Calculating Function Sensitivity for Synthetic Data Algorithms

Markus Pettersson[*]
markus.pettersson1998@gmail.com
Unaffiliated
Göteborg, Sweden

Johannes Ljung Ekeroth[†]
johannes.ljung@gmail.com
Unaffiliated
Göteborg, Sweden

Alejandro Russo
russo@chalmers.se
Chalmers University
DPella AB
Göteborg, Sweden

ABSTRACT

Differential privacy (DP) provides a robust framework for ensuring individual privacy while analyzing population data. To achieve DP, statistical noise is added to query results before publication, but accurately determining the required noise is challenging, especially for user-defined functions. Existing approaches often rely on limited pre-defined functions with known sensitivities, limiting the expressivity of DP systems. In this paper, we present a novel embedded domain-specific language (eDSL) in Haskell to automatically approximate the sensitivity of user-defined linear functions commonly used in synthetic data generation. Our approach leverages Haskell's expressive type system and generic programming principles to infer function ranges, enabling us to compute sensitivities efficiently. We demonstrate the effectiveness of our eDSL by integrating it into the Multiplicative Weights Exponential Mechanism (MWEM) for synthetic data generation. Our solution provides guidance for users when updating functions, ensuring proper sensitivity consideration, enhancing the robustness and reliability of synthetic data algorithms. By adopting this straightforward yet effective approach, we streamline the sensitivity calculation process for user-defined functions, making it more accessible and user-friendly. The contributions of our work include an eDSL capable of approximating sensitivity for linear functions and its evaluation within the context of MWEM workloads.

CCS CONCEPTS

• **Security and privacy** → *Data anonymization and sanitization*; • **Theory of computation** → **Program analysis**; **Type structures**.

KEYWORDS

to be done

[*]This work was done as part of author's master studies at Chalmers.
[†]This work was done as part of author's master studies at Chalmers.

ACM Reference Format:

Markus Pettersson, Johannes Ljung Ekeroth, and Alejandro Russo. 2018. Calculating Function Sensitivity for Synthetic Data Algorithms. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

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 and then adding calibrated statistical noise to the results before their publication [8]. This simple idea has spawned a series of works (e.g., [12, 20, 23, 24, 30]) 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 parameters (ϵ, δ), 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 developed Fuzz [25], a functional programming language that employs linear indexed types to track a programs' sensitivity. This approach has been extended in subsequent works [10, 11, 24, 29], incorporating additional language features such as partial evaluation, linear, and modal types, to enhance 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, further impeding their adoption. Recent studies [1, 19] suggest that linear and complex types in general are not strictly needed for the task of determining programs' sensitivity.

One application of Differential Privacy is in the generation of synthetic data. For instance, the Multiplicative Weights Exponential Mechanism (MWEM) [14] is a simple and practical algorithm for differentially private data release. MWEM is capable of producing synthetic datasets that respect any set of *linear queries*, which are queries that apply a function $f : D \rightarrow [-1, 1]$ to each record from an universe D, and then aggregates the result for the whole dataset. Users can define their custom set of queries, which is called a *workload*. After given a privacy parameter and a workload, MWEM can be used to generate the synthetic data by learning from the original data distribution with a differentially private algorithm to benefit from the theoretical guarantees that differential privacy provides. Numerous synthetic data algorithms follow workflows similar to MWEM with user-defined functions guiding the generation of data.

In this paper, we focus on addressing the challenge faced by synthetic data algorithms when dealing with user-defined functions' sensitivity. Specifically, we aim to provide solutions that accurately determine the sensitivity of these functions, ensuring the proper application of differential privacy techniques during data generation. Additionally, our solution is capable to assist users when they update their functions, ensuring they are informed about the correct sensitivity values to be used with the synthetic data algorithms. Users may inadvertently overlook the sensitivity considerations when introducing new functions, which can lead to privacy breaches and compromised data protection. By providing effective sensitivity calculation methods and guidance for function updates, we aim to enhance the robustness and reliability of synthetic data algorithms.

By focusing on the scenario of synthetic data generation, we show that sophisticated language features (e.g., linear types) are not essential for effectively determining the sensitivity of user-defined functions. Instead, we propose a novel idea of *obtaining the range of linear functions, which enables us to approximate their sensitivity.* This work presents a embedded domain specific language (eDSL) in Haskell capable to approximate the sensitivity of linear functions commonly used in synthetic data generation, such as k-way marginals. Our eDSL makes use of generic programming principles and harnesses Haskell's expressive type system to automatically infer the range of functions—specially on functions which perform pattern-matching (i.e., branch) on input data. An essential aspect of your work is that the eDSL is designed as a library, eliminating the need for modifications to the Haskell compiler or language itself. Our eDSL is lightweight and consists of only 322 lines of code. The contributions of our work are:

(1) an eDSL in Haskell capable to approximate the sensitivity of linear functions commonly used in synthetic data generation.

(2) Evaluation of our technique by integrating it into user-defined workloads for MWEM.

(3) As a by-product of our work, the technique that we have developed can be seen as an improve version of the partial evaluation technique known as "The Trick".

By adopting this more straightforward yet effective approach, we aim to streamline the sensitivity calculation process for user-defined functions, making it more accessible and user-friendly.

