CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Sensitivity computation for user-defined functions in Differential Privacy systems

Master's thesis in Computer science and engineering

Johannes Ljung Ekeroth & Markus Pettersson

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2022
www.chalmers.se

Master's thesis 2022

Sensitivity computation for user-defined functions in Differential Privacy systems

A technique for co-domain range analysis on user-defined functions

Johannes Ljung Ekeroth
Markus Pettersson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2022

Sensitivity computation for user-defined functions in Differential Privacy systems

Johannes Ljung Ekeroth & Markus Pettersson

Supervisor: Alejandro Russo, Department of Computer Science and Engineering
Examiner: David Sands, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Sensitivity computation for user-defined functions in Differential Privacy systems
Johannes Ljung Ekeroth & Markus Pettersson
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Differential privacy (DP) is emerging as a viable solution to release statistical information about a population without compromising data subjects' privacy. A standard way to achieve DP is by adding calibrated noise to the result of some statistical analysis. To account for the impact an individual's data have on the result of an analysis, the noise needs to be calibrated to the maximal change in the observable result of the analysis that occurs as an individual's data changes. This is formalized through the notion of *sensitivity*.

In this work, we construct a small DSL for writing queries on datasets. The DSL is capable of automatically computing the global sensitivity of said queries. Using this DSL, we improve an existing implementation of the MWEM algorithm by stepping away from manual tuning of the sensitivity parameter for every set of queries instead of adjusting it automatically.

The underlying mechanism for computing the sensitivity is a technique for analyzing the range of user-defined functions, implemented as a data generic library in Haskell. The technique works on enumeration types and may be used à la carte in scenarios where range analysis of user-defined functions is desirable.

Keywords: Functional Programming, Generic Programming, Domain-specific languages, Differential Privacy, Data Synthesis

Acknowledgements

First, we want to thank our supervisor at Chalmers and DPella - Alejandro Russo - for his guidance, enthusiasm and good spirit. Second, we want to thank our supervisor at Boston University and DPella - Marco Gaboardi - for sharing his expertise in Differential Privacy and for inviting us to the Boston Differential Privacy Summer School 2022. Third, we thank everyone at DPella for welcoming us at their office, and for all the fika sessions we enjoyed together!

Johannes Ljung Ekeroth & Markus Pettersson, Gothenburg, August 2022

Contents

1

Introduction

The widespread use of machine learning and statistical analysis in many areas of modern engineering relies heavily on the availability and quality of data sets. To ensure both availability and quality, large amounts of data from a diverse group of individuals need to be collected. But a society built on data collecting faces several ethical dilemmas. One of these dilemmas is how to ensure the privacy of the individuals whose data is collected.

Differential privacy is emerging as a viable solution to release statistical information about a population without compromising data subjects' privacy. A standard way to achieve differential privacy is by adding some statistical noise to the result of a query [1]. If the noise is carefully calibrated, data analysts can still reason about the true answer while the privacy of individuals is protected. This principle of differential privacy carries over to data synthesis, or data sanitation, where the goal is to generate anonymous data which explains the statistics you are interested in.

One popular algorithm for deferentially private data synthesis is Multiplicative Weights Exponential Mechanism (MWEM), which in theory is guaranteed to yield near-optimal results on differentially private synthetic data release [2]. MWEM maintains an approximate distribution of the underlying dataset, while iteratively improving the accuracy of this approximation with respect to the private dataset and a set of user-defined queries. A query is defined as the mapping from a dataset to some result, and a collection of multiple queries is called a *workload*.

MWEM is used at DPella[1], a company working with differential privacy techniques, for synthetic data generation. In their current implementation, one of the parameters of the MWEM algorithm, the sensitivity parameter, is hard-coded. In reality, the value of the sensitivity parameter is dependent on the workload. Updating the sensitivity is both

[1] `https://www.dpella.io`

tedious and error-prone, as the programmer can write new queries but forget to update the sensitivity parameter accordingly. An incorrect sensitivity breaks the differential privacy property of the Exponential Mechanism [2], which means that we lose this property in MWEM as well.

1.1 Motivation

To maximise the privacy guarantees DPella can give to their users, we aim to find a way of automatically computing the sensitivity of user-defined functions.

Some differentially private data synthesis algorithms, such as MWEM, can approximate the distribution of an underlying dataset only if the sensitivity of every statistical query is known [2].

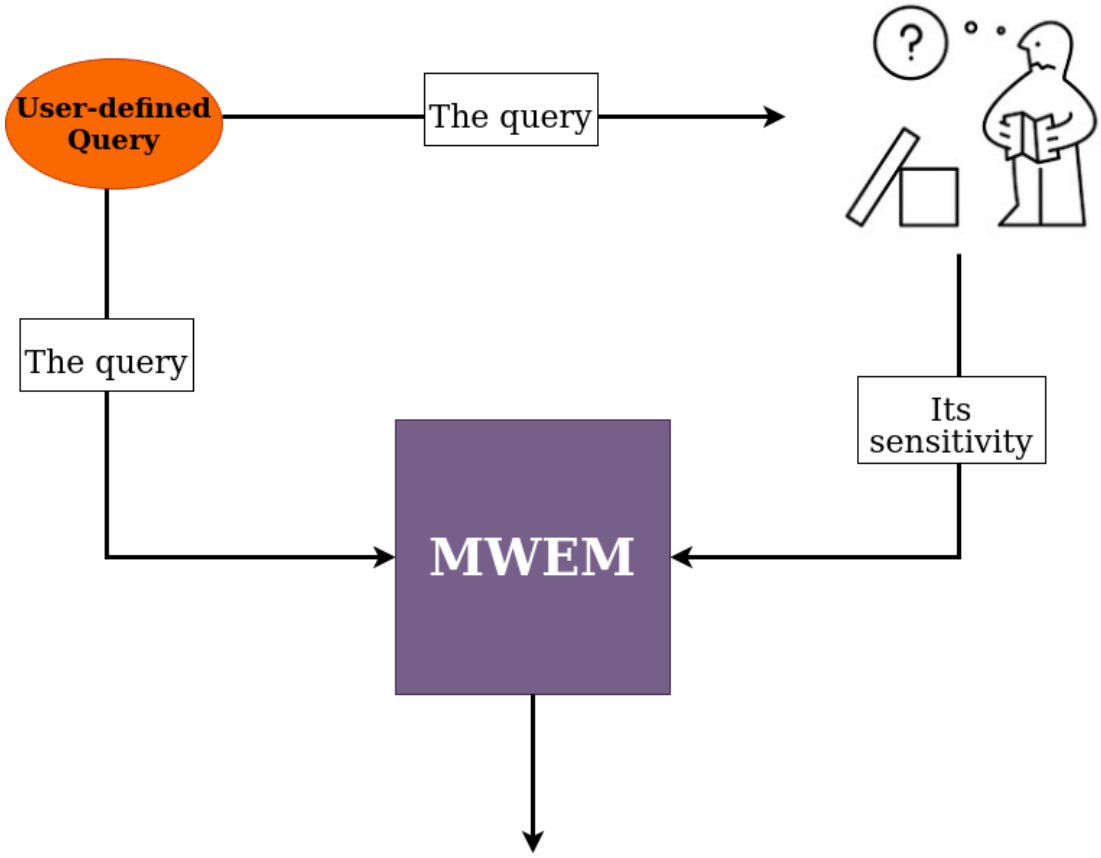

Figure 1.1: Analysing a query to extract its sensitivity, which is then propagated to the MWEM algorithm.

To extract the sensitivity of a user-defined query, we first need to compute all possible values the query can return, which is also known as the *range* of the query. Given the range of a query, the sensitivity is interpreted as the difference between its maximum and

minimum values.

1.2 Goal

Since DPella internally uses the Haskell programming language, our goal is to create a Haskell *domain-specific language* (DSL) capable of computing the co-domain of user-defined functions that pattern-match on enumeration types. Given a function, this DSL will compute an interval of possible outputs. We are looking to implement a type-based mechanism that allows us to identify significant inputs of user-defined functions at runtime.

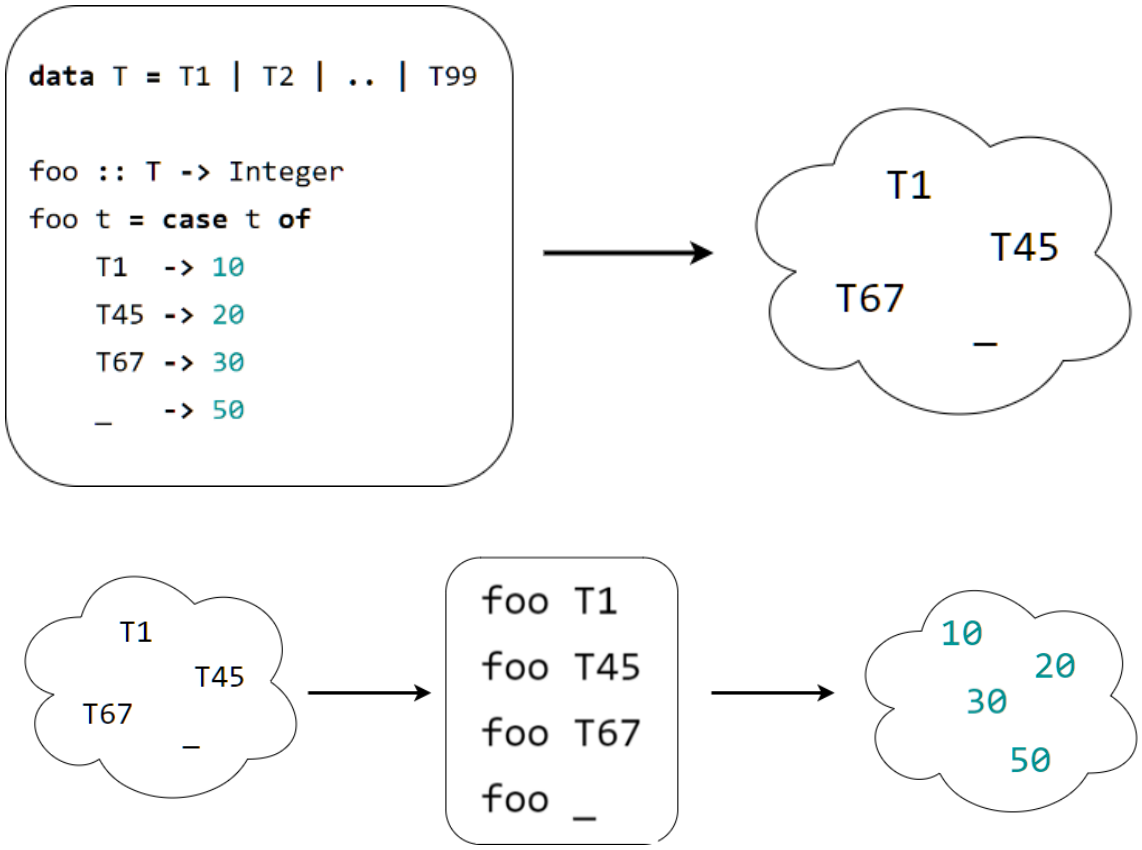

Figure 1.2: Identify explicitly matched inputs and obtain all possible outputs.

In this work, we propose to compute the sensitivity of user-defined queries by automatically computing their range. For that, we implement an internal range-analysis interpreter as a library in Haskell. This interpreter has two main tasks (which are depicted in Figure 1.2):

1. Extract information on explicitly used patterns inside pattern matching functions.

2. Given the information about explicit pattern matches, apply each pattern to the function to obtain a collection of results. This collection will be the range of the function.

The library is integrated into a DSL for creating queries, which are part of the workloads of MWEM. However, the technique is not domain-specific and can be used in many scenarios where range analysis is desirable. It is an explicit design goal that the mechanism must be compatible with Haskell's native `case of` expressions, to make usage convenient for all Haskell developers.

