Really Naturally Linear Indexed Type Checking

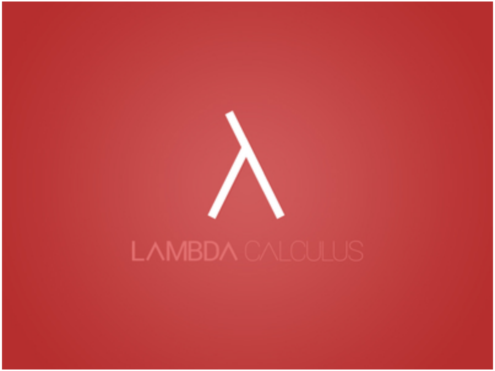
Arthur Azevedo de Amorim[1], Marco Gaboardi[2],
Emilio Jesús Gallego Arias[1], Justin Hsu[1]

[1]University of Pennsylvania
[2]University of Dundee

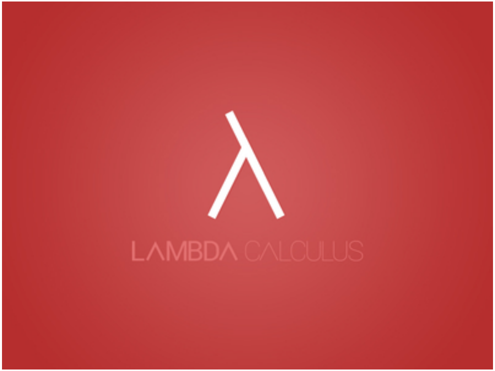
October 2, 2014

In the beginning...

In the beginning...



In the beginning...



Check properties via types

- Type safety
- Parametricity
- Non-interference

More recently

Properties model quantitative information

- Numerical robustness
- Probabilistic assertions
- Differential privacy

Properties model quantitative information

- Numerical robustness — how robust?
- Probabilistic assertions
- Differential privacy

Properties model quantitative information

- Numerical robustness — how robust?
- Probabilistic assertions — how likely?
- Differential privacy

More recently

Properties model quantitative information

- Numerical robustness — how robust?
- Probabilistic assertions — how likely?
- Differential privacy — how private?

More recently

Properties model quantitative information

- Numerical robustness
- Probabilistic assertions
- Differential privacy

how robust?

how likely?

how private?

Properties not just true or false

But what about typechecking?

Typechecking quantitative languages is tricky

- May need to solve numeric constraints
- Typechecking may not be decidable
- May need heuristics to make typechecking practical

But what about typechecking?

Typechecking quantitative languages is tricky

- May need to solve numeric constraints
- Typechecking may not be decidable
- May need heuristics to make typechecking practical

Our goal

- Design and implement a typechecking algorithm for DFuzz, a language for verifying differential privacy

The plan today

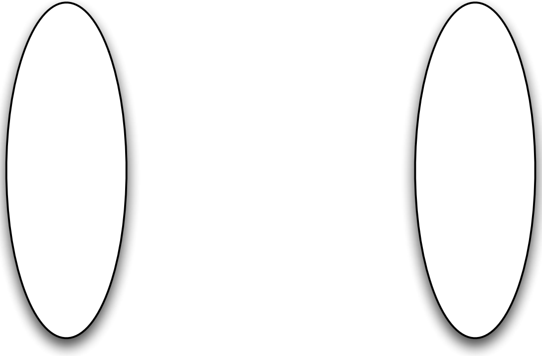
- A DFuzz crash course
- The problem with standard approaches
- Modifying the DFuzz language to ease typechecking
- Decidability and heuristics

