Let's not make a fuzz about it

Elisabet Lobo-Vesga
Chalmers University of Technology
Gothenburg, Sweden
elilob@chalmers.se

I. Motivation

One of the most popular recipes to achieve differential privacy is to add noise calibrated to the global sensitivity of the data analysis that one wants to make private [1]. This simple idea has also found important applications in the design of programming frameworks to help programmers to implement differentially private programs. This approach has been first pioneered by Reed and Pierce [2] who have designed a functional programming language, named Fuzz, where types allow one to reason about the sensitivity of ones programs. The technical devices that Reed and Pierce use are linear indexed types, where the index represents the sensitivity of a function. This approach has been further extended in a series of works [3]–[6]. In order to extend the expressivity of the sensitivity analysis proposed by Fuzz, these works have added to the type system originally proposed by Reed and Pierce additional programming languages features, such as linear types, modal types and partial evaluation. These features are not mainstream and usually require to design a new language from scratch. Furthermore, these features are part of the classical toolbox of programming language researchers but they are often rather obscure to users who do not have such a background. This makes the languages using these feature only accessible to programmers which are also expert in programming language research.

In this work, we explore a different direction. Instead of creating a new language from scratch, we design a library capable of calculating the sensitivity of programs. For that, we show that the library should be implemented in programming languages where the type-system has some particular features. Specifically, the library is built on a novel use of polymorphism to represent (and prove) sensitivity of functions at the type level, together with the use of type constraints [7], [8], and type-level natural numbers. This idea allows us to design a library whose basic components can be used as a sound calculus for reasoning about sensitivity. Thus, we present DSencity, a concrete implementation of our ideas for the programming language Haskell. We discuss how DSencity can be used to reason about the sensitivity of classical examples working over vectors, such as sum, map, and sort, and we leave reasoning about more complex programs for future work. While our library-based approach is more limited in term of expressivity than some of the previous works, it can be used more directly in the programming workflow, e.g., by integrating it with other Haskell-based DP frameworks [9].

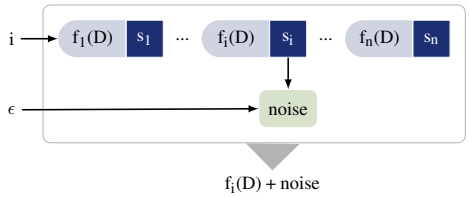

Fig. 1: Differential privacy workflow

II. Background

Differential privacy [1] is a quantitative notion of privacy that bounds how much an individual's private data can affect the result of a data analysis. Intuitively, a query function f is called ϵ-differentially private when for any two datasets D_1 and D_2 differing in one row, the probabilities of obtaining the same output, when f is executed in each dataset, differs by at most ϵ. In other words, ϵ imposes a limit on the *privacy loss* that an individual can incur in, as a result of running the analysis. A standard way to achieve ϵ-differential privacy is by adding some calibrated noise to the result of the function. To protect all the different ways in which an individual's data can affect the output, the noise needs to be scaled by the maximal change that the result of the function can have when changing an individual's data. Such quantification is known as the *sensitivity* [1] of the query.

Determining the sensitivity of a data analysis is a crucial but requires advance reasoning. In practice, frameworks for differential private queries offer a handful of functions whose sensitivity has been proven. Figure 1 depicts a simplification of the general workflow in such systems. Initially, the data analyst selects the i^{th} function (out of range of options) together with the desired privacy loss. Internally, the system has the associated known sensitivity (named s_1, s_i, and s_n) for each supported function. Given user's inputs the sensitivity s_i, alongside ϵ, is used to calibrate the noise that is later added to the result of executing $f_i(D)$, thus the output is randomized protecting data subjects' privacy.

III. Approach

In this section, we present the library DSencity, where developers can write functions, and by doing so, *discover* and *provide a proof* about their sensitivity. Obtaining a proof is merely to convince the type-checker. We argue that the ideas presented in this section are not limited to Haskell but also

```
Lit :: a → Rel 0 a
(:+:) :: Rel d1 Int → Rel d2 Int → Rel (d1+d2) Int
(:*:) :: d~(da+db) ⇒ Rel da a → Rel db b → Rel d (a, b)
```

(a) Relational primitives

```
(:!)   :: (∀(d :: Nat) . Rel d a → Rel (s*d) b)
       → Sen s (a → b)
(:@:) :: Sen s (a → b) → Rel d a → Rel (s*d) a
```

(b) Sensitivity primitives

Fig. 2: DSencity API

scale to other programming languages with advanced type-systems.

DSencity introduces the abstract datatype **data** `Rel (d :: Nat)` a to write programs. This data type is indexed by type-level (`d`) natural numbers—the `d` stands for *distance*. When programming, developers can think about a term `t :: Rel d a` simply as a term of type a. For sensitivity calculations, however, a term `t :: Rel d a` is viewed as a *collection of values of type* a, where the distance among any two values is, at most, `d`. From now on, we will refer to terms `t :: Rel d a`, for a given `d` and a, as relational terms or values.

Basic relational operations

Figure 2a shows some of the basic relational operations: constants (`Lit`), addition (`:+:`), building relational pairs (`:*:`), and changing the distance relation (`:⟨⟩`). In essence, the types of these primitives encode how *distances of the resulting relational values change with respect to the distances of the inputs*. Primitive `Lit x` creates a collection which only contains `x`—thus, the distance is set to `0`. Primitive `:+:` indicates that the distance of added values are at a distance, at most, d_1+d_2. Finally, primitive `:*:` creates relational pairs at distance `d`, where `d` is the addition of the distance of their components.

