Automatic Sensitivity Analysis using Linear Dependent Types

Marco Gaboardi[1,2], Emilio Jesús Gallego Arias[1],
Andreas Haeberlen[1], Justin Hsu[1], and Benjamin C. Pierce[1]

[1] University of Pennsylvania
{emilioga,ahae,justhsu,bcpierce}@cis.upenn.edu
[2] Università di Bologna - INRIA Focus team
gaboardi@cs.unibo.it

Abstract. Function *sensitivity*—how much a function output can change with respect to changes in the input—is a key property in many research areas. For instance, in the field of *differential privacy*, a common mechanism for making a (possibly non-private) query private involves establishing a bound on its sensitivity.
One way to bound the sensitivity of functional programs is by using a type-based analysis combining linear indexed types and dependent types — the main concept behind *DFuzz*, a language for differentially private queries. In this paper we propose a type checking algorithm for the system of [1] to make this analysis automatic, by reducing sensitivity analysis to constraint solving — a solution to these constraints provides an upper bound on program sensitivity.

1 Introduction

The *sensitivity* of a function $f(x)$ is an upper bound on how much $f(x)$ can change in response to a change to x—in other words, if f has sensitivity k, then $|f(x+\delta) - f(x)| \leq k \cdot |\delta|$ for all x and δ. This property, also known as *Lipschitz continuity*, can be extended to entire programs with multiple inputs, and it has important applications in many parts of computer science, including control theory [2], dynamic systems [3], program analysis [4], and data privacy [5].

In these applications, it is often necessary to *verify* a claim that a given function or program has a particular sensitivity. For instance, consider the following scenario from the differential privacy literature [5]: A querier Q asks the owner of a private database D to run some program f on D and then report the result $f(D)$ back to Q. It is known that this can be done safely, as long as a) the program f has a finite sensitivity, and b) the answer $f(D)$ is perturbed with a bit of random "noise", the amount of which depends on the sensitivity of f. Note that f can be an arbitrary program, and that the safety of the response depends on knowing the correct sensitivity; if the owner of the database underestimates the sensitivity of f, the result can be a serious privacy breach.

One promising approach is the use of a *linear type system* to approximate the sensitivity of a function [6,1,7].

In particular, Reed and Pierce [6] proposed *Fuzz*, a functional programming language for writing differentially private queries. *Fuzz* type system is based on linear logic, with functional types indexed by real numbers reflecting the sensitivity of functions. Thus, a k-sensitive function f from σ to τ has type $f : !_k\sigma \multimap \tau$, with k a bound on the usually unbounded ! operator. For instance, the function $\lambda x.x + x : \mathbb{R} \to \mathbb{R}$ has a sensitivity-aware type $!_2\mathbb{R} \multimap \mathbb{R}$.

As is standard in functional programming, the typing rules enable compositional reasoning. Thus, the sensitivity of the overall program can be verified from the known sensitivities of the language primitives. This approach is attractive because it produces a formal proof that a given program has a certain sensitivity; however, it requires the programmer to make extensive sensitivity annotations throughout the program, which is tedious, error-prone, and hard to maintain, since a small change in some part may require the user to redo many annotations. In addition, the use of linear logic imposes some burden with context management [8,9], In [10,11] the problems of type-based sensitivity checking, inference and minimization are addressed.

However, some applications of sensitivity analysis require a finer-grain analysis than the *Fuzz* type system can provide. Indeed, in differential privacy it is common for the sensitivity of a function to depend on its input. The extension of *Fuzz*'s linear type system to data-dependent analysis leads to dependent types. In [1], the authors present *DFuzz*, an extension of *Fuzz* with *linear dependent types* [12].

DFuzz's new features are singleton dependent types [13], dependent pattern matching, and subtyping (which is also present in [11]). This new rich type system greatly improves the expressivity of sensitivity analysis; sensitivies depending on the number of iterations or on the number of elements of a list can be expressed and checked. However, as the power of the analysis increases, typechecking either becomes more challenging, or requires complex annotations from the user.

In this paper, we propose a type checking algorithm for a *DFuzz*. First, the introduction of size and sensitivity variables means that our domain of analysis shifts from reals to polynomials over several variables; second, dependent pattern matching introduces typing under refinement of size variables. We handle subtyping using standard approaches.

The type checking algorithm is developed in three steps. First, we define a syntax-directed typing system to eliminate the non-determinism consequence of the subtyping rules. Second, an algorithmic system generates a set of constraints over an extended language of real and integer polynomials. Finally, the constraints are elaborated to first order formulas. Additionally, we are extending the implementation of [10] in order to incorporate these features.