The quantitative property

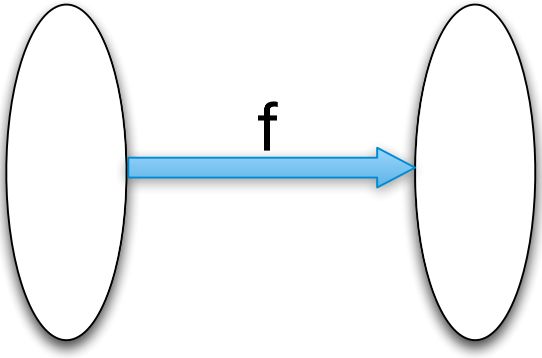
Differential privacy [DMNS06]

- Rigorous definition of privacy for randomized programs
- Idea: random noise should "conceal" an individual's data
- Quantitative: measure how private a program is
- Close connection to sensitivity analysis

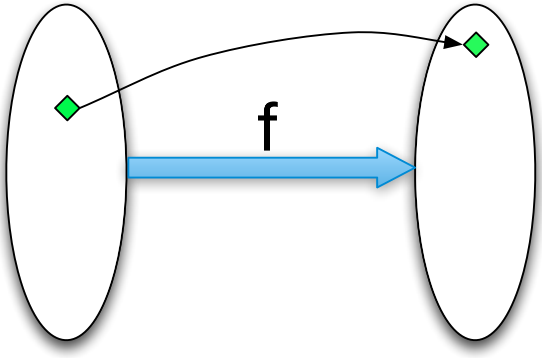
R-sensitive function



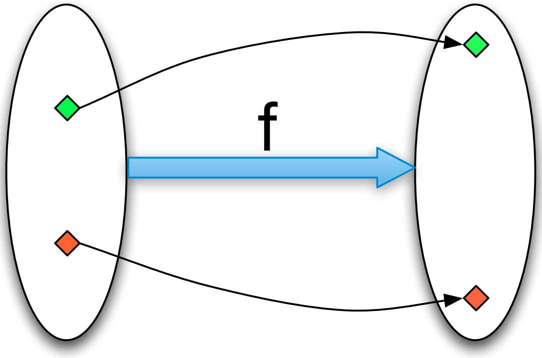
R-sensitive function



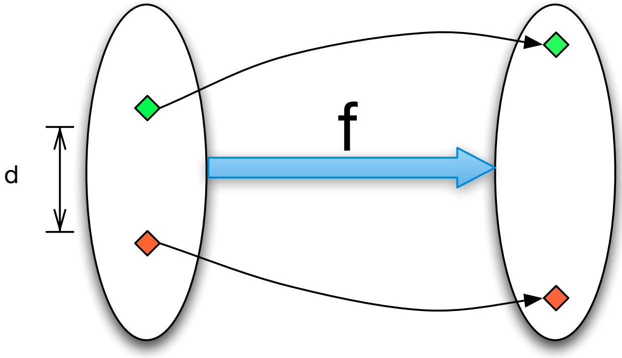
R-sensitive function



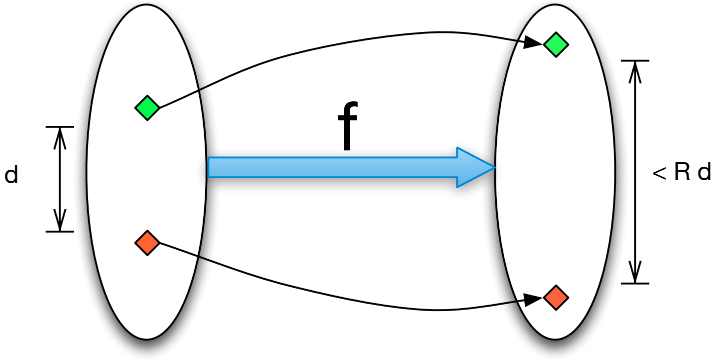
R-sensitive function



R-sensitive function



R-sensitive function



A language for differential privacy

DFuzz [GHHNP13]

- Type system for differentially private programs
- Use linear logic to model sensitivity
- Combine with (lightweight) dependent types

In a little more detail...

Types

$$\tau ::= \mathbb{N}\,[R] \;\mid \tau \oplus \tau \mid \tau \otimes \tau \mid\, !_R\, \tau \multimap \tau \mid \forall i.\, \tau$$

In a little more detail...

