Sensitivity by Parametricity

ELISABET LOBO-VESGA[*], DPella AB, Sweden
ALEJANDRO RUSSO, Chalmers University of Technology, Sweden and DPella AB, Sweden
MARCO GABOARDI, Boston University, USA and DPella AB, Sweden
CARLOS TOMÉ CORTIÑAS, Chalmers University of Technology, Sweden

The work of Fuzz has pioneered the use of functional programming languages where types allow reasoning about the sensitivity of programs. Fuzz and subsequent work (e.g., DFuzz and Duet) use advanced technical devices like linear types, modal types, and partial evaluation. These features usually require the design of a new programming language from scratch—a significant task on its own! While these features are part of the classical toolbox of programming languages, they are often unfamiliar to non-experts in this field. Fortunately, recent studies (e.g., Solo) have shown that linear and complex types in general, are not strictly needed for the task of determining programs' sensitivity since this can be achieved by annotating base types with static sensitivity information. In this work, we take a different approach. We propose to enrich base types with information about the metric relation between values, and we present the novel idea of applying *parametricity* to derive direct proofs for the sensitivity of functions. A direct consequence of our result is that *calculating and proving* the sensitivity of functions is reduced to simply type-checking in a programming language with support for polymorphism and type-level naturals. We formalize our main result in a calculus, prove its soundness, and implement a software library in the programming language Haskell–where we reason about the sensitivity of canonical examples. We show that the simplicity of our approach allows us to exploit the type inference of the host language to support a limited form of sensitivity inference. Furthermore, we extend the language with a privacy monad to showcase how our library can be used in practical scenarios such as the implementation of differentially private programs, where the privacy guarantees depend on the sensitivity of user-defined functions. Our library, called Spar, is implemented in less than 500 lines of code.

CCS Concepts: • **Security and privacy** → **Logic and verification**; • **Theory of computation** → **Program reasoning**.

Additional Key Words and Phrases: sensitivity, functional programming languages, differential privacy, Haskell

ACM Reference Format:
Elisabet Lobo-Vesga, Alejandro Russo, Marco Gaboardi, and Carlos Tomé Cortiñas. 2024. Sensitivity by Parametricity. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 286 (October 2024), 27 pages. https://doi.org/10.1145/3689726

1 Introduction

Differential privacy (DP) is a mathematical definition of privacy that tackles the challenge of extracting informative insights from a population while protecting the privacy of each individual. The standard approach to achieving DP involves computing the desired analysis in a dataset

[*]The majority of this work was done while the author was affiliated with Chalmers University of Technology.

Authors' Contact Information: Elisabet Lobo-Vesga, DPella AB, Gothenburg, Sweden, lobo@dpella.io; Alejandro Russo, Chalmers University of Technology, Gothenburg, Sweden and DPella AB, Gothenburg, Sweden, russo@chalmers.se; Marco Gaboardi, Boston University, Boston, USA and DPella AB, Gothenburg, Sweden, gaboardi@bu.edu; Carlos Tomé Cortiñas, Chalmers University of Technology, Gothenburg, Sweden, carlos.tome@chalmers.se.

ACM 2475-1421/2024/10-ART286
https://doi.org/10.1145/3689726

and then adding calibrated statistical noise to the results before their publication [Dwork et al. 2006]. This simple idea has spawned a series of works (e.g., [Gaboardi et al. 2020; Lobo-Vesga et al. 2020; McSherry 2010; Near et al. 2019; Zhang et al. 2020]) focused on designing programming languages that enable analysts to implement differentially-private consults when accessing sensitive information. At the backbone of every DP programming language resides the noise-calibration mechanism, which determines the amount of noise necessary to mask a person's inclusion in the population. This calibration depends on the desired level of privacy, determined by a parameter ϵ, as well as the global sensitivity of the query. The sensitivity quantifies the extent to which a function's outputs can vary due to modifications in its inputs.

The task of automatically calculating the global sensitivity of arbitrary functions is known to be challenging. As a result, designers of DP systems have conventionally relied on supporting a limited set of pre-defined functions that have a known global sensitivity. Although this approach has enabled several compelling analyses, it significantly restricts the range of queries that can be expressed. To address this limitation, Reed and Pierce [2010] developed Fuzz, a functional programming language that employs linear indexed types tracking programs' sensitivity. This approach has been extended in subsequent works [Eigner and Maffei 2013; Gaboardi et al. 2013; Near et al. 2019; Winograd-Cort et al. 2017], incorporating additional language features such as partial evaluation, linear, and modal types, enhancing Fuzz's expressivity. However, these features are not mainstream and usually require designing a new language from scratch, which can pose significant barriers to adoption for non-experts in programming languages. Moreover, complex language features like linear and modal types are not commonly known outside academic circles and are currently in an experimental phase in mature compilers like GHC [Bernardy et al. 2017], further impeding their adoption.

Recent studies [Abuah et al. 2022; Lobo-Vesga 2021] suggest that linear and complex types in general are not strictly needed for the task of determining programs' sensitivity. Notably, Abuah et al. [2022] introduced Solo, a system for static verification of differential privacy where programs' sensitivity is determined with respect to a set of data sources. In this system, base types are annotated with a sensitivity environment which is tracked and modified by the various operations in a taint-analysis fashion; effectively circumventing linear typing disciplines. Solo shows that polymorphism of indexed types, which are used to represent sensitivity environments at the type level, is sufficient to compute the sensitivity of user-defined functions statically. However, the formal model presented in [Abuah et al. 2022] is inherently *monomorphic* in the sensitivity environments, i.e. in the type indices, creating a disconnect between what is used in the language and its formal model. This critical gap becomes a tangible issue when examining Solo's `foldr` primitive—the system's sole resource of recursion—that is introduced with a type that is polymorphic in the sensitivity environments. Indeed, the reader can see that its type signature is unsound in Section **??** of the accompanying material. Although it is possible to recover the soundness of `foldr` in Solo by making its type signature monomorphic in the sensitivity annotations, there remains an uncertainty regarding the correctness of others Solo's polymorphic sensitivity annotations. (Section 8 provides a more detail comparison between Solo and our work.)

It is against this backdrop of unsoundness in Solo that we highlight the importance of a sound model connecting the use of polymorphism with sensitivity calculations, which is the main contribution of this work. Concretely, in this paper, we introduce Spar, a ready-to-use Haskell library aimed at obtaining the sensitivity of user-defined functions via tracking the distance between values and utilizing the type checker to provide evidence of how much a program will amplify the inputs' distance—thus offering a *direct proof of function sensitivity*. Our approach is rooted in a novel use of *parametricity* that in combination with type constraints [Jones 1994; Wadler and

Blott 1989], and type-level numbers can verify the sensitivity of functions—including higher-order ones—by simply type-checking.

We formalize our ideas in λ_{SPAR}: a sound calculus capable of proving the sensitivity of programs by just relying on features currently available in strongly-typed functional programming languages. Additionally, by having SPAR as a concrete implementation of λ_{SPAR} in the Haskell programming language, we demonstrate how this approach can be effectively used to reason about the sensitivity of functions through classical examples such as summing, mapping, and sorting elements of a vector. Because SPAR is provided as an embedded domain-specific language, our implementation leverages Haskell's advanced type inference to offer limited support for sensitivity inference.

We argue that the main result of this work opens the door to integrating procedures for automatically proving the sensitivity of functions in the programming workflow, e.g., by using SPAR's sensitivity proofs as an input to other Haskell-based DP frameworks [Lobo-Vesga et al. 2020].

To support this claim, we show how λ_{SPAR} and SPAR can be equipped with a privacy monad that can be used to implement a generalization of the Laplace mechanism, which is a standard mechanism for adding noise to differentially private queries. Intuitively, for a given SPAR function, we use the discovered sensitivity to add a proportional noise to its output, thus ensuring that the function is differentially private. We use this approach to implement canonical differentially private queries, and while this is not the central focus of the paper, it serves as a demonstration of the versatility and potential inherent in SPAR to write highly reliable code, i.e., with sensitivity proofs. Moreover, the implementation of these canonical examples allows us to explore SPAR's expressivity in comparison to existing frameworks. By implementing the Laplace mechanism (a generalized version of Fuzz's `addNoise` primitive), we demonstrate that Spar can effectively perform a range of privacy-preserving computations on par with those in Fuzz and SOLO. Additionally, by leveraging Haskell's lightweight dependent types, we showcase SPAR's ability to express generic folds, similar to those in DFuzz's [Gaboardi et al. 2013] linear-dependent system. Notably, these generic folds cannot be expressed by Fuzz or SOLO, highlighting SPAR's unique strength in managing complex and higher-order functions. This comprehensive demonstration underscores SPAR's advanced expressivity while retaining the simplicity and accessibility of a Haskell library.

In summary, the main contributions of this paper are as follows:

- The first use, to the best of our knowledge, of parametrically polymorphic functions to prove sensitivity via distance tracking.
- A formal calculus that captures our ideas (Section 2), and the first soundness proof, to the best of our knowledge, of the use of parametricity to compute the sensitivity of functions (Section 3.1).
- An implementation of our calculus as a ready-to-use Haskell library with case studies of user-defined sensitive functions like `map`, `filter`, `folds`, and `sort` (Section 4), as well as canonical differentially-private algorithms such as Cumulative Distribution Functions (CDFs) and k-means clustering (Section 6).

1.1 Motivating Examples

Before we dive into our formalism and soundness proof, we showcase our main contributions by examples using our software library SPAR. The library is designed for developers to write functions and, by doing so, *discover* and *provide proof* of their sensitivity. Despite SPAR being implemented in the functional programming language Haskell, we argue that the ideas presented here can be deployed in other programming languages supporting parametric polymorphism and a type system with support for type-level natural numbers—a detailed discussion about the required features can be found in Section 5.

We consider a function to be k-sensitive (or have sensitivity k) if it magnifies the distance of its inputs by a factor of *at most* k. Formally:

DEFINITION 1.1 (SENSITIVITY [REED AND PIERCE 2010]). *Given two metric spaces (A, d_A) and (B, d_B), a function $f : A \to B$ is k-sensitive iff: $\forall\, x_1, x_2 \in A.\ d_B(f(x_1), f(x_2)) \leqslant k * d_A(x_1, x_2)$*

The definition above concerns *multiplicative sensitivity*, which is the notion used when considering differential privacy as an application and it is the focus of this work. We do not consider other notions of sensitivity, e.g., where the output of the function is bounded exponentially or quadratically with respect to the inputs' distance. Furthermore, in this work, we will employ an equivalent definition of function sensitivity that differs from the commonly presented one (Definition 1.1).

DEFINITION 1.2 (ALTERNATIVE SENSITIVITY DEFINITION). *Given two metric spaces (A, d_A) and (B, d_B), a function $f : A \to B$ is k-sensitive iff:*
$\forall n \in \mathbb{R},\ x_1, x_2 \in A.\ d_A(x_1, x_2) \leqslant n \Rightarrow d_B(f(x_1), f(x_2)) \leqslant k * n$

We have deliberately chosen to embrace this alternative definition due to its seamless integration with SPAR's type system. Nonetheless, the equivalence between definitions 1.1 and 1.2 follows from the monotonicity of multiplication.

We start by considering a simple function that adds the constant 42 to any given numerical value; in Haskell, we write `add42 x = x+42` with type `Int → Int`. If $d_{\texttt{Int}}$ is defined as the Euclidean distance, we can intuitively notice that `add42` is 1-sensitive since for any two possible inputs at a certain distance, it produces outputs that are at the same distance. For instance, if we consider inputs 5 and 23, where $d_{\texttt{Int}}(5, 23) = 18$, then outputs 47 and 65 are also at distance 18, i.e., $d_{\texttt{Int}}(47, 65) = 18$.

User-defined functions, however, are often more complex than `add42`, and as functions' complexity increases, the less intuitive it is to reason about their sensitivity. For instance, the function `nest x = (x, (add42 x, (x, x)))` of type `Int → (Int, (Int, (Int, Int)))` utilizes its input several times to create a series of nested pairs. If we define the distance between two pairs to be the sum of the distances between each pair of components, what is `nest`'s sensitivity? While the sensitivity-knowledgeable reader could quickly answer this question, the everyday programmer might struggle to perform such a cumbersome analysis. More importantly, we argue that the responsibility of conducting such critical calculations should not fall on the (error-prone) programmer but rather on their (reliable) programming tools.

With this in mind SPAR introduces the abstract datatype `Rel (d :: Nat) a` indexed by type-level natural numbers (`d :: Nat`), where `d` stands for distance. When programming, developers can think about the term `t :: Rel d a` simply as a term of type `a`—i.e., `Rel d a` is isomorphic to `a`. However, for sensitivity calculations, a term `t :: Rel d a` will be interpreted as the *collection* of all pairs of values of type `a` whose distance is at most `d`. From now on, we will refer to the terms `t :: Rel d a`, for a given `d` and `a`, as relational terms or values.