2 Linear Dependent Types and Sensitivity Analysis: DFuzz

Fig. 1 describe the various DFuzz language constructs: size expressions S are either natural numbers or variables, sensitivities R are real polynomials over two-kinded variables κ ranging over the extended positive reals $\mathbb{R}^\infty$ and the natural numbers $\mathbb{N}$.

Expressions e are drawn from a vanilla linear lambda calculus, with real and natural constants, additive pairs, sensitivity abstraction/application, and dependent pattern matching over sizes.

Types σ are standard but for the functional type, which is annotated with a polynomial denoting the function sensitivity. We have regular typing environments, which are extended with a sensitivity annotation, sensitivity variables environments used in sensitivity abstraction and constraints over sizes, which refine knowledge about in-context size indexes.

$$
\begin{array}{lll}
\kappa & ::= \mathrm{r} \mid \mathrm{n} & \text{(kinds)} \\
N & ::= \mathbb{N} & \text{(natural numbers)} \\
\mathbb{R}^\infty & ::= \mathbb{R} \geq 0 \cup \{\infty\} & \text{(ext. positive reals)} \\
S & ::= i \mid 0 \mid S + 1 & \text{(size expressions)} \\
R & ::= \mathrm{r} \mid i \mid S \mid R + R \mid R * R & \text{(sens. expressions)} \\
\sigma, \tau & ::= \mathbb{R} \mid \mathbb{N}[n] \mid !_R\sigma \multimap \tau \mid \sigma \,\&\, \tau \mid \forall i.\sigma & \text{(types)} \\
v & ::= x \mid N \mid \mathbb{R}^\infty \mid \lambda x : !_R\sigma.\ e \mid \langle v_1, v_2 \rangle \mid \Lambda i.\ e & \text{(values)} \\
e & ::= x \mid N \mid \mathbb{R}^\infty \mid \lambda x : !_R\sigma.e \mid e_1\ e_2 \mid \langle e_1, e_2 \rangle \mid \pi_i\ e \mid \Lambda i.\ e \mid e[R] \mid & \\
 & \quad \mathbf{case}\, e\, \mathbf{of}\, 0 \Rightarrow e_l \mid n_{[i]} + 1 \Rightarrow e_r & \text{(expressions)} \\
\Gamma, \Delta & ::= \emptyset \mid \Gamma, x : !_R\sigma & \text{(environments)} \\
\phi, \psi & ::= \emptyset \mid \phi, i : \kappa & \text{(sens. environments)} \\
\Phi, \Psi & ::= \top \mid \Phi \wedge S = 0 \mid \Phi \wedge S = i + 1 & \text{(constraints)}
\end{array}
$$

Fig. 1: *DFuzz* syntax

Linear dependent types usually are carry an associated subtyping relation (Fig. 2), which in this case captured the fact that a k-sensitive function is also k'-sensitive for any $k' \geq k$. The subtyping relation is largely standard, but we should remark that the subtyping is defined with respect to a set of variables and a constraint store, representing the context under which the sensitivity expressions are defined.

The typing relation is presented in Fig. 3. It comprises the usual subtyping at both sides of the judgment, variables, constants and introduction and elimination rules for the linear constructions. The most interesting rule is the ($\mathbb{N}\ E$), which describes dependent pattern matching over sizes. If the matched-against expression e can be typed as dependent size $\mathbb{N}[S]$, then we each branch must be typed under a refinement of the index S. This involves introducing a new sensitivity variable in the successor case. In order to define the main soundness

$$\frac{}{\phi; \Phi \models \sigma \sqsubseteq \sigma} \sqsubseteq\text{-Refl} \qquad \frac{\phi; \Phi \models \sigma' \sqsubseteq \sigma \qquad \phi; \Phi \models \tau \sqsubseteq \tau'}{\phi; \Phi \models \sigma \,\&\, \tau \sqsubseteq \sigma' \,\&\, \tau'} \; (\sqsubseteq . \,\&)$$

$$\frac{\models \forall \phi.(\Phi \supset R \leq R') \qquad \phi; \Phi \models \sigma' \sqsubseteq \sigma \qquad \phi; \Phi \models \tau \sqsubseteq \tau'}{\phi; \Phi \models !_R \sigma \multimap \tau \sqsubseteq !_{R'} \sigma' \multimap \tau'} \; \sqsubseteq\text{-Fun}$$

$$\frac{\phi; \Phi, i : * \models \sigma \sqsubseteq \tau \qquad i \text{ fresh in } \phi}{\phi; \Phi \models \forall i.\sigma \sqsubseteq \forall i.\tau} \; \sqsubseteq\text{-SFun}$$

$$\frac{\forall (x : !_{R_i} \sigma_i, x : !_{R_i'} \tau_i) \in (\Gamma, \Delta), \qquad \models \forall \phi.(\Phi \supset R_i \geq R_i') \qquad \phi; \Phi \models \tau_i \sqsubseteq \sigma_i}{\phi; \Phi \models \Gamma \sqsubseteq \Delta} \; \sqsubseteq\text{-Env}$$