Types

$$\tau ::= \mathbb{N}\ [R] \mid \tau \oplus \tau \mid \tau \otimes \tau \mid !_R\ \tau \multimap \tau \mid \forall i.\ \tau$$

Contexts

$$\Gamma ::= \cdot \mid \Gamma, x :_{[R]} \tau$$

Types

$$\tau ::= \mathbb{N}\ [R] \ \mid \tau \oplus \tau \mid \tau \otimes \tau \mid !_{R}\ \tau \multimap \tau \mid \forall i.\, \tau$$

Contexts

$$\Gamma ::= \cdot \mid \Gamma, x :_{[R]} \tau$$

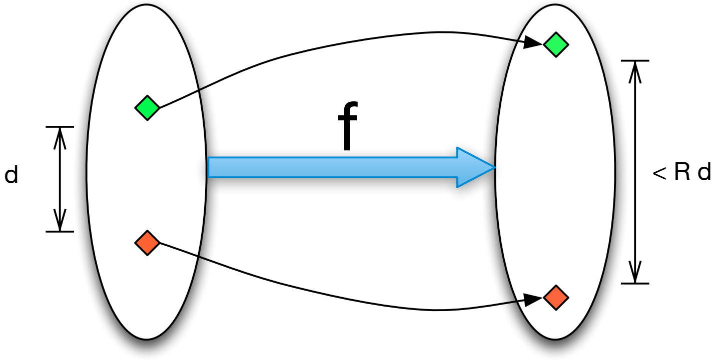
Typing judgment

$$\Gamma \vdash e : \tau$$

Sensitivity reading

- Functions $!_R \tau_1 \multimap \tau_2$: R-sensitive functions
- Changing input by d changes output by at most $R \cdot d$

R-sensitive function



Reading the types

Sensitivity reading

- Functions $!_R \tau_1 \multimap \tau_2$: R-sensitive functions
- Changing input by d changes output by at most $R \cdot d$

Sensitivity reading

- Functions $!_R \tau_1 \multimap \tau_2$: R-sensitive functions
- Changing input by d changes output by at most $R \cdot d$

Subtyping

- "A 1-sensitive function is also a 2-sensitive function"
- Subtyping: weaken sensitivity bound

$$!_R \tau \multimap \tau_2 \sqsubseteq !_{R'} \tau_1 \multimap \tau_2 \qquad \text{if} \qquad R \leq R'$$

In a little more detail...

Types

$$\tau ::= \mathbb{N}\,[R] \mid \tau \oplus \tau \mid \tau \otimes \tau \mid !_{R}\, \tau \multimap \tau \mid \forall i.\, \tau$$

Contexts

$$\Gamma ::= \cdot \mid \Gamma, x :_{[R]} \tau$$

Typing judgment

$$\Gamma \vdash e : \tau$$

In a little more detail...

Types

$$\tau ::= \mathbb{N}\,\boxed{[R]} \mid \tau \oplus \tau \mid \tau \otimes \tau \mid !_{\boxed{R}}\, \tau \multimap \tau \mid \forall i.\, \tau$$

Contexts

$$\Gamma ::= \cdot \mid \Gamma, x :_{\boxed{[R]}} \tau$$

Typing judgment

$$\Gamma \vdash e : \tau$$

Grammar

$$R ::= i_{\mathbb{R}} \mid i_{\mathbb{N}} \mid \mathbb{R} \mid R + R \mid R \cdot R$$

Grammar

$$R ::= \boxed{i_{\mathbb{R}} \mid i_{\mathbb{N}}} \mid \mathbb{R} \mid R + R \mid R \cdot R$$

variables over
real/naturals

Grammar

$$R ::= \boxed{i_{\mathbb{R}} \mid i_{\mathbb{N}}} \mid \mathbb{R} \mid R + R \mid R \cdot R$$

variables over
real/naturals

Sensitivity not known statically

- DFuzz is dependent!
- Sensitivity may depend on inputs (length of list, number of iterations, etc.)

