}]
\]
The Probability Polymonad

Reasoning about distance is compositional:

**SUB-M**
\[
\frac{G \vdash T \leq U \quad \forall \theta. \theta \vdash G, x :: T \Rightarrow [\epsilon_1 \leq \epsilon_2 \land \delta_1 \leq \delta_2]_\theta}{G \vdash M_{\epsilon_1,\delta_1}[T] \leq M_{\epsilon_2,\delta_2}[U]}
\]

**UNITM**
\[
\frac{G \vdash e :: T}{G \vdash \text{unit}_M e :: M_{\epsilon,\delta}[T]}
\]

**BINDM**
\[
\frac{G \vdash e_1 :: M_{\epsilon_1,\delta_1}[T_1] \quad G, x :: T_1 \vdash e_2 :: M_{\epsilon_2,\delta_2}[T_2]}{G \vdash \text{bind}_M x = e_1 \text{ in } e_2 :: M_{\epsilon_1+\epsilon_2,\delta_1+\delta_2}[T_2]}
\]

Bind is distance-adjusting sampling.
Recall the Laplace Mechanism:

For a $k$-sensitive $f$, $f$ plus $k/\epsilon$-scaled Laplacian noise is DP. This is captured by the type:
Recall the Laplace Mechanism:

For a $k$-sensitive $f$, $f$ plus $k/\epsilon$-scaled Laplacian noise is DP. This is captured by the type:

$$\text{lap} : \Pi(\epsilon :: \mathbb{R}). \Pi(x :: \mathbb{R}). \mathcal{M}_{\epsilon \ast |x_\downarrow - x_\uparrow|,0}[\{r :: \mathbb{R} \mid r_\downarrow = r_\uparrow\}]$$

Note that the actual distance $\epsilon \ast |x_\downarrow - x_\uparrow|$ depends on the distance of the inputs. This is a better alternative than using a precondition on $x$. 

Barthe et al. Verifying MD and DP 32
Recall the Laplace Mechanism:
For a \( k \)-sensitive \( f \), \( f \) plus \( k/\epsilon \)-scaled Laplacian noise is DP. This is captured by the type:

\[
\text{lap} : \Pi(\epsilon :: \mathbb{R}). \Pi(x :: \mathbb{R}). \mathcal{M}_{\epsilon^* |x_\downarrow - x_\uparrow|,0}[\{r :: \mathbb{R} \mid r_\downarrow = r_\uparrow\}]
\]

Note that the actual distance \( \epsilon^* |x_\downarrow - x_\uparrow| \) depends on the distance of the inputs. This is a better alternative than using a precondition on \( x \).
Using the bind rule, we can sample from \text{laplace} and assume the sampled value equal in both runs.
Example: Private Histogram

We add noise to an histogram to make it private.

```ocaml
let rec histogram {l :: L(R) | Adj x_< x_> } : M[e * d(l_<,l_>)] { r :: L(R) | r_< = r_> } =
match l with
| [] → unit []
| x :: xs →
  mlet y = lap eps x in
  mlet ys = histogram xs in
  munit (y :: ys)
```

The main proof obligation is:

\[
e \ast d(x_{<::xs}_{<}, x_{>::xs}_{<}) \geq e \ast (d(x_{<::xs}_{<}, x_{>::xs}_{<}) + d(xs_{<::xs}_{<}, xs_{>::xs}_{<}))
\]

which is implied by the adjacency precondition.
Example: Private Histogram

We add noise to an histogram to make it private.

\[
\text{let rec histogram } \{ l :: L(R) | \text{Adj } x_{\Leftarrow} x_{\Rightarrow} \} \}\quad:\text{M}\{ e \ast d(l_{\Leftarrow},l_{\Rightarrow}) \}\{ r :: L(R) | r_{\Leftarrow} = r_{\Rightarrow} \} = \text{match } l \text{ with} \\
| [] \rightarrow \text{unit } [] \\
| x :: xs \rightarrow \\
\text{mlet } y = \text{lap } \text{eps } x \text{ in} \\
\text{mlet } ys = \text{histogram } xs \text{ in} \\
\text{munit } (y :: ys)
\]