Together with the data type `Rel d a`, SPAR exposes a set of basic relational operations such as:

```
lit  :: Int → Rel d Int
(:+:) :: Rel d₁ Int → Rel d₂ Int → Rel (d₁+d₂) Int
(:⋆:) :: Rel dₐ a → Rel d_b b → Rel (dₐ+d_b) (a, b)
```

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` lifts a regular integer into a relationl one, which means that an integer can be at any distance from another one—thus, the distance is parametric on `d`. Primitive `(:+:)` indicates that the distance of added values is, at most, d_1+d_2. The primitive `(:⋆:)` creates relational pairs at distance d_a+d_b, thus encoding pairs under the ℓ_1 norm. For simplicity, we demonstrate how SPAR works for the norm ℓ_1 but adding more primitives to

```
1 add42 x = x :+: (lit @0 42)
2
3 > :type add42
4 Rel d Int → Rel d Int
5
6 sen_add42 :: Sen 1 Int Int
7 sen_add42 = add42
```

```
 8 nest x = x :★: (add42 x :★: (x :★: x)))
 9
10 > :type nest
11 Rel d Int → Rel (4*d) (Int, (Int, (Int, Int)))
12
13 sen_nest :: Sen 4 Int (Int, (Int, (Int, Int)))
14 sen_nest = nest
```

Fig. 1. Examples in Spar

support other norms like ℓ_∞ is possible by either adding an extra constructor (as in Fuzz [Reed and Pierce 2010]) (:&:) :: Rel d_a a → Rel d_b b → Rel (Max d_a d_b) (a, b) or by simply indexing the pair constructor (:★:) by the metric being used (as in Solo [Abuah et al. 2022]).

With these simple operations, we can rewrite our previous examples as the Spar functions shown in Figure 1 lines 1 and 8, where we use Haskell's type applications (@) to indicate that 42 will be a constant, thus its distance is set to 0. Aided by Haskell's type system we can inspect how much add42 and nest magnify the distance between their inputs (see lines 3-4 and lines 10-11, respectively). Function add42 preserves its inputs' distances while nest quadruples them. Spar, by construction, tracks how many occurrences of x affect the distance inferred for the results—which diverges from previous work [Gaboardi et al. 2013; Near et al. 2019; Reed and Pierce 2010] in requiring the utilization of linear type systems. Moreover, 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 add42 and nest can be applied to relational terms at distance 1 (i.e., terms of type Rel 1 a), terms at distance 2 (i.e., terms of type Rel 2 a), etc.

The central insight of this work is that parametric polymorphism can capture the fact that outputs' distances in functions are bounded in the same manner independently of the inputs' distances, which is no more than the definition of sensitivity! For instance, the type of nest indicates that this function magnifies the distance of its inputs by a factor of 4, thus nest has sensitivity 4.

Spar uses the type synonym Sen (k :: Nat) a b = ∀ d . Rel d a → Rel (k*d) b that represents functions from a to b with proven sensitivity k. The proof comes from the explicit use of parametricity. Specifically, to create a function of type Sen k a b, we need to implement a *polymorphic function on the distance of its inputs* (observe the ∀ d and the Rel d in the type-signature) whose outputs' distance is scaled by k. Technically, Spar derives sensitivity not only from parametricity of the universal quantification over d, but also from the *parametric use of the indexed abstract data type* Rel d—which can be interpreted as an existentially quantified type from the context. Later in Section 2, we provide a custom relational interpretation for Rel d using a logical relation that exploits parametricity in this (indexed) type.

With this data type, we can provide a proof of sensitivity for our previous examples, shown in lines 6-7 and 13-14 in Figure 1. Observe that the definitions of sen_add42 and sen_nest do not require us to perform any transformations to our functions, meaning that the proof of sensitivity reduces to being able to type-check these expressions.

Parametric polymorphism gives us a simple proof mechanism for sensitivity, but how far can we go with it? How expressive can our programs be? In the following sections, we show that the core calculus of this library is sound, how it can cover some advanced examples similar to the ones found in previous works [Gaboardi et al. 2013; Near et al. 2019; Reed and Pierce 2010], and how to connect these results with the field of differential privacy.

$$r \in \mathbb{R}_{\geqslant 0} \qquad i \in \text{dvar} \qquad j \in \text{lvar} \qquad x \in \text{evar} \qquad n \in \mathbb{R}$$

$$\begin{aligned} e \in \text{expr} ::=\ & x \mid n \mid e + e \mid (e, e) \mid \textbf{fst}\ e \mid \textbf{snd}\ e \mid \lambda x.e \mid e\ @\ e \mid [\,] \mid e :: e \\ & \mid \textbf{case}\ e\ \textbf{of}\ \{[\,].e\}\{(x :: xs).e\} \mid \textbf{vec-rec}\ e\ \{x.xs.r.e\}\ e \mid \Lambda i.e \mid e\langle\rangle \\ & \mid \mathrm{N}\ e \mid e + e \mid (e, e) \mid \textbf{let}\ (x, y) = e\ \textbf{in}\ e \mid [\,] \mid e :: e \\ & \mid \textbf{case}\ e\ \textbf{of}\ \{[\,].e\}\{(x :: xs).e\} \mid \textbf{vec-rec}\ e\ \{x.xs.r.e\}\ e \end{aligned}$$

$$d \in \text{dist} ::= i \mid r \mid d + d \mid d * d \qquad l \in \text{length} ::= j \mid 0 \mid l + 1$$

$$\tau \in \text{type} ::= \sigma \mid \tau \times \tau \mid \vec{\tau}_l \mid \tau \to \tau \mid \text{Rel}\ d\ \sigma \mid \forall i.\tau \qquad \sigma \in \text{relational type} ::= \mathbb{R} \mid \sigma \times \sigma \mid \vec{\sigma}_l$$

$$\Gamma \in \text{tenv} ::= \emptyset \mid \Gamma, x : \tau \qquad \Psi \in \text{denv} ::= \emptyset \mid \Psi, i \qquad C \in \text{cenv} ::= \emptyset \mid d = d \mid C \wedge C$$

Fig. 2. λ_{SPAR} syntax

2 λ_{SPAR}: A Calculus for Distance Tracking

This section introduces λ_{SPAR}, a calculus that annotates types with *distances* and keeps track of them using special operators. Importantly, functions in this calculus can be defined on parametric distances, a feature that is essential for obtaining sensitivity proofs. For clarity, we will present a simplified version of λ_{SPAR} showcasing the main technical ideas; however, any simplifications will be communicated to the reader.

2.1 Syntax

Types. As depicted in Figure 2, our formalism includes a basic numeric type $\mathbb{R}$, pairs $(\_ \times \_)$, functions $(\_ \to \_)$, fixed-length lists or vectors $\vec{(\_)}_l$, quantification over distances $\forall i.(\_)$, and the novel relational type Rel d $(\_)$. As previously stated, the relational type is essentially a regular type annotated with an upper bound on the distance between its inhabitants. To keep distance reasoning as intuitive and straightforward as possible, we introduce a hierarchical structure in the types preventing nesting within relational types and relational types over functions. We overload the use of the word type to refer to both τ and σ when the context allows disambiguation. The reader should also consider the overloaded usage of $(\_ \times \_)$ and $\vec{(\_)}_l$ as the same type constructor.

Distances are represented as terms in a small language at the type level including variables i, non-negative real constants r (i.e., $r \in \mathbb{R}_{\geqslant 0}$), and two operators denoting addition $(\_ + \_)$ and multiplication $(\_ * \_)$. We remark that, without loss of generality, the implementation of λ_{SPAR} in Section 4 encodes distances as type-level natural numbers with their respective operations. Vectors' lengths are represented by type-level natural numbers with variables, this language is similar to that of distance and DFuzz's size terms.

Contrary to previous sensitivity analysis calculi, λ_{SPAR}'s types do not carry *sensitivity* information associated with variables (as in Fuzz [Reed and Pierce 2010]) or values (as in Solo [Abuah et al. 2022]). Instead, the use of relational types to track *distances* allows us to explicitly model bounded metric spaces from which function sensitivity can be *proven* as a uniform continuity property.

Terms. It includes the canonical introduction and elimination forms of an ordinary typed functional programming language: variables; literals and arithmetic operations; pairs and projections; abstractions and applications; indexed lists and pattern discrimination; a vector recursion operator; type abstraction (over distances) and type application. Moreover, the language contains similar introduction and elimination forms of relational terms–marked with a distinguishing color. The only difference between non-relational and relational terms resides in how pairs get eliminated.

T.BigLam
$$\frac{\Psi, i; C; \Gamma \vdash e : \tau \qquad i \notin \mathbf{FV}(C, \Gamma)}{\Psi; C; \Gamma \vdash \Lambda i.e : \forall i.\tau}$$

T.BigApp
$$\frac{\Psi; C; \Gamma \vdash e : \forall i.\tau \qquad \Psi \vdash d}{\Psi; C; \Gamma \vdash e\langle\rangle : \tau[d/i]}$$

T.⊑
$$\frac{\Psi; C; \Gamma \vdash e : \tau_1 \qquad \Psi; C \models \tau_1 \sqsubseteq \tau_2}{\Psi; C; \Gamma \vdash e : \tau_2}$$

T.Num$_\mathrm{R}$
$$\frac{\Psi; C; \Gamma \vdash e : \mathbb{R} \qquad \Psi \vdash d}{\Psi; C; \Gamma \vdash \mathrm{N}\ e : \mathrm{Rel}\ d\ \mathbb{R}}$$

T.Add$_\mathrm{R}$
$$\frac{\Psi; C; \Gamma \vdash e_1 : \mathrm{Rel}\ d_1\ \mathbb{R} \qquad \Psi; C; \Gamma \vdash e_2 : \mathrm{Rel}\ d_2\ \mathbb{R} \qquad i \notin \Psi}{\Psi, i;\ C \wedge i = (d_1 + d_2);\ \Gamma \vdash e_1 + e_2 : \mathrm{Rel}\ i\ \mathbb{R}}$$

T.Pair$_\mathrm{R}$
$$\frac{\Psi; C; \Gamma \vdash e_1 : \mathrm{Rel}\ d_1\ \sigma_1 \qquad \Psi; C; \Gamma \vdash e_2 : \mathrm{Rel}\ d_2\ \sigma_2 \qquad i \notin \Psi}{\Psi, i;\ C \wedge i = (d_1 + d_2);\ \Gamma \vdash (e_1, e_2) : \mathrm{Rel}\ i\ (\sigma_1 \times \sigma_2)}$$

T.Let$_\mathrm{R}$
$$\frac{\begin{array}{c}\Psi; C; \Gamma \vdash e_1 : \mathrm{Rel}\ d\ (\sigma_1 \times \sigma_2) \\ \Psi, i_1, i_2;\ C \wedge d = (i_1 + i_2);\ \Gamma, x : \mathrm{Rel}\ i_1\ \sigma_1, y : \mathrm{Rel}\ i_2\ \sigma_2 \vdash e_2 : \tau \qquad i_1, i_2 \notin \Psi\end{array}}{\Psi; C; \Gamma \vdash \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2 : \tau}$$

Fig. 3. Selected typing rules

While non-relational pairs use projections (terms **fst** and **snd**), the relational ones are eliminated via pattern-matching (term **let** $(x, y) = e$ **in** e).

Typing Rules. Besides the usual environment Γ for term variables, a judgment $\Psi; C; \Gamma \vdash e : \tau$ contains two extra parameters, Ψ and C, conforming to the grammar in Figure 2. In a simplified setting, the set Ψ represents the environment for distance variables. Distinctly from Γ, this environment does not map variables to a type (or *kind*) since all distances are implicitly typed as positive real numbers. Moreover, the set C records the constraints under which typing is obtained. These constraints are carried out to ensure that the interpretation of expressions with free (distance) variables preserves the metric of their respective types. For instance, in the rule T.Pair$_\mathrm{R}$ (see Figure 3), the consequent can be interpreted as follows: regardless of the specific value assigned to the distance of the pair (i.e., when substituting variable i), it must always equal the sum of the distances of its components. This condition adheres to the ℓ_1 metric, also known as the Manhattan distance.

In the full formalization, sets Ψ and C will keep track of the variables and constraints of the lengths of vectors as well those of distances. Technically, a more precise definition of Ψ would be $\Psi \in \mathrm{dlenv} ::= \emptyset \mid (\Psi_\mathrm{d}, \Psi_\mathrm{l})$ with $\Psi_\mathrm{d} \in \mathrm{denv} ::= \emptyset \mid \Psi_\mathrm{d}, i$ and $\Psi_\mathrm{l} \in \mathrm{lenv} ::= \emptyset \mid \Psi_\mathrm{l}, j$; however, we will omit this distinction in favor of readability.