Fig. 2: *DFuzz* Subtyping relation

$$\frac{\phi; \Phi \mid \Delta \vdash e : \sigma \qquad \phi; \Phi \models \Gamma \sqsubseteq \Delta}{\phi; \Phi \mid \Gamma \vdash e : \sigma} \; (\sqsubseteq .\text{L})$$

$$\frac{\phi; \Phi \mid \Gamma \vdash e : \sigma \qquad \phi; \Phi \models \sigma \sqsubseteq \tau}{\phi; \Phi \mid \Gamma \vdash e : \tau} \; (\sqsubseteq .\text{R}) \qquad \frac{}{\phi; \Phi \mid \Gamma \vdash \mathrm{r} : \mathbb{R}} \; \text{Const}$$

$$\frac{}{\phi; \Phi \mid \Gamma, x : !_1 \sigma \vdash x : \sigma} \; (\text{Var}) \qquad \frac{\phi; \Phi \mid \Gamma \vdash e_1 : \sigma \qquad \phi; \Phi \mid \Gamma \vdash e_2 : \tau}{\phi; \Phi \mid \Gamma \vdash \langle e_1, e_2 \rangle : \sigma \,\&\, \tau} \; (\&\, I)$$

$$\frac{\phi; \Phi \mid \Gamma \vdash e : \sigma_1 \,\&\, \sigma_2}{\phi; \Phi \mid \Gamma \vdash \pi_i \; e : \sigma_i} \; (\&\, E)$$

$$\frac{\phi; \Phi \mid \Gamma, x : !_R \sigma \vdash e : \tau}{\phi; \Phi \mid \Gamma \vdash \lambda x : \sigma.e : !_R \sigma \multimap \tau} \; (\multimap I)$$

$$\frac{\phi; \Phi \mid \Gamma \vdash e_1 : !_R \sigma \multimap \tau \qquad \phi; \Phi \mid \Delta \vdash e_2 : \sigma}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash e_1 \; e_2 : \tau} \; (\multimap E)$$

$$\frac{\phi, i : \kappa; \Phi \mid \Gamma \vdash e : \sigma \qquad i \text{ fresh in } \Phi}{\phi; \Phi \mid \Gamma \vdash \Lambda i.e : \forall \sigma} \; (\forall I) \qquad \frac{\phi; \Phi \mid \Gamma \vdash e : \forall i.\sigma \qquad s \;\mathbf{wf}\; \phi}{\phi; \Phi \mid \Gamma \vdash e[s] : \sigma[s/i]} \; (\forall E)$$

$$\frac{}{\phi; \Phi \mid \Gamma \vdash 0 : \mathbb{N}[0]} \; (0 \; I) \qquad \frac{\phi; \Phi \mid \Gamma \vdash e : \mathbb{N}[S]}{\phi; \Phi \mid \Gamma \vdash e + 1 : \mathbb{N}[S+1]} \; (\mathrm{S} \; I)$$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Delta \vdash e : \mathbb{N}[S] \\ \phi; \Phi \wedge S = 0 \mid \Gamma \vdash e_l : \sigma \qquad \phi, i : \mathrm{n}; \Phi \wedge S = i + 1 \mid \Gamma, n : !_R \mathbb{N}[i] \vdash e_r : \sigma\end{array}}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash \mathbf{case}\, e \,\mathbf{of}\, 0 \Rightarrow e_l \mid n_{[i]} + 1 \Rightarrow e_r : \sigma} \; (\mathbb{N} \; E)$$

Fig. 3: *DFuzz* type assignment

property of the type system just presented, we need to define a type-indexed distance relation on values and expressions:

Definition 1 (Distance relations). *Assume the metric-compatible relations* $\vdash v_1 \sim_r v_2 : \sigma$*, which relates two values* v_1, v_2 *of type* σ *at distance* $r \in \mathbb{R}^\infty$ *and;* $\Gamma \vdash e_1 \approx_r e_2 : \sigma$*, which relates two expressions* e_1, e_2 *of type* σ *in environment* Γ *at distance* r*.*

For instance, $\vdash \mathrm{r}_1 \sim_k \mathrm{r}_2 : \mathbb{R} \iff |\mathrm{r}_1 - \mathrm{r}_2| \leq k$ and $\Gamma + R \cdot \Delta \vdash (\lambda x.\ e)\ e_1 \approx_k (\lambda x.\ e)\ e_2 : \tau$ iff $\emptyset; \top \mid \Gamma \vdash \lambda x.\ e : !_i\sigma \multimap \tau$, $\Delta \vdash e_1 \approx_m e_2 : \sigma$, and $i \cdot m \leq k$.