The main proof obligation is:

\[
e \ast d(x_{\Leftarrow} :: xs_{\Leftarrow}, x_{\Rightarrow} :: xs_{\Rightarrow}) \geq e \ast (d(x_{\Leftarrow}, x_{\Rightarrow}) + d(xs_{\Leftarrow}, xs_{\Rightarrow}))
\]

which is implied by the adjacency precondition.
Combining MD and DP: Aggregative Games

- We verify the computation of an approximate Nash-equilibrium.
- $n$ agents can choose over a space of actions $a_i \in A$. 

$\left( a_1, \ldots, a_n \right)$ is an $\alpha$-approximate Nash-equilibrium if no single agent $i$ can gain more than $\alpha$ payoff by unilateral deviation: For all agents $i$ and actions $a'_i$:

$$E[P_i(a_1, \ldots, a_i, \ldots, a_N)] \geq E[P_i(a_1, \ldots, a'_i, \ldots, a_N)] - \alpha.$$ 

Assumption: Payoff for $i$ depends only on $a_i$ plus a signal, a positive (bounded) real number depending on the aggregated actions of all players.
We verify the computation of an approximate Nash-equilibrium.

$n$ agents can choose over a space of actions $a_i \in A$.

$(a_1, \ldots, a_n)$ is an $\alpha$-approximate Nash-equilibrium if no single agent $i$ can gain more than $\alpha$ payoff by unilateral deviation: For all agents $i$ and actions $a_i'$:

$$E[P_i(a_1, \ldots, a_i, \ldots a_N)] \geq E[P_i(a_1, \ldots, a'_i, \ldots a_N)] - \alpha.$$
We verify the computation of an approximate Nash-equilibrium.

\( n \) agents can choose over a space of actions \( a_i \in A \).

\((a_1, \ldots, a_n)\) is an \( \alpha \)-approximate Nash-equilibrium if no single agent \( i \) can gain more than \( \alpha \) payoff by unilateral deviation: For all agents \( i \) and actions \( a'_i \):

\[
E[P_i(a_1, \ldots, a_i, \ldots a_N)] \geq E[P_i(a_1, \ldots, a'_i, \ldots a_N)] - \alpha.
\]

Assumption: Payoff for \( i \) depends only on \( a_i \) plus a signal, a positive (bounded) real number depending on the aggregated actions of all players.
The key: use differential privacy to compute the equilibria.

Mediator: The mechanism suggests the equilibria action $a_i$.

We prove that the player gets optimal utility if she does $a_i$.

We reason over a deviation function $dev_i$ for player $i$. 
The key: use differential privacy to compute the equilibria.

Mediator: The mechanism suggests the equilibria action $a_i$.

We prove that the player gets optimal utility if she does $a_i$.

We reason over a deviation function $dev_i$ for player $i$.

In types:

```haskell
let aggregative_utility ( ... )
  { dev :: act → act | ∀ a : act. dev a = a ) }
  : { u :: real | u ⊳ > u ⊲ - alpha }
```
The key: use differential privacy to compute the equilibria.

Mediator: The mechanism suggests the equilibria action $a_i$.

We prove that the player gets optimal utility if she does $a_i$.

We reason over a deviation function $dev_i$ for player $i$.

In types:

```markdown
let aggregative_utility ( ... )

\{ dev :: act → act | \forall a : act. dev_\triangleleft a = a \} 

: \{ u :: real | u_\triangleleft >= u_\triangleright - alpha \}
```

Relate expectation to distance on the distributions:

$$E : \Pi(\mu :: M_{\epsilon, \delta}[\{ x :: l | x_\triangleleft <= x_\triangleright + c \}]). \{ e :: l | e_\triangleleft <= e_\triangleright + \epsilon + c + \delta e^{-\epsilon} \}$$
The Implementation