Figure 3 presents the typing rules that capture the formation of (the most simple) relational terms, along with rules for type abstraction, type application, and subtyping—the remaining typing judgments can be found in Section **??** of the accompanying material. Essentially, these rules describe how distances are altered and propagated based on the underlying operation. More importantly, this system captures Fuzz's metric relation [Reed and Pierce 2010], so type safety not only ensures that "well-typed programs don't go wrong" but also guarantees that "they can't go too far." In our case, Rel d σ directly indicates that they are at most d far apart.

$$\Psi; C \models d_1 \dot{\leqslant} d_2 \iff \forall \rho.\ \Psi \subseteq \text{dom}(\rho) \land \rho \models C \Rightarrow [\![d_1]\!]_\rho \leqslant [\![d_2]\!]_\rho$$

$$\textsc{st.Refl}\ \frac{}{\Psi; C \models \tau \sqsubseteq \tau} \qquad \textsc{st.Trans}\ \frac{\Psi; C \models \tau_1 \sqsubseteq \tau_2 \qquad \Psi; C \models \tau_2 \sqsubseteq \tau_3}{\Psi; C \models \tau_1 \sqsubseteq \tau_3} \qquad \textsc{st.Rel}\ \frac{C; \Psi \models d_1 \dot{\leqslant} d_2}{\Psi; C \models \text{Rel}\ d_1\ \sigma \sqsubseteq \text{Rel}\ d_2\ \sigma}$$

Fig. 4. Selected subtyping rules and distance comparison

Rules T.NUM$_\text{R}$, T.ADD$_\text{R}$, and T.PAIR$_\text{R}$ correspond to the operators `lit`, (:+:), and (:★:) introduced in Section 1.1. The rule for relational numbers has the premise $\Psi \vdash d$ preventing d from referring to unbounded variables; when a distance expression d satisfies this condition we say that d is *covered by* Ψ. Rule T.LET$_\text{R}$ allows deconstructing a relational pair under a scoped environment where the distance of the pair components is represented by the fresh variables i_1 and i_2, additionally, the set of constraints is extended requiring that the addition of these variables is equal to the distance of the original pair; this way, it is ensured that the metric for pairs is preserved under elimination.

Our calculus considers vectors with statically known lengths captured by the types $\vec{\tau}_l$ and Rel $d\ \vec{\sigma}_l$. By employing length-indexed vectors, we ensure that values of different lengths are never compared by our logical relation semantics (later explained in Section 3). Enabling lists instead of vectors would have required expressing infinite distances since lists of varying lengths are deemed infinitely apart (see Fuzz [Reed and Pierce 2010]). Furthermore, this choice is advantageous compared to SOLO, as it permits users to define their own recursive functions on vectors—SOLO's lists are opaque in the implementation and recursive functions such as fold are built-in (see Section 5.4 in Abuah et al. [2022]). The introduction and elimination rules for vectors are excluded as they require explicit handling of length and distance variables which pollutes the typing judgments—these details can be found in the accompanying material. Intuitively, given that vectors of fixed lengths can be represented as nested pairs, their introduction, and elimination typing rules impose the same distance constraints as pairs.

The T.BIGLAM rule represents the standard universal quantification over types. However, in λ_{SPAR}, this quantification is limited to distance variables resulting in a system with restricted polymorphism. We shall demonstrate shortly that this is the minimal level of type-abstraction required to derive a proof of sensitivity. Type-level application is captured in rule T.BIGAPP. As customary, this rule requires the desired instantiation to be well-formed, this is, that the distance term for the specialized type should be covered by the distance variable context Ψ.

Lastly, rule T. $\sqsubseteq$ follows the standard procedure for subtyping. The main purpose of the subtyping relation $\sqsubseteq$ (see Figure 4) is to capture the fact that values that are at distance d_1 are also at distance d_2 for $d_1 \leqslant d_2$. To compare distance expressions under $\leqslant$, we provide an interpretation of distance expressions over the domain $\mathbb{R}_{\geqslant 0}$. Concretely, given an assignment $\rho \in \text{dvar} \to \mathbb{R}_{\geqslant 0}$ such that $\text{var}(d) \subseteq \text{dom}(\rho)$, the inductive interpretation $[\![d]\!]_\rho$ is defined as:

$$[\![i]\!]_\rho = \rho(i) \qquad [\![r]\!]_\rho = r \qquad [\![d_1 + d_2]\!]_\rho = [\![d_1]\!]_\rho + [\![d_2]\!]_\rho \qquad [\![d_1 * d_2]\!]_\rho = [\![d_1]\!]_\rho * [\![d_2]\!]_\rho$$

Observe that while the arithmetic operators (_ + _ and _ ∗ _) were symbolic in distance terms, their usage on the right-hand side of the equations refers to the addition and multiplication of real numbers.

With this interpretation, we define an ordering relation over distances $\Psi; C \models d_1 \dot{\leqslant} d_2$ taking into account the constraints they must satisfy (see Figure 4). This relation encapsulates the fact that a term d_1 can be considered less than or equal to term d_2 under the constraints C, only if for every closing assignment ρ satisfying the constraints, the interpretation of d_1 is less than or equal than that of d_2. The subtyping relation relies on such a condition to increase the distance of a relational term (rule ST.REL). In other words, a relational term can only be subtyped if its distance is upgraded (or remains intact).

Lastly, when considering vectors' length variables we have established that sets Ψ and C are extended to track these variables and their constraints. Accordingly, assignments ρ and interpretations $[\![\cdot]\!]_\rho$ are adapted to handle distance and length expressions separately.

2.2 Operational Semantics

We provide a big-step environment-based operational semantics. The grammar below defines the values that an expression can evaluate to:

$$v \in \text{val} \ ::= \ n \mid (v, v) \mid \langle \lambda x.e \mid \gamma \rangle \mid [\,] \mid v :: v \mid \langle \Lambda i.e \mid \gamma \rangle \mid \mathrm{N}\, v \mid (v, v) \mid [\,] \mid v :: v$$

With γ a value environment (also referred to as substitution) mapping variables to values (i.e.,$\gamma \in$ var $\rightarrow$ val). The rules for evaluation are standard (refer to the accompanying material Section ??) with judgment $\gamma \vdash e \Downarrow v$ stating that a configuration with value environment γ and term e evaluates to value v.

3 Formal Guarantees

Soundness for λ_{SPAR} is defined as a *metric preservation* property. Intuitively, the metric preservation property ensures that closing an open term with two distinct but related substitutions will produce related expressions whose distance is bounded.

We adopt the technique of logical relations in which we determine how two well-typed expressions can be considered related at a determined distance. As is customary, we define mutually recursive logical relations $\mathcal{V}[\![\tau]\!]^\rho$, $\mathcal{E}[\![\tau]\!]^\rho$, and $\mathcal{S}[\![\Gamma]\!]^\rho$ relating values, open expressions and typing contexts, respectively (see Figure 5). Moreover, we introduce a new relation $\mathcal{D}[\![\sigma]\!]_d^\rho$ specifying the metric relation between relational values (i.e., values of type Rel d σ).

All of these relations are indexed with an assignment ρ mapping distance variables to non-negative real numbers. Such an assignment is used by the relation $\mathcal{D}[\![\sigma]\!]_d^\rho$ to evaluate the distance term d and use it as an upper bound on the distances of its components. We use $(v_1, v_2) \in \mathcal{D}[\![\sigma]\!]_d^\rho$ to denote that given an assignment ρ, relational values v_1 and v_2 are related at type σ and distance d—similarly for non-relational values, expressions, and environments. The notation $\rho \models C$ states that conditions C holds under assignment ρ, e.g., $\rho \models d = d_1 + d_2$ denotes that $[\![d]\!]_\rho = [\![d_1]\!]_\rho + [\![d_2]\!]_\rho$.

At a high level, our logical relations state:

- Two relational values are related under $\mathcal{D}[\![\sigma]\!]_d^\rho$ if the distance between their operands is less or equal to the value resulting from interpreting the distance term d with the assignment ρ.
- Two values are related under $\mathcal{V}[\![\tau]\!]^\rho$ if they are equivalent (base types) or their components are related accordingly. For example, two functions of type $\tau_1 \rightarrow \tau_2$ are related if they map related inputs $(v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]^\rho$ to related outputs $(\gamma_1 \vdash \lambda x.e_1 \ @ \ v_1, \gamma_2 \vdash \lambda x.e_2 \ @ \ v_2) \in \mathcal{E}[\![\tau_2]\!]^\rho$.
- Two expressions are related under $\mathcal{E}[\![\tau]\!]^\rho$ when both reduce to values that are related at $\mathcal{V}[\![\tau]\!]^\rho$
- Two substitutions are related under $\mathcal{S}[\![\Gamma]\!]^\rho$ when both map all of their variables to related values.

Note that these relations depart from previous work as they do not assign a metric interpretation to all of the types in the calculus. Instead, such interpretation will be restricted only to relational types since these are the ones modeling the metric relation. More precisely, those expressions where none of its components have been annotated with a distance will be assumed to have distance zero—i.e., related under equivalence. For instance, consider the non-relational pairs $(5, 42) : \mathbb{R} \times \mathbb{R}$ and $(1, 42) : \mathbb{R} \times \mathbb{R}$, then $((5, 42), (1, 42)) \notin \mathcal{V}[\![\mathbb{R} \times \mathbb{R}]\!]^\rho$ since $5 \not\equiv 1$. However, non-relational

$$
\begin{aligned}
\mathcal{D}[\![\mathbb{R}]\!]^{\rho}_{d} &= \{(\mathrm{N}\ n_1, \mathrm{N}\ n_2) \mid |n_1 - n_2| \leqslant [\![d]\!]_{\rho.\mathrm{d}}\} \\
\mathcal{D}[\![\sigma_1 \times \sigma_2]\!]^{\rho}_{d} &= \{((v_{11}, v_{21}), (v_{12}, v_{22})) \mid \exists d_1, d_2.\ \rho \models d = d_1 + d_2 \\
&\qquad \wedge (v_{11}, v_{12}) \in \mathcal{D}[\![\sigma_1]\!]^{\rho}_{d_1} \wedge (v_{21}, v_{22}) \in \mathcal{D}[\![\sigma_2]\!]^{\rho}_{d_2}\} \\
\mathcal{D}[\![\vec{\sigma}_0]\!]^{\rho}_{d} &= \{([\,], [\,])\} \\
\mathcal{D}[\![\vec{\sigma}_{(l+1)}]\!]^{\rho}_{d} &= \{(v_{11} :: v_{21}, v_{12} :: v_{22}) \mid \exists d_1, d_2.\ \rho \models d = d_1 + d_2 \\
&\qquad \wedge (v_{11}, v_{12}) \in \mathcal{D}[\![\sigma]\!]^{\rho}_{d_1} \wedge (v_{21}, v_{22}) \in \mathcal{D}[\![\vec{\sigma}_l]\!]^{\rho}_{d_2}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}[\![\mathbb{R}]\!]^{\rho} &= \{(v_1, v_2) \mid v_1 \equiv v_2\} \\
\mathcal{V}[\![\tau_1 \times \tau_2]\!]^{\rho} &= \{((v_{11}, v_{21}), (v_{12}, v_{22})) \mid (v_{11}, v_{12}) \in \mathcal{V}[\![\tau_1]\!]^{\rho} \wedge (v_{21}, v_{22}) \in \mathcal{V}[\![\tau_2]\!]^{\rho}\} \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!]^{\rho} &= \{(\langle \lambda x.e_1 \mid \gamma_1 \rangle, \langle \lambda x.e_2 \mid \gamma_2 \rangle) \mid \forall v_1, v_2.\ (v_1, v_2) \in \mathcal{V}[\![\tau_1]\!]^{\rho} \\
&\qquad \Rightarrow (\gamma_1 \vdash \lambda x.e_1 \mathbin{@} v_1, \gamma_2 \vdash \lambda x.e_2 \mathbin{@} v_2) \in \mathcal{E}[\![\tau_2]\!]^{\rho}\} \\
\mathcal{V}[\![\vec{\tau}_0]\!]^{\rho} &= \{([\,], [\,])\} \\
\mathcal{V}[\![\vec{\tau}_{(l+1)}]\!]^{\rho} &= \{(v_{11} :: v_{21}, v_{12} :: v_{22}) \mid (v_{11}, v_{12}) \in \mathcal{V}[\![\tau]\!]^{\rho} \wedge (v_{21}, v_{22}) \in \mathcal{V}[\![\vec{\tau}_l]\!]^{\rho}\} \\
\mathcal{V}[\![\mathrm{Rel}\ d\ \sigma]\!]^{\rho} &= \mathcal{D}[\![\sigma]\!]^{\rho}_{d} \\
\mathcal{V}[\![\forall i.\tau]\!]^{\rho} &= \{(\langle \Lambda i.e_1 \mid \gamma_1 \rangle, \langle \Lambda i.e_2 \mid \gamma_2 \rangle) \mid \forall r \in \mathbb{R}_{\geqslant 0}.(\gamma_1 \vdash e_1, \gamma_2 \vdash e_2) \in \mathcal{E}[\![\tau]\!]^{\rho[i\, :=\, r]}\}
\end{aligned}
$$

$$
\mathcal{E}[\![\tau]\!]^{\rho} = \{(\gamma_1 \vdash e_1, \gamma_2 \vdash e_2) \mid \forall v_1, v_2.\gamma_1 \vdash e_1 \Downarrow v_1 \wedge \gamma_2 \vdash e_2 \Downarrow v_2 \Rightarrow (v_1, v_2) \in \mathcal{V}[\![\tau]\!]^{\rho}\}
$$

$$
\begin{aligned}
\mathcal{S}[\![\Gamma]\!]^{\rho} &= \{(\gamma_1, \gamma_2) \mid \mathrm{dom}(\gamma_1) \equiv \mathrm{dom}(\gamma_2) \equiv \mathrm{dom}(\Gamma) \wedge \forall (x : \tau).\ x \in \mathrm{dom}(\Gamma) \\
&\qquad \Rightarrow (\gamma_1(x), \gamma_2(x)) \in \mathcal{V}[\![\tau]\!]^{\rho}\}
\end{aligned}
$$

Fig. 5. Mutually-recursive logical relations

pairs with relational components can be related in a metric relation; this is: $((\mathrm{N}\ 5, 42), (\mathrm{N}\ 1, 42)) \in \mathcal{V}[\![\mathrm{Rel}\ 7\ \mathbb{R} \times \mathbb{R}]\!]^{\rho}$ since $|5 - 1| \leqslant 7$ and $42 \equiv 42$

With these logical relations, we establish the notion of type soundness via the fundamental lemma of logical relations (i.e., well-typed terms are related to themselves), which also corresponds to the metric preservation theorem [Reed and Pierce 2010].