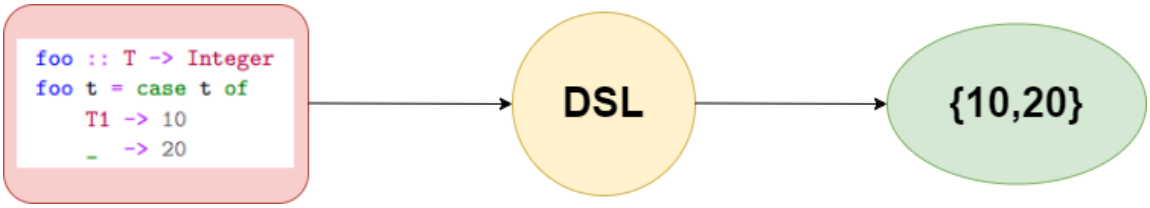

Figure 1.3: Extracting the possible outputs of a user-defined function

1.3 Overview

We start by giving some context to our work in Section 2, where we briefly state what differential privacy means and some related concepts. We also dedicate a portion of Section 2 to some intuition regarding the range of functions. In Section 3 we state the three main problems we face in our work. Section 4.2 covers the technical aspects of our main contribution, which includes the algorithm we developed to solve three main problems stated in Section 3. In section 5 we take use our algorithm in an existing differential privacy system to automatically compute the sensitivity of user-defined functions. In section 6, 7, and 8, we discuss the resulting technique and give examples of reasonable extensions to our work. Finally, we close the report by discussing related work in Section 9.

2

Background

In this chapter, we explain what differential privacy is, how it can be used in real-world scenarios and the difficulties of programming in a differential privacy system. We give a motivation for using Haskell to implement differential privacy systems, which involves a brief discussion on what embedded domain-specific languages (EDSLs) are and how they relate to our work. These discussions are backed up by code examples that emphasize the idea behind differential privacy, EDSLs, and how they can be combined to write programs with certain security properties.

2.1 Differential Privacy

A standard way to achieve differential privacy is by adding some carefully calibrated noise to the result of a query. To protect all the different ways in which an individual's data can affect the result of a query, the noise needs to be calibrated to the maximal change that the result of the query can have when changing an individual's data. This is formalized through the notion of sensitivity. The sensitivity is a measure of the amount of noise needed to ensure differential privacy.

2.2 Queries

Analogous to mathematical functions, a *query* can be viewed as a function f to be evaluated on a dataset. A dataset can be modelled as a vector $x \in D^n$, where each record x_i represents the information contributed by one individual and D^n denotes all datasets with n records [3].

A query written in a general-purpose programming language can be viewed in the following way: Given a vector of records, each containing some kind of data, a query is a function from that vector, call it `Dataset`, to some `Result`. In a more specific example, illustrated

in Listing 2.1, the function `countAllergic` is a query that counts the number of allergic people stored in a dataset.

```
type Table  = [Row]
type Result = Int

data Row = MkRow
  { name     :: String
  , allergic :: Bool
  }

countAllergic :: Dataset -> Result
countAllergic = length . filter allergic
```

Listing 2.1: Queries as functions.

Queries that map records in a dataset to values and sum up the results are a common special case called *linear queries* [2].

2.3 Sensitivity

Sensitivity describes the maximal change in output of a function f from one input to another. Sensitivity is a useful metric when we want to add calibrated noise to the result of a query, which we will see an example of later in this section. As a rule of thumb, the higher the sensitivity of a function, the more noise needs to be added.

In the context of differential privacy and private data analysis, one can make the distinction between *local sensitivity* and *global sensitivity*. However, in this work, we are only concerned with global sensitivity.

For simplicity, we restrict the formal definition of global sensitivity to functions f with co-domain $\mathbb{R}$.

2.3.1 Notation

$||\cdot||$ is the $L1$ norm[1] on $\mathbb{R}$, and denote the distance metric on the outputs of f. Note that both local and global sensitivity may be generalized to other metric spaces than $(\mathbb{R}, ||\cdot||)$ [3].

[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 $|x_1 - x_2|$.

The Hamming distance $d(x, y)$ between two datasets is the number of entries on which x and y differ, and is defined as $d(x, y) = |\{i : x_i \neq y_i\}|$. Two datasets are *neighbours* if they differ in just one individual's data, i.e., $d(x, y) = 1$ [3].

2.3.2 Definition of global sensitivity

For a function $f : D^n \to \mathbb{R}$, the global sensitivity $GS_f(x)$ is defined as:

$$GS_f = \max_{x,y:d(x,y)=1} ||f(x) - f(y)||$$

Global sensitivity considers all neighbouring datasets at once. Concerning the notion of sensitivity laid out in the introduction to this section, the input is made up of the entire domain.

We note that the global sensitivity of f is $GS_f = \max_x LS_f(x)$. Thus, by restricting our work to global sensitivity, we only approximate the sensitivity of a function $f : D^n \to \mathbb{R}$.

2.3.3 Examples

Individual	Allergic
Johannes	False
Markus	True

Table 2.1: A (small) table containing personal data.

Because the column 'Allergic' illustrated in Table 2.1 only assumes two possible values, namely `False` and `True`, the *range* of the query `countAllergic` changes by at most 1 if any individual x_i is changed, added or removed from the dataset. We say that the sensitivity[2] of `countAllergic` is 1. This is true for counting queries in general.

2.4 The range of a function

Looking at Table 2.1 and Listing 2.1, we notice that to approximate the sensitivity of `countAllergic` we first collect the set of possible outputs. We formalize this definition as follows: Given a function $f : \tau_1 \to \tau_2$, the range R_f can be defined as:

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

[2] This is both the local and global sensitivity

The range R_f is a subset of the co-domain of f.

With this in mind, we can transform Table 2.1 into a corresponding Haskell function, and extract its range using the specification given above.

```
allergic :: Individual -> Allergic
allergic Johannes = False
allergic Markus   = True
allergic _        = False
```

Listing 2.2: Function which map values of some record to some result.

In this case, where the dataset is small and the inputs of `allergic` are known, it is easy to extract the range from the function's definition.

$$R_{allergic} = \{False, True\}$$

However, there is an inconsistency between the formal description of R_f and $R_{allergic}$. If the domain τ_1 is infinite, R_f becomes impossible to produce because it would require us to apply the function an infinite number of times. Even if the domain τ_1 is finite but very large, such as the set of every existing individual, R_f becomes hard to produce because it could require us to apply the function many times. For example, imagine the following definition of `allergic`:

```
allergic :: Individual -> Allergic
allergic _ = False
```

Listing 2.3: Function which makes no assumptions about individuals.

Now, the range of `allergic` has changed, which is once again easy to see from the definition of the function.

$$R_{allergic} = \{False\}$$

If the range was to be derived by applying the definition we presented previously, when would we conclude that the function `allergic` only ever takes on the value `False`?

Therefore, while the formal description of R_f is sound, we may need to know which inputs in the domain produce unique outputs for a function f. We will refer to these inputs as the *significant inputs*.

2.5 Haskell, GHC

Haskell is a statically typed, functional programming language with strong typing guarantees [4]. Its type system is expressive and allows for automatic verification of type correctness at compile-time via the most widely used implementation GHC.

In this section, we outline features of Haskell which are ubiquitous when implementing DSLs in a functional programming language. We also cover two language extensions, *Patterns Synonyms* and *Template Haskell*, which will be part of the solution that this report presents.

2.5.1 Pattern matching in Haskell

One of the main selling points of functional programming languages has long been *pattern matching*. Pattern matching allows the programmer to express a program's execution path based on the shape of values (patterns) at runtime. The anatomy of pattern matching consists of **an expression to match on** (matchee), and **an enumeration of patterns** to compare the value of the matchee against.

Haskell makes heavy use of pattern matching to promote a declarative style of programming, where a prominent example is function definitions as in Listing 2.4. Notice how Haskell optionally allows binding variables to be used on the right-hand side of the definition.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 0
fib n = fib (n-1) + fib (n-2)
```

Listing 2.4: Definition of the Fibonacci sequence in Haskell.

The pattern matching feature can also be expressed using `case of` expressions as in Listing 2.5. This example highlights the *wildcard* pattern `_`, which is used as a catch-all pattern without binding the matchee to a new variable name.

```
fib' :: Int -> Int
fib' n = case n of
    0 -> 0
    1 -> 1
    _ -> fib' (n-1) + fib' (n-2)
```

Listing 2.5: Second stab at defining the Fibonacci sequence.

2.5.2 Algebraic data types

Algebraic data types (ADTs) are very common in Haskell. The "Algebraic" property of these data types is that they are created using two algebraic operations analogous to sums and products.

A sum dictates a choice between two types `A` and `B`, and a product is the combination of `A` and `B`.

As an example, Listing 2.6 demonstrates a type `Tree a`, consisting of two choices (a sum): A base construction, `Leaf a`; A combination of two trees (a product), `Branch (Tree a) (Tree a)`.

```
data Tree a
    = Leaf a
    | Branch (Tree a) (Tree a)
```

Listing 2.6: Definition of an Algebraic data type `Tree`.

A special case of sum types is called *enumeration types*, which can be seen in Listing 2.7. These are distinguished by only having constructors which do not take any arguments.

```
data Weekday
    = Monday | Tuesday  | Wednesday | Thursday
    | Friday | Saturday | Sunday
```

Listing 2.7: Definition of an enumeration type `Weekday`.

2.5.3 Pattern synonyms

PatternSynonyms is a language extension to GHC that allows the user to create *aliases* for Haskell patterns. They can be used to define abbreviations of a pattern, bind that abbreviation to a name, and use it as a replacement for the initial pattern.

```haskell
data EmbeddedInfo = E Info

data Info  = T Type | V Value
data Type  = T1 | T2
data Value = V1 | V2

-- No pattern synonym is used. If we want to match
-- on constructors of Value or Type, we need
-- to be explicit about the, currently non-informative,
-- constructors E, T, and V.
explicitMatch :: EmbeddedInfo -> String
explicitMatch (E (T T2)) = "Type T2"
explicitMatch (E (V V1)) = "Value V1"
explicitMatch ...

-- We can use pattern synonyms to create a new abbreviated pattern.
pattern T2_ = E (T T2)
pattern V1_ = E (V V1)

-- The explicit patterns can now be replaced by the new pattern synonyms.
abbreviatedMatch :: EmbeddedInfo -> String
abbreviatedMatch T2_ = "Type T2"
abbreviatedMatch V1_ = "Value V1"
abbreviatedMatch ...
```

Listing 2.8: Using a pattern synonym to abbreviate a verbose pattern.

We can see in Listing 2.8 how pattern synonyms can be used to globally[3] bind verbose patterns to an abbreviated, more accessible version.

2.5.4 Template Haskell

Template Haskell (TH) is a language extension to Haskell, implemented in GHC, capable of analysing and producing Haskell code at compile-time [5]. TH allows for a meta-programming workflow where the user can convert Haskell code to an abstract representation, freely manipulate that representation, and convert it back to Haskell code. This workflow depends on two major features which are both exposed to the user: At compile-time, analyze the abstract representation of Haskell code; generate fresh Haskell code that will be compiled as if it was originally written inside the module by the user. In this work,

[3] Available to all functions. As opposed to function local definitions.

we will only be concerned with the code generation facilities of TH to reduce boilerplate code.

2.5.5 Domain-specific languages in Haskell

A DSL is a language tailored to express programs for a certain problem domain particularly well. The intent when designing DSLs is often to embed domain-specific knowledge as language primitives, which are then exposed to the programmer.

Haskell is often attributed as being particularly well suited for writing DSLs [6]. ADTs is a Haskell feature that can be used to abstractly describe the language constructs of the DSL as plain Haskell data. This data can be composed to form an abstract syntax tree (AST) of DSL language terms. Constructing DSL language terms in this fashion is called *deep embedding*. By leveraging pattern matching to deconstruct these language terms as was done in the function `explicitMatch` in Listing 2.8, it is straight forward give the DSL multiple semantics via run-functions [7].

Embedded domain-specific languages (EDSLs) are DSLs written and executed entirely within the runtime of the host language used to implement the EDSL. The key characteristics of EDSLs are that they fit naturally into the host language environment as they can reuse libraries and language features of the host. They are also trivial to re-use as libraries within the host language.

Implementing an EDSL is arguably easier than implementing non-embedded DSLs since the EDSL by definition reuses a lot of the host language infrastructure, such as the lexer, parser and code generator.

3

Problems

This chapter presents the main problems we need to solve to implement our DSL. Some of the related problems are related to DSL design, where the solutions may be subjective in nature (aesthetics and style, ergonomics of programming, etc.). Other problems belong to the domain of mathematics, and there we have to convey what assumptions and prerequisites that have to exist before we can arrive at a solution.

3.1 Pattern matching

The expressiveness and power that pattern matching offers does come with a few caveats. 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.

Specifically, given two distinct functions, `f1 :: Bool -> String` and `f2 :: Bool -> String`, it is possible that one of them utilizes pattern matching and the other does not. Given how equally they are treated from a developer's perspective, and that branching functions cannot be distinguished from non-branching ones based on the type signature alone, how could one expect to extract specific information about any possibly occurring case expressions. Even worse, if pattern matching is being used inside of a function, how could one extract the exact patterns used inside the case expression(s)?

3.2 Range analysis

Given the inherent difficulty of directly analysing specific patterns used in pattern matching, the difficulty of computing the range of a function increases. Given a user-defined function and its domain, what inputs are we expected to try to predict its range?

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. [8].

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

-- A user-defined function
foo :: T -> Int
foo t = case t of
    T1 -> 1
    T2 -> 2
    _  -> 3

-- Extracting all possible outputs by applying all possible inputs
foos :: [Int]
foos = [foo x | x <- [T1,T2,T3]]
```

Listing 3.1: Extracting all possible outputs by applying a function to all possible inputs.