2 MOTIVATING EXAMPLE

We consider the Adult data set, which is one of the most popular datasets from the UCI Machine Learning Repository [7], to concretely show the motivation behind our work. This dataset is widely used in other papers concerning the evaluation of synthetic data algorithms like MWEM [14, 18]. The dataset was acquired from US Census data (1994) and contains more than 30000 records, each with 15 associated attributes (columns). For simplicity, we focus on generation of synthetic data that respects 4 out of the available 15 attributes. The 4 attributes of interest are sex, race, work class and hours working per week. They are all modelled as an enumeration data type in Haskell.

```haskell
-- Sex: two options
data Sex  = Male | Female
-- Race: five options
data Race = White | Black | Asian | Eskimo | Other
-- Workclass: nine options
data Workclass = Private  | SelfEmployed | ... | Unknown
-- HoursPerWeek: 100 options
data HoursPerWeek  = Zero | One | Two | ... | NinetyNine

type Adult = (Sex, Race, Workclass, HoursPerWeek)
```

At this point, a user wanted to use MWEM needs to start providing linear queries to the workload. For instance, one such a function could be counting the number of unemployeed black females (q1), the number of entrepreneurs white females working regular hours (q2), and the gender difference between white male and white female in the federal government with regular working hours (q3).

```haskell
q1 :: Adult -> Int
q1 (Female, Black, Unknown, Zero) = 1
q1 _ = 0

q2 :: Adult -> Int
q2 (Female, White, SelfEmployed, Forty) = 1
q2 _ = 0

q3 :: Adult -> Int
q3 (Male, White, FederalGov, Forty)   = -1
q3 (Female, White, FederalGov, Forty) =  1
```

From now on, we use linear query and query as interchangeable terms. The user could then put all these functions into a workload and pass it to MWEM to generate synthetic data respecting (as much as possible) the quantities returned by such functions when applied to the rows of the original dataset. As an example, we show two potential workloads using the queries mentioned above.

```haskell
type Workload row = [row -> Int]

w1 :: Workload Adult
w1 = [q1, q2, q3]
```

```
w2 :: Workload Adult
w2 = [q1, q2]
```

Sensitivity of queries. What are the sensitivity of workload w1 and w2? One way to answer that question is to *approximate the sensitivity* of each function into the workload by its *range* and taking the maximum. If we could statically obtain all the possible values that a query can return, we could approximate its sensitivity by the difference between its maximum and minimum values. From the examples above, we have that the ranges for q1, q2, and q3 are $[0, 1]$, $[0, 1]$, and $[-1, 1]$ which yield sensitivity 1, 1, and 2, respectively—using the $L1$ norm[1]. In this light, the sensitivity for w1 is 2. Analogously, the sensitivity of w2 is 1.

How can the range of query be statically obtained in an automatic manner? Functions q1, q2, and q3 pattern-matches on the input to deliver a result; and pattern-matching is a deeply rooted language feature and therefore the entire mechanism and the techniques used to accomplish it take place at the language level, behind the curtains. This means that a function utilizing pattern matching cannot be distinguished from other functions by just inspecting its static type. Analyzing the `case of` expression in Haskell is fundamentally hard since information about each pattern is not accessible at the language-level. Instead, one way to extract the range, if the domain is finite, is to apply all possible inputs. This technique has been named *The Trick* by N. Jones et al. [16] and comes from partial evaluation literature. The biggest caveat with this technique is that it needs to enumerate the entire domain. For instance, The Trick needs to try with $2 \times 9 \times 5 \times 100 = 9000$ possible values to obtain the range of each function in the workloads above, which may become infeasible with large workloads.

Another alternative is to embed `case of` expressions as data in our eDSL. The benefit of that is to have good control over what information will be part of a program's AST—assuming a deep embedded encoding. As a simple example, consider the following basic eDSL for expressing pattern matching on Haskell values.

```
data CaseDSL p a where
  Case :: p -> [(Alt p, a)] -> CaseDSL p a

-- A pattern in a pattern matching expression.
data Alt a where
  Pat      :: Eq a => a -> Alt a
  Wildcard :: Alt a

instance Eq (Alt a) where
  (Pat a)  == (Pat b)  = a == b
  Wildcard == Wildcard = True
  _        == _        = False

naiveTrick :: CaseDSL p a -> [a]
naiveTrick (Case p arms) = map snd arms
```

However, the merits of this approach pretty much stops here. From a usability perspective, the eDSL does not reuse the Haskell native `case of` expression and thus might seem foreign even to Haskell-programmers.

In our approach, instead, the queries q1, q2, and q3 can be rewritten as follows.

```
q1' :: QueryT Adult -> Int
q1' =
  fourAttr $ \case
    (Female_, Black_, Unknown_, Zero_) -> lit 1
    _                                  -> lit 0

q2' :: QueryT Adult -> Int
q2' =
  fourAttr $ \case
    (Female_, White_, SelfEmployed_, Forty_) -> lit 1
    _                                        -> lit 0

q3' :: QueryT Adult -> Int
q3' =
  fourAttr $ \case
    (Male_, White_, FederalGov_, Forty_)  -> lit 1
    (Female_, White_, FederalGov_, Forty) -> lit (-1)
    _                                     -> lit 0
```

To deduce the range of each function, we develop a new version of The Trick—which we called trick'—capable to obtain the ranges of each function by applying elements of the domain which contribute to unique output values rather than trying with all of them. For instance, to deduce the range of q1', trick' only tries with $2^4 = 16$, which is a much lesser number than 9000! Observe that our eDSL uses the native pattern matching from Haskell `\case` [2]. The attentive reader have notice the underlyne in each constructor, e.g., Male_. We leave its detailed explanation for later (see Section 4), but intuitivey such constructors are given information to the type-system about the constructor being pattern-matched. The type-system then collects all that information and we can easily obtain the range of functions with *trick'*:

```
ghci> trick' q1'
[0,1]

ghci> trick' q3'
[-1,1]
```

In the rest of the paper, we will explain how to implement trick' and how it can be used to statically obtain ranges of linear queries so that the synthetic data algorithm always uses the right information.

3 BACKGROUND

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 3.1 (Sensitivity [25]). Given two metric spaces (A, d_A) and (B, d_B), a function $f : A \rightarrow B$ is k-sensitive iff: $\forall\, x_1, x_2 \in A.\ d_B(f(x_1), f(x_2)) \leq k * d_A(x_1, x_2)$

[1] Also known as the Manhattan distance. In a one-dimensional space the distance between two points x_1 and x_2 is defined as $|x_1 - x_2|$.

[2] `\case { p1 -> e1; ...; pN -> eN }` is equivalent to `\x -> case x of { p1 -> e1; ...; pN -> eN }`, where x is a fresh variable.