Theorem 3.1 (Metric preservation). *Let a well-typed expression $\Psi; C; \Gamma \vdash e : \tau$ be given. For any ρ for which $\Psi \subseteq \mathrm{dom}(\rho)$ and $\rho \models C$; suppose γ_1, γ_2 are two substitutions for Γ such that $(\gamma_1, \gamma_2) \in \mathcal{S}[\![\Gamma]\!]^{\rho}$, then we have $(\gamma_1 \vdash e, \gamma_2 \vdash e) \in \mathcal{E}[\![\tau]\!]^{\rho}$.*

Proof. By induction on the typing derivations of e. The cases for non-relational terms are standard. A full-blown proof of all relational and non-relational terms can be found in the accompanying material.

□

3.1 Sensitivity by Parametricity

In this section, we explore the connection between parametricity and function sensitivity. In a nutshell, we show that by assigning a relational interpretation for λ_{SPAR}'s types, a proof of function sensitivity can be derived from such an interpretation given that λ_{SPAR} is parametric on distances.

The concept of *parametricity* [Reynolds 1983; Wadler 1989] refers to a generic property of programming languages supporting parametric polymorphism. This property captures the intuition that every instance of a polymorphic function should behave the same. Wadler's key observation is that by interpreting types as relations, instead of sets, one can produce useful theorems about programs directly from their types. For instance, when considering any polymorphic list-transformation function $r : \forall A.[A] \to [A]$, one can use parametricity to obtain the (free) theorem: $\forall\ f\ xs.\ \text{map}\ f\ (r\ xs) \equiv r\ (\text{map}\ f\ xs)$. This theorem tells us insightful information about the way r interacts with its input: it works on the structure of the input list in a way that is independent of the elements of the list. Formally, parametricity states that any closed term e of type τ is related to itself under a relational interpretation of its types, this is:

$$\Psi; C; \emptyset \vdash e : \tau \Rightarrow (e, e) \in [\![\tau]\!] \tag{1}$$

with $[\![\tau]\!] \in \tau \times \tau$ denoting the relational interpretation for τ. In the previous section, we defined a set of logical relations providing a metric interpretation to our types. Then, if we define $[\![\tau]\!]$ as $\mathcal{E}[\![\tau]\!]^{\rho}$, the parametricity lemma corresponds to the fundamental lemma of logical relations—i.e, metric preservation Theorem 3.1—where the substitutions γ_1 and γ_2 are empty; thus trivially related.

With this in mind, we argue that given a distance-parametric closed function $\Lambda i.f$, with type $\forall i.\text{Rel}\ i\ \sigma_1 \to \text{Rel}\ (k * i)\ \sigma_2$, we can prove that it satisfies k-sensitivity via metric preservation. Concretely, when we consider functions such as: $\Psi; C; \emptyset \vdash \Lambda i.f : \forall i.(\text{Rel}\ i\ \sigma_1 \to \text{Rel}\ (k * i)\ \sigma_2)$, the previous statement describes a sensitivity soundness theorem of the form:

Theorem 3.2 (Sensitivity soundness). *Given a polymorphic function f with distance variable i, it holds that*

$$\Psi; C; \emptyset \vdash \Lambda i.f : \forall i.(\mathit{Rel}\ i\ \sigma_1 \to \mathit{Rel}\ (k * i)\ \sigma_2) \Rightarrow f \textit{ is } k\textit{-sensitive}$$

Proof. Recall Definition 1.2 stating that a function f is k-sensitive if the distance between its outputs is bounded by k times the distance between its inputs—for whatever distance they might have. In terms of our logical relations, k-sensitivity for closed functions can be expressed as follows:

$$\begin{aligned} &\forall\ \Psi, C, \rho, v_1, v_2, r \in \mathbb{R}_{\geqslant 0}.\ \Psi \subseteq \text{dom}(\rho) \wedge \rho \models C \wedge (v_1, v_2) \in \mathcal{V}[\![\text{Rel}\ i\ \sigma_1]\!]^{\rho[i\ :=\ r]} \\ &\quad \Rightarrow (\cdot \vdash f\ @\ v_1, \cdot \vdash f\ @\ v_2) \in \mathcal{E}[\![\text{Rel}\ (k * i)\ \sigma_2]\!]^{\rho[i\ :=\ r]} \end{aligned} \tag{2}$$

By parametricity (i.e, metric preservation) over $\Lambda i.f$ we know:

$$\forall\ \Psi, C, \rho.\ \Psi \subseteq \text{dom}(\rho) \wedge \rho \models C \Rightarrow (\cdot \vdash \Lambda i.f, \cdot \vdash \Lambda i.f) \in \mathcal{E}[\![\forall i.(\text{Rel}\ i\ \sigma_1 \to \text{Rel}\ (k * i)\ \sigma_2)]\!]^{\rho} \tag{3}$$

Now, let's expand on the conclusion of this implication:

$$\begin{aligned} &(\cdot \vdash \Lambda i.f, \cdot \vdash \Lambda i.f) \in \mathcal{E}[\![\forall i.(\text{Rel}\ i\ \sigma_1 \to \text{Rel}\ (k * i)\ \sigma_2)]\!]^{\rho} \\ \equiv&\langle \textit{By definition of } \mathcal{E}[\![\_]\!]^{\rho}_{\emptyset} \rangle \\ &\forall F_1, F_2. \cdot \vdash \Lambda i.f \Downarrow F_1 \wedge \cdot \vdash \Lambda i.f \Downarrow F_2 \Rightarrow (F_1, F_2) \in \mathcal{V}[\![\forall i.\text{Rel}\ i\ \sigma_1 \to \text{Rel}\ (k * i)\ \sigma_2]\!]^{\rho} \\ \equiv&\langle \textit{By determinism of } (\_ \Downarrow \_) \textit{ with } \cdot \vdash \Lambda i.f \Downarrow \langle \Lambda i.f \mid \cdot \rangle \rangle \\ &(\langle \Lambda i.f \mid \cdot \rangle, \langle \Lambda i.f \mid \cdot \rangle) \in \mathcal{V}[\![\forall i.(\text{Rel}\ i\ \sigma_1 \to \text{Rel}\ (k * i)\ \sigma_2)]\!]^{\rho} \\ \equiv&\langle \textit{By definition of } \mathcal{V}[\![\_]\!]^{\rho} \textit{ at } \forall i.(\_) \rangle \\ &\forall r \in \mathbb{R}_{\geqslant 0}.(\cdot \vdash f, \cdot \vdash f) \in \mathcal{E}[\![\text{Rel}\ i\ \sigma_1 \to \text{Rel}\ (k * i)\ \sigma_2]\!]^{\rho[i\ :=\ r]} \end{aligned}$$

$$
\begin{aligned}
&\equiv\langle \textit{By definition of } \mathcal{E}[\![\_]\!]^{\rho}_{\emptyset} \rangle \\
&\quad \forall r \in \mathbb{R}_{\geqslant 0}, f_1, f_2. \cdot \vdash f \Downarrow f_1 \land \cdot \vdash f \Downarrow f_2 \Rightarrow (f_1, f_2) \in \mathcal{V}[\![\mathrm{Rel}\ i\ \sigma_1 \to \mathrm{Rel}\ (k * i)\ \sigma_2]\!]^{\rho[i\,:=\,r]} \\
&\equiv\langle \textit{By determinism of } (\_ \Downarrow \_) \textit{ with } \cdot \vdash f \Downarrow \langle \lambda x.e \mid \cdot \rangle \rangle \\
&\quad \forall r \in \mathbb{R}_{\geqslant 0}. (\langle \lambda x.e \mid \cdot \rangle, \langle \lambda x.e \mid \cdot \rangle) \in \mathcal{V}[\![\mathrm{Rel}\ i\ \sigma_1 \to \mathrm{Rel}\ (k * i)\ \sigma_2]\!]^{\rho[i\,:=\,r]} \\
&\equiv\langle \textit{By definition of } \mathcal{V}[\![\_]\!]^{\rho} \textit{ at } (\_ \to \_) \rangle \\
&\quad \forall v_1, v_2, r \in \mathbb{R}_{\geqslant 0}.\ (v_1, v_2) \in \mathcal{V}[\![\mathrm{Rel}\ i\ \sigma_1]\!]^{\rho[i\,:=\,r]} \\
&\qquad \Rightarrow (\cdot \vdash \lambda x.e\ @\ v_1, \cdot \vdash \lambda x.e\ @\ v_2) \in \mathcal{E}[\![\mathrm{Rel}\ (k * i)\ \sigma_2]\!]^{\rho[i\,:=\,r]} \\
&\equiv\langle \textit{Since } \cdot \vdash f \Downarrow \langle \lambda x.e \mid \cdot \rangle \textit{ and } \cdot \vdash \lambda x.e \Downarrow \langle \lambda x.e \mid \cdot \rangle \rangle \\
&\quad \forall v_1, v_2, r \in \mathbb{R}_{\geqslant 0}.\ (v_1, v_2) \in \mathcal{V}[\![\mathrm{Rel}\ i\ \sigma_1]\!]^{\rho[i\,:=\,r]} \\
&\qquad \Rightarrow (\cdot \vdash f\ @\ v_1, \cdot \vdash f\ @\ v_2) \in \mathcal{E}[\![\mathrm{Rel}\ (k * i)\ \sigma_2]\!]^{\rho[i\,:=\,r]} \qquad (4)
\end{aligned}
$$

When rewriting (3) with (4) we obtain:

$$
\begin{aligned}
&\forall\ \Psi, C, \rho, v_1, v_2, r \in \mathbb{R}_{\geqslant 0}.\ \Psi \subseteq \mathrm{dom}(\rho) \land \rho \models C \land (v_1, v_2) \in \mathcal{V}[\![\mathrm{Rel}\ i\ \sigma_1]\!]^{\rho[i\,:=\,r]} \\
&\qquad \Rightarrow (\cdot \vdash f\ @\ v_1, \cdot \vdash f\ @\ v_2) \in \mathcal{E}[\![\mathrm{Rel}\ (k * i)\ \sigma_2]\!]^{\rho[i\,:=\,r]}
\end{aligned}
$$

Which is no less than the definition (2) of a k-sensitive function expressed in terms of logical relations. □

Our reasoning showcases the usefulness of parametricity as a technique for obtaining insightful theorems about parametric functions. In particular, we have shown that by giving a metric interpretation to λ_{SPAR}' types, sensitivity soundness (Theorem 3.2) follows directly from the metric preservation Theorem 3.1. In a nutshell, we have shown that tracking programs' distances and allowing parametricity on distance terms are sufficient conditions to prove sensitivity soundness.

It's essential to recognize that our findings concerning function sensitivity are grounded in a specific interpretation of parametricity that deviates from its conventional understanding. As previously presented, the traditional form of parametricity involves the re-interpretation of universal quantifiers as relational properties that are later instantiated with specific relations, often functions, leading to the discovery of novel theorems or facts. In our approach, we do not rely solely on universal quantification over distances but also on the the parametric use of the indexed abstract data type `Rel`. Consequently, we don't strictly adhere to the classical notion of parametricity that associates universal quantifiers with relation properties; instead, we leverage a relational interpretation of `Rel d` within a logical relation, which allows us to establish function sensitivity capitalizing on the parametric characteristics associated with the indexed type `Rel d`. While this may appear restrictive since we cannot derive generic properties in the vein of "theorems for free", it's important to clarify that our primary goal is not to uncover broad insights into how our functions interact with their inputs in a general sense. Instead, our focus is on understanding how our functions impact the *distance* of their inputs, a concept directly tied to function sensitivity. This emphasis prompts an intriguing question: what novel properties related to function distances can be derived from this framework? Beyond straightforward theorems, such as proving that a zero-sensitive function is constant, it remains unclear which other interesting properties may emerge— an interesting direction for future work.