The biggest caveat with The Trick is that it needs to enumerate the entire domain. This approach does not work on functions with infinite domains, such as the natural numbers. But even for finite domains, The Trick may become infeasible to run due to the limitation in speed and memory of computer hardware if the function itself is expensive to compute.

3.3 Embedding pattern matching

Analysing the `case of` expression in Haskell is fundamentally hard since information about each pattern is not accessible at runtime. Because of this, one needs to somehow keep that information around.

One may try to improve upon The Trick by embedding `case of` expressions as data in an EDSL. The benefit of this is that the language author has very good control over what information will be part of a program's AST. As a simple example, consider the following basic EDSL for expressing pattern matching on Haskell values.

```haskell
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
  Wild :: Alt a

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

evalCase :: CaseDSL p a -> a
evalCase (Case p arms) = apply p arms
  where
    apply :: p -> [(Alt p, a)] -> a
    apply p [] = error $ "Non exhaustive pattern matching in case expression."
    apply p ((a, rhs):as) = case a of
                          Wild -> rhs
                          Pat c -> if p == c
                                   then rhs
                                   else f p as
```

Listing 3.2: A small and restricted DSL for pattern matching on Haskell values.

Implementing an exact and more efficient version of The Trick in this DSL is trivial, since enough information about every `Case` expression is preserved until evaluation, as is shown in Listing 3.3.

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

Listing 3.3: The Trick implemented for CaseDSL.

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

The pattern matching in this language is not even as general as Haskell's native approach.

Every pattern need to implement the `Eq` typeclass, and no variable names get bound as part of the matching process. While solving the main problem of whole-program range analysis, the expressiveness of this DSL is arguably too different from Haskell's native `case of` expressions, making it less intuitive for the general user and therefore not desirable.

4

A mechanism for range analysis on Haskell functions

This chapter outlines the steps we carried out in the research phase of this work. We intend to build up the reader's intuition behind our proposed solution to the range analysis problem, implemented as a library in Haskell.

First, we revisit Haskell's type system and cover generic programming, specifically, the Sums of Products (SOP) view for datatype-generic programming [9]. We deep dive into some abstractions that prove helpful when we want to encode the use of certain values at the type level.

Finally, we explore the possibility of implementing an efficient and accurate version of The Trick using type level programming. The solution is applicable for functions that pattern match on user-defined enumeration types. By leveraging Template Haskell we can expose all the necessary facilities to programmers with minimal manual boilerplate.

4.1 Lifting information to the type level

Many languages with a strong type system, such as Haskell, can infer types when no specific type information is provided by the programmer. Type inference takes place at any stage where the type of a program needs to be deduced to make sure that the program is indeed well-typed.

Consider the function `foo x = x + (2 :: Int)`. There is only one type that `foo` can be assigned to be well-typed, namely: `Int -> Int`[1].

[1] (+) enforces that both arguments have equal types.

Whether or not it was the user's intention, the applied argument must be of type `Int`, or else type checking will not succeed.

Sometimes a programmer omits an explicit type signature for reasons like laziness, feeling like trivial functions do not deserve explicit types, or not being able to figure out the type on their own. However, sometimes type inference can be used as a tool, like any other, to mitigate the tediousness of writing verbose type signatures.

Consider the scenario in Listing 4.1, where we use a `Proxy`[2] type to obtain specific information about constructors explicitly used inside a case expression.

In Haskell, the type of a type is called a *kind*. For example, the kind of `[]` is `* -> *`. With the use of the language extensions[3] *PolyKinds* and *DataKinds*, Haskell allows for user-defined kinds as well as *promoting* data constructors to type constructors. We can refer to a promoted constructor using the `'` syntax. With this functionality, it is possible to replace standard kinds with more informative ones. We utilize this functionality frequently, and show an example of this in Listing 4.1.

```
data T = T1 | T2 | T3

pattern T1_ :: Proxy '['T1,o2,o3]
pattern T1_ = Proxy

 ...

pattern T3_ :: Proxy '[o1,o2,'T3]
pattern T3_ = Proxy

-- foo :: ?
foo t = case t of
 T1_ -> 10
 T3_ -> 20
```

Listing 4.1: Storing explicit type information inside a `Proxy`.

The type of `foo` is `foo :: Proxy [T1,o2,T3] -> Integer`, which successfully gives specific information about what constructors were matched on explicitly. Those constructors

[2] Proxy holds no data and its only purpose is to store type information in its parameter. It is defined as `data Proxy a = Proxy`

[3] A complete list of used language extensions can be found in appendix C.

which were not matched explicitly are kept as *type variables*. However, would this type be written explicitly to the type-signature of `foo`, it would restrict future changes to the function in tedious ways. Consider the following changes to the example given in Listing 9.

```
-- Adding the explicit type signature given by ghci
foo :: Proxy [T1,o2,T3] -> Integer
foo t = case t of
 T1_ -> 10
 T2_ -> 15 -- Adding a new pattern to match on
 T3_ -> 20
```

Listing 4.2: Foo is given the explicit type signature, and the user chose to match on `T2_` as well.

If the type signature is given explicitly as in Listing 10, further changes to the function definition will most likely be rejected. When introducing the new pattern `T2_` the type of `foo` changes, therefore making the type
`foo :: Proxy [T1,o2,T3] -> Integer` invalid. The problem can be solved by, at every occurrence of case expressions matching on `T`, synchronizing the type signature with the definition. For small examples that might be feasible, but for larger examples, it will become a tedious, time-consuming, and monotonous labour. Haskell's type inference does not mind this tedious and monotonous labour, and so it can be used as a tool to accomplish this synchronization automatically.

This example shows two things:

1. It is possible to lift specific information about patterns, used inside case expressions, to the type level

2. This type information can be synchronized with the code doing pattern matching by omitting an explicit type signature and letting type inference do the work.

4.2 The technique

One of the pitfalls of The Trick, and perhaps the most crucial one regarding efficiency, is that it applies *all* the possible inputs of a domain. For large domains, this is an immensely inefficient property, impractical for many real-life scenarios.

The purpose of these techniques is to specialize the trick so that it does not apply more

inputs than necessary to extract the range of possible outputs. We look to only apply the significant inputs when doing range analysis. By doing so, we can accurately assess the range of a function.

4.2.1 Reifying type level information

We demonstrated in section 4.1 how to lift specific information about patterns to the type level with `Proxy`. It would be desirable to be able to turn this information back into some concrete structure. Then, we could programmatically use type information that is otherwise only accessible by the compiler. Luckily the SOP library, written by A.Löh et al., provides precisely this functionality [9].

```
hcpure :: All c os -> Proxy c -> (c a -> f a) -> NP f os
```

The SOP library exports a function called `hcpure`, which takes type level information and transforms it into a concrete term-level structure called *NP* (N-ary Product) by applying a common type class method on each element of some type level list `os`[4].

This function has a rather complex type signature, and we will omit some technical details regarding its internal implementation, but we will showcase it using the following example:

```
-- mempty :: [a]      == []
-- mempty :: Maybe a == Nothing

transform :: (All Monoid os) => NP I os
transform = hcpure (Proxy :: Proxy Monoid) mempty

-- > transform @'[Maybe [Bool], [Int]]
-- > I Nothing :* I [] :* Nil
```

Listing 4.3: Applying `transform` to the type level information yields a concrete structure.

In the function `transform`, we apply `hcpure` to two things: A `Proxy` of a `Monoid` and the method `mempty` from the `Monoid` typeclass.

`All` is a typeclass parameterized by a constraint and a type level list. `All` requires that the constraint applies to every element of the type level list. In this example, every element of the type level list `os` is a `Monoid`.

[4]The product is parameterized by a type constructor and indexed by a type level list. The length of the list determines the number of elements in the product. If the type constructor is `I`, and the i-th element of the list is of type `os`, then the i-th element of the product is of type `I os`. [10]

Given this information, we can read out the function `transform` as: Given a typeclass `tc` and a method `tcMethod`, transform type level descriptions of these `Monoid` types into an `NP` using the given method. So, `transform @'[Maybe [Bool], [Int]]` is given two different type level descriptions, `Maybe` τ_1 and $[\tau_2]$, and applies the corresponding `Monoid` method `mempty` to inhabit the `NP I Nothing :* I [] :* Nil`. As expected, the type `Maybe` τ_1 gets transformed using the `Maybe` instance for `Monoid`, and $[\tau_2]$ gets transformed using the `List` instance.

4.2.2 Working with N-ary Products

Once we have reifyed type level information into a term-level `NP`, we need some structured way of analysing them. In Listing 4.3 the resulting `NP` was specialized to type `I` which is just an Identity constructor wrapping some argument. This is useful when the type level information describes a heterogeneous collection of types, allowing `hcpure` to wrap all results inside the `I` constructor. But what if we wanted to analyze the information inside the result? For example, if we would like to construct a homogeneous list from the result, or have some option to filter out specific parts of the result, it would be beneficial to specialize the `NP` to some other structure than the `Identity` type. Let us make a new transformation called `defined`, and specialize it to the `Maybe` type instead.

```haskell
class Evidence t where
  evidence :: Maybe t

data T1 = T1 deriving Show
data T2 = T2 deriving Show
data T3 = T3 deriving Show

instance Evidence T1 where
  evidence = Just T1

instance Evidence T2 where
  evidence = Just T2

instance Evidence T3 where
  evidence = Just T3

instance {-#INCOHERENT#-} Evidence a where
  evidence = Nothing

defined :: All Evidence os => NP Maybe os
defined = hcpure (Proxy :: Proxy Evidence) evidence

-- > defined @'[T1,T2,Int,Bool]
-- > Just T1 :* Just T2 :* Nothing :* Nothing :* Nil
```

Listing 4.4: Specializing `hcpure` to work on a `Evidence` class and the `Maybe` type.

We create the type class *Evidence* to capture the notion of having an evidence of a specific data type. Namely, if some data type has an explicit instance of `Evidence`, we return its constructor inside the `Just` constructor else we return `Nothing`.

Notice that we need to help GHC here, because the instances overlap. GHC requires exactly one instance to satisfy the constraint it is trying to solve during instance resolution [11]. In Listing 4.4 GHC could pick either the instance `Evidence T1` or `Evidence a` based on the first type list element `T1` during the invocation of `defined`, which means that GHC can not single out just one possible instance given the context. The same reasoning applies for second element `T2` as well. Since there are potentially two matching instances for every resolution when the type variable is instantiated, we tell GHC to pick the more specific

one by marking the most general instance with the *{-# INCOHERENT #-}* pragma[5] [6].