The main soundness result is that distance is preserved by evaluation:

Theorem 1 (Metric preservation). *Let* $\hookrightarrow$ *be the standard call-by-value big-step evaluation relation for* DFuzz*. Then, if* $\vdash e_1 \approx_m e_2 : \sigma$ *and* $e_1 \hookrightarrow v_1$*, then* $\exists v_2$ *such that* $e_2 \hookrightarrow v_2$ *and* $\vdash v_1 \sim_m v_2 : \sigma$*.*

3 Type Checking for DFuzz

Given a typing judgment $\Gamma \vdash e : \sigma$, we want to answer the question of whether a derivation for it exists. Usually, the term e is understood as a proof of the proposition σ which — assuming Church style proofs — makes type checking straightforward. However, enriched type systems will either significantly increase the annotation burden or require the presence of additional machinery to infer temporal propositions.

In this paper, *DFuzz* is presented in a *Church*-style, meaning that every variable introduced is annotated by the user with its sensitivity and type. Thus, checking a *DFuzz* judgment requires solving two particular sub-problems:

1. The subtyping rules $(\sqsubseteq .\mathrm{L}), (\sqsubseteq .\mathrm{R})$ introduce non-determinism in type assignment.
2. The application rule (*ruleeapp*) introduces non-determinism given that there are multiple ways of splitting a given context, which cannot be predicted without prior analysis of the subexpressions involved in the application.

The difficulty of both of these problems increases for the presence of polynomials and dependent pattern matching.

To overcome challenge 1), we may use the standard algorithmic approaches to subtyping (i.e. [14, p. 255]).

The main idea is to force subtyping to happen at the points where actual types are compared for equality. For this idea to work, the notion of minimal subtype with respect to the subtyping relation must exist. Then, type checking keeps record of the *minimal* synthesized type — the minimal sensitivity of a function in this case — broadening it under demand. The new system is syntax directed, providing a direct path to the algorithmic version of the type checking procedure.

The solution for challenge 1) proves useful for solving challenge 2). If inferred sensitivities are minimal, we already have the best possible assignments for context splitting. We may understand $(\multimap E)$ algorithmicaly as a request to performing resource analysis on both sub-terms of the application.

3.1 Syntax-directed typing

The purpose of this section is to introduce a syntax-directed version of the *DFuzz* type system. This amounts to eliminating the rules concerning subtyping. Would we remain in a core lambda calculus, every expression would feature a minimal sensitivity, making this approach feasible. However, this is not the case in the full *DFuzz* system. In particular, lets focus in $(\& \ I)$ rule. We could synthesize two different sensitivities for a given variable in context, but the rule may need to return the least upper bound of two polynomials, and this is easily seen not to exists in general (take $x + 1$ and x^2).

Our solution is to extend the synthesized sensitivities adjoining to them best upper bounds for polynomials. Note that these new best upper bounds are not polynomials themselves, but admit interpretation over a standard n-dimensional space. We must also calculate best sensitivities in other two extra cases: the best sensitivity that doesn't mention a given variable i, and the sensitivity depending on a refinement constraint:

Definition 2 (Extended sensitivities). *The extended language of sensitivities $\hat{R}$ is defined as:*

$$\begin{aligned}\hat{R} ::= & \ R \\ & \mid \hat{R} + \hat{R} \mid \hat{R} \cdot \hat{R} \\ & \mid \mathbf{lub}(\hat{R}, \hat{R}) \\ & \mid \mathbf{free}(i\#\hat{R}) \\ & \mid \mathbf{nref}(S \| \hat{R}^{|_0}, \hat{R}^{|_{i+1}})\end{aligned}$$

- $\mathbf{lub}(\hat{R}, \hat{R})$ *represents the least upper bound of two polynomials or extended sensitivities.*
- $\mathbf{free}(i\#\hat{R})$ *represents the least polynomial or extended sensitivity that is free of the variable i and greater than $\hat{R}$*
- $\mathbf{nref}(S \| \hat{R}_1^{|_0}, \hat{R}_2^{|_{i+1}})$ *represents the least upper bound for $\hat{R}_1$ and $\hat{R}_2$ assuming $S = 0$ for the first one and $S = i + 1$ for the second extended sensitivity, with i a fresh size variable.*

The extended language implies that the subtyping relation must be accordingly extended. We write $\sqsubseteq_{\mathcal{S}}$ for the extended relation and study it on Sec. 3.3. The new sensitivity constructors are extended to environment and type operators in the standard way.