4 λ_{SPAR} as a Library

In this section, we present SPAR, the software library that realizes the calculus from Section 2. As captured by our main result in Section 3.1, the library relies on *parametrically polymorphic*

```
data Rel (d :: Nat) a

   -- Numbers
lit :: Int → Rel d Int
(:+:) :: Rel d₁ Int → Rel d₂ Int → Rel (d₁+d₂) Int

   -- Pairs
(:★:) :: Rel dₐ a → Rel d_b b → Rel (dₐ+d_b) (a, b)

   -- Sub-typing
up :: Rel d a → Rel (d+c) a

   -- Vectors
Nil  :: Rel d (Vec 0 a)
(:>) :: Rel dₐ a → Rel dᵥ (Vec l a)
       → Rel (dₐ+dᵥ) (Vec (l+1) a)

   -- Sensitivity type synonym
type Sen (k :: Nat) a b =
     ∀ (d :: Nat) . Rel d a → Rel (k*d) b

   -- Using sensitivity functions
run :: (Rf a, Rf b) ⇒ Sen k a b → (a → b)
```

Fig. 6. Spar's API

functions to compute sensitivity for user-defined functions. While λ_{SPAR} introduces special terms for type-level abstraction and application; as well as pattern matching and recursion over vectors (i.e., **vec-rec** e $\{x.xs.r.e\}$ e and **vec-rec** e $\{x.xs.r.e\}$ e whose typing rules can be found in Section **??** of the accompanying material), Spar will outsource these features from the host language, Haskell. Besides polymorphism, our implementation uses some of the advanced features of Haskell's type system to facilitate the usage of the library—we defer explanations about such features to Section 5.

Spar is a domain-specific language[Fowler 2010] (DSL) embedded in Haskell, i.e., the language constructs are given as a library of ordinary Haskell functions. Not all language constructs need to be part of the Spar core language. One of the great benefits of an embedded language is the ability to use the host language to create programs. For instance, it is possible to leverage any of Haskell's higher-order functions to compactly describe functions and prove their sensitivity.

The full API of Spar is presented in Figure 6. Type `Rel` is indexed by a type-level natural number. Unlike our calculus, Spar only considers natural numbers as distances. This limitation mainly arises from Haskell's type system not being powerful enough to represent real numbers, and their operations, at the type level. Primitives `lit` and (:+:) correspond to the relational numbers and addition from our calculus. Similarly, primitive (:★:), `Nil`, and (:>) correspond to relational pairs, and relational vectors, respectively. As we mentioned in Section 2, the construction of pairs (:★:) and vectors (:>) work similarly when tracking distances at the type-level, i.e., d_a+d_b and d_a+d_v. Primitive up encodes the sub-typing rule from our calculus. The type synonym `Sen` directly refers to functions from `a` to `b` with proven sensitivity `k`, where the proof comes from the explicit use of parametric polymorphism. Finally, function `run` takes a function with proven sensitivity `k` and obtains the underlying function so that it can be executed by the host language—we defer the explanation about how to use it and its constraints until the next section. In the remainder of this section, we will see some examples showing how Spar can support reasoning about the sensitivity of complex higher-order recursive functions.

4.1 Discovering Function's Sensitivity

We start with an example where developers can write functions, and by doing so, *discover* and *provide a proof* of their sensitivity. Obtaining a proof is merely convincing the type-checker. We focus on analyzing the sensitivity of the well-known map function:

```
map :: (a → b) → [a] → [b]
```

which takes a function from `a` to `b` (`a → b`), a list of elements of type `a` (`[a]`), and applies the function to each element to obtain a list of elements of type `b` (`[b]`). What is the sensitivity of the `map` function? To answer that question, we proceed to implement `map` using Spar. We assume that the argument of the `map` function is of sensitivity `k`, that is, `smap :: Sen k a b → ...` (with

`...` denoting code that is irrelevant to the point being made.) Since we need to reason about how the output changes with respect to the input, we make `smap` take and return relational values: `smap :: Sen k a b → Rel d (Vec l a) → ...` What should be the distance of the relational output? At this point, we can simply write the function and assign a type variable waiting for the type system to guide the implementation.

```
smap :: Sen k a b → Rel d (Vec l a) → Rel x (Vec l b)
smap f Nil = Nil
smap f (x :> xs) = f x :> smap f xs
```

Running this code through the type checker prompts the error message

```
 Could not deduce: ((k * da) + dv) ~ x arising from a use of ‘:>’
 from the context: (Vec l a ~ Vec (l1 + 1) a1, (da + dv) ~ x)
```

Observe that the information provided by the type system is helping us figure out the shape of `x`, i.e., it should be, at least, `k*da+dv`, where we know that `da+dv` is equal (unifies) to `x`. With such information, we can propose that `x` increases its value to `k*da+k*dv`, which is equivalent to `k*(da+dv)` or simply `k*d`, this is: `smap :: Sen k a b → Rel d (Vec l a) → Rel (k*d) (Vec l b)`. With this type-signature `smap` type-checks! and we can proceed to use the `Sen` type-synonym `smap :: Sen k a b → Sen k (Vec l a) (Vec l b)` indicating that `smap f` has sensitivity `k` given that `f` has sensitivity `k`. As shown by this example, Spar leverages Haskell's type-system to provide some restricted support for sensitivity inference. Sensitivity inference for fully unannotated definitions challenges the limits of Haskell's type system (e.g., by producing errors involving untouchable variables). As such, the best way we have found to leverage Haskell's type system for inference is as presented in this section, where partial type signatures are given and type variables get introduced for distances were we require help from the type system. To the best of our knowledge, previous sensitivity languages [Abuah et al. 2022; Gaboardi et al. 2013; Near et al. 2019] do not specifically address sensitivity inference. While D'Antoni *et al.* [D'Antoni et al. 2013] present an automatic analysis that infers and verifies the sensitivity annotations of Fuzz's programs, supporting this approach requires the modification of the host language's type checker.

Using this methodology Spar can be used to prove the sensitivity of many list-related functions such as summation, concatenation, folding, and zipping/unzipping operations providing users with useful information (see their typing in Section **??** of the accompanying material). For instance, from the type-signature of `sfoldl :: Sen 1 (a, b) b → Sen 1 (b, Vec l a) b`, we can assert that the left-folding of a vector with a function of sensitivity `1` has also sensitivity `1`.

When folding with a function of sensitivity `k`, Spar uses the length of the vector to determine the sensitivity of the output:

```
sGFoldr :: (1 ⩽ k) ⇒ Rel 0 b → Sen k (a, b) b → Sen (l*(k^l)) (Vec l a) b
```

For simplicity, we consider the base case as a constant value[1]. When considering vectors whose distance metric is determined by the ℓ_∞ norm instead, we will get that:

sGFoldr$_\infty$ `:: (1 ⩽ k) ⇒ Rel 0 b → Sen k (a & b) b → Sen (k^l) (Vec` ℓ_∞ `l a) b`

where the sensitivity of the output is exponentially increased by the length of the vector `l`. Spar can also express generic versions of `foldr1` as well as left folds. Calculating sensitivities based on the length of vectors cannot be expressed in Fuzz-like systems or Solo unless lightweight linear dependent types are considered, e.g., as done by DFuzz [Gaboardi et al. 2013].

Extending Spar with primitives to handle pairs and vectors under the ℓ_∞ norm is straightforward. We can add the following constructors to the API:

[1]If the base case is a relation value, i.e., `Rel n b`, then the sensitivity of the results becomes `Sen (l*((k+n)^l)) (Vec l a) b`

-- Pairs under the ℓ_∞ norm
(:&:) :: Rel d_a a → Rel d_b b → Rel (Max d_a d_b) (a & b)
-- Vectors under the ℓ_∞ norm
Nil$_\infty$ :: Rel 0 (Vec ℓ_∞ 0 a)
(:>$_\infty$) :: Rel d_a a → Rel d_v (Vec ℓ_∞ l a) → Rel (Max d_a d_v) (Vec ℓ_∞ (l+1) a)

where `Max` is a type-level function that returns the maximum of two natural numbers, `Norm` is a datatype that can be either ℓ_1 or ℓ_∞, there is a new type of tuples (&), and the vector type `Vec` is enriched with the norm information, i.e., **data** `Vec (nF :: Norm) (l :: Nat) a`. For simplicity and conciseness, we continue referring to the vectors under the ℓ_1 norm `Vec` ℓ_1 `l a` simply as `Vec l a`.

To conclude this section, we consider the case of obtaining the sensitivity of an insertion sort operation. Sort's implementation requires the conditional swap, or *cswp*, operation introduced by Fuzz: *"... it takes in a pair, and outputs the same pair, swapped if necessary so that the first component is no larger than the second one."* [Reed and Pierce 2010]. This primitive outsources from the program the ability to compare the elements of a pair.

Similar to Fuzz, our calculus is not powerful enough to encode `cswp` and we need to consider it as an add-on primitive with type `Sen 1 (Int, Int) (Int, Int)`. The implementation of `cswp` requires comparing elements of type `Rel d Int` to determine when to flip the pair. With `cswp` in place, we can implement insert sort as shown in Figure **??** in the accompanying material, and verify that it is a 1 sensitive function with type `Sen 1 (Vec l Int) (Vec l Int)`, as expected.

5 Implementation

In this section, we describe some of our design decisions and insights gained while realizing our calculus in Haskell. We expect that these insights could help in implementing Spar in other programming languages.

5.1 Handling Type-Level Natural Numbers

To compute distances, Spar *relies heavily on the type system's ability to manipulate, either concretely or symbolically, type-level natural number expressions and decide their equality.* We utilize the Glasgow Haskell Compiler (GHC) and some plugin extensions for injecting new axioms into GHC's type equality relation while not breaking type safety.

Notably, Spar makes use of the GHC plugin called `TypeLits.Normalize`[2] to expand the compiler's capabilities for handling equality involving type-level natural number expressions. This plugin equips the compiler with syntactic equality checks, enabling it to demonstrate equivalences between types such as `1+n` and `n+1`.

Additionally, there are a few instances—four exactly—where GHC requires additional assistance in understanding the interactions between the associativity of type-level operators like + and unification. In these cases, we make use of Haskell's `Constraint` library[3], in conjunction with GHC's `ConstraintKinds` extension[4]. This combination allows us to perform type-level programming to manipulate type constraints effectively. By employing this library, we can derive arithmetic facts from an expression's constraints, and these derivations can in turn be used to provide the necessary information for the type system to resolve type unifications.

When it comes to handling subtractions, the type system encounters challenges in resolving associations between operands. This limitation becomes evident in the typing of `splitVec` primitive (and its auxiliary functions), where the type system must ensure that the partitions of the original

[2]https://hackage.haskell.org/package/ghc-typelits-natnormalise
[3]https://hackage.haskell.org/package/constraints
[4]https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/constraint_kind.html

vector contains all the elements. In such a scenario, we turn to the Thoralf plugin [Otwani and Eisenberg 2018], which translates unification constraints into queries directed at an external SMT solver. For a more detailed discussion of the applications and implications of these extensions refer to the accompanying material Section ??.

5.2 Spar as an Embedded DSL

Spar is a *deeply embedded* DSL when it comes to implementing introduction and elimination constructs from our calculus, but a *shallowly embedded* DSL when it comes to operations among relational values [Svenningsson and Axelsson 2015]. The deeply embedded components of our DSL do not perform any actual computation; instead, they result in a data structure representing the relational values being constructed. When it comes to elimination rules, we utilize pattern-synonyms [Pickering et al. 2016] which allows us to expose specific constructors and hide others. For instance, the pattern-synonym (:⋆:) enables constructing and eliminating (i.e., pattern-matching) pairs—thus enabling conveniently writing programs like λ(`d1` :⋆: `d2`) $\rightarrow$ `d1` :+: `d1` :+: `d2`. Importantly, we do not enable pattern-matching on relational numbers. If we did, programmers could write functions that break Spar's guarantees.

To execute Spar's functions, we need to call `run` :: (`Rf a`, `Rf b`) $\Rightarrow$ Sen `k a b` $\rightarrow$ `a` $\rightarrow$ `b`, which involves taking a value of type `a` in the host semantics and *reifying* it into Spar, applying the function given as an argument, and *reflecting* the result back into the host language. (The type class `Rf` ensures that values can be reified and reflected.) Function `run` should be considered carefully as it *forgets* about distances among relational values. Luckily, a safe workflow can be enforced by the use of Safe Haskell [Terei et al. 2012]—more details about the internal representation of Spar and its correct usage are described in the accompanying material Sections ?? and ??.