Now the `defined` function returns an `NP` containing `Maybe` types instead of `Identity` types. With this new `NP` at our disposal, we can analyze and distinguish between evidenced, and non-evidenced types applied to the `defined` function.

```
collect :: All Show os => NP Maybe os -> [String]
collect Nil = []
collect (Nothing :* rest) = collect rest
collect (Just v  :* rest) =
    ("Evidence found: " ++ show v) : collect rest

-- ghci> collect (transform @'[T1,T2,Int,Bool]
-- ["Evidence found: T1", "Evidence found: T2"]
```

Listing 4.5: Analysing the internal structure of an `NP`, and use it to construct a list of strings.

Since Haskell will help us lift the information about explicitly matched patterns to the type level (using type inference), together with the `NP` mechanics described above, we have a way of acquiring an analyzable data structure containing information about these patterns. In Listing 4.4 we provided each constructor of `T` as separate data types, `data T1`, `data T2`, `data T3`. If we instead use a common singleton type, we can collect the constructors and put them in the same data type and work with them inside the singleton type instead. We provide an example of this in Listing 4.6.

[5] A directive passed on to the compiler from within Haskell source code.
[6] A more thorough explanation of instance resolution and `INCOHERENT` can be found in appendix A

```haskell
data T = T1 | T2 | T3 deriving Show

data Singleton (d :: T) where
  ST1 :: Singleton (T1 :: T)
  ST2 :: Singleton (T2 :: T)
  ST3 :: Singleton (T3 :: T)

deriving instance Show (Singleton t)

pattern T1_ :: Proxy '[ Singleton T1,o2,o3]
pattern T1_ = Proxy

pattern T2_ :: Proxy '[o1, Singleton T2,o3]
pattern T2_ = Proxy

pattern T3_ :: Proxy '[o1,o2, Singleton T3]
pattern T3_  = Proxy

instance Evidence (Singleton T1) where
  evidence = Just ST1

instance Evidence (Singleton T2) where
  evidence = Just ST2

instance Evidence (Singleton T3) where
  evidence = Just ST3

instance {-# INCOHERENT #-} Evidence x where
  evidence = Nothing

-- A function foo mapping from T constructors to Int
-- Matching on two constructors explicitly
foo t = case t of
  T1_ -> 10
  T2_ -> 20

-- showArgs takes a function foo and collects information about
-- evidenced patterns inside that function.
showArgs :: forall os r. (All Show os, All Evidence os)
         => (Proxy os -> r) -> [String]
showArgs _fun = collect (defined @os)

-- ghci> showArgs foo
--["Evidence found: ST1","Evidence found: ST2"]
```

Listing 4.6: Analysing the internal structure of an NP, and use it to construct a list of Strings.

With `showArgs` in Listing 4.6, we now have a way of extracting what constructors were explicitly matched inside a user-defined function, without mentioning anything about the specific data type `T`.

4.2.3 Constructing unique patterns using N-ary sums

So far, we have successfully demonstrated how we can build type level lists using `NP`, and how to transform the type level list into a runtime representation which we in turn could collect into a list of `String`s.

Notice in Listing 4.6, when the `NP` is transformed into a list of `String`s, the function `showArgs` ignores the argument `_fun :: (Proxy os -> r)`. If we instead transform the `NP` into a list of `(Proxy os)`, we can map the supplied function `_fun` over that list to obtain the actual results of applying the function `_fun`.

The process of transforming an `NP` to a representation matching the patterns reveals two limitations of the current implementation: Proxies do not reflect the length of the type level list, and since they, by definition, are *value-less*, different Proxies cannot be distinguished at runtime. Consider the example in Listing 4.7 that emphasizes the type level limitation:

```
toProxy :: forall os . NP Maybe os -> [Proxy os]
toProxy Nil = []
toProxy (Nothing :* r) = toProxy r
toProxy (Just v  :* r) = (Proxy @os) : toProxy r

-- ghci> :load
-- • Couldn't match type ‘xs’ with ‘x : xs’
--       Expected: [Proxy os]
--         Actual: [Proxy xs]
```

Listing 4.7: Trying to convert an `NP` to a list of `Proxy os`.

The type checker rejects the function `toProxy` because of the recursive calls. The initial type `toProxy :: forall os ...` describes a specific type level list, such as `'[Singleton 'T1, Singleton 'T2,o3]`, which is reflected in the input `NP`: `(Just ST1 :* Just ST2 :* Nothing :* Nil)`. However, when applying `toProxy` to the tail of the original `NP`, the new `NP` no longer reflects the original type level list which results in a type mismatch.

The example in Listing 4.8 bypasses the typing mismatch by telling GHC the type of `os` in the `toProxy` and `buildProxies` functions using the *type equality* operator `~`. But this workaround forces us into another problem. As the example demonstrate, there is no way to distinguish the values from the result of `buildProxies`.

```haskell
toProxy :: forall os o1 o2 o3 . os ~ '[o1,o2,o3]
        => NP Maybe os -> [Proxy os]
toProxy _ = [Proxy, Proxy, Proxy]

-- Expected result if all significant inputs are applied: [10, 20, 30]
foo t = case t of
    T1_ -> 10
    T2_ -> 20
    T3_ -> 30

buildProxies :: forall o1 o2 o3 os r .
                ( All Evidence os
                , os ~ '[o1,o2,o3])
             => (Proxy os -> r) -> [Proxy os]
buildProxies _f = toProxy (defined @os)

-- ghci> buildProxies foo
-- [Proxy, Proxy, Proxy]
-- ghci> map foo (buildProxies foo)
-- [10,10,10]
```

Listing 4.8: Trying to convert an `NP` to a list of `Proxy os`.

It is clear from the previous examples that we need some other representation than `Proxy`, capable of constructing distinct runtime values and keeping the structure of the type level list intact through recursion. Again we turn our attention to the SOP library and a new N-ary structure, `NS` (N-ary Sum).

The `NS`, just like the `NP` structure, is parameterized over a type constructor and some type level list, and comes with two different constructors `Z` and `S`, that can be nested to produce a sum where each element is unique.

```
ghci> let x = Z (I 'x')      :: NS I '[Char,Bool]
ghci> let y = S (Z (I True)) :: NS I '[Char,Bool]
ghci> :t [x,y]
[x,y] :: [NS I '[Char,Bool]]
```

Listing 4.9: Constructing two distinguishable `NS` values.

We see in Listing 4.9 that we can produce two unique representations by nesting `Z` and `S`

and still reflect both to the same type level list. Intuitively, the nested constructors can be read out as (Z)ero and (S)uccessor, and points to the corresponding index in the type level list.

We can adapt our previous attempts at collecting patterns by simply replacing the `Proxy` representations with `NS` representations, and instead of collecting a list of `Proxy`, we collect a list of `NS`.

```
pattern T1_ :: NS I '[ Singleton 'T1,o2,o3]
pattern T1_ = Z (I ST1)

pattern T2_ :: NS I '[o1,Singleton 'T2,o3]
pattern T2_ = S (Z (I ST2))
...

foo t = case t of
  T1_ -> 10
  T2_ -> 20

indexes :: NP Maybe os -> [NS I os]
indexes Nil = []
indexes (Nothing :* r) = map S (indexes r)
indexes (Just v  :* r) = Z (I v) : map S (indexes r)

getIndexes :: forall os r . (All Evidence os)
           => (NS I os -> r) -> [NS I os]
getIndexes _f = indexes (defined @os)

-- ghci> getIndexes foo
-- [Z (I ST1), S (Z (I ST2))]
-- ghci> :t it
-- it :: [NS I '[Singleton 'T1, Singleton 'T2, o3]]
-- ghci> map foo it
-- [10,20]
```

Listing 4.10: Constructing an NS from an NP.

In Listing 4.10, the function `indexes` is now transforming an `NP` to a representation where each element is distinguishable, and when combined with `defined`, gives a list of evidenced singletons inside a type level list (corresponds to explicitly matched patterns in `foo`). Notice how the reified type signature of the patterns lines up with the application of `Z` and `S`.

As noticed in the *ghci* sequence in Listing 4.10, we were able to map the original function `foo` on the index list to obtain the results. Thus, we now have everything needed to transform a user-defined function into a list of possible outputs, limited to enumeration types and without a wildcard. A small, but complete, example is given in Listing 4.11.

```
data T = T1 | T2 | T3 deriving Show

data ST (t :: T) where
  ST1 :: ST (T1 :: T)
  ST2 :: ST (T2 :: T)
  ST3 :: ST (T3 :: T)

class Evidence t where
  evidence :: Maybe t

instance {-#INCOHERENT#-} Evidence x where
  evidence = Nothing

instance Evidence (ST T1) where
  evidence = Just ST1

instance Evidence (ST T2) where
  evidence = Just ST2

instance Evidence (ST T3) where
  evidence = Just ST3

pattern T1_ :: NS I '[ST 'T1, o2, o3]
pattern T1_ = Z (I ST1)

pattern T2_ :: NS I '[o1, ST 'T2, o3]
pattern T2_ = S (Z (I ST2))

pattern T3_ :: NS I '[o1, o2, ST 'T3]
pattern T3_  = S (S (Z (I ST3)))

defined :: forall os . All Evidence os => NP Maybe os
defined = hcpure (Proxy :: Proxy Evidence) evidence

indexes :: NP Maybe os -> [NS I os]
indexes Nil            = []
indexes (Nothing :* r) = map S (indexes r)
```

```haskell
indexes (Just v  :* r) = Z (I v) : map S (indexes r)

trickNoWC :: forall os r. (All Evidence os)
          => (NS I os -> r) -> [r]
trickNoWC f = map f (indexes (defined @os))

foo t = case t of
  T1_ -> 10
  T2_ -> 20
  _   -> 30

-- ghci> trickNoWC foo
-- [10,20]
```

Listing 4.11: First version of an optimized trick.

4.2.4 Targeting wildcards

Until this point, we have successfully built a framework for pattern matching on a representation of enumeration types in a way that is reflected in the type of the pattern match expression. But 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 significant inputs to try to apply.

Given an inferred type level list from a `case of` expression in our framework, we need to generate a constructor that was *not* matched explicitly.

To reason about constructors that were not matched explicitly, we need to be able to discover those constructors and infer how to construct them. The discovery is pretty simple; unmatched constructors have not been reified and thus remain as type variables in the type level list. Consequently, during the transformation in `indexes` from an `NP Maybe os` into a list of patterns `[NS I os]`, all unmatched constructors will coincide with `Nothing` values. Even if we can detect `Nothing`, the identity functor `I` imposes that we need a value of type `os` to construct a value of type `NS I os`. We need a way to infer what constructor each type level element would correspond to if it had been reified, even in the absence of a runtime value.

Let us take a moment to reflect on how patterns are constructed. In Listing 4.11, `I` is used to reflect the type of a singleton value, i.e. `I ST1, I ST2, I ST3`, to the type level list. While it is clear which pattern corresponds to which constructor from the original enumeration type, i.e. `data T`, this bookkeeping is not strictly required to create distinguishable

patterns. Indeed, we can create distinct patterns by composing `Z` and `S` without specifying the value of each singleton. We are free to choose whichever inner value we see fit. The composition of `Z` and `S` enforces correct type-level element reification.

For instance, if we exchange `I` for a new functor `Proxy`, we can replace every occurrence of a singleton value with a `Proxy` value, as is done in Listing 4.12.

```
pattern T1_ :: NS Proxy '[ST 'T1, o2, o3]
pattern T1_ = Z Proxy

pattern T2_ :: NS Proxy '[o1, ST 'T2, o3]
pattern T2_ = S (Z Proxy)

pattern T3_ :: NS Proxy '[o1, o2, ST 'T3]
pattern T3_  = S (S (Z Proxy))
```

Listing 4.12: Patterns using `Proxy` as the functor.