As expected, we define the range of a function f, noted R_f, as a subset of its co-domain.

Definition 3.2 (Range of a function). Given a function $f : \tau_1 \rightarrow \tau_2$, the range R_f can be defined as:

$$R_f = \{f\ x \mid x \in \tau_1\}$$

It is easy to see that if a function has a finite range, then it is $d_B(\min R_f, \max R_f)$-sensitive.

3.1 Proxy types

A proxy type is a specialized construct that does not store any data itself but includes a phantom parameter of arbitrary type.

```
data Proxy a = Proxy
```

Its primary purpose is to convey type information, even in situations where an actual value of that type is either unavailable or too expensive to create. Imagine that a function wait for a proxy parameters.

```
f :: Proxy t -> [t]
```

Then, when the function is called, type information gets provided by an explicitly typing annotation.

```
e = f (Proxy :: Proxy Int)
```

3.2 SOP library

The Sum-of-Products library [21] provides a datatype-generic programming interface to define programs in such a way that they automatically work for a large class of datatypes.

3.2.1 N-ary products. This library provides n-ary products of type `NP f ts` for a given functor `f` and a type-level list `ts`. Intuitively, this type allows to implement heterogeneous lists, i.e., lists where elements can be of different types. More specifically, a n-product of type `NP f [t1,t2,t3,...,tn]` should be thought as a list of elements `[x1,x2,x3,...,xn]` where `x1 :: f t1`, `x2 :: f t2`, and so on. Values of n-products are built with constructors `Nil` and `:*`.

```
npExample1 :: NP Maybe [Int,Char]
npExample1 = Just 42 :* Just 'a' :* Nil

npExample2 :: NP Maybe [Int,Char]
npExample2 = Nothing :* Just 'b' :* Nil
```

The library allows to write generic functions on n-ary products. The next example produces an string for each element of an optional value.

```
npString :: All Show ts => NP Maybe ts -> [String]
npString Nil = []
npString (Nothing :* np) = npString np
npString (Just v  :* np) = show v : npString np
```

For `npString` to work, it requires that every possible type in `ts` has an instance of the `Show` type-class. The constraint `All Show ts` requests precisely that. We can now run our generic function.

```
ghci> npString npExample1
["42", "'a'"]
ghci> npString npExample2
["'b'"]
```

The SOP library provides many utilities for doing generic programming. One of the features relevant for our work is the function `hcpure`[3].

```
hcpure :: All c ts
       => Proxy c -> (forall a. c a => f a) -> NP f ts
```

This function is used to create an n-ary product `NP f ts` based *solely on type-level information found in* `ts`. For that, `hcpure` applies a method of a given type class `c` (`forall a. c a => f a`). For instance, the next example shows the type-class `SomeValue` that provides examples of values based on their type via the method `values`.

```
class SomeValue a where
   values :: [a]

instance SomeValue Int where
   values = [42]

instance SomeValue Char where
   values = ['x','y']

npValues :: All SomeValue ts => NP [] ts
npValues = hcpure (Proxy :: Proxy SomeValue) values
```

If we apply `npValues` to any type-level list, we will get an n-ary product with example values for such types.

```
ghci> npValues @[Int,Int,Char]
[42] :* [42] :* "xy" :* Nil
```

3.2.2 N-ary sums. SOP also supports n-ary sums by providing the type `NS f [t1, t2, ..., tn]`, which encodes a value of type `f ti` for some type `ti` appearing in the type-level list. For instance, the value `v :: NS Maybe [Int,Char]` can encode either a value of type `Maybe Int` or `Maybe Char`. N-ary sums are represented by a well-typed index into the list `ts`. The index must fall within the length of the type-level list. Furthermore, each index i must contain an element of type `f ti`. The constructors of indexes are as follows.

```
Z :: f t -> NS f (t : ts)
S :: NS f ts -> NS f (t : ts)
```

The following example are valid indexes of a n-ary sum of type `NS Maybe [Int,Char]`.

```
v1 :: NS Maybe [Int,Char]
v1 = Z (Just 42)

v2 :: NS Maybe [Int,Char]
v2 = S (Z (Just 'a'))
```

Observe that `Z (Just 'a')` is ill-typed—the first position of the type-level list is of type `Maybe Int`, not of type `Maybe Char`!

[3] `hcpure` type is more general but here we present a specialized version for `NP` to simplify the exposition.

4 A NEW TRICK

We outline our main result capable to implement `trick'` as described in Section 2. Our solution includes different key ideas that we will proceed to present incrementally.

4.1 Lifting pattern-matches to the type level

Our first step is to find a way that information about what a function pattern-matches upon gets reflected at the type level. To concrete, we take the following running example of a sum type and a partial function.

```
data T = T0 | T1 | T2 | T3

foo :: T -> Int
foo T0  = 10
foo T2  = 5
```

The type `T` does not reflect over which constructors the function pattern-matches. What would be a good type capable of capturing pattern-matching options? Would such a type still work when adding/removing pattern-matching options?

We propose the idea to consider a type-level list that encodes all the possible options of `T`, i.e., [t0, t1, t2, t3, t4] where `ti` is either a type variable or the unit type `()`. Each position in the type-level list correspond to a constructor in `T`. If `ti` is a type-variable, it means that no pattern-matching has done on the ith-constructor. Instead, if `ti` is the unit type `()`, then a pattern-matches is performed on the ith-constructor. For instance, the type-level list associated to function `foo` is `[(),t1,(),t3,t4]` where `t1`, `t3`, and `t4` are type-variables.