6 Beyond Sensitivity

This section presents evidence of the practical application of Spar's proof of sensitivity and distance tracking in the deployment of differentially private user-defined algorithms on PINQ-like systems (i.e., LINQ-inspired query languages with DP-primitives) [Ebadi and Sands 2017; Lobo-Vesga et al. 2020; McSherry 2010; Proserpio et al. 2014].

To lay the groundwork, we commence with a review of the fundamental concepts of differential privacy. At its core, differential privacy aims to safeguard individual-level information by ensuring that the outcomes of computations remain statistically indistinguishable, regardless of whether a particular individual's data is included or excluded. This means that the results should not be significantly affected or reveal sensitive details by the presence or absence of any specific individual. Concretely, consider the scenario where private data is collected and stored in a database consisting of rows with the same structure, where each individual's data is contained in a single row. This type of database will be denoted as **db**. For simplicity, we focus on queries that produce real-valued results. In formal terms, differential privacy is defined as a property of randomized queries $\tilde{q}$:

Definition 6.1 (Differential Privacy [Dwork et al. 2006]). *A randomized function $\tilde{q} : \mathbf{db} \rightarrow \mathbb{R}$ is ϵ-differentially private if, for all possible sets of outputs $S \subseteq \mathbb{R}$ and for any* adjacent *datasets in* **db**, *written $D_1 \sim_1 D_2$, then $\Pr[\tilde{q}(D_1) \in S] \leqslant e^{\epsilon} \cdot \Pr[\tilde{q}(D_2) \in S]$.*

With $D_1 \sim_1 D_2 \equiv d_{\mathbf{db}}(D_1, D_2) \leqslant 1$, and $d_{\mathbf{db}}(\cdot)$ the distance metric between datasets. Intuitively, two adjacent datasets differ on one record at most [Kifer and Machanavajjhala 2011]. As such, Definition 6.1 captures the essence that the presence or absence of an individual in the dataset should have a limited and controlled effect on the distribution of outputs produced by $\tilde{q}$. The degree of this impact is determined by the privacy parameter ϵ, often referred to as the *privacy cost*

```
set       :: Set∅ a → Rel d (Set a)
size      :: Rel d (Set a) → Rel d Int
setMap    :: (a → b) → Rel d (Set a) → Rel d (Set b)
setFilter :: (a → Bool) → Rel d (Set a) → Rel d (Set a)
setSplit  :: (a → Bool) → Rel d (Set a) → Rel d (Set a, Set a)
∪, ∩, \   :: Ord a ⇒ Rel d (Set a, Set a) → Rel d (Set a)
```

(a) Set operators

```
newtype PM (χ :: Nat) a
return  :: a → PM 0 a
(≫=)    :: PM χ1 a → (a → PM χ2 b)
           → PM (χ1+χ2) b
```

(b) Privacy monad

Fig. 7. Spar's extensions

associated with executing $\tilde{q}$. A larger value of ϵ implies a higher potential disclosure of information about individuals through the query result.

A standard way to satisfy this property is to take k-sensitive functions and convert them into ϵ-DP queries via the Laplace mechanism.

Theorem 6.1 (Laplace mechanism [Dwork et al. 2006]). *Let $f : \mathbf{db} \to \mathbb{R}$ be a deterministic k-sensitive function, and let $\tilde{q} : \mathbf{db} \to \mathbb{R}$ be the randomized function $\tilde{q}(D) = f(D) + Y$, where Y is a random variable drawn from $\mathcal{L}\left(0, {}^{k}/_{\epsilon}\right)$. Then $\tilde{q}$ is ϵ-differentially private.*

Observe that the *magnitude* of the added noise is inversely related to the privacy cost and directly related to the sensitivity of the query. In other words, the stronger the privacy requirement (small values of ϵ) and the more sensitive the query (larger values of k), the more noise needs to be added to f in order to preserve the privacy of the individuals.

Until now, our focus has primarily revolved around *aggregations*, which are functions that combine multiple records or data points to generate summarized results (i.e., real-valued outputs), like counting, adding, or averaging. However, in the domain of databases, there are other types of operations called *transformations* that modify the structure or content of a dataset without necessarily summarizing it. Transformations, such as filtering, joining, and splitting, focus on altering the dataset itself rather than providing a summary. Nevertheless, like aggregations, it is crucial to quantify the sensitivity of these transformations in order to ensure the proper protection of individuals' privacy within the dataset.

Definition 6.2 (Stable transformations [McSherry 2010]). *A transformation $t : \mathbf{db} \to \mathbf{db}$ is c-stable if for any two datasets D_1, D_2, then $d_{\mathbf{db}}(t(D_1), t(D_2)) \leqslant c \cdot d_{\mathbf{db}}(D_1, D_2)$*

Stable transformations play a critical role as they ensure the propagation of differential privacy from their outputs back to their inputs, with the privacy guarantees scaled by their stability constant. This property is formalized as:

Theorem 6.2 ([McSherry 2010]). *Let $\tilde{q}$ be ϵ-differentially private, and let t be an arbitrary c-stable transformation. Then, the composition of these operations $\tilde{q} \circ t$ satisfies $(c \cdot \epsilon)$-differential privacy.*

In other words, the level of privacy protection in the transformed data is directly related to the stability constant.

6.1 A Generic Laplace Mechanism

Now that we have established a solid foundation of the fundamental concepts of differential privacy, we can dive into the utilization and extension of Spar to support differentially private queries via the Laplace mechanism. To start, we enhance Spar's with support for (relational) finite sets and their canonical primitive operations, similar to those presented in Fuzz (see Figure 7a). It is well known that introducing primitives like `setMap`, `setFilter`, and `setSplit` may lead to non-termination when applied to diverging argument functions. Note that Spar guarantees apply only to terminating functions. To ensure termination, a dynamic timeout mechanism similar to Fuzz's can be utilized, requiring `setMap` to take an additional argument for a default element when termination fails. This detail is omitted for brevity.

These extensions serve as the fundamental building blocks for representing datasets and their transformations. As presented in Fuzz and subsequent systems, the distance between sets is determined by the cardinality of their symmetric difference. In other words, two sets are considered to be at a distance d if they differ in at most d elements. Hence, the expression `db :: Rel 1 (Set a)` encodes the universe of datasets whose rows are of type `a` and differ in only one row—thus capturing the adjacency concept from Definition 6.1.

Subsequently, we introduce the primitive `laplace` implementing a generic version of the Laplace mechanism with the following type-signature (we refer the reader to Section **??** of the accompanying material for its full implementation):

```
laplace :: ∀ k c a b . (Rf a, Rf b, KnownNat k, KnownNat c) ⇒ Epsilon → Sen k (Set b) Int
           → Sen c (Set∅ a) (Set b) → Rel 1 (Set∅ a) → IO Double
laplace ϵ senA senT (RSet∅ db) = do...
```

We consider this primitive a general version of the Laplace mechanism since it also accounts for table transformations. This function accepts several arguments: the privacy parameter ϵ, an aggregating k-sensitive function called `senA`, a c-stable transformation denoted as `senT`, and the *original dataset* (`RSet∅ db`). Its purpose is to create a randomized version of the composite function `senA . senT` of type `Sen (k*c) (Set∅ db) Int`, in such a way that it satisfies ϵ-differential privacy, for any given value of ϵ.

To ensure accurate noise injection, the primitive `laplace` restricts the original datasets to the type `Rel 1 (Set∅ a)`. This type encompasses datasets that differ by at most one row, aligning with the requirements outlined in the Definition 6.1. The new type `Set∅ a` serves to prevent users from manipulating datasets in ways that are not accounted for by the noise-calibrating mechanism. While `Set∅ a` can be seen as synonymous with `Set a`, they differ in the fact that Spar does not support any set operations on the former except by the injection function `set`. In other words, users are unable to perform transformations on this dataset that are not specified by `senT` and, as such, accounted by the stability c.

To calibrate the noise, the `laplace` function leverages the type-level information regarding the sensitivity of aggregations and the stability of transformations. This capability is facilitated by the type constraint `KnownNat`—a mechanism provided by GHC for demoting type-level naturals to term expression. Utilizing this information, the function samples a random number from a Laplace distribution with mean 0 and scale $^{(k\cdot c)}/_{\epsilon}$. This configuration satisfies ϵ-differential privacy.

In preparation for query execution, the Spar function `senA . senT` will be transformed into a Haskell function using the `run` function presented in Section 5. Bringing everything together, `laplace` calculates the result of running the query on the dataset and returns the addition of the sampled noise and the actual value of the query. It is important to note that the use of `run` to obtain the underlying query is appropriate in this context as `laplace` is not intended as part of the toolkit to explore functions' sensitivity, but rather serves as a trustworthy primitive.

6.2 A Privacy Monad

The return type of `laplace`, i.e., the `IO` monad, does not exactly indicate the Differential Privacy guarantees implemented by the function at the type level. To accommodate for that, we extend Spar with a *privacy monad* `PM` capturing randomized computations (see Figure 7b). Such privacy monads are common in other sensitivity calculi (e.g., [Abuah et al. 2022; Gaboardi et al. 2013; Reed and Pierce 2010]). Different from previous work, PM is a graded monad on a number of *privacy units* $1/\phi$, where ϕ :: Nat is decided when executing the computation. For instance, if $\phi = 1000$, then PM 2 Double denotes a $\epsilon = 2 \cdot (1/1000)$ differentially private computation. With the privacy monad in place, we can adapt the type of laplace as follows:

```
laplace :: ∀ k c a b . (Rf a, Rf b, KnownNat k, KnownNat c, KnownNat χ, KnownNat ϕ, ?ϕ :: Proxy ϕ)
          ⇒ Proxy χ → Sen k (Set b) Int → Sen c (Set∅ a) (Set b) → Rel 1 (Set∅ a) → PM χ Double
```

The type constraint ?ϕ :: Proxy ϕ represents an *implicit parameter* [Lewis et al. 2000] to capture the privacy unit. Intuitively, an implicit parameter a of type b, i.e., ?a :: b, acts as a “global” parameter that needs to be defined in order to execute the function. The run function for monadic computations requires the value for ?ϕ to be concretized—we omit the details for brevity, but interested readers can refer to the accompanying artifact for details.

By using PM χ, Spar can be easily extended with a generalized version of the primitive addNoise from Fuzz:

```
addNoise :: ∀ d k a χ ϕ . (?ϕ :: Proxy ϕ, KnownNat ϕ, KnownNat χ, KnownNat n, KnownNat k, Rf a)
           ⇒ Proxy χ → Sen k a Int → Rel d a → PM χ Double
```

In order to integrate the privacy monad in our model, following previous work, we use an approach based on probabilistic coupling. See Section **??** of the accompanying material for a detailed explanation of the monad’s integration.

6.3 Cumulative Distribution Functions

With these extensions, we are now able to implement canonical differentially-private algorithms such as the following $(l \cdot \epsilon)$-differentially private cumulative distribution functions (CDFs), where l is the number of partitions or buckets (represented as type synonym of Int) and $\epsilon = \chi/\phi$.

```
1  cdf :: ∀ l ϕ χ . (?ϕ :: Proxy ϕ, KnownNat ϕ, KnownNat χ)
2      ⇒ Proxy χ → Vec l Bucket → Rel 1 (Set∅ Int) → PM (l∗χ) [(Bucket, Double)]
3  cdf χ VNil          dataset = return [ ]
4  cdf χ (VCons b bks) dataset =
5    let senT :: ∀ d . Bucket → Rel d (Set Int) → Rel d (Set Int)
6        senT b s = case setSplit @d @Int (λz → b ⩾ z) s of
7                         cBucket :⋆: _rest → up cBucket
8        noisyCount :: Bucket → PM χ Double
9        noisyCount b = laplace @1 @1 χ size (senT b) dataset
10   in do x ← fmap (b, ) (noisyCount b); r ← cdf χ bks dataset; return (x : r)
```

In this case, the aggregating function is the primitive size, which counts the number of elements in a set. The transformations, on the other hand, are captured by the local function senT. Given a bucket b and a dataset s, senT returns the subset of elements in s that are less than or equal to b (lines 5-7). Using the laplace primitive, an χ/ϕ-differentially private noisyCount query is defined. This query takes a bucket b, divides the original dataset accordingly, and provides a noisy count of the number of elements belonging to that bucket (lines 8-9). Finally, this query is called on the remaining buckets to finish computing the CDF of dataset (line 10).