Remark 1. We may interpret the language of extended sensitivities in a piecewise manner.

Using the extended language, we define a syntax-directed system shown in Fig. 4 (straightforward rules like $((\& \ E))$ are omitted for brevity). Every derivation in the syntax-directed system induces a derivation on the original, non-deterministic one. Every original derivation in the non-deterministic system can be recovered by a syntax-directed derivation plus a subtyping step.

$$\frac{}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \mathrm{r} : \mathbb{R}} \text{ Const} \qquad \frac{\mathrm{zctx}(\Gamma)}{\phi; \Phi \mid \Gamma, x : !_1 \sigma \vdash_{\mathcal{S}} x : \sigma} \text{ (Var)}$$

$$\frac{\phi; \Phi \mid \Gamma_1 \vdash_{\mathcal{S}} e_1 : \sigma \qquad \phi; \Phi \mid \Gamma_2 \vdash_{\mathcal{S}} e_2 : \tau}{\phi; \Phi \mid \mathbf{lub}(\Gamma_1, \Gamma_2) \vdash_{\mathcal{S}} \langle e_1, e_2 \rangle : \sigma \,\&\, \tau} \ (\& \ I)$$

$$\frac{\phi; \Phi \mid \Gamma, x : !_{\hat{R}'} \sigma \vdash_{\mathcal{S}} e : \tau \qquad \models \forall \phi.\ (\Phi \supset R \geq_{\mathcal{S}} \hat{R}')}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} \lambda x : \sigma . e : !_R \sigma \multimap \tau} \ (\multimap I)$$

$$\frac{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e_1 : !_R \sigma \multimap \tau \qquad \phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e_2 : \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq_{\mathcal{S}} \sigma}{\phi; \Phi \mid \Gamma + R \cdot \Delta \vdash_{\mathcal{S}} e_1\ e_2 : \tau} \ (\multimap E)$$

$$\frac{\phi \mid \Phi, i : \kappa; \Gamma \vdash_{\mathcal{S}} e : \sigma}{\phi; \Phi \mid \mathbf{free}(i \# \Gamma) \vdash_{\mathcal{S}} \Lambda i . e : \forall \sigma} \ (\forall I) \qquad \frac{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e : \forall i . \sigma \qquad \phi \vdash s}{\phi; \Phi \mid \Gamma \vdash_{\mathcal{S}} e[s] : \sigma[s/i]} \ (\forall E)$$

$$\frac{\begin{array}{c}\phi; \Phi \mid \Delta \vdash_{\mathcal{S}} e : \mathbb{N}[S] \\ \phi; \Phi \wedge S = 0 \mid \Gamma_l \vdash_{\mathcal{S}} e_l : \sigma_l \qquad \phi, i : \mathrm{n}; \Phi \wedge S = i + 1 \mid \Gamma_r, n : !_R \mathbb{N}[i] \vdash_{\mathcal{S}} e_r : \sigma_r\end{array}}{\begin{array}{c}\phi; \Phi \mid \mathbf{nref}(S \| {\Gamma_l}^{\mid 0}, {\Gamma_r}^{\mid i+1}) + R \cdot \Delta \vdash_{\mathcal{S}} \mathbf{case}\, e\, \mathbf{of} \\ 0 \Rightarrow e_l \mid n_{[i]} + 1 \Rightarrow e_r : \mathbf{nref}(S \| {\sigma_l}^{\mid 0}, {\sigma_r}^{\mid i+1})\end{array}} \ (\mathbb{N}\ E)$$

$$\mathrm{zctx}(\Gamma) \iff \forall (\_ : !_{i} \_) \in \Gamma.\ i = 0$$

Fig. 4: *DFuzz* type assignment, syntax directed version

Lemma 1 (Syntax-directed correctness). *If $\Gamma \vdash_{\mathcal{S}} e : \sigma$ has a derivation, then $\Gamma \vdash e : \sigma$ has.*

Lemma 2 (Syntax-directed completeness). *If $\Gamma \vdash e : \sigma$ has a derivation, then exists Γ' such that $\Gamma' \vdash_{\mathcal{S}} e : \sigma$ has a derivation and $\Gamma' \sqsupseteq_{\mathcal{S}} \Gamma$.*

3.2 Sensitivity generation

The syntax-directed system of the previous section enables us to derive an algorithmic type-checking system presented in Fig. 5.