Now, we can forge matches by constructing a pattern that we know has not been matched upon by injecting a `Proxy` value. This is encoded by the Haskell function `addWild`, shown in Listing 4.13, which recurses on the explicit matches until it finds a term that has not been reified.

```
-- Given a set of explicitly matched constructors,
-- add an unmatched constructor to the resulting set.
addWild :: forall os . All Evidence os
        => NP Maybe os -- matched
        -> NP Proxy os -- significant inputs
addWild Nil = Nil
addWild (Nothing :* r) = Proxy :* r -- This is where the wildcard is injected.
addWild (Just x  :* r) = Proxy :* addWild r
```

Listing 4.13: Initial definition of `addWild`.

But this definition of `addWild` does not type check. In the second case of `addWild`, when we coincide with a `Nothing` constructor, the type of `r` is `r :: NP Maybe os`, and as such we are unable to use it as a return value.

Our approach to solving this problem was to create a new functor `Match`, which unifies the idea behind the presence/absence detection of `Maybe` and value-agnostic type reflection of

`Proxy`. The definition of `Match` is given in Listing 4.14. The idea is that `Yes` signify an explicit match on some pattern, while `No` indicates the opposite.

```
data Match a = Yes | No
```

Listing 4.14: Definition of functor `Match`.

The patterns and function definitions have to be updated slightly, which we show in Listing 4.15. Notably, we can use the new `Match` functor in the definition of `defined`, `indexes`, and `trick`, which brings some more uniformity to the solution.

```haskell
data T = T1 | T2 | T3 deriving Show

data ST (t :: T) where
  ...

class Evidence t where
  evidence :: Match t

instance {-#INCOHERENT#-} Evidence x where
  evidence = No

instance Evidence (ST T1) where
  evidence = Yes

instance Evidence (ST T2) where
  evidence = Yes

instance Evidence (ST T3) where
  evidence = Yes

pattern T1_ :: NS Match '[ST 'T1, o2, o3]
pattern T1_ = Z Yes

pattern T2_ :: NS Match '[o1, ST 'T2, o3]
pattern T2_ = S (Z Yes)

pattern T3_ :: NS Match '[o1, o2, ST 'T3]
pattern T3_  = S (S (Z Yes))

defined :: forall os . All Evidence os => NP Match os
defined = hcpure (Proxy :: Proxy Evidence) evidence

indexes :: NP Match os -> [NS Match os]
indexes Nil         = []
indexes (No :* r)   = map S (indexes r)
indexes (Yes :* r) = Z Yes : map S (indexes r)

trickNoWC :: forall os r. (All Evidence os)
          => (NS Match os -> r) -> [r]
trickNoWC f = map f (indexes (defined @os))
```

Listing 4.15: Updated solution using `Match` functor.

After some slight modifications, the types line up correctly and the function `addWild`, shown in Listing 4.16, type checks.

```
-- Given a set of explicitly matched constructors,
-- add an unmatched constructor to the resulting set.
addWild :: forall os . All Evidence os
        => NP Match os -- matched
        -> NP Match os -- significant inputs
addWild Nil = Nil
addWild (No  :* r) = Yes :* r -- This is where the wildcard is injected.
addWild (Yes :* r) = Yes :* addWild r
```

Listing 4.16: Final definition of `addWild`.

Notice how `os` remains unchanged, even though we are now potentially matching on yet another constructor. The implications of this are discussed in section 4.2.5.

Finally, we can construct all significant inputs, which concludes a significant part of creating a more efficient and accurate version of 'The Trick'.

4.2.5 Completing the new Trick

Finally, we give a definition of 'The Trick' that uses type information to build the significant inputs of a function `f`, including wildcard, and applies them to cover the set of possible outputs of `f`.

```
foo t = case t of
  T1_ -> 10
  T2_ -> 20
  _   -> 30

trick :: forall os r . (All Evidence os)
      => (NS Match os -> r) -- ^ Concrete case expression
      -> [r]                -- ^ Possible outputs
trick f = let explicit    = defined @os
              significant = addWild explicit
          in map f (indexes significant)

-- ghci> trick foo
-- [10,20,30]
```

Listing 4.17: Final version of the Trick, including wildcards.

After a wildcard match has been added by `addWild`, one could still not tell the difference between the two values `explicit :: NP Match os` and `significant :: NP Match os` by looking at their types. It is an (expected) consequence of using `NP` with the functor `Match` since the i-th element of the N-ary product `NP Match os` is either `Yes` or `No` independent of the instantiation of the i-th element of `os`. For instance, defining the following value is valid:

```
Yes :* No :* Nil :: NP Match '[o1, o2]
```

It is via the instances of `Evidence` that we attach some meaning to reified terms of `os`, which in turn is only used by `defined`. While the runtime value of `significant` thus deviates from the value communicated by the type level list `os`, we deem this to be okay as this desynchronized type remains local within `trick`. Keeping the value and type of `significant` desynchronized is also needed to match the input type of the user-defined function `f`.

Because of the inclusion of a wildcard pattern, we require the user-defined function `f` to be exhaustive. As alluded to, this is not enforced via the type system. But if the pattern matching inside function `f` is not complete, Haskell will throw a **Non-exhaustive patterns** error at runtime when applying `trick` to `f`. We think this is a reasonable trade-off for two reasons: from a correctness perspective, we know that every input has been considered and handled appropriately and because we have seen this requirement from alternative implementations of the trick [12].

4.3 Improving the technique

Noticeable when looking at a complete example, such as in Listing 4.11, the majority of lines of code are specific to the actual data type, resulting in lots of boilerplate code. This boilerplate code will scale linearly in the size of the target data type, making it tedious to write and impractical to use. To make the technique more accessible, we will automate the process of producing singletons and patterns using Template Haskell.

Furthermore, when looking at the specified instances of `Evidence` implemented in Listing 4.15, we see that they are identical. And most importantly, the type arguments of the type class instances contain all the necessary information to implement the class functions `evidence` and `universe`. With some additional abstractions, we show how it is possible to make these instances general and remove the overhead & necessity of generating many identical instances.

4.3.1 Generalized instances

The `Evidence` class provides the functionality to convert type level information to concrete runtime values via `NP`. It does so by specifying instances for each singleton constructor. From the instances in Listing 4.15, it is painfully obvious that the only thing differentiating any two instances is the instantiation of the type parameter. Thus, there is an opportunity to reduce the amount of boilerplate if we can find a way to unify all explicit matches under the same type reification.

As it turns out, even the explicitness of using type level singletons to represent matches at the type level does not provide any necessary information to the compiler. If the correct type level list element is reified, both `defined` and `indexes` work correctly. Since no particular type is needed, we choose to use the standard Haskell unit type `()`, shown in Listing 4.18.

```
pattern T1_ :: NS Match '[(), o2, o3]
pattern T1_ = Z Yes

pattern T2_ :: NS Match '[o1, (), o3]
pattern T2_ = S (Z Yes)

pattern T3_ :: NS Match '[o1, o2, ()]
pattern T3_  = S (S (Z Yes))
```

Listing 4.18: Updated patterns using `()` for type reification.

With this generalization of the patterns' type signatures, we are only ever concerned with two instances of `Evidence`:

```haskell
class Evidence t where
    evidence :: Match t

-- This instance generalizes explicit matches.
-- Therefore, any occurring pattern will match this instance.
instance Evidence () where
    evidence = Yes

-- This instance generalizes absence of a pattern.
-- Be aware, this instance will only be selected if
-- the type variable has not been instantiated with '()'
instance {-# INCOHERENT #-} Evidence x where
    evidence = No
```

Listing 4.19: Generalized instances of the `Evidence` class.

With the generalization in Listing 4.19, we can convert a type level list of matches to the corresponding concrete patterns without needing an `Evidence` instance for each singleton type. In fact, it allows us to remove singletons altogether, further simplifying the code. But now, we take a small step back and reflect on this simplification. Looking at the type signatures in Listing 4.18, some Haskell programmers may be disturbed by the fact that the connection back to the original enumeration type τ has vanished. Indeed, lacking this piece of information could pose some problems in some scenarios.

Consider the example depicted in Listing 4.20. `foo` passes type checking even tough, conceptually, we matched on constructors from different data types! There is no way to tell patterns of different data types apart from the type signatures alone, if the data types have the same number of constructors.

```haskell
data T = T1 | T2 | T3
data ABC = A | B | C

pattern T1_ :: NS Match '[(), o2, o3]
pattern T2_ :: NS Match '[o1, (), o3]
pattern T3_ :: NS Match '[o1, o2, ()]

pattern A_  :: NS Match '[(), o2, o3]
pattern B_  :: NS Match '[o1, (), o3]
pattern C_  :: NS Match '[o1, o2, ()]

-- !?
foo t = case t of
  T1_  -> 10
  C_   -> 20
  _    -> 42

-- ghci> trick foo
-- [10,42,20]
```

Listing 4.20: The dangers of over-simplifying type reification! Pattern definitions omitted for brevity.

This bug was present in many of the previously proposed solutions, which the keen-eyed Haskeller might have inferred from the example presented in Listing 4.9.

The error can be fixed by introducing a binary proxy type which we call `TTup`, as done in Listing 4.21. Using `TTup` we can associate each pattern with its original type τ. This gives us the added type safety of only matching on values of the same type.

```haskell
data TTup t idx where
  TSnd :: idx -> TTup t idx

data T = T1 | T2 | T3
data ABC = A | B | C

pattern T1_ :: TTup T (NS Match '[(), o2, o3])
pattern T2_ :: TTup T (NS Match '[o1, (), o3])
pattern T3_ :: TTup T (NS Match '[o1, o2, ()])
pattern T1_ = TSnd (Z Yes)
pattern T2_ = TSnd (S (Z Yes))
pattern T3_ = TSnd (S (S (Z Yes)))

pattern A_ :: TTup ABC (NS Match '[(), o2, o3])
pattern B_ :: TTup ABC (NS Match '[o1, (), o3])
pattern C_ :: TTup ABC (NS Match '[o1, o2, ()])
pattern A_ = TSnd (Z Yes)
pattern B_ = TSnd (S (Z Yes))
pattern C_ = TSnd (S (S (Z Yes)))

foo t = case t of
  T1_  -> 10
  C_   -> 20
  _    -> 42

-- ghci> :reload
-- error:
--     • Couldn't match type ‘T’ with ‘ABC’
--       Expected type: TTup T (NS Match '[(), o2, ()])
--         Actual type: TTup ABC (NS Match '[(), o2, ()])
--     • In the pattern: C_
--     ..
```

Listing 4.21: `foo` failing type checking, as expected.

To accommodate the use of the `TTup` type with its `TSnd` constructor, some of the code presented in Listings 4.15, 4.16, 4.17 need to change yet again. Since this modification is trivial and does not affect the algorithm, it has been omitted for brevity.

4.3.2 Reducing boilerplate

In contrast to the generic nature of the `Evidence` class, every new data type τ requires a unique collection of pattern synonyms to construct its correct type level representation. While it would be desirable to not depend on the overhead introduced by implementing large chunks of patterns, there is no obvious way to generalize them. Thus, we instead created a TH pipeline to generate this code automatically. The general structure of these patterns can be seen in Listing 4.22, and a showcase of the result is given at the end of this section in Listing 4.24.

```
data T = T1 | ... | TC | ... | TN deriving Show

-- The general structure of a pattern synonym.
-- The dots specifies parts that need to be specialized
-- for each unique singleton.
pattern T1_ ...

pattern TC_ :: TTup T (NS Match '[... , () , ...])
pattern TC_ = TSnd ( ... S (Z Yes))

pattern TN_ ...
```

Listing 4.22: General structure of pattern definitions.

The details of our TH pipeline are not of any interest with regards to the technique described in section 4.2. Therefore they are omitted here. In short, we have made it possible to replace the example in Listing 4.11 with the code in Listing 4.23:

```haskell
data T = T1 | T2 | T3 deriving Show

$(mkTCBoiler ''T)

class Evidence t where
    evidence :: Match t

instance Evidence () where
    evidence = Yes

instance {-# INCOHERENT #-} Evidence x where
    evidence = No

defined = ...
indexes = ...
trickNoWC = ...
foo = ...
```

Listing 4.23: Reducing boilerplate code.