- Hybrid SMT/Bidirectional type checking.
- Why3 as the SMT backend, multiple solvers required.
- Verification using top-level annotations (+2 cuts).
- Top-level types act as the specification.
- Support for debug of type-checking failures important.
<table>
<thead>
<tr>
<th>Example</th>
<th># Lines</th>
<th>Verif. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram</td>
<td>25</td>
<td>2.66 s.</td>
</tr>
<tr>
<td>dummysum</td>
<td>31</td>
<td>11.95 s.</td>
</tr>
<tr>
<td>noisysum</td>
<td>55</td>
<td>3.64 s.</td>
</tr>
<tr>
<td>two-level-a</td>
<td>38</td>
<td>2.55 s.</td>
</tr>
<tr>
<td>two-level-b</td>
<td>56</td>
<td>3.94 s.</td>
</tr>
<tr>
<td>binary</td>
<td>95</td>
<td>18.56 s.</td>
</tr>
<tr>
<td>idc</td>
<td>73</td>
<td>27.60 s.</td>
</tr>
<tr>
<td>dualquery</td>
<td>128</td>
<td>27.71 s.</td>
</tr>
<tr>
<td>competitive-b</td>
<td>81</td>
<td>2.80 s.</td>
</tr>
<tr>
<td>competitive</td>
<td>75</td>
<td>4.19 s.</td>
</tr>
<tr>
<td>fixedprice</td>
<td>10</td>
<td>0.90 s.</td>
</tr>
<tr>
<td>summarization</td>
<td>471</td>
<td>238.42 s.</td>
</tr>
</tbody>
</table>

Table: Benchmarks
Future Work:

- More examples from the algorithms community.
- More examples from the security/cryptography domain.
- More properties: accuracy, fancier distributions.
- Extensions to the language.

Conclusions:

- Higher-Order Approximate Probabilistic Relational Refinement Types: HOAR2
- Built-in support for approximate reasoning.
- Logic seems to capture many examples.
- Automatic verification worked reasonably well.
- SMT interaction is still a challenge.
Future work and Conclusions:

Future Work:

▶ More examples from the algorithms community.
▶ More examples from the security/cryptography domain.
▶ More properties: accuracy, fancier distributions.
▶ Extensions to the language.

Conclusions

▶ Higher-Order Approximate Probabilistic Relational Refinement Types: HOARe2
▶ Built-in support for approximate reasoning.
▶ Logic seems to capture many examples.
▶ Automatic verification worked reasonably well.
▶ SMT interaction is still a challenge.
Questions?
More on the Aggregative Example:

Expected Payoff for the deviating agent

```plaintext
let expay br* dev* br dev =
    E (mlet sums = mkSums k br* br in
         let s* = search k br* br sums in
         let a* = dev* (br* s*) in
         let a = dev (br s*) in
         let p* = pay* a* (sign a* a) in
         munit p*) (λx. x)

s* is close to the true signal on the strategy profile.
```
More on the Aggregative Example:

Type for `expay`

\[
\{ \text{br}^* :: \mathbb{R} \rightarrow A \mid \forall s, a. \text{pay}^*(\text{br}^* \triangleleft s) s \geq \text{pay}^* a s \} \\
\rightarrow \{ \text{dev}^* :: A \rightarrow A \mid \forall x. \text{dev}^* \triangleleft x = x \} \\
\rightarrow \{ \text{br} :: \mathbb{R} \rightarrow A \mid \text{br} \triangleleft = \text{br} \triangleright \} \\
\rightarrow \{ \text{dev} :: A \rightarrow A \mid \forall a. \text{dev} \triangleleft a = \text{dev} \triangleright a = a \} \\
\rightarrow \{ \text{u} :: \mathbb{R}^+ \mid \text{u} \triangleleft \geq \text{u} \triangleright - \alpha \}.
\]

Extended type for Laplace `lap` with a refinement type capturing `accuracy`:

\[
\Pi(x :: \mathbb{R}). \mathcal{M}^{(\epsilon \mid x \triangleleft - x \triangleright \mid, \beta)}[\{ u :: \mathbb{R} \mid u \triangleleft = u \triangleright \land |x \triangleleft - u \triangleleft| < T \}]
\]