This algorithmic type system is based on judgments of the form:

$$\phi; \Phi; \Gamma^{\bullet}; e \Longrightarrow \Gamma; \sigma$$

capturing the expression e, the candidate environment $\Gamma^{\bullet}$ (containing types with no external sensitivity), a constraint Φ and a sensitivity environment ϕ as inputs, the algorithm outputs a typing environment Γ and a type σ.

$$\frac{}{\phi; \Phi; \Gamma^{\bullet}; 0 \Longrightarrow \mathrm{zctx}(\Gamma); \mathbb{N}[0]} \ \text{Zero} \qquad \frac{\phi; \Phi; \Gamma^{\bullet}; e \Longrightarrow \Gamma; \mathbb{N}[S]}{\phi; \Phi; \Gamma^{\bullet}; \mathbf{s}\ e \Longrightarrow \Gamma; \mathbb{N}[S+1]} \ \text{Succ}$$

$$\frac{}{\phi; \Phi; \Gamma^{\bullet}; \mathrm{r} \Longrightarrow \mathrm{zctx}(\Gamma); \mathbb{R}[\mathrm{r}]} \ \text{Const}$$

$$\frac{}{\phi; \Phi; \Gamma^{\bullet}, x : \sigma; x \Longrightarrow \mathrm{zctx}(\Gamma), x : !_1\sigma; \sigma} \ (\text{Var})$$

$$\frac{\phi; \Phi; \Gamma^{\bullet}, x : \sigma; e \Longrightarrow \Gamma, x : !_{R'}\sigma; \tau \qquad \phi; \Phi \models R \geq_{\mathcal{S}} R'}{\phi; \Phi; \Gamma^{\bullet}; \lambda x : !_R\sigma.e \Longrightarrow \Gamma; !_R\sigma \multimap \tau} \ (\multimap I)$$

$$\frac{\phi; \Phi; \Gamma^{\bullet}; e_1 \Longrightarrow \Gamma; !_R\sigma \multimap \tau \qquad \phi; \Phi; \Gamma^{\bullet}; e_2 \Longrightarrow \Delta; \sigma' \qquad \phi; \Phi \models \sigma' \sqsubseteq_{\mathcal{S}} \sigma}{\phi; \Phi; \Gamma^{\bullet}; e_1\ e_2 \Longrightarrow \Gamma + R \cdot \Delta; \tau} \ (\multimap E)$$

$$\frac{\phi, i; \Phi; \Gamma^{\bullet}; e \Longrightarrow \Gamma; \sigma}{\phi; \Phi; \Gamma^{\bullet}; \Lambda i.e \Longrightarrow \mathbf{free}(i \# \Gamma); \forall i.\sigma} \ (\forall I) \qquad \frac{\phi; \Phi; \Gamma^{\bullet}; e \Longrightarrow \Gamma; \forall i.\sigma \qquad \phi \vdash S}{\phi; \Phi; \Gamma^{\bullet}; e[S] \Longrightarrow \Gamma; \sigma[S/i]} \ (\forall E)$$

$$\frac{\phi; \Phi; \Gamma^{\bullet}; e_1 \Longrightarrow \Gamma_1; \sigma_1 \qquad \phi; \Phi; \Gamma^{\bullet}; e_2 \Longrightarrow \Gamma_2; \sigma_2}{\phi; \Phi; \Gamma^{\bullet}; \langle e_1, e_2 \rangle \Longrightarrow \mathbf{lub}(\Gamma_1, \Gamma_2); \sigma_1 \ \& \ \sigma_2} \ (\& \ I)$$

$$\frac{\begin{array}{c}\phi; \Phi; \Gamma^{\bullet}; e \Longrightarrow \Gamma_1; \mathbb{N}[S] \\ \phi; \Phi, S = 0; \Gamma^{\bullet}; e_1 \Longrightarrow \Gamma_1; \sigma_1 \qquad \phi, i; \Phi, S = i+1; \Gamma^{\bullet}; e_2 \Longrightarrow \Gamma_2; \sigma_2 \\ \Delta \equiv \mathbf{nref}(S \| {\Gamma_1}^{|_0}, {\Gamma_2}^{|_{i+1}}) \qquad \sigma \equiv \mathbf{nref}(S \| {\sigma_1}^{|_0}, {\sigma_2}^{|_{i+1}})\end{array}}{\phi; \Phi; \Gamma^{\bullet}; \mathbf{case}\, e\, \mathbf{of}\, 0 \mapsto e_1 \mid x[i]+1 \mapsto e_2 \Longrightarrow \Delta; \sigma} \ \textit{case}$$

Fig. 5: Algorithmic rules for *DFuzz*

Assuming the correctness and completeness of the subtyping relation, the algorithm is sound and complete with respect to the original type system.