6.4 K-Means Clustering

Another example of a differentially private algorithm that can be implemented in Spar is the k-means clustering algorithm. This algorithm partitions a dataset of points into k clusters, where each cluster is represented by its centroid. The corresponding type-signature for kmeans is presented below with points represented as pairs of integers (one for each coordinate) and centroids as points.

```
kmeans :: ∀ d min max iter k χ ϕ . (?ϕ :: Proxy ϕ, KnownNat ϕ, KnownNat χ, min ⩽ max
  , KnownNat n, KnownNat min, KnownNat max, KnownNat (max-min))
   ⇒     Vec iter () → Proxy χ → Range min max → Vec l Centroid → Rel d (Set Point)
   →     PM (l*(3*χ*iter)) (Vec l Centroid)
```

where the third parameter Range min max encodes the range of the x and y coordinates. The first parameter (Vec iter ()) is a vector of unit elements encoding the number of iterations. The implementation of kmeans (see details in Section **??** of the accompanying material) uses the following auxiliary functions which are defined using Spar primitives and addNoise:

```
vecSeq       :: Vec l (PM χ a) → PM (l*χ) (Vec l a)
partition    :: Vec l Centroid → Rel d (Set Point) → Vec l (Rel d (Set Point))
findCenter   :: ∀ d min max χ ϕ . (KnownNat χ, ?ϕ :: Proxy ϕ, KnownNat ϕ, min ⩽ max,
                KnownNat d, KnownNat min, KnownNat max)
              ⇒ Proxy χ → Range min max → Rel d (Set Point) → PM (3*χ) Centroid
```

Comparing Spar with DFuzz's kmeans implementation, we find that the former scales privacy cost not only by the number of iterations iter but also by the number of clusters l due to the implicit sequential composition encoded by vecSeq, unlike DFuzz's parallel approach. Sequentially composing differentially private computations accumulates the privacy cost of the operations, whereas composing them in parallel maintains the cost of the maximum singular operation provided the operations are executed over *disjoint datasets*. Adding support for parallel composition in Spar requires incorporating mechanisms to check for disjoint datasets, which can be done by a simple information-flow control analysis [Lobo-Vesga et al. 2020].

The implementation of cdf and kmeans demonstrates how Spar's proof of sensitivity and distance tracking can be utilized to deploy differentially private advanced user-defined algorithms.

7 Discussion

Non-Termination. Our use of logical relations in Section 3 states that if two computations over metrically related inputs *do both terminate*, then their outputs are metrically related. Since our calculus has no fixed-point primitive of the kind **fix** of type $\tau \to \tau$; our formal guarantees align with our formalization. Nevertheless, adding such primitive demands the use of the well-known mechanism of step-indexed logical relations [Ahmed 2006], which makes it possible to prove the fundamental theorem of logical relations even in the presence of fixpoints. Note that our recursive operators **vec-rec** e $\{x.xs.r.e\}$ e and **vec-rec** e $\{x.xs.r.e\}$ e do not jeopardize λ_{Spar}'s termination as they are defined using structural recursion over vectors, which is a well-founded recursion scheme that guarantees termination. We leave as future work how to extend our formalization with fixpoints and address abnormal termination—a good starting point is to consider the mitigation techniques proposed in Fuzz [Reed and Pierce 2010].

Branching on Relational Values. As shown in Section 5.2, enabling branching on relational terms is problematic—a well-known limitation shared in many related works (e.g., [Abuah et al. 2022; Gaboardi et al. 2013; Near et al. 2019]). We foresee a possible manner to overcome this limitation by adopting the program continuity verification framework by Chaudhuri et al. [Chaudhuri et al. 2012]. This framework characterizes how a small perturbation to the input variables of a given branch condition can cause to exercise different branches, which could lead to syntactically divergent behaviors. A step in this direction has been recently taken in [Freiermuth 2023].

Distance Using Real Numbers. Our implementation of λ_{SPAR} only considers distances as natural numbers. This design decision is based on the limitations of Haskell's type system. Dependently-typed languages—like Coq, Idris, and Agda—can easily support (axiomatic[5] or constructive [Cruz-Filipe et al. 2004; Geuvers and Niqui 2002]) encodings of real numbers at the type level in a natural way. To illustrate this point, we reformulate the type-signature of `lit` from Figure 6 in a type-dependent fashion: `lit : ∀ {A : Set} {d : ℝ} Rel d A` where $\mathbb{R}$ is the type for representing real numbers and `d` is a term of that type. Different from Spar, the type `Rel` is indexed by a term-level real number. There is a series of works on formulating parametricity with dependent types [Bernardy et al. 2010, 2012; Bernardy and Moulin 2012]. In this light, we expect that our soundness results also hold in such a setting—an interesting direction for future work.

Function Space on Relational Types. When designing DSLs, it is often the case that either the function space is introduced in the DSL, i.e., `Exp (a → b)`, or it is avoided given that the host language's function space, i.e., `Exp a → Exp b`, is sufficient to build the type of programs one is interested in. As shown in Sections 2-4 we have opted for the latter approach, hence, functions of type `a → b` do not have an associated relational interpretation. Equipping Spar with an internalized function space imposes significant implementation challenges—more details are provided in Section **??** of the accompanying material. While interesting and attainable endeavor, we leave it as future work as it is unclear whether the advantages gained from extending the language are substantial enough to justify the increased complexity imposed on the system.

Spar's Guarantees. Given that Spar is an embedded language, programmers can use the full expressive power of the host language while constructing expressions of `Rel` type through terminating pure functions. Importantly, this practice does not compromise the formal guarantees established by λ_{SPAR}, as delineated in Theorem 3.2. Specifically, any total Haskell function `f` with type signature of the form `∀ d . Rel d a → Rel (k*d) b` inherently adheres to k-sensitivity, thus upholding the integrity of our computational model. As such, we cannot claim that Spar allows programmers to reason about any function that is k-sensitive, just those that are terminating and possess this specific type signature.

While the implementation of Spar aligns closely with the theory, there are some differences worth noting. Specifically, the treatment of recursion and the nature of indices in the implementation differ from the theoretical framework. While the former mainly impacts the precision of sensitivity analysis, the latter could influence the soundness of the analysis. These could be addressed by embedding Spar in a language with a termination checker, like Agda, or by implementing recursion operators similar to those in λ_{SPAR}, making Spar a standalone DSL rather than an embedded one. These approaches are complementary to our main message about capturing polymorphic sensitivity using parametricity.

Previously, we have noted that Spar leverages Haskell's ability for manipulating type-level natural numbers expressions and their equalities to reason about distances and sensitivities. The arithmetic constraints that arbitrary programs can create are intricate and we suspect them undecidable; this is also DFuzz's case as it has a similar type-level reasoning [de Amorim et al. 2014]. However, this limitation seems mostly theoretical. In practice, the type-checker handles all our examples, except for one occasion where we needed to resort to an SMT solver, which resolved it efficiently. Alternatively, one could consider embedding Spar in a language with a more powerful type system, such as Liquid Haskell's [Rondon et al. 2008] refinement types, offloading the verification of arithmetic constraints to the SMT solver.

[5]Like in https://coq.inria.fr/library/Coq.Reals.Raxioms.html

Spar's Extensibility. Spar can be easily extended to incorporate list types, sum types, or booleans, akin to other core calculi such as Fuzz. Adapting distances and metric spaces to accommodate infinite distances is a straightforward extension, maintaining the meta-theory with minimal alterations. Yet, these extensions might yield limited benefits, as most operations default to infinite distances. As such, these types are intentionally omitted from λ_{Spar}, and modeled by Spar's host language instead. Note that Spar's host language operates outside the sensitivity calculus, thus residing in the realm of infinite distances, hence, we can easily employ a list of relational integers with the type `[Rel d Int]`, which is a simpler approach than expanding the index language to track infinity.

8 Related Work

Sensitivity by Linear Types. Several works have studied techniques to reason about program sensitivity by typing, most of which in the context of differential privacy. An early approach is the work by Reed and Pierce [2010]. They designed an indexed linear type system for differential privacy where types explicitly track sensitivities using types of the form $!_r A \multimap B$. In their work, this type can only be assigned to terms representing functions from A to B which have sensitivity less than r. Functions of these forms could be turned into differentially private programs by adding noise carefully calibrated to r. The type system by Reed and Pierce was implemented in the language Fuzz which was also extended with a timed runtime to avoid side channels with respect to the differential privacy guarantee [Haeberlen et al. 2011]. Automated type inference for this type system was studied by D'Antoni et al. [2013], and its semantic foundation were studied by de Amorim et al. [2017]. Fuzz was further extended in several directions: Eigner and Maffei [2013] extended Fuzz to reason about distributed data and differentially private security protocols. Gaboardi et al. [2013] extended Fuzz's type checker by means of a simple form of dependent types. Winograd-Cort et al. [2017] extended Fuzz type checker and runtime system to an adaptive framework. Zhang et al. [2019] extended the ideas of Fuzz to a three-level logic for reasoning about sensitivity for primitives that are not captured in Fuzz. de Amorim et al. [2019] add to Fuzz more general rules for reasoning about the sensitivity of programs returning probability distributions.

Departing from Fuzz's line of work, λ_{Spar} does not require the use of linear types; instead, we use distance-annotated types to provide a practical system for analyzing the sensitivity of functions. A way to understand λ_{Spar} with respect to Fuzz is by noting that programs of type $x_1 :_{s_1} \sigma_1, x_2 :_{s_2} \sigma_2, \ldots, x_n :_{s_n} \sigma_n \vdash e : \sigma$ represent Fuzz's $\sum_i s_i$-sensitive functions whose guarantees are provided by the metric preservation theorem. In its simplified form, this theorem states that if each x_i varies by d_i the result of evaluating e will vary by at most $\sum_i d_i * s_i$. In λ_{Spar}, these sensitive functions are typed as $x_1 : \text{Rel } d_1\ \sigma_1, x_2 : \text{Rel } d_2\ \sigma_2, \ldots, x_n : \text{Rel } d_n\ \sigma_n \vdash e : \text{Rel } (\sum_i d_i * s_i)\ \sigma$, where s_i are determined by the operations in e. Here we can see that Rel $d\ \sigma$ explicitly captures the otherwise hidden metric relation between values of type σ. As such, λ_{Spar} demonstrates that is feasible to encode Fuzz's relational semantics as a type system without the need for linearity.

Solo. The closest work aligning with our goal is the Solo system introduced by Abuah et al. [2022]. This system is a differentially-private language that tracks the sensitivity of programs without requiring linear types. The authors' key insight for eliminating the reliance on linearity is that base types can be annotated with Fuzz's sensitivity environments from where the notion of k-sensitivity can be recovered. Their type system is also embedded in Haskell and leverages polymorphism for some specific parts of the implementation.

Notably, our approach differs from that of Solo in three significant ways. Firstly, our sensitivity language annotates base types with *distance* information instead of *sensitivity*, constituting a fundamental departure from Solo's approach and every Fuzz-like system. By tracking distances, we gain the capability to provide *a proof of sensitivity by the compiler* without losing expressivity with respect to sensitivity tracking. Had we chosen sensitivity types instead, the proofs would have eventually needed to be unfolded to distances, e.g., as done in Fuzz or Solo's semantics.

Secondly, and as a result of using distance information, Spar avoids Solo's limitation of assuming the correctness of typing signatures of higher-order functions, e.g., `map` or `foldr`. Given that Solo's formalism and soundness proofs are based on a *monomorphic calculus*, the formal guarantees of their built-in polymorphic high-order primitives are not developed. In Spar, correctness follows directly from the soundness of the system itself. For instance, since in our language `foldr` is implemented and the type system ensures that we can only assign its correct type—even if it depends on the length of the input list [Gaboardi et al. 2013]. In Solo's, unfortunately, the type signature of `foldr` (its only source of recursion) is unsound—see Section A in the accompanying material for details. While the soundness of `foldr` in Solo can be recovered by assigning it a monomorphic type signature taking only 1-sensitive functions, there are no guarantees that other parts of Solo polymorphic notations are correct. We think that this illustrates well why we argue for the importance that Spar builds on the concept of distance which is the same concept on which the metric preservation proof is built, and it illustrates well the importance of a model of parametricity and its soundness.

Thirdly, our primary emphasis is on developing a sound system for calculating function sensitivity, in contrast to Solo, which aims to provide a comprehensive framework for differentially-private consults. Unfortunately, the Solo library appears to be incomplete and lacks proper implementation, e.g., out of the eleven functions included in the standard library, nine are essentially defined as $\perp$ [6]. In contrast, Spar is presented as a ready-to-use Haskell library with a more focused scope.

Other Type-Based Approaches. Near et al. [2019] designed the language Duet to support other notions of differential privacy. The Duet approach is based on the design of a two-layer language. The underlying layer is similar to Fuzz, and the other layer is a linear type system without annotations for sensitivity. While the second layer enables support for approximate differential privacy and other relaxed forms of differential privacy by not imposing constraints on the distance of elements, this approach does restrict the capacity of Duet to provide support for higher-order functions. Toro et al. [2023] further extended this approach by combining linear types with contextual effects, resulting in a system that supports various notions of differential privacy and higher-order functions. It allows for potentially tighter bounds on the sensitivity of user-defined functions compared to those provided by Spar. However, this is only available for type systems with support for tracking both linear and contextual effect information.