Types

$$\tau ::= \mathbb{N}\,[R] \mid \tau \oplus \tau \mid \tau \otimes \tau \mid !_{R}\,\tau \multimap \tau \mid \forall i.\, \tau$$

Contexts

$$\Gamma ::= \cdot \mid \Gamma, x :_{[R]} \tau$$

Typing judgment

$$\Gamma \vdash e : \tau$$

Grammar

$$R ::= \boxed{i_{\mathbb{R}} \mid i_{\mathbb{N}}} \mid \mathbb{R} \mid R + R \mid R \cdot R$$

variables over
real/naturals

Sensitivity not known statically

- DFuzz is dependent!
- Sensitivity may depend on inputs (length of list, number of iterations, etc.)

Grammar

$$R ::= \boxed{i_{\mathbb{R}} \mid i_{\mathbb{N}}} \mid \mathbb{R} \mid R + R \mid R \cdot R$$

variables over real/naturals

Sensitivity not known statically

- DFuzz is dependent!
- Sensitivity may depend on inputs (length of list, number of iterations, etc.)

What does this mean for typechecking?

- Sensitivities are polynomials over reals and naturals
- How to check subtyping?

Reading the types

Sensitivity reading

- Functions $!_R \tau_1 \multimap \tau_2$: R-sensitive functions
- Changing input by d changes output by at most $R \cdot d$

Subtyping

- "A 1-sensitive function is also a 2-sensitive function"
- Subtyping: weaken sensitivity bound

$$!_R \tau \multimap \tau_2 \sqsubseteq \, !_{R'} \tau_1 \multimap \tau_2 \qquad \text{if} \qquad R \leq R'$$

Sensitivity reading

- Functions $!_R \tau_1 \multimap \tau_2$: R-sensitive functions
- Changing input by d changes output by at most $R \cdot d$

Subtyping

- "A 1-sensitive function is also a 2-sensitive function"
- Subtyping: weaken sensitivity bound

$$!_R \tau \multimap \tau_2 \sqsubseteq \, !_{R'} \tau_1 \multimap \tau_2 \qquad \text{if} \qquad \boxed{R \leq R'}$$

compare polynomials

The typechecking problem

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

type without sensitivities

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

The typechecking problem

Assume

type without sensitivities

- Can extract type skeleton from term
- Given annotated term, compute best type

The typechecking problem

Assume

type without sensitivities

- Can extract type skeleton from term
- Given annotated term, compute best type

w.r.t. subtyping

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

Annotations

- We need: fully annotated argument types of all functions

$$!_{??}\ \tau_1 \multimap \tau_2$$

The typechecking problem

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

Annotations

- We need: fully annotated argument types of all functions

$$!_{??}\ \underset{\text{annot.}}{\boxed{\tau_1}} \multimap \tau_2$$

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

Annotations

- We need: fully annotated argument types of all functions

$$!^{??}\ \tau_1 \multimap \tau_2$$

no annot.

annot.

The typechecking problem

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

Annotations

- We need: fully annotated argument types of all functions

$$! \; ?? \;\; \tau_1 \multimap \boxed{\tau_2}$$

no annot.

annot.

no annot.

The typechecking problem

Assume

- Can extract type skeleton from term
- Given annotated term, compute best type

Annotations

- We need: fully annotated argument types of all functions

$$!_{??}\ \tau_1 \multimap \tau_2$$

no annot. annot. no annot.

- Other more minor annotations

Input

- Annotated term e
- Annotated context skeleton $\Gamma^\bullet$:

$$x :_{??} \tau$$

Input

- Annotated term e
- Annotated context skeleton $\Gamma^\bullet$:

$$x : \boxed{??} \quad \tau$$

no annot.