Theorem 2 (Algorithmic Soundness). *Suppose* $\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma$ *is derivable in the algorithmic system. Then, there is a derivation showing* $\phi; \Phi; \Gamma \vdash_{\mathcal{S}} e : \sigma$.

Theorem 3 (Algorithmic Completeness). *Suppose* $\phi; \Phi; \Gamma \vdash_{\mathcal{S}} e : \sigma$ *is derivable. Then, the algorithm system outputs:* $\phi; \Phi; \Gamma^\bullet; e \Longrightarrow \Gamma; \sigma$.

3.3 Constraint solving

The missing bit of the type checking algorithm is the definition of the extended subtyping relation.

The basic operation of the extended relation is the checking of the inequality $\forall \phi.\Phi \supset R \geq_{\mathcal{S}} \hat{R}$, where $\hat{R}$ is an extended sensitivity, R a sensitivity annotation and ϕ, Φ are regular sensitivity/size variable contexts and constraint stores.

We elaborate the extended sensitivity language by characterizing the extended operations logically, that is to say, every extended sensitivity will get replaced by an existential variable that is logically equivalent to its intended meaning.

Definition 3 (FO elaboration). *We define the function* **elab** *from extended sensitivities to pairs of regular sensitivities and first-order formulas:*

$$
\begin{array}{ll}
\mathbf{elab}(\mathbf{lub}(R_1, R_2)) & = \{X, \exists X1, X2.(X \geq X_1 \wedge X \geq X_2 \wedge C_1 \wedge C_2\} \\
& \quad \textit{where} \quad \mathbf{elab}(R_i) = \{X_i, C_i\} \quad i \in \{1, 2\} \\
\mathbf{elab}(\mathbf{free}(i \# R)) & = \{X, \exists X_R.\forall i.X \geq X_R \wedge C_R\} \\
& \quad \textit{where} \quad \mathbf{elab}(R) = \{X_R, C_R\} \\
\mathbf{elab}(\mathbf{nref}(S \| {R_1}^{|_0}, {R_2}^{|_{i+1}})) & = \{X, (S = 0 \supset X \geq X_1 \wedge C_1) \wedge \\
& \qquad (\forall i.S = i + 1 \supset X \geq X_2 \wedge C_2)\} \\
& \quad \textit{where} \quad \mathbf{elab}(R_i) = \{X_i, C_i\} \quad i \in \{1, 2\} \\
\mathbf{elab}(R) & = \{R, \top\} \qquad \textit{otherwise}
\end{array}
$$

Then, the problem reduces as follows:

$$\forall \phi.\Phi \supset R \geq_{\mathcal{S}} \hat{R} \iff \forall \phi.\Phi \supset (\exists X.(R \geq X \wedge C)) \qquad \mathbf{elab}(R) = \{X, C\}$$

Note that for instance we don't assert the *least* part of the *least upper bound* in the elaboration, as we are already checking for a bound. Indeed the constraint would look like:

$$\forall Z, (Z \geq X_1 \wedge Z \geq X_2) \supset Z \geq X$$

but the reader can check that it would have no effect, as we already are provided for a bound in every problem.

4 Related Work

Sensitivity analyses. Chaudhuri et al. in [15] study an automatic program analysis that can verify robustness of imperative programs. Their notion of robustness is same as program sensitivity: a program is K-robust if an ε-variation of the input can cause the output to vary by at most $\pm K\varepsilon$. In the paper, they further extend this notion by parametrizing it over the "size" of the input (e.g. the number of elements in an array). Their technique performs a numerical analysis of the function computed by the program and then reasons about such function. Our approach differs from the one in [15] in several respects. Their analysis only considers terminating programs for which bounds on the number of loop iterations are known a priori. While our analysis does not impose this restriction, their approach allows to prove robustness of programs for which our analysis fails. On the other hand, we are able to prove potentially non-terminating program to be K-sensitive. On this respect, the two approaches seem complementary. The main difference is however that we consider higher order programs.

Palamidessi and Stronati [16] recently proposed a constraint-based approach to compute the sensitivity of relational algebra queries. In particular, their analysis is able to compute the minimal sensitivity of wide range of queries. In contrast, the goal of our approach is to provide an upper bound on the sensitivity not of relational queries but for higher order functional programs.

Type systems. Many previous works have reduced type inference and type checking to constraint satisfiability (see [17,18] for an introduction).