The function `mkTCBoiler` in Listing 4.23 automatically generate the code specific to the data type `T`. From a user's perspective, since the pipeline described in 4.2 is generic given the automatically produced code, we can do even better. Listing 4.24 shows a potential workflow from a user's perspective.

```haskell
import TheTrick

data T = T1 | T2 | T3 deriving Show

$(mkTCBoiler ''T)

foo = ...
```

Listing 4.24: Working with the technique from a user perspective.

This TH pipeline, together with the generalized instances of `Evidence`, adds up to an expressive and easily accessible workflow for the user.

4.3.3 Conversions between concrete values and patterns

It may be apparent that there should exist a bijection between every enumeration type and the generated patterns. And as we will see during the case study, converting functions that work on the generated patterns to work on the original enumeration type is important in order to integrate our solution smoothly into an existing differential privacy system. With this in mind we defined the following type class in Listing 4.25, which encapsulates this bijection.

```
class Reifyable t os where
  reify   :: t -> MatchT t os
  reflect :: MatchT t os -> t
```

Listing 4.25: Definition of the bijection between some enumeration type and the corresponding, generated patterns.

Instances for these are generated automatically as part of the TH pipeline, but an example instance would look like Listing 4.26.

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

Listing 4.26: Example implementation of the bijection `Reifyable` for an enumeration type `T`.

4.4 Summarizing the technique

Finally, we summarize the technique presented in section 4.2, as well as the improvements of the technique from section 4.3.

In Listing 4.27, we provide an example of the technique from the programmer's perspective, the generated boilerplate code, and the final version of our technique.

```haskell
-- Programmers module
import TheTrick

data T = T1 | T2 | T3 deriving Show

$(mkTCBoiler ''T)

-- Generated code by mkTCBoiler
pattern T1_ :: TTup T (NS Match '[(),o2,o3])
pattern T1_ = TSnd (Z Yes)

pattern T2_ :: TTup T (NS Match '[o1,(),o3])
pattern T2_ = TSnd (S (Z Yes))

pattern T3_ :: TTup T (NS Match '[o1,o2,()])
pattern T3_ = TSnd (S (S (Z Yes)))

-- The technique
defined :: forall os . All Evidence os => NP Match os
defined = hcpure (Proxy :: Proxy Evidence) ev

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

addWild :: forall os. All Evidence os => NP Match os -> NP Match os
addWild Nil       = Nil
addWild (No :* r) = Yes :* r
addWild (p  :* r) = p :* addWild r

trick :: forall os r a . (All Evidence os)
      => (MatchT a os -> r)
      -> [r]
trick f = map f' matches
  where matches = indexes (addWild (defined @os))
        f'      = f . TSnd
```

Listing 4.27: The technique, summarized.

With this, our technique is complete. It is up to a developer to decide how it should be deployed to best fit their needs. Section 5 covers how we used the technique to do automatic sensitivity computations.

5

Case study: Automatic sensitivity computations for MWEM's workloads

In this chapter, we discuss the process of retrofitting a DSL capable of automatic range analysis into an existing implementation of MWEM at DPella. To assess how well our implementation performs in a real-world scenario, we will evaluate the success or failure of the integration with two key metrics in mind:

1. Synthetic data output quality: ability to mimic statistical characteristics and patterns of the original MWEM implementation.

2. Ease of configuration: limited knowledge required during implementation and subsequent reconfiguration for including new patterns.

If at least metric (2) is fulfilled, we have successfully factored out the process of manually tuning the sensitivity parameter for every workload before every run of the MWEM algorithm.

We leave out any metrics that are focused on the performance and applicability of MWEM itself because we deem those to be out of scope for this work.

5.1 Prerequisites

DPella had already prepared a small environment for us to test our range analysis mechanism. The test environment consisted of a data set, a workload on the dataset as well as their MWEM implementation. All three components were accessible as Haskell code, and

we will present their purpose briefly in the following sections.

5.1.1 Adult data set

To evaluate our implementation we use the Adult data set, which is one of the most popular datasets from the UCI Machine Learning Repository [13]. In particular, the dataset is widely used in other papers concerning the evaluation of MWEM [2, 14]. The dataset was acquired from US Census data (1994) and contains more than 30000 records, each with 15 associated attributes. The different attributes are further explained in appendix B.

We limit this case study to queries on 4 out of the available 15 attributes. The 4 attributes of interest are sex, race, work class and age. Sex, race and work class are all discrete attributes, i.e. the values we expect them to take on are distinct and clearly defined. They are modelled as an enumeration data type in Haskell.

The fourth attribute, age, is continuous in the underlying dataset but has been discretized in DPella's test environment. The age attribute is defined as 9 distinct intervals to capture adults within the range from 0 to 99 years, which is seen in Listing 5.1.

```
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
```

Listing 5.1: `Age` modelled as a discrete enumeration type.

5.1.2 Workload

We were given a workload of 50 linear queries that each discriminates on 2 of the 4 attributes of interest[1]. The queries cover all possible domains that can be constructed by combining

[1] An exhaustive list of queries, i.e. a query on every combination of the two attributes, would consist of 400 queries.

the 4 attributes in pairs of 2^2. 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, but we will give examples in section 5.3.

Listing 5.2 depicts how the actual workload is modelled.

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

Listing 5.2: Definition of `Workload` type.

As we can see, the `Workload` is parameterized over a `row`, called a record in the literature. A record in this test suite would be of type `Adult`. For all of our given queries to be bundled in the same workload, we have a way of lifting each query from its 2-tuple input domain to work on the `Adult` data type.

It would be unwieldy to lift every query in this report whenever we need to talk about queries on `Adult`, so we assume that lifting of the given queries happens implicitly.

Finally, we can encode the given workload as Haskell code, which is found in Listing 5.3.

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

Listing 5.3: The given queries as a `Workload`.

[2] Also called 2-way marginals. The concept extends to n-way marginals.

5.1.3 Evaluating MWEM

We have previously stated that MWEM runs on a workload (a set of queries), but the algorithm requires more parameters than that. We want to fix as many of these parameters as possible to construct a fair comparative test between the original implementation and our integration project.

In Listing 5.4 we give the type signature of MWEM, which reveals the input parameters. We give a brief explanation of each parameter and their default value(s) in our test.

```
mwem :: Ord row
     => Epsilon          -- ^ Privacy budget ϵ
     -> Iterations       -- ^ The number of iterations to run MWEM
     -> Workload row     -- ^ The set of queries to work on
     -> [row]            -- ^ Dataset rows, x_i ∈ x
     -> [row]            -- ^ Dataset universe
     -> [(row, Double)] -- ^ An approximate, differentially private distribution.
```

Listing 5.4: The type signature of the MWEM algorithm.

- **Epsilon**: What is referred to as the *privacy budget* in the literature [2, 14]. We will test the integration with standard ϵ values of 0.01, 0.1 and 1 [15].
- **Iterations**: The original paper suggests running the algorithm with 10 iterations [2]. This is not always optimal, but for our use case, it will suffice [14].
- **Workload row**: The set of queries to run MWEM with. We will produce the results using both the given `dpellaWorkload` and our adapted `autoWorkload` presented in section 5.3.
- **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 $0 \leq d(x_i) \leq 1$[3]. From this distribution, one can synthesize an arbitrarily large dataset that respects the

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

characteristics[4] of the underlying dataset.

To evaluate our integration, we will run MWEM on the given `dpellaWorkload` with hard-coded sensitivity as well as our adapted workload where the sensitivity is computed. As a baseline, we will run MWEM on the `dpellaWorkload` 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. 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).

5.2 DSL

With the technique described in section 4.2 at our disposal, we set out to implement a DSL capable of computing the sensitivity of user-defined queries.

5.2.1 The Query DSL

To capture the notion of a query in MWEM, we construct the `Query` DSL. The `Query` DSL is special in that it has an interval semantic, `intervalSem`, which is used to compute a query's sensitivity.

```
data Query t r where
  Single :: (All Evidence os, Num r, Ord r, Reifyable t os)
         => (MatchT t os -> r) -- User-defined query
         -> Query t r
```

Listing 5.5: A Query DSL with a constructor `Single`, that encapsulates a user-defined query

Once a user-defined query has been encapsulated inside the `Single` constructor, we can run it through the pipeline described in Section 4.2 to extract its range.

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

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

intervalSem :: Query t r -> Interval r
intervalSem (Single q) = foldl1 hull (trick q)

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

Listing 5.6: Definition of interval semantics for `Query` DSL.

With the function `sensitivity` in Listing 5.6, we can now compute the sensitivity of a user-defined query inside of the DSL. Since the query on the generated patterns is encapsulated inside the DSL, it is possible to work with the inner structure when defining semantics, while keeping the type of the outer structure intact. This is advantageous because we can keep the entire sensitivity computation separated from MWEM, and the user constructing the queries does not need to know about the underlying technique.

Since the given workload contains queries matching 2 attributes, we introduce a special 2-tuple constructor `Tuple` in the `Query` language[5].

```
data Query t r where
  Single :: ...
  Tuple  :: ( Num r, Ord r
            , All Evidence os1, All Evidence os2
            , Reifyable a os1 , Reifyable b os2)
         => ((MatchT a os1, MatchT b os2) -> r)
         -> Query (a, b) r

intervalSem :: Query t r -> Interval r
...
intervalSem (Tuple q) =
    let ras = trick (curry q)
    in foldl1 hull (concatMap trick ras)
```

Listing 5.7: Adding support for tuples in `Query` DSL.

[5]We leave the job of integrating support for n-tuples to future work.

With the `Tuple` extension shown in Listing 5.7, it is possible to encapsulate queries matching a combination of two attributes in a convenient way.

5.2.2 Example programs

```
-- Module B
import TheTrick

data T = T1 | T2 | T3 deriving Show
$(mkTCBoiler ''T)

-- Module C
import DSL
import B

fooQuery :: Query T Int
fooQuery = Single match
  where match t = case t of
    T1_ -> lit 1
    T2_ -> lit 15
    _   -> lit 30

fooQuery :: Query (T, T) Int
fooQuery = Tuple match
  where match t = case t of
    (T1_, T2_) -> lit 10
    _          -> lit 20

-- ghci> intervalSem fooQuery
-- (1 ... 30)
-- ghci> intervalSem fooQuery2
-- (10 ... 20)
```

Listing 5.8: Constructing two `Query`s, and testing the interval semantics on both constructions.

By restricting the results of the DSL to numerical values, we can re-use one of the many libraries for Haskell which define arithmetic on intervals when defining the interval semantics of the DSL[16, 17, 18]. We chose `Numeric.Interval` for two particular reasons: it defines infimum, supremum and the complex hull between two intervals [16]; we did not need support for open intervals or rational numbers, which is offered by other, more

featureful interval libraries [18].

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 said functions.

5.3 Integration

As mentioned in Section 5.1.2, the current implementation of `mwem` uses a workload where the sensitivity is manually set. Listing 5.9 gives a few examples of the queries inside that workload[6].

```
data Sex = Male | Female
data Race = White | Black | Asian | Eskimo | Other
data Workclass = Private |  ...  -- See Appendix B for remaining definitions.
data Age = Teens |  ...

query1 :: (Sex, Race) -> Double
query1 = \case
  (Female, Eskimo) -> 1.0
  _                -> 0.0

query2 :: (Sex, Workclass) -> Double
query2 = \case
  (Male, Private) -> 1.0
  _               -> 0.0
```

Listing 5.9: Examples query of the `dpellaWorkload` from Listing 5.3.

Compare these given queries to the equivalent queries in the `Query` DSL, as depicted in Listing 5.10.

[6]The category `Eskimo` is derived from the race classification 'Amer-Indian-Eskimo' present in the Adult data set. The use of 'Eskimo' can be seen as derogatory, which is why we choose to point out its use in this context.

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

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

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

Listing 5.10: Adapting an existing query from the workload to the `Query` DSL.