Let's rewrite `foo` with this new type representation in mind. For simplicity, we use a `Proxy` type to wrap the type-level list. So, we propose `foo'` to have the following type.

```
foo' :: Proxy [(),t1,(),t3] -> Int
```

We then proceed to introduce constructors—in the form of pattern synonym—that make sure that their corresponding position in the type-level list gets set to `()`.

```
pattern T0_ :: Proxy [(),t1,t2,t3]
pattern T0_ = Proxy

pattern T1_ :: Proxy [t0,(),t2,t3]
pattern T1_ = Proxy

pattern T2_ :: Proxy [t0,t1,(),t3]
pattern T2_ = Proxy

pattern T3_ :: Proxy [t0,t1,t2,()]
pattern T3_ = Proxy
```

We complete the definition of `foo'` with our new constructors.

```
foo' T0_  = 10
foo' T2_  = 5
```

If we ask for the type of `foo'`, the type-system infers in which constructors the function has pattern-matched.

```
ghci> :t foo'
foo' :: Proxy [(),t1,(),t3] -> Int
```

Observe that if we add (or remove) pattern-matches to `foo'`, the type-system will automatically update the type-level list about the matches done by the function.

```
foo' T0_  = 10
foo' T2_  = 5
foo' T1_ = 0

ghci> :t foo'
foo' :: Proxy [(),(),(),t3] -> Int
```

Observe that if the type signature of `foo'` were to be written explicitly, it would restrict future changes to the function in tedious ways. Instead, we rely on Haskell's type inference to deduce the constructors `foo'` pattern-matches upon. Although function `foo` and `foo'` have different types, they are isomorphic (see Section 4.5 for a explicit connection among them).

4.2 Synthesizing matches at the term-level

The new version of The Trick that we propose, called `trick'`, needs as an input the exact constructors that the function has pattern-matched upon. As show in the previous Section, that information has been collected by the Haskell's type inference. We need to synthesize at the term-level the constructors that the analyzed function pattern-matches on in order for `trick'` being able to invoke the function only on them. For instance, we would like a function that when given the type-list `[(),t1,(),t3]`, it synthesizes the list of options `[T0_,T2_]`. To implement such a function, the challenge relies on how to deduce if a type is either `()` or a *type-variable*. There are no mechanism to explicit ask the type-system if a type is a variable or a concrete type. Nevertheless, we found a manner to achieve that, which constitutes our second key idea.

We start by introducing a datatype to answer if a match has occurred.

```
data Matched a = Yes | No
```

We use that datatype for building a n-ary product `NP Matched ts`. Intuitively, an element of type `Matched ()` is going to be `Yes` in the n-ary product. In contrast, an element of type `Matched t`, for a given type-variable `t`, is going to be `No`. We introduce the type-class `HasBeenMatched` to generate such answers.

```
class HasBeenMatched t where
   answer :: Matched t

instance HasBeenMatched () where
   answer = Yes

instance {-# INCOHERENT #-} HasBeenMatched a where
   answer = No
```

The method `answer` produces `Yes` for unit types.

```
ghci> answer @()
Yes
```

However, when answer is applied to a type variable, it will produce No.

```
ghci> let t' :: forall x. Matched x ; t' = answer @x in t'
No
```

The way that we achieve that is because when answer has type HasBeenMatched x for a type-variable x, there is an ambiguity for the GHC. There are two possible instances that could match: the generic instance HasBeenMatched a or the more specific one HasBeenMatched () (x could unify later with ()). We use the {-## INCOHERENT ##-} pragma that indicates that, in case of ambiguity, so GHC chooses the more generic instance.

We write a function that collects all the answers in a n-ary product and provide an example of its use.

```
defined :: All HasBeenMatched ts => NP Matched ts
defined = hcpure (Proxy :: Proxy HasBeenMatched) answer

matches :: forall t1 t3. NP Matched [(),t1,(),t3]
matches = defined @[(),t1,(),t3]
```

```
ghci> matches
Yes :* No :* Yes :* No :* Nil
```

Once we obtained the list of yeses and noes, we write another function that converts all the yeses into indexes in the type-level list, so effectively indicating the constructors that have been matched.

```
indexes :: forall ts . NP Matched ts -> [NS Matched ts]
indexes Nil          = []
indexes (No  :* rs) = map S (indexes rs)
indexes (Yes :* rs) = Z Yes : map S (indexes rs)
```

Observe the use of n-ary sums. As an example, the list of indexes for matches contains zero and two as expected.

```
ghci> indexes matches
[Z Yes,S (S (Z Yes))]
```

4.3 Connecting patterns with the original type

We define patterns synonyms as indexes into the type-level list of potential constructors to be matched upon.

```
pattern T0_ :: NS Matched [(),t1,t2,t3]
pattern T0_ = Z Yes

pattern T1_ :: NS Matched [t0,(),t2,t3]
pattern T1_ = S (Z Yes)

pattern T2_ :: NS Matched [t0,t1,(),t3]
pattern T2_ = S (S (Z Yes))

pattern T3_ :: NS Matched [t0,t1,t2,()]
pattern T3_ = S (S (S (Z Yes)))
```

As the patterns are generic, these can be not only associated with the type T but also with any other isomorphic data type with four constructors, e.g., data W = W0 | W1 | W2 | W3. So, having the patterns for T and W, a function could match, for instance, on both T_1 and W0_. To avoid such behavior, we introduce a new data type that connects the original type with the patterns being introduced by our technique and adapt the definitions of patterns accordingly.

```
data PatternsOf t ts where
     Constructor :: NS Matched ts -> PatternsOf t ts

pattern T0_ :: PatternsOf T '[(),t1,t2,t3]
pattern T0_ = Constructor (Z Yes)

pattern T1_ :: PatternsOf T '[t0,(),t2,t3]
pattern T1_ = Constructor (S (Z Yes))

pattern T2_ :: PatternsOf T '[t0,t1,(),t3]
pattern T2_ = Constructor (S (S (Z Yes)))

pattern T3_ :: PatternsOf T '[t0,t1,t2,()]
pattern T3_ = Constructor (S (S (S (Z Yes))))
```

We can now rewrite foo' with our new pattern synonyms and ask for its type.

```
foo' T0_ = 10
foo' T2_ = 5
foo' T1_ = 0
```

```
ghci> :t foo'
foo' :: PatternsOf T '[(),(),(),t3] -> Int
```