Linear types are a key tool to support fine grained reasoning about resource management. Type checking and type inference for linear types usually involve some type decoration problem that can be reduced to a corresponding problem on integer constraints [19,20]. These constraints can usually be solved efficiently using constraint solvers for integer programming. Dal Lago and Petit [7] have recently proposed an inference algorithm for a type system for implicit complexity that combines linear indexed types with dependent types. Their algorithm generates integer constraints that can be solved using Why3 [21], a platform combining automatic and interactive solvers. ATS [22] is a language that combines linear types with automatic and interactive solvers for integer constraints. ATS helps reasoning about memory and pointers properties. However, linear types as used in ATS are not enough to reason about the sensitivity of programs. Our approach differs from all of these in the use of non linear constraints over the reals.

Differential privacy. Differential privacy [23] is one of the strongest privacy guarantees that has been proposed to date. Other linguistic tools besides Fuzz have been proposed to ensure differential privacy. PINQ [24] is an SQL-like differentially private query language embedded in C#; Airavat [25] is a MapReduce-based solution using a modified Java VM. CertiPriv [26]. is a machine-assisted framework—built on top of the Coq proof assistant—for reasoning about differ-

ential privacy from first principles. None of these tools provides a static sensitivity analysis method.

References

1. Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A., Pierce, B.C.: Linear dependent types for differential privacy. In: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '13, New York, NY, USA, ACM (2013) 357–370
2. Zames, G.: Input-output feedback stability and robustness, 1959-85. Control Systems, IEEE **16**(3) (June 1996) 61–66
3. Bournez, O., Graça, D.S., Hainry, E.: Robust computations with dynamical systems. In: MFCS'10. LNCS, Springer-Verlag (2010) 198–208
4. Chaudhuri, S., Gulwani, S., Lublinerman, R., NavidPour, S.: Proving programs robust. In: SIGSOFT FSE. (2011)
5. Dwork, C.: Differential privacy: A survey of results. Proc. TAMC (2008)
6. Reed, J., Pierce, B.C.: Distance makes the types grow stronger: A calculus for differential privacy. In: Proc. ICFP. (September 2010)
7. Dal Lago, U., Petit, B.: Geometry of types. In: Proc. ACM POPL. (2013)
8. Hodas, J.S., Miller, D.: Logic programming in a fragment of intuitionistic linear logic: Extended abstract. In: In Kahn, G. (Ed.), Sixth Annual Symposium on Logic in Computer Science. (1991) 32
9. Cervesato, I., Hodas, J.S., Pfenning, F.: Efficient resource management for linear logic proof search. Theor. Comput. Sci. **232**(1-2) (2000) 133–163
10. D'Antoni, L., Gaboardi, M., Gallego Arias, E.J., Haeberlen, A., Pierce, B.C.: Type-based sensitivity analysis (2013) preprint.
11. Eigner, F., Maffei, M.: Differential privacy by typing in security protocols. In: Proceedings of the Computer Security Foundations Symposion. (2013)
12. Dal Lago, U., Gaboardi, M.: Linear dependent types and relative completeness. In: IEEE LICS '11. (2011) 133–142
13. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proc. POPL. (1999)
14. Pierce, B., Steffen, M.: Higher-order subtyping. Theoretical Computer Science **176**(1–2) (1997) 235 – 282
15. Chaudhuri, S., Gulwani, S., Lublinerman, R., NavidPour, S.: Proving programs robust. In: Proc. FSE. (2011)
16. Palamidessi, C., Stronati, M.: Differential privacy for relational algebra: Improving the sensitivity bounds via constraint systems. In Wiklicky, H., Massink, M., eds.: QAPL. Volume 85 of EPTCS. (2012) 92–105
17. Odersky, M., Sulzmann, M., Wehr, M.: Type inference with constrained types. TAPOS **5**(1) (1999) 35–55
18. Pottier, F., Rémy, D.: The essence of ML type inference. In Pierce, B.C., ed.: Advanced Topics in Types and Programming Languages. MIT Press (2005)
19. Atassi, V., Baillot, P., Terui, K.: Verification of ptime reducibility for system F terms: Type inference in dual light affine logic. Logical Methods in Computer Science **3**(4) (2007)
20. Baillot, P., Hofmann, M.: Type inference in intuitionistic linear logic. In: PPDP. (2010) 219–230

21. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011) 53–64
22. Chen, C., Xi, H.: Combining programming with theorem proving. In: Proc. ICFP. (2005) 66–77
23. Dwork, C.: Differential privacy. In: Proc. ICALP. (2006)
24. McSherry, F.: Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In: Proc. SIGMOD. (2009)
25. Roy, I., Setty, S.T.V., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: security and privacy for MapReduce. In: Proc. NSDI. (2010)
26. Barthe, G., Köpf, B., Olmedo, F., Zanella Béguelin, S.: Probabilistic relational reasoning for differential privacy. In: Proc. POPL. (2012)