Relational Type Systems. Several works have also explored how to reason about sensitivity using relational type systems. This line of work was pioneered by Barthe et al. [2015] and Zhang and Kifer [2017] and further extended afterward, e.g. [Barthe et al. 2016; Wang et al. 2020, 2019]. Barthe et al. [2015] introduce a relational version of refinement types in a higher-order functional language. This general form of refinement types supports the encoding of metric spaces and functions between them, and so it supports the encoding of Fuzz.

Zhang and Kifer [2017] introduce a relational type system for a simple imperative programming language. Their basic types are similar in spirit to our `Rel d` type. In the imperative setting, since programs represent only first-order functions, distance information on basic types is enough to capture the sensitivity. In a functional language like the one we consider, having distance information on basic types is not sufficient. Our work shows how to extend this approach to higher-order functions too. Relational type systems like the ones described above are not readily available and require specialized implementations. They are not easy to use, even for specialists.

[6] https://zenodo.org/record/7079930

Program Analysis. Other approaches to reason about program sensitivity were based on program analysis. To reason about the continuity of programs, Chaudhuri et al. [2012] designed a program analysis tracking the usage of variables and giving an upper bound on the program's sensitivity. Johnson et al. [2020] proposed a static analysis to track the sensitivities of queries in a SQL-like system. Abuah et al. [2021] designed a dynamic sensitivity analysis that tracks sensitivity and metric information at the values level. This dynamic analysis is used to guarantee differential privacy in an adaptive setting, similar to the one explored in Adaptive Fuzz [Winograd-Cort et al. 2017].

Parametricity. Studies of parametricity and its variants abound in the literature. It all started with the seminal paper by Reynolds [1983], where the polymorphic semantic of System F's types is captured in a suitable model. Wadler [1989] then popularized this result as a tool to deduce theorems for polymorphic functions. Kennedy [1994, 1997] showed how parametricity can be useful to reason about units of measure for numeric types. In these works, one can express how the scaling of the input can affect the output for some transformations. Atkey et al. [2013] generalized this approach and showed how they can also capture notions of distance-indexed types. Using distance-indexed types they also showed that they can express sensitivity for functions over reals, but it is unclear if their system can express sensitivity for functions over other types. Our main result proving the sensitivity of functions can be seen as a theorem arising for parametrically polymorphic functions on distances. To be best of our knowledge, we are the first ones to provide soundness proof of the use of parametricity to determine the sensitivity of functions. There exists a series of works on obtaining parametricity results for dependent-typed languages [Bernardy et al. 2010, 2012; Bernardy and Moulin 2012]—which constitutes interesting results when realizing $\lambda_{\textsc{Spar}}$ in languages like Agda, Coq, or Idris.

9 Conclusions

We have presented $\lambda_{\textsc{Spar}}$, a sound calculus that uses parametricity to prove the sensitivity of functions by type-checking. The calculus is simple, and that is its beauty. We also showed how to implement the calculus as the library Spar for the programming language Haskell—where the total library consists of 460 lines of code. We expect that Spar can serve as a basis for providing a light-weight verification tool to certify the sensitivity of functions.

Data-Availability Statement

The software that supports Sections 4-6 is available on Zenodo [Lobo-Vesga et al. 2024].

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. We are also grateful to Andreas Abel for his insightful comments and suggestions on early drafts of the formalization. This work was supported by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011) as well as the Swedish research agency Vetenskapsrådet. Marco Gaboardi's work was partially funded by the National Science Foundation under Grants No. 1845803 and 2040249

References

Chiké Abuah, David Darais, and Joseph P. Near. 2022. Solo: A Lightweight Static Analysis for Differential Privacy. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 150 (oct 2022), 30 pages. https://doi.org/10.1145/3563313

Chiké Abuah, Alex Silence, David Darais, and Joseph P. Near. 2021. DDUO: General-Purpose Dynamic Analysis for Differential Privacy. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021.* IEEE, 1–15. https://doi.org/10.1109/CSF51468.2021.00043

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Proceedings of the 15th European Conference on Programming Languages and Systems (ESOP'06)*. Springer-Verlag. https://doi.org/10.1007/11693024_6

Robert Atkey, Patricia Johann, and Andrew Kennedy. 2013. Abstraction and Invariance for Algebraically Indexed Types. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 87–100. https://doi.org/10.1145/2429069.2429082

Gilles Barthe, Gian Pietro Farina, Marco Gaboardi, Emilio Jesús Gallego Arias, Andy Gordon, Justin Hsu, and Pierre-Yves Strub. 2016. Differentially Private Bayesian Programming. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 68–79. https://doi.org/10.1145/2976749.2978371

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 55–68. https://doi.org/10.1145/2676726.2677000

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158093

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2010. Parametricity and Dependent Types. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 345–356. https://doi.org/10.1145/1863543.1863592

Jean-philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming* 22, 2 (March 2012), 107–152. https://doi.org/10.1017/S0956796812000056

Jean-Philippe Bernardy and Guilhem Moulin. 2012. A Computational Interpretation of Parametricity. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science* (New Orleans, Louisiana) *(LICS '12)*. IEEE Computer Society, USA, 135–144. https://doi.org/10.1109/LICS.2012.25

Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2012. Continuity and robustness of programs. *Commun. ACM* 55, 8 (Aug. 2012), 107–115. https://doi.org/10.1145/2240236.2240262

Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. 2004. C-CoRN, the Constructive Coq Repository at Nijmegen. In *Mathematical Knowledge Management*, Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec (Eds.). Springer, Berlin, Heidelberg, 88–103. https://doi.org/10.1007/978-3-540-27818-4_7

Loris D'Antoni, Marco Gaboardi, Emilio Jesús Gallego Arias, Andreas Haeberlen, and Benjamin C. Pierce. 2013. Sensitivity analysis using type-based constraints. In *Proceedings of the 1st annual workshop on Functional programming concepts in domain-specific languages, FPCDSL@ICFP 2013, Boston, Massachusetts, USA, September 22, 2013*, Richard Lazarus, Assaf J. Kfoury, and Jacob Beal (Eds.). ACM, 43–50. https://doi.org/10.1145/2505351.2505353

Arthur Azevedo de Amorim, Marco Gaboardi, Emilio Jesús Gallego Arias, and Justin Hsu. 2014. Really Natural Linear Indexed Type Checking. In *Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages, IFL '14, Boston, MA, USA, October 1-3, 2014*, Sam Tobin-Hochstadt (Ed.). ACM, 5:1–5:12. https://doi.org/10.1145/2746325.2746335

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2019. Probabilistic Relational Reasoning via Metrics. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. IEEE, 1–19. https://doi.org/10.1109/LICS.2019.8785715

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A semantic account of metric preservation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 545–556. https://doi.org/10.1145/3009837.3009890

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography*, Shai Halevi and Tal Rabin (Eds.). Springer, Berlin, Heidelberg, 265–284. https://doi.org/10.1007/11681878_14

Hamid Ebadi and David Sands. 2017. Featherweight PINQ. *Journal of Privacy and Confidentiality* 7, 2 (Jan. 2017). https://doi.org/10.29012/jpc.v7i2.653

Fabienne Eigner and Matteo Maffei. 2013. 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, 272–286. https://doi.org/10.1109/CSF.2013.25

Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.

Daniel Freiermuth. 2023. A type-driven approach for sensitivity checking with branching.

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy, Vol. 48. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2480359.2429113

Marco Gaboardi, Michael Hay, and Salil Vadhan. 2020. A programming framework for OpenDP. *Manuscript, May* (2020).

Herman Geuvers and Milad Niqui. 2002. Constructive Reals in Coq: Axioms and Categoricity. In *Types for Proofs and Programs*, Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack (Eds.). Springer, Berlin, Heidelberg, 79–95. https://doi.org/10.1007/3-540-45842-5_6

Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. 2011. Differential privacy under fire. In *Proceedings of the 20th USENIX Conference on Security* (San Francisco, CA) *(SEC'11)*. USENIX Association, USA, 33.

Noah M. Johnson, Joseph P. Near, Joseph M. Hellerstein, and Dawn Song. 2020. Chorus: a Programming Framework for Building Scalable Differential Privacy Mechanisms. In *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*. IEEE, 535–551. https://doi.org/10.1109/EuroSP48549.2020.00041

Mark P. Jones. 1994. A theory of qualified types. *Science of Computer Programming* 22, 3 (1994), 231–256. https://doi.org/10.1016/0167-6423(94)00005-0

Andrew J. Kennedy. 1994. Dimension Types. In *Proceedings of the 5th European Symposium on Programming: Programming Languages and Systems (ESOP '94)*. Springer-Verlag, Berlin, Heidelberg, 348–362. https://doi.org/10.1007/3-540-57880-3_23

Andrew J. Kennedy. 1997. Relational Parametricity and Units of Measure. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 442–455. https://doi.org/10.1145/263699.263761

Daniel Kifer and Ashwin Machanavajjhala. 2011. No free lunch in data privacy. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 193–204. https://doi.org/10.1145/1989323.1989345

Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. 2000. Implicit parameters: dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Boston, MA, USA) *(POPL '00)*. Association for Computing Machinery, New York, NY, USA, 108–118. https://doi.org/10.1145/325694.325708

Elisabet Lobo-Vesga. 2021. Let's not Make a Fuzz about it. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 114–116. https://doi.org/10.1109/ICSE-Companion52605.2021.00051

Elisabet Lobo-Vesga, Alejandro Russo, and Marco Gaboardi. 2020. A Programming Framework for Differential Privacy with Accuracy Concentration Bounds. In *Proc. IEEE Symp. on Security and Privacy (SP '20)*. IEEE Computer Society. https://doi.org/10.1109/SP40000.2020.00086

Elisabet Lobo-Vesga, Alejandro Russo, Marco Gaboardi, and Carlos Tomé Cortiñas. 2024. *Paper Artifact: Sensitivity by Parametricity.* https://doi.org/10.5281/zenodo.13622515

Frank D. McSherry. 2010. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. *Commun. ACM* 53, 9 (Sept. 2010), 89–97. https://doi.org/10.1145/1810891.1810916

Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. 2019. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 172 (oct 2019), 30 pages. https://doi.org/10.1145/3360598

Divesh Otwani and Richard A. Eisenberg. 2018. The Thoralf plugin: for your fancy type needs. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. ACM, New York, NY, USA, 106–118. https://doi.org/10.1145/3242744.3242754

Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern Synonyms. In *Proceedings of the 9th International Symposium on Haskell* (Nara, Japan) *(Haskell 2016)*. Association for Computing Machinery, New York, NY, USA, 80–91. https://doi.org/10.1145/2976002.2976013

Davide Proserpio, Sharon Goldberg, and Frank McSherry. 2014. Calibrating Data to Sensitivity in Private Data Analysis. *PVLDB* 7, 8 (2014), 637–648. https://doi.org/10.14778/2732296.2732300

Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. *SIGPLAN Not.* 45, 9 (Sept. 2010), 157–168. https://doi.org/10.1145/1932681.1863568

John C. Reynolds. 1983. Types, Abstraction, and Parametric Polymorphism. In *Information Processing 83: Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. Elsevier Science Publishers B. V. (North-Holland), 513–523.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

Josef Svenningsson and Emil Axelsson. 2015. Combining Deep and Shallow Embedding of Domain-Specific Languages. *Computer Languages, Systems & Structures* 44 (dec 2015), 143–165. https://doi.org/10.1016/j.cl.2015.07.003

David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe Haskell. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 137–148. https://doi.org/10.1145/2364506.2364524

Matías Toro, David Darais, Chike Abuah, Joseph P. Near, Damián Árquez, Federico Olmedo, and Éric Tanter. 2023. Contextual Linear Types for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 8 (may 2023), 69 pages. https://doi.org/10.1145/3589207

Philip Wadler. 1989. Theorems for free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, United Kingdom) *(FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

Yuxin Wang, Zeyu Ding, Daniel Kifer, and Danfeng Zhang. 2020. CheckDP: An Automated and Integrated Approach for Proving Differential Privacy or Finding Precise Counterexamples. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 919–938. https://doi.org/10.1145/3372297.3417282

Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. 2019. Proving differential privacy with shadow execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 655–669. https://doi.org/10.1145/3314221.3314619

Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. 2017. A framework for adaptive differential privacy. *PACMPL* 1, ICFP (2017), 10:1–10:29. https://doi.org/10.1145/3110254

Danfeng Zhang and Daniel Kifer. 2017. LightDP: towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 888–901. https://doi.org/10.1145/3009837.3009884

Dan Zhang, Ryan McKenna, Ios Kotsogiannis, Michael Hay, Ashwin Machanavajjhala, and Gerome Miklau. 2020. EKTELO: A Framework for Defining Differentially Private Computations. *ACM Trans. Database Syst.* 45, 1, Article 2 (Feb. 2020), 44 pages. https://doi.org/10.1145/3362032

Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C. Pierce, and Aaron Roth. 2019. Fuzzi: a three-level logic for differential privacy. *Proc. ACM Program. Lang.* 3, ICFP (2019), 93:1–93:28. https://doi.org/10.1145/3341697

Received 2024-04-05; accepted 2024-08-18