4.4 Implementing the trick'

We can now present the definition of the trick'.

```
trick' :: forall t ts r . All Matched ts
       => (PatternsOf t ts -> r) -> [r]
trick' f = map (f. Constructor) matches
   where matches = indexes (defined @ts)
```

It takes a function f that pattern-matches on the pattern synonyms associated to the type T (PatternsOf t ts -> [r]), and returns the list of the possible results ([r]). We can observe that the trick' simply obtains the indexes that have been pattern-matched by invoking indexes (defined @ts), and then it proceeds to apply them to f with map. Applying the trick' to foo' outputs its range.

```
ghci> trick' foo'
[10,5,0]
```

4.5 Running functions

We have seen that users writing functions with our patterns synonyms and proposed types enable the use of the trick' to obtain the range of functions—and thus approximated its sensitivity. However, the function to be executed should operate on the datatype t rather than PatternsOf t ts. With this in mind, we defined the following type class which gives an injection between the original datatype and our generated patterns.

```
class Reifyable t ts where
  reify   :: t -> PatternsOf t ts
```

With the method reify, function convert translate functions written with our pattern synonyms into functions working on the original data type.

```
convert :: (PatternsOf t ts -> r) -> t -> r
convert f = f . reify
```

Obtaining instances for `Reifyable t ts` is straightforward as shown by the following examples.

```
instance Reifyable T [(),(),(),()] where
  reify t = case t of
    T0 -> T0_
    T1 -> T1_
    T2 -> T2_
    T3 -> T3_
```

It is easy to see that `convert foo'` is equivalent to the function `foo`.

We note that function `convert` forces all the elements of the type-level list to be of type unit. Hence, it should be called after `trick'` has been used since it will unify all the type-variables (representing unmatched constructors) to `()`.

Removing boilerplate. Every new simple sum type that we would like to apply the `trick'` to will require a unique collection of pattern synonyms. For instance, a type `T` with hundred constructors will require to write hundred pattern synonyms with the right type-level information. To avoid such undesirable overhead, we use Template Haskell (TH) to generate this code automatically since it follows a clear pattern. The details of our TH pipeline are not of any interest with regards to the technique described here. Therefore, they are omitted here. From the user perspective, we provide the function `mkTCBoiler` that is responsible to automatically generate the patterns specific to a given sum type `T`.

```
data T = T0 | T1 | T2 | T3

$(mkTCBoiler ''T)

foo' T0_ = 10
foo' T2_ = 5
foo' T1_ = 0
```

4.6 Wildcards

Until this point, we have successfully built a framework for analyzing pattern-matching of enumeration types at the type-level. However, we have omitted *wildcard* matches, which are an integral part of pattern matching. It is part of every Haskell developer's toolbox, and from the perspective of range analysis, it could be considered one of the inputs to try to apply. For instance, let us consider that function `foo` is instead written as follows.

```
foo :: T -> Int
foo T0 = 10
foo T2 = 5
foo _  = 20  -- wildcard
```

To be able to infer the range of the function, `trick'` needs to also hit the wildcard with a constructor that is *not* matched explicitly, e.g., `T3` or `T1` in this case. The information about constructors not matched explicitly is already present at the type-level as type variables. Consequently, during the the call of `indexes`, all unmatched constructors will coincide with `No` values—recall Section 4.2. *It is then enough to convert one of such* `No` *values into a* `Yes` *for the* `trick'` *to consider a constructor that will "hit" the wildcard.* The following function flips the first `No` into a `Yes`.

```
addWildcard :: All Matched ts
            => NP Matched ts -> NP Matched ts
addWildcard Nil        = Nil
addWildcard (No :* r) = Yes :* r
addWildcard (p  :* r) = p :* addWildcard r
```

Notice how `ts` remains unchanged, even though we are now potentially matching on yet another constructor. We can present now the adapted version of the `trick'`.

```
trick' :: forall t ts r . All Matched ts
       => (PatternsOf t ts -> r) -> [r]
trick' f = map (f. Constructor) matches
   where matches = indexes (addWildcard (defined @ts))
```

We carefully insert `addWildcard` after bringing the type-level information about explicitly matches constructors down to the term-level. From that point on, function `indexes` generates the constructors to be applied by `trick'` only based on the `Yes`/`No` values—so, an unmatched constructor will be considered in the mix.

```
foo' T0_ = 10
foo' T2_ = 5
foo' _   = 20  -- wildcard

ghci> trick' foo'
[10,20,5]
```

Because of the inclusion of a wildcard pattern, we require the user-defined function to be exhaustive. If not, a function could throw a **Non-exhaustive patterns** error at runtime when calling the `trick'`. We think this is a reasonable demand for two reasons: from a correctness perspective, we know that every input has been considered and handled appropriately, and because it is the same requirement as other libraries implementing The Trick [28].

5 WRITING LINEAR QUERIES

With the technique in Section 4 at our disposal, we set out to implement a DSL capable of approximating the sensitivity of user-defined linear queries.

To capture the notion of a query, we construct the `Query` DSL. Programs in this DSL can be interpreted in two different ways: as a query which execution is as expected or as an static analysis based on an interval-based semantics—which we use to reason about range of function. In this section, we will focus on the later since the former is straightforward.

We start by defining a query that only matches on one attribute of the data set.

```
data Query t r where
  OneAttr :: ( All Matched ts, Reifyable t ts
             , Num r, Ord r)
          => (PatternsOf t ts -> r) -- User-defined query
          -> Query t r
```

By restricting the results of the queries to numerical values (`Ord r`, `Num r`), we can leverage one of the many libraries for Haskell which define arithmetic on intervals when defining the interval semantics of the DSL[17, 22, 26]. We choose the library `Numeric.Interval` [17] for two particular reasons: it defines infimum, supremum and the complex hull between two intervals; we did not need support for open intervals or rational numbers, which is offered by other, more featureful interval libraries [26].