Once adapted versions of the given queries haven been encapsulated inside the `Query` DSL, we need to construct a new workload. When constructing a workload we also need to include the sensitivity parameter, which can now be done automatically through the `sensitivity` function. Thus, we give a way of constructing a workload by passing a list of `Query`s, and the sensitivity computation takes place under the hood. The construction is given in Listing 5.11.

```
constructWL :: [Query Adult Double] -> Workload Adult
constructWL qs = let maxSens = maximum $ map sensitivity qs
                     funs    = map extractFun qs
                 in Workload funs maxSens
  where
    extractFun :: Query a r -> (a -> r)
    extractFun (Single q) = \a -> q (reify a)
    extractFun (Tuple q)  = \(a,b) -> q (reify a,reify b)


-- lifting of queries is omitted here
autoWorkload :: Workload Adult
autoWorkload = constructWL [query1, query2]
```

Listing 5.11: Combining the construction of a workload with a sensitivity computation for each query.

All that remains is to replace the old workload with the newly adapted version, as in Listing 5.12

```
-- The algorithm used to run with the original workload
mwem epsilon iterations dpellaworkload ... = ...

-- The algorithm now runs with the adapted workload
mwem epsilon iterations autoWorkload ... = ...
```

Listing 5.12: Running MWEM with a new workload, where a sensitivity is computed automatically.

5.4 Results

In this section, we present a few results of running the MWEM algorithm, both with the previous workload input and our new adapted workload. Each figure shows the result of running each query to the raw data and two sets of synthetic data.

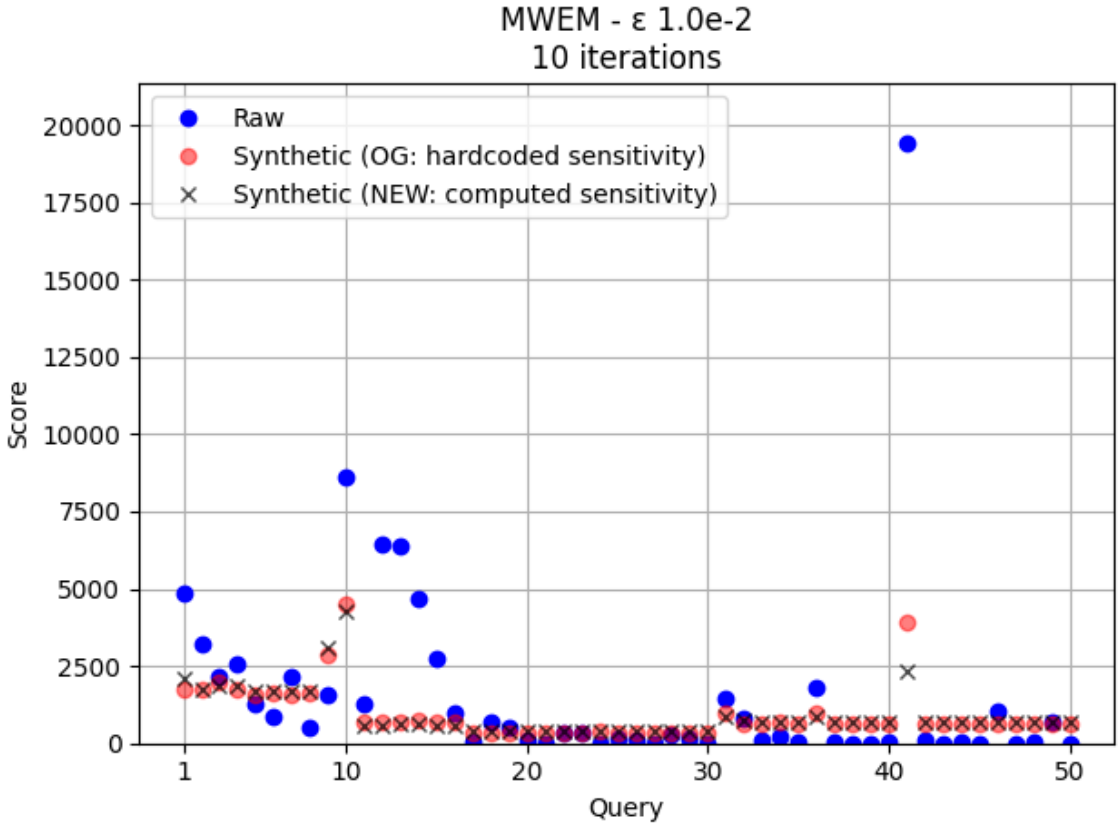

Figure 5.1: Result of running the MWEM algorithm with $\epsilon = 0.01$

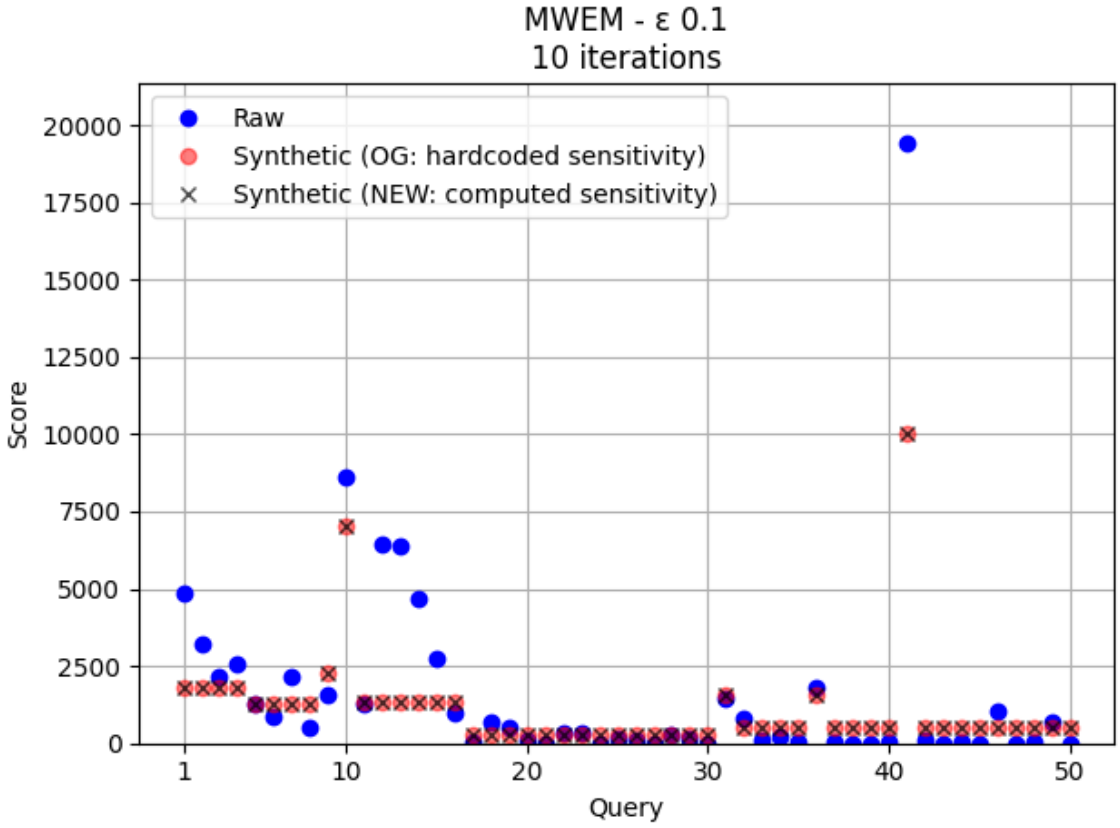

Figure 5.2: Result of running the MWEM algorithm with $\epsilon = 0.1$

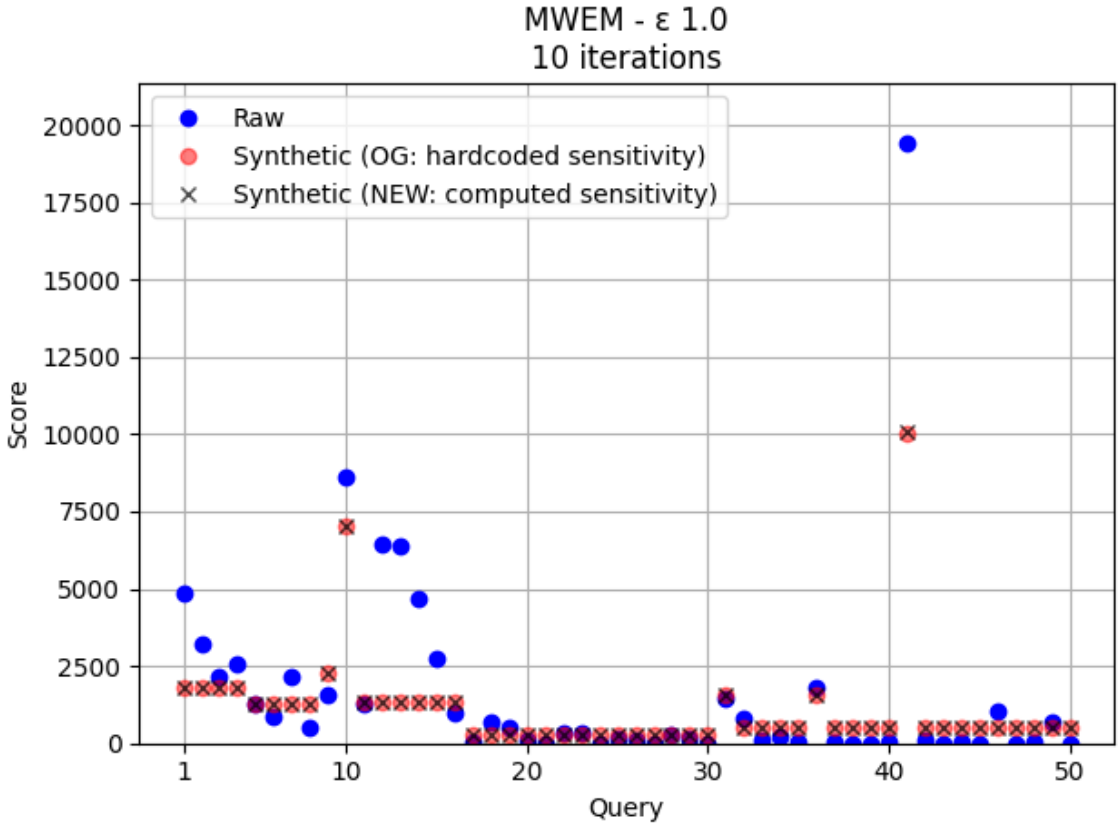

Figure 5.3: Result of running the MWEM algorithm with $\epsilon = 1.0$

The information of interest in these figures is how well the new solution (marked with a cross) lines up with the old solution (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 our adapted workload.

The blue circles represent the result of applying the queries to the raw data. However, recall the metrics outlined in Section 5 that emphasizes that we in this work are not interested in how well the synthetic data mimics the raw data.

We conclude that we have fulfilled both metrics.

The quality of the synthetic data, when generated with our adapted workload, was not significantly different than had been synthesised using the existing workload. Thereby, we

have fulfilled metric (1).

We also conclude that we fulfilled metric (2), since the additional cognitive overhead of a programmer that will write queries in our DSL is low. There are primarily three constructs that they would need to be aware of: Using `_` constructors instead of the regular data constructors, wrapping each query in the `Tuple` constructors and creating the `Workload`'s through `constructWL`. The queries in our DSL look similiar to how they would have been written before, with the added benefit that the sensitivity never can get outdated.

6

Conclusion

This report has presented a data-generic approach to analysing explicit pattern matches in functions working on enumeration types. It was accomplished by reflecting explicitly matched patterns in the function's type signature, taking advantage of GHC's advanced unification mechanisms. The function's type signature information could then be recursed over to construct the appropriate runtime values to feed back into the analysed function.

Using this approach, we developed a DSL capable of interpreting ranges of functions on enumeration types and subsequently integrated this DSL as a query language for a data synthesis algorithm in an existing differential privacy system. The integration successfully factored out the manual labour of setting the sensitivity parameter of workloads used in MWEM, while inquiring minimal runtime overhead[1]. Since no DSL internals are present in the MWEM implementation, the programmer is free to choose between writing queries in our DSL, or in the way they previously did[2].