Input

- Annotated term e
- Annotated context skeleton $\Gamma^\bullet$:

$$x : \underbrace{??}_{\text{no annot.}} \quad \boxed{\tau}$$

annot.

The typechecking problem

Input

- Annotated term e
- Annotated context skeleton $\Gamma^\bullet$:

$$x : \; ?? \quad \tau$$

annot.

no annot.

Output

- Type τ^* and context Γ with $\Gamma \vdash e : \tau^*$
- Most precise context and type (with respect to subtyping)

"Bottom-up" typechecking

- For each premise, compute best context and type
- Combine outputs from premises to get context and type

"Bottom-up" typechecking

- For each premise, compute best context and type
- Combine outputs from premises to get context and type

Example: function application

$$\frac{\Gamma \vdash e_1 : !_R\sigma \multimap \tau \qquad \Delta \vdash e_2 : \sigma}{\Gamma + R \cdot \Delta \vdash e_1\ e_2 : \tau}$$

1. Given $(e_1\ e_2, \Gamma^\bullet)$

"Bottom-up" typechecking

- For each premise, compute best context and type
- Combine outputs from premises to get context and type

Example: function application

$$\frac{\Gamma \vdash \boxed{e_1} : !_R \sigma \multimap \tau \qquad \Delta \vdash e_2 : \sigma}{\Gamma + R \cdot \Delta \vdash e_1 \; e_2 : \tau}$$

1. Given $(e_1 \; e_2, \Gamma^\bullet)$
2. Call typechecker on $(e_1, \Gamma^\bullet)$, get $(!_R \sigma \multimap \tau, \Gamma)$

"Bottom-up" typechecking

- For each premise, compute best context and type
- Combine outputs from premises to get context and type

Example: function application

$$\frac{\Gamma \vdash e_1 : !_R\sigma \multimap \tau \qquad \Delta \vdash \boxed{e_2} : \sigma}{\Gamma + R \cdot \Delta \vdash e_1\ e_2 : \tau}$$

1. Given $(e_1\ e_2, \Gamma^\bullet)$
2. Call typechecker on $(e_1, \Gamma^\bullet)$, get $(!_R\sigma \multimap \tau, \Gamma)$
3. Call typechecker on $(e_2, \Delta^\bullet)$, get (σ', Δ)

"Bottom-up" typechecking

- For each premise, compute best context and type
- Combine outputs from premises to get context and type

Example: function application

$$\frac{\Gamma \vdash e_1 : !_R \sigma \multimap \tau \qquad \Delta \vdash e_2 : \sigma}{\boxed{\Gamma + R \cdot \Delta} \vdash e_1\ e_2 : \boxed{\tau}}$$

1. Given $(e_1\ e_2, \Gamma^\bullet)$
2. Call typechecker on $(e_1, \Gamma^\bullet)$, get $(!_R \sigma \multimap \tau, \Gamma)$
3. Call typechecker on $(e_2, \Delta^\bullet)$, get (σ', Δ)
4. Check $\sigma' \sqsubseteq \sigma$, output $(\tau, \Gamma + R \cdot \Delta)$

A problem with the bottom-up approach

- Some DFuzz rules have form

$$\frac{\Gamma \vdash \; e_1 \; : \sigma_1 \qquad \Gamma \vdash \; e_2 \; : \sigma_2}{\Gamma \vdash \cdots : \cdots}$$

A problem with the bottom-up approach

- Some DFuzz rules have form

$$\frac{\Gamma \vdash \boxed{e_1} : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash \cdots : \cdots}$$

A problem with the bottom-up approach

- Some DFuzz rules have form

$$\frac{\Gamma \vdash \boxed{e_1} : \sigma_1 \qquad \Gamma \vdash \boxed{e_2} : \sigma_2}{\Gamma \vdash \cdots : \cdots}$$

A problem with the bottom-up approach

- Some DFuzz rules have form

$$\frac{\Gamma \vdash \boxed{e_1} : \sigma_1 \qquad \Gamma \vdash \boxed{e_2} : \sigma_2}{\Gamma \vdash \cdots : \cdots}$$

- Running algorithm gives (σ_1, Γ_1) and (σ_2, Γ_2)

A problem with the bottom-up approach

- Some DFuzz rules have form

$$\frac{\Gamma \vdash \boxed{e_1} : \sigma_1 \qquad \Gamma \vdash \boxed{e_2} : \sigma_2}{\Gamma \vdash \cdots : \cdots}$$

- Running algorithm gives (σ_1, Γ_1) and (σ_2, Γ_2)
- But what context do we output?