Once a user-defined query has been encapsulated inside the `OneAttr` constructor, we can use the `trick'` from Section 4 to obtain its range.

```
import Numeric.Interval ( hull
                        , width -- |max - min|
                        )
import NewTrick (trick')

intervalSem :: Query t r -> Interval r
intervalSem (OneAttr q) = foldl1 hull (trick' q)

sensitivity :: Num r => Query t r -> r
sensitivity = width . intervalSem
```

We use the primitive `hull` to create a convex, closed set of points from the elements in the range of the query. Then, we approximate the sensitivity of the query by the *widht* of the interval.

Since workloads could include queries matching on two attributes as well, we proceed to also introduce an special constructor for such cases and extend the interval semantics accordingly.

```
  data Query t r where
       OneAttr :: ( All Matched ts, Reifyable t ts
                  , Num r, Ord r )
               => (PatternsOf t ts -> r)
               -> Query t r
       TwoAttr :: ( All Matched ts1, All Matched ts2
                  , Reifyable t1 ts1, Reifyable t2 ts2
                  , Num r, Ord r )
               => (    PatternsOf t1 ts1
                    -> PatternsOf t2 ts2
                    -> r )
               -> Query (t1, t2) r

intervalSem :: Query t r -> Interval r
intervalSem (OneAttr q)     = foldl1 hull (trick' q)
intervalSem (TwoAttr q1 q2) =
   foldl1 hull (trick' q1 ++ trick' q2)
```

We also provide an *uncurryfied* interface for convenience of the developers.

```
oneAttr = OneAttr
twoAttr = TwoAttr . curry
```

In a similar way, we add constructors for queries which consider three and four attributes—omitted here for brevity. We leave as future work to consider a generic constructor that could work with *n*-attributes.

In what follows, we show an example to illustrate our DSL.

```
import TheNewTrick

data T = T0 | T1 | T2 | T3

$(mkTCBoiler ''T)

import DSL

bar :: Query T Int
bar = oneAttr $ \case
        T1_ -> lit 1
        T2_ -> lit 15
        _   -> lit 30

bar' :: Query (T, T) Int
bar' = twoAttr $ \case
         (T1_, T2_) -> lit 10
         _          -> lit 20

-- ghci> intervalSem bar
-- (1 ... 30)
-- ghci> intervalSem bar'
-- (10 ... 20)
```

From here on, we can extend the functionality of the DSL to support more expressive queries, such as comparisons and conditional branching. However, at this stage, we have everything we need to achieve the main purpose of this DSL: *turn user-defined functions into a range of possible output values.* All that remains is to integrate this functionality into an existing differential privacy system so that it can be used to approximate the sensitivity of workloads.

6 CASE STUDY

We evaluate our technique with DPella's implementation of MWEM applied to the Adult dataset. We, however, limit this case study to queries on four out of the available fifteen attributes. The attributes we focused on are sex, race, work-class and age. Sex, race and work-class are all discrete attributes. They are modelled as an enumeration data type in Haskell—recall Section 2.

The fourth attribute, age, is modeled as an integer but we discretize it into nine distinct intervals to capture adults within the range from 0 to 99 years.

```
data Age
  = Teens     -- age < 20
  | Twenties  -- 20 <= age < 30
  | Thirties  -- 30 <= age < 40
  | Fortys    -- 40 <= age < 50
  | Fiftys    -- 50 <= age < 60
  | Sixtys    -- 60 <= age < 70
  | Seventees -- 70 <= age < 80
  | Eighties  -- 80 <= age < 90
  | Nineties  -- 90 <= age < 100
```

We were given a workload of fifty linear queries that each discriminates on two out of the four attributes of interest[4]. The queries cover all possible domains that can be constructed by combining the four attributes in pairs of two[5]. We have queries on the following forms:

$$\begin{aligned}
Sex \times Workclass &\rightarrow Result\\
Sex \times Age &\rightarrow Result\\
Sex \times Race &\rightarrow Result\\
Age \times Race &\rightarrow Result\\
Age \times Workclass &\rightarrow Result\\
Race \times Workclass &\rightarrow Result
\end{aligned}$$

Where the *Result* type is implemented as Haskell's `Double` type. The implementation of these queries is not important at this stage.

The workloads are modeled using the following record data type.

```
data Workload row = Workload
  { getWorkload    :: [row -> Double]
  , getSensitivity :: Sensitivity
  }
```

As we can see, the `Workload` is parameterized over a type `row`, i.e., the type of records in the data set. In our case, `row` will get instantiated to `Adult`.

```
type Adult = (Sex,Race,Workclass,Age)
```

For all of the given queries to be bundled in the same workload, we need a way of lifting each query from its two attributes input domain to work on the whole `Adult` type—we omit these details since it is not hard to imagine how to do it. Finally, we can encode the given workload as follows.

```
workload :: Workload Adult
workload = Workload
    [query1, query2, . . . , query50] -- given queries
    1.0  -- hard-coded sensitivity.
         -- Someone must have manually reviewed
         -- query1 .. query50.
```

6.1 Writing the workload using the Query DSL

We rewrite the given queries using our DSL.

```
$(mkTCBoiler ''Sex)
$(mkTCBoiler ''Race)
$(mkTCBoiler ''Workclass)
$(mkTCBoiler ''Age)

query1' :: Query (Sex, Race) Double
query1' = twoAttr $ \case
  (Female_,Eskimo_) -> 1.0
  _      -> 0.0

query2' :: Query (Sex, Workclass) Double
query2' = twoAttr $ \case
  (Male_,Private_) -> 1.0
  _                -> 0.0

...  -- the rest of the queries
```

[4]An exhaustive list of queries, i.e. a query on every combination of the two attributes, would consist of 400 queries.
[5]Also called 2-way marginals.