With this simple API, we can start writing DSencity functions like:

```
f1 x = x:+:(Lit 42)
f2 x = x:*:((f1 x):*:(x:*:x))
```

Function f_1 simply adds the constant `42` to the input, while function f_2 utilizes the argument `x` many times to create a complex pair. What are the distances of the outputs produced by f_1 and f_2 with respect to their inputs? We can simply ask Haskell's type-system about them:

```
> : type f1
Rel d Int → Rel d Int
> : type f2
Rel d Int → Rel (4*d) (Int,(Int,(Int, Int)))
```

Observe how the type-system is doing the distance calculations for us. As a first observation, we see that the API, by construction, tracks how many occurrences of `x` affects the distance inferred for the result—and different from previous work [2], [3], [6], DSencity does not require the utilization of linear type-systems. Furthermore, since the code that we wrote is generic, the type signatures produced by the type-system are polymorphic in `d`. In other words, the code of f_1 and f_2 can be applied to relational terms at distance `1` (i.e., terms of type `Rel 1 a`), `2` (i.e., terms of type `Rel 2 a`), etc.

Parametricity and Sensitivity

The main insight of this work is that parametric polymorphism can capture the fact that outputs' distances in functions are modified in the same manner independently of the inputs' distances, which is no more than the definition of sensitivity. For instance, the type of f_2 indicates that f_2 magnifies the distance of the input by `4` and thus have sensitivity `4`.

DSencity uses the datatype **data** `Sen (s::Nat) (a → b)` for functions which have been proven to have sensitivity `s`. The proof comes from the explicit use of parametricity. Specifically, to create a value of sensitivity `s`, primitive `(:!)`—see Figure 2b—needs to receive, as an argument, a *polymorphic function on the distance of the input*—observe the ∀ `d` in the type-signature—, and where the output's distance is scaled by `s`. By applying `(:!)`, we prove that f_1 and f_2 have sensitivity `1` and `4`, respectively:

```
senf1 :: Sen 1 (Int → Int)
senf1 = (:!) f1
senf2 :: Sen 4 (Int → (Int,(Int,(Int, Int))))
senf2 = (:!) f2
```

Figure 2b also shows primitive `:@:` that enables to apply functions, which have been certified to have a certain sensitivity, to an specific relational value. For instance, if `t :: Rel 2 Int`, then `senf2:@:t` creates a relational pair at distance `8`, i.e., `senf2:@:t :: Rel 8 (Int,(Int,(Int, Int)))`.

Parametricity gives us a simple proof mechanism for sensitivity, but how far can we go with it? How expressive our programs can be? We are still exploring these questions. However our idea seems promising since we have been able to cover some advanced examples similar to the ones found in previous work [2], [3], [6], specifically, we have been able to prove the sensitivity of map, left fold, sum of a vector, and sort.

IV. Final Remarks

We have presented the design of DSencity, a Haskell library which perform sensitivity calculations of programs while proving proofs by parametricity. DSencity is simple and that is its beauty. It is implemented with just 360 lines of Haskell code, which also includes a testing framework of relational operations omitted for brevity. We expect DSencity to serve as a basis for providing a light-weight verification tool to certify the sensitivity of user-defined functions to expand the usability of frameworks for differentially private queries as shown in Figure 3.

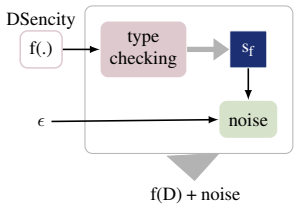

Fig. 3: Integration

REFERENCES

[1] C. Dwork, F. McSherry, K. Nissim, and A. Smith, "Calibrating noise to sensitivity in private data analysis," in *Proceedings of the Third Conference on Theory of Cryptography*, ser. TCC'06, 2006, pp. 265–284.

[2] J. Reed and B. C. Pierce, "Distance makes the types grow stronger: a calculus for differential privacy," in *Proc. ACM SIGPLAN International Conference on Functional Programming*, 2010.

[3] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce, "Linear dependent types for differential privacy," in *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013.

[4] F. Eigner and M. Maffei, "Differential privacy by typing in security protocols," in *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*. IEEE Computer Society, 2013, pp. 272–286. [Online]. Available: https://doi.org/10.1109/CSF.2013.25

[5] D. Winograd-Cort, A. Haeberlen, A. Roth, and B. C. Pierce, "A framework for adaptive differential privacy," *PACMPL*, vol. 1, no. ICFP, pp. 10:1–10:29, 2017. [Online]. Available: https://doi.org/10.1145/3110254

[6] J. P. Near, D. Darais, C. Abuah, T. Stevens, P. Gaddamadugu, L. Wang, N. Somani, M. Zhang, N. Sharma, A. Shan, and D. Song, "Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 172:1–172:30, 2019.

[7] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad-hoc," in *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.

[8] M. P. Jones, "A theory of qualified types," *Sci. Comput. Program.*, vol. 22, no. 3, 1994.

[9] E. Lobo-Vesga, A. Russo, and M. Gaboardi, "A programming framework for differential privacy with accuracy concentration bounds," in *Proc. IEEE Symp. on Security and Privacy*, ser. SP '20. IEEE Computer Society, 2020.