“Minimal” context?

First try

- Have $x :_{[R_1]} \sigma$ and $x :_{[R_2]} \sigma$
- Most precise context should be $x :_{[\mathbf{max}(R_1, R_2)]} \sigma$
- But DFuzz doesn’t have $\mathbf{max}(R_1, R_2)$...

Grammar

$$R ::= \; i_{\mathbb{R}} \mid i_{\mathbb{N}} \; \mid \mathbb{R} \mid R + R \mid R \cdot R$$

“Minimal” context?

First try

- Have $x :_{[R_1]} \sigma$ and $x :_{[R_2]} \sigma$
- Most precise context should be $x :_{[\mathbf{max}(R_1, R_2)]} \sigma$
- But DFuzz doesn’t have $\mathbf{max}(R_1, R_2)$...

“Minimal” context?

First try

- Have $x :_{[R_1]} \sigma$ and $x :_{[R_2]} \sigma$
- Most precise context should be $x :_{[\mathbf{max}(R_1, R_2)]} \sigma$
- But DFuzz doesn’t have $\mathbf{max}(R_1, R_2)$...

$\boxed{\text{Max of two polynomials may not be polynomial!}}$

The idea: enrich DFuzz

EDFuzz: E(xtended) DFuzz

- Sensitivity language in DFuzz is "incomplete" for typechecking
- Add constructions like $\mathbf{max}(R_1, R_2)$ to sensitivity language
- Typecheck EDFuzz programs instead

The idea: enrich DFuzz

EDFuzz: E(xtended) DFuzz

- Sensitivity language in DFuzz is “incomplete” for typechecking
- Add constructions like $\mathbf{max}(R_1, R_2)$ to sensitivity language
- Typecheck EDFuzz programs instead

Relation with DFuzz

- Extension: all DFuzz programs still valid EDFuzz programs
- Preserve metatheory
- Bottom-up typechecking simple, works

How does this fix the problem?

Previously problematic rule

$$\frac{\Gamma \vdash \ e_1 \ : \sigma_1 \qquad \Gamma \vdash \ e_2 \ : \sigma_2}{\Gamma \ \vdash \cdots : \cdots}$$

How does this fix the problem?

Previously problematic rule

$$\frac{\Gamma \vdash \ e_1 \ : \sigma_1 \qquad \Gamma \vdash \ e_2 \ : \sigma_2}{\Gamma \ \vdash \cdots : \cdots}$$

Now: no problem

- Running algorithm gives (σ_1, Γ_1) and (σ_2, Γ_2)

How does this fix the problem?

Previously problematic rule

$$\frac{\Gamma \vdash \boxed{e_1} : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash \cdots : \cdots}$$

Now: no problem

- Running algorithm gives (σ_1, Γ_1) and (σ_2, Γ_2)

How does this fix the problem?

Previously problematic rule

$$\frac{\Gamma \vdash \boxed{e_1} : \sigma_1 \qquad \Gamma \vdash \boxed{e_2} : \sigma_2}{\Gamma \ \vdash \cdots : \cdots}$$

Now: no problem

- Running algorithm gives (σ_1, Γ_1) and (σ_2, Γ_2)

How does this fix the problem?