Once the adapted versions of the queries haven been written with the `Query` DSL, we need to construct a workload with them. When constructing a workload we also need to include the sensitivity parameter, but this time it gets computed automatically through the `sensitivity` function.

```
-- Converts a linear query in the DSL into
-- a query that works directly on the data set
extractFun :: Query a r -> a -> r
extractFun (OneAttr q)     a     = q (reify a)
extractFun (TwoAttr q1 q2) (a,b) = q (reify a,reify b)

-- Put together queries written in our DSL into a workload
constructWL :: [Query Adult Double] -> Workload Adult

constructWL qs = Workload funs sen
   where sen  = maximum $ map sensitivity qs
         funs = map extractFun qs
```

Observe how the sensitivity in the workload is now computed by the function `sensitivity`. We can now build a workload based on the queries written in our DSL.

```
workload :: Workload Adult
workload = constructWL
   [query1', query2', . . . , query50']
```

6.2 Evaluation

We have previously stated that MWEM runs on a workload (a set of queries), but the algorithm requires more parameters than that.

- **Epsilon**: What is referred to as the *privacy budget* in the literature [14, 18]. We will test the integration with standard ϵ values of 0.01, 0.1 and 1 [9].
- **Iterations**: The original paper suggests running the algorithm with 10 iterations [14]. This is not always optimal, but for our use case, it will suffice [18].
- **Workload row**: The set of queries to run MWEM with. We will produce the results using both the given `workload` and our adapted `workload'` presented in section **??**.
- **Dataset rows**: The underlying dataset that MWEM will try to approximate.
- **Dataset universe**: The entire universe of all possible records. In this case, it is every combination of the 4 attributes sex, race, work class and age.

The result of running MWEM is an approximate distribution of the records in the underlying dataset. MWEM will try to replicate the characteristics of the underlying dataset with respect to the queries in the supplied workload. The result gives back the entire universe of records, where each record has been coupled with a distribution

$0 \leq d(x_i) \leq 1$[6]. From this distribution, one can synthesize an arbitrarily large dataset that respects the characteristics[7] of the underlying dataset.

To evaluate our integration, we will run MWEM on the given `workload` with hard-coded sensitivity as well as our adapted `workload'` where the sensitivity is computed. As a baseline, we will run MWEM on the `workload` three times for different values of ϵ. The resulting distributions are scaled up by the number of records in the original dataset: 32561. Let us call these synthesised datasets (A). Finally, we run every query on the original dataset as well as the synthezied datasets (A). Since every query is a linear query, we simply sum up the result of every query for all datasets. This allows us to compare how well the synthesised datasets (A) perform, compared with the original dataset, with respect to every query.

We then proceed by performing the same evaluation using our adapted workload, i.e., `workload'`. The difference is that the sensitivity of the workload has been computed from the set of queries. What we want to find out is if the newly synthesised datasets (B) perform as well as the previously synthesised datasets (A).

Figure 1 presents a few results of running the MWEM algorithm, both with `workload` and `workload'`. Each figure shows the result of running each query to the raw data and two sets of synthetic data (A) and (B). The information of interest in these figures is how well MWEM with `workload'` (marked with a cross) lines up with the one running with `workload` (the red circle). A cross lined up perfectly inside a red circle means that the algorithm produced the same result when run twice; one time using the existing workload, and one time using the workload generated from queries written by our DSL. The blue circles represent the result of applying the queries to the raw data. However, we emphasize that we in this work are not interested in how well the synthetic data mimics the raw data. We can observe that the quality of the synthetic data, when generated with our adapted workload, was not significantly different than had been synthesised using the existing workload.

7 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 [25]. They designed an indexed linear type system for differential privacy where types explicitly track sensitivities thanks to 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 [25] 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 [13]. Automated type inference for this type system was studied by D'Antoni *et al.* [4], and its semantics foundation was studied by Azevedo de Amorim *et al.* [6]. Fuzz was further extended in several directions: Eigner and Maffei [10] extended Fuzz to reason about distributed data and differentially private security protocols. Gaboardi *et al.* [11] extended Fuzz's type checker by means of a simple form of dependent types. Winograd-Cort *et al.* [29] extended Fuzz type checker and runtime system to an adaptive framework. Zhang *et al.* [31] extended the ideas of Fuzz to a three-level logic for reasoning about sensitivity for primitives that are not captured in Fuzz. Azevedo de Amorim *et al.* [5]

[6]The total sum of all distributions should equal 1

[7]Again, this is with respect to the queries on the dataset.

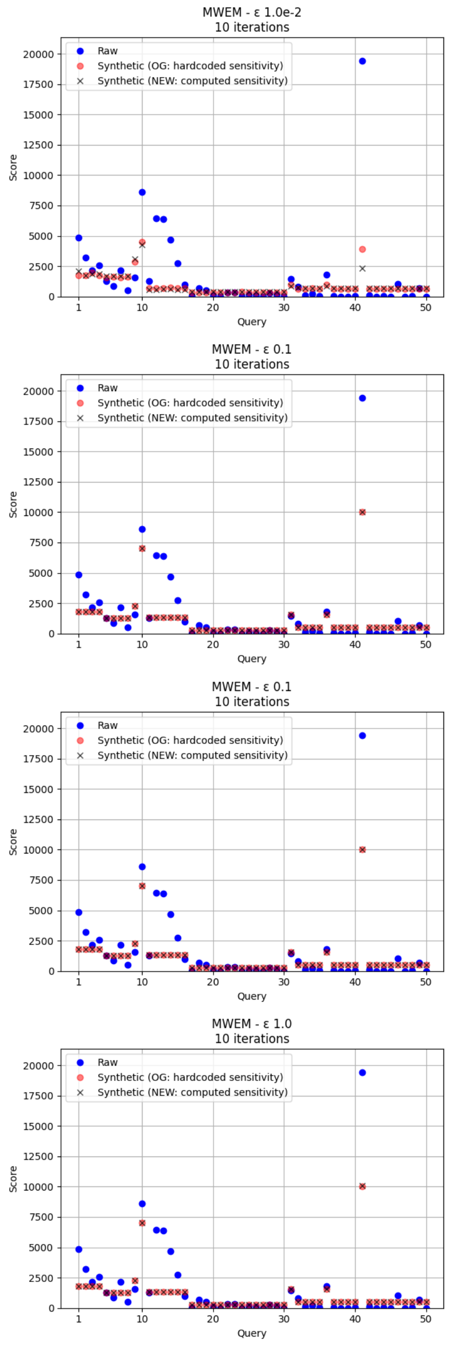

Figure 1: Result of running the MWEM algorithm with different ϵ

add to Fuzz more general rules for reasoning about the sensitivity of programs returning probability distributions. Departing from Fuzz's line of work, our work does not require the use of linear types; instead, we simply approximate sensitivity by *the width* of functions' ranges, which seems an acceptable over approximation for linear queries used in synthetic data algorithms.

Other type-based approaches. Near et al. [24] designed the language Duet to support other notions of differential privacy. The Duet approach is based on the design of a two-layers language. The underlying layer is similar to Fuzz, and the other layer is a linear type system without annotations for sensitivities. 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.* [27] 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. By incorporating contextual effects, their system enables a detailed analysis of program sensitivity, particularly when dealing with complex data structures. As a result, it allows for potentially tighter bounds on the sensitivity of user-defined functions compared to our work. However, this is only available for type systems with support for tracking both linear and contextual effect information.

Another work aligned with our goal is Abuah *et al.*'s [1] system named Solo. This system is a fully-fledged differentially-private language that allows for tracking the sensitivity of programs without requiring linear types. The authors' main 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.

Program analysis. Other approaches to reason about program sensitivity were based on program analysis. To reason about the continuity of programs, Chaudhuri et al. [3] designed a program analysis tracking the usage of variables and giving an upper bound on the program's sensitivity. Johnson *et al.* [15] proposed a static analysis to track sensitivities of queries in a SQL-like system. Abuah *et al.* [2] 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 [29].

8 FINAL REMARKS

In this paper, we presented a novel approach to approximate the sensitivity for functions with pattern-matching on enumeration types. We introduced a type-level representation of pattern-matching information and used it to approximate the sensitivity of functions. Our technique allows us to automatically identify the matched constructors and synthesize them at the term level, enabling us to evaluate the function on all relevant inputs. By leveraging Template Haskell, we provided a seamless integration of our method into existing Haskell codebases. We demonstrated the effectiveness of our approach through a case study which uses MWEM and involves a workload of linear queries on the Adult dataset. Our technique not only provides valuable insights into the sensitivity of functions but also paves the way for more robust and accurate privacy analysis in differential privacy systems implemented in Haskell. By automatically generating the necessary pattern synonyms and handling wildcard patterns, we reduce the burden on developers, facilitating the adoption of privacy-aware programming practices in Haskell-based systems. Overall, our method contributes to enhancing the privacy guarantees and usability of Haskell-based differential privacy solutions for synthetic data generation.

ACKNOWLEDGMENTS

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and by the Swedish research agency Vetenskapsrådet.

REFERENCES

[1] 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
[2] Chike 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
[3] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2012. Continuity and robustness of programs. *Commun. ACM* 55, 8 (2012), 107–115.
[4] 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
[5] 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
[6] 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
[7] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/datasets/adult
[8] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Proceedings of the Third Conference on Theory of Cryptography* (New York, NY) *(TCC'06)*. 265–284.
[9] Cynthia Dwork, Aaron Roth, et al. 2014. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.* 9, 3-4 (2014), 211–407.
[10] 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
[11] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*
[12] Marco Gaboardi, Michael Hay, and Salil Vadhan. 2020. A programming framework for opendp. *Manuscript, May* (2020).
[13] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. 2011. Differential Privacy Under Fire. In *Proc. of USENIX Security Symposium.*
[14] Moritz Hardt, Katrina Ligett, and Frank McSherry. 2012. A simple and practical algorithm for differentially private data release. *Advances in neural information processing systems* 25 (2012).
[15] 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
[16] Neil Jones, Carsten Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation.*
[17] Edward Kmett. 2021. intervals: Basic interval arithmetic. *Available online https://hackage.haskell.org/package/intervals* (2021).

[18] Dan Josephus Knoors. 2018. Utility of Differentially Private Synthetic Data Generation for High-Dimensional Databases. (2018). http://www.diva-portal.se/smash/get/diva2:1252390/FULLTEXT01.pdf

[19] 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

[20] Elisabet Lobo-Vesga, Alejandro Russo, and Marco Gaboardi. 2020. A Programming Framework for Differential Privacy with Accuracy Concentration Bounds *(SP '20)*. IEEE Computer Society.

[21] Andres Löh. [n. d.]. SOP NP (n-ary products). *Available online https://hackage.haskell.org/package/sop-core-0.5.0.2/docs/Data-SOP-NP.html* ([n. d.]).

[22] Robert Massaioli. 2019. range: An efficient and versatile range library. *Available online https://hackage.haskell.org/package/range* (2019).

[23] Frank D. McSherry. 2009. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD*. ACM.

[24] 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 (2019), 172:1–172:30.

[25] Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proc. ACM SIGPLAN International Conference on Functional Programming*.

[26] Masahiro Sakai. 2021. data-interval: Interval datatype, interval arithmetic and interval-based containers. *Available online https://hackage.haskell.org/package/data-interval* (2021).

[27] 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

[28] Nachiappan Valliappan, Robert Krook, Alejandro Russo, and Koen Claessen. 2020. Towards Secure IoT Programming in Haskell. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell* (Virtual Event, USA) *(Haskell 2020)*. Association for Computing Machinery, New York, NY, USA, 136–150. https://doi.org/10.1145/3406088.3409027

[29] 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

[30] Dan Zhang, Ryan McKenna, Ios Kotsogiannis, Michael Hay, Ashwin Machanavajjhala, and Gerome Miklau. 2018. EKTELO: A Framework for Defining Differentially-Private Computations. In *Proc. International Conference on Management of Data*.

[31] 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