Previously problematic rule

$$\frac{\Gamma \vdash \; e_1 \; : \sigma_1 \qquad \Gamma \vdash \; e_2 \; : \sigma_2}{\Gamma \; \vdash \cdots : \cdots}$$

Now: no problem

- Running algorithm gives (σ_1, Γ_1) and (σ_2, Γ_2)
- For

$$x :_{[R_1]} \sigma \in \Gamma_1 \qquad \text{and} \qquad x :_{[R_2]} \sigma \in \Gamma_2,$$

put $x :_{[\mathbf{max}(R_1, R_2)]} \sigma$ in output context

How does this fix the problem?

Previously problematic rule

$$\frac{\Gamma \vdash \; e_1 \; : \sigma_1 \qquad \Gamma \vdash \; e_2 \; : \sigma_2}{\Gamma \vdash \cdots : \cdots}$$

Now: no problem

- Running algorithm gives (σ_1, Γ_1) and (σ_2, Γ_2)
- For

  $$x :_{[R_1]} \sigma \in \Gamma_1 \qquad \text{and} \qquad x :_{[R_2]} \sigma \in \Gamma_2,$$

  put $x :_{[\mathbf{max}(R_1, R_2)]} \sigma$ in output context
- Return $\mathbf{max}(R_1, R_2)$ as context

Decidability?

Bad news

- Must check inequalities over reals and natural polynomials
- Subtype relation is undecidable
- Even checking validity of derivations is undecidable
- Problem for both DFuzz and EDFuzz

Decidability?

Bad news

- Must check inequalities over reals and natural polynomials
- Subtype relation is undecidable
- Even checking validity of derivations is undecidable
- Problem for both DFuzz and EDFuzz

Good news

- Constraint solvers are pretty good in practice
- Typical DFuzz programs rely on easy constraints

Special structure of constraints

- Allow standard (DFuzz) annotations only
- Subtyping only needs to check

  $$R \geq R^*,$$

  where R is a DFuzz sensitivity and R^* is a EDFuzz sensitivity
- R understood by standard numeric solvers
- R^* has extended terms like $\mathbf{max}(R_1, R_2), \ldots$

Checking the constraints

Idea: eliminate extended terms

- Change $R \geq \mathbf{max}(R_1^*, R_2^*)$ to

$$R \geq R_1^* \wedge R \geq R_2^*$$

- Recursively eliminate comparisons $R \geq R^*$
- Similar technique for other new sensitivity constructions

It works!

- Dispatches numeric constraints to Why3
- Typechecks examples from the DFuzz paper with no problems
- Annotation burden light on these examples

Final thoughts

Lessons learned

- Typechecking with quantitative constraints is tricky
- Numeric solvers are quite good, even for undecidable problems
- Minor details in original language can have huge effects on how easy it is to use standard solvers
- Keep typechecking in mind!

Final thoughts

Lessons learned

- Typechecking with quantitative constraints is tricky
- Numeric solvers are quite good, even for undecidable problems
- Minor details in original language can have huge effects on how easy it is to use standard solvers
- Keep typechecking in mind!

Open questions

- Does this technique of "completing" a language to ease typechecking apply to other quantitative type systems?
- Can we remove the argument type annotation in functions?

Really Naturally Linear Indexed Type Checking

Arthur Azevedo de Amorim[1], Marco Gaboardi[2],
Emilio Jesús Gallego Arias[1], Justin Hsu[1]

[1]University of Pennsylvania
[2]University of Dundee

October 2, 2014

Another example

Problematic rule

$$\frac{\Gamma \vdash e : \sigma \qquad i \text{ fresh in } \Gamma}{\Gamma \vdash \Lambda i : \kappa .\ e : \forall i : \kappa .\ \sigma}$$

Avoidance problem

- Running typechecker on $(e, \Gamma^\bullet)$ yields (σ, Γ)
- For $x :_{[R]} \sigma \in \Gamma$, want smallest R^* bigger than R but independent of i
- Again: R^* may lie outside sensitivity language
- Add construction $\mathbf{sup}(R, i)$ to EDFuzz