While our DSL was designed with differential privacy systems in mind, the technique described in Section 4.2 is not domain-specific. Therefore, this technique can be used in other scenarios where co-domain investigations are desirable.

Our technique is successful in its task, but there are limitations. In its current state, it can only work on enumeration types. For large enumeration types, compile-time increases due to the boilerplate code generated by TH. These shortcomings are discussed more thoroughly in Section 7.

[1] The overhead that exists is calculating the range and converting concrete values into generated patterns at runtime.

[2] At the loss of automatic sensitivity computation.

7

Discussion

In this chapter, we reflect on our implementation. In particular we discuss its pain points and things to improve. Finally, we close the discussion by bringing up some ethical implications, and our stance on them.

7.1 Limitations

In this section we state some aspects of the solution that could be considered limitations. Limitations differ widely depending on how the technique is used, and for what purposes. Therefore, the different aspects discussed in this section strongly connects to the restrictions we found when doing our case study.

7.1.1 ADT support

As for now, our solution only supports range analysis using enumeration types, and every unique construction of some type need to be represented as its own constructor. For example, consider the example in Listing 7.1 where we represent the natural numbers and arithmetic expressions on those numbers as data types.

```haskell
data Nat = Zero
         | Suc Nat

data Expr = Lit Nat
          | Add Expr Expr

one :: Nat
one = Suc Zero

two :: Nat
two = Suc (Suc Zero)

oneplustwo :: Expr
oneplustwo = Add (Lit one) (Lit two)
```

Listing 7.1: Representing arithmetic expressions using data types.

With the representation of `Nat` and `Expr`, we can represent an infinite number of arithmetic expressions by combining small representations into larger ones. Recursive types let us construct arbitrarily large values, and product types let us combine two or more of such values to construct larger ones. However, with merely enumeration types we are required to represent each expression in its entirety as a unique constructor. Representing large domains in this fashion is impractical, and for infinite domains, it is impossible. Listing 7.2 illustrates this restriction, by attempting to convert the data types in Listing 7.1 to enumeration types.

```
data Nat = Zero | One | Two | Three | Four | ...

data Expr = LitOne
          | LitTwo
          | LitThree
          | AddOneTwo
          | AddTwoThree
          | ...

one :: Nat
one = One

two :: Nat
two = Two

oneplustwo :: Expr
oneplustwo = AddOneTwo
```

Listing 7.2: Each arithmetic expression is created with a unique constructor.

7.1.2 Threading type level lists

The technique depends heavily on the usage of pattern synonyms, which unify to reveal constructor information at the type level. In order to combine the types of these patterns into a unified type, we utilize type level lists. The type level list describes, for each constructor in the enumeration type, if a constructor was explictly matched or not. When the number of constructors for some data type increases, so do the type level list encapsulated in each pattern synonym and with the limitation described in 7.1.1 in mind, it is fair to assume that these data types can become rather large, thus generating large type level lists.

```
data T = T1 | T2 | ... | T99

pattern T1_ :: TTup T (NS Match '[(),o2,o3, ..., o99])
pattern T1_ = ...
```

Listing 7.3: A pattern synonym for a type with 99 constructors need to construct type level lists with 98 type variables.

While this may seem like a non-issue, these lists threaded throughout the entire technique consist of lots of overhead information. In cases where only pattern `T1_` from Listing 7.3

is explicitly matched in some function, the type level list contains unnecessary information about 98 constructors. The advantage of this approach is that GHC can unify these lists automatically, but since the code needs to be generated and compiled, the overall compile-time increases for each introduced data type.

7.1.3 Increased compile-times

As the data types grow, so does the generated boilerplate code and the compile-time overhead. There is a long-standing issue related to compiling large generics, which affects us via the dependency on SOP[1].

However, we note that data types upwards of 200 constructors are totally feasible, as is shown in Figure 7.1. For instance, that number of constructors would be sufficient to represent all the countries in the world. The current compile-times are measured using GHC 8.6.5, a AMD Ryzen 1950x CPU and 32GB 3000mhz RAM.

[1]The issue is actively being worked on: Issue #5642 on GHC's official GitLab issue tracker.

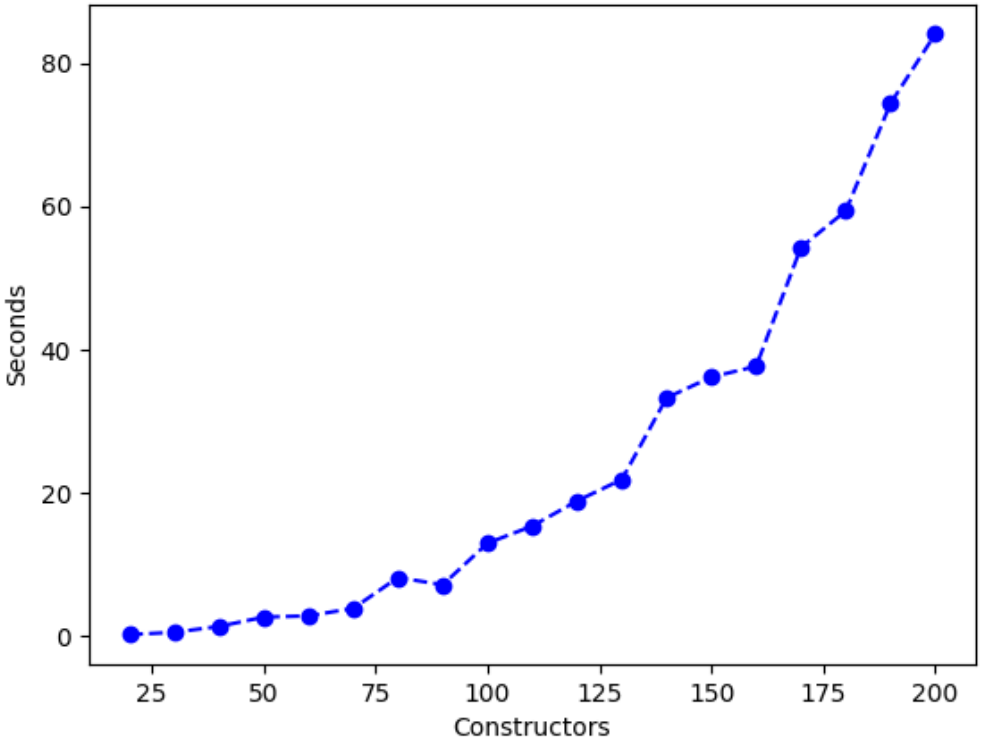

Figure 7.1: Correspondence between number of constructors and compile-time.

Looking at Figure 7.1 how the compile-time increases, it is fair to assume that representing large data types, given that all domains must be enumeration types, will be practically impossible. For example, the Swedish transport administration reports 7 536 309 registered vehicles in Sweden year 2016 [19]. Representing that domain using enumeration types would be vastly impractical. Although such a representation may never be desirable, the limitation exists and should be mentioned. We show the known compile-time correspondence using real numbers in Table 7.1.

Constructors	compile-time (Seconds)
20	0.24
30	0.56
40	1.37
50	2.68
60	2.85
70	3.90
80	8.20
90	7.20
100	13.00
110	15.40
120	18.90
130	21.90
140	33.30
150	36.20
160	37.70
170	54.30
180	59.50
190	74.40
200	84.10

Table 7.1: Correspondence between number of constructors and compile-time

7.1.4 Separation between concrete types and their synonyms

As stated previously, the pattern synonyms play a central role in the technique. Because they need to be used as a replacement for original data type constructors, we have a separation between the source of type-level information and the information itself. When users want to write functions that pattern match on some data type `data T = T1 | T2`, what they do instead is to write functions pattern matching on the generated synonyms `T1_` or `T2_`. It is not necessarily an intrinsic restriction on the technique, but it can be a source of confusion when many different data types and functions are involved.

7.2 Ethics

Communicating the benefits and risks associated with differential privacy is notoriously hard. If you tell someone without knowledge of statistics that differential privacy does not turn non-private data into private data but that it preserves the privacy of individuals on statistical aggregates (based on some parameter ϵ), they would most likely be overwhelmed

[20]. But simplifying the technology to promote the benefits of differential privacy to a point where the 'data utility versus privacy' aspect is compromised may play down the risks too much. [21].

Even if intricate aspects of differential privacy, such as different parameters and noise, could be communicated well to a broad audience, it is questionable how much of an impact it would have on the level of trust the users have for the company using differential privacy.

> Through large-scale online surveys, we investigate whether explanations of differential privacy that hides important information about algorithm parameters persuade users to share more browser history data. Surprisingly, we find that the explanations have little effect on individuals' willingness to share data; in fact, most people make up their minds about whether or not to share before they even learn about the privacy protection. [21]

This study indicates that you either need an audience which already trusts you. Or perhaps you need to make a compelling sales argument for differential privacy which does not include detailed parameter choices.

An example of a company that boasted about its efforts in preserving its customers' privacy through differential privacy is Apple. In a self-published report on how Apple uses differential privacy, they simultaneously try to convince their customers that the affected data will remain anonymous when subject to analysis.

> It [local differential privacy] is a technique that enables Apple to learn about the user community without learning about individuals in the community. Differential privacy transforms the information shared with Apple before it ever leaves the user's device such that Apple can never reproduce the true data. [22]

It turns out, as stated in [23], that the privacy preservation was not as strong as initially claimed. There is some privacy loss going on in their data analysis, but Apple never specified the different parameters used to determine this privacy loss. Without giving the audience enough transparency regarding the implementation and parameters of the privacy mechanism used, it is hard to review and determine how well a deployed mechanism works.

By releasing our report to the public, we give external parties the ability to review and comment on the design of our library. We would also like the source code to be released eventually, such that external parties can verify that there are no major discrepancies between the design outlined in this report and the source code. While our core library is not strictly tied to differential privacy systems, it was conceived because of a need to

automate the approximation of sensitivity in them. A bug or soundness-hole in the core library could lead to weakened differential privacy guarantees of the MWEM algorithm at DPella. Not only would this be dishonest to their customers, but it could also risk leaking the individuals' private information in subtle ways.

We acknowledge that no matter the amount of source code and design verification, the software that DPella run is ultimately in their control. This means that DPella needs its users to trust the company, not their software [24].

Transparency is one way of gaining trust, but transparency is of little value if the subject of transparency is hard to grasp or ill-communicated. This is why we suggest that DPella make a serious effort to release a whitepaper detailing their thoughts on the promise and troubles of differential privacy, and how they use the technology in their systems. In particular, what differentially private algorithms they use, the relevant parameters (ϵ,δ ...) and their corresponding values could be conveyed through such a report. In the end, trust is both a reflection of how much the users trust DPella as well as DPella's technology.

8

Future work

In this chapter, we present some additions and improvements that we think would make our core library even more useful. The suggested areas to work on would bring new features to the core library and lower the overhead of including it in an existing project.

8.1 Add support for more complex ADT structures

One limitation of our solution, and perhaps the most restricting one for general scenarios, is that it only supports enumeration types such as `data T = T1 | T2 | T3`. While it is possible, in theory, to represent many types as enumeration types, it starts to become tedious when the domain of the type is large, and outright impossible when the domain is infinite.

```
data IntEnum = Zero | One | Two | ...
data MaybeInt = INothing | JustZero | JustOne | ...
```

Listing 8.1: Representing Haskell types as user-defined enumeration types.

Not only is the example in Listing 8.1 painful to read, but also impractical. The utility of the technique, as presented in Section 4.2, would increase if it supported more complex ADT constructions, such as product types and recursive types.

```
data IntEnum = Zero | Suc IntEnum
data MaybeInt = INothing | IJust IntEnum
```

Listing 8.2: Product types and recursive types makes it possible to represent more useful types.

8.2 Reflect wildcards at type level

Recall in Section 4.2.5 that a local desynchronization occurs between the concrete NP and the type level list. While this inconsistency does not impose any problems for the technique itself, it could restrict other use cases where type information regarding the wildcard ought to be explicit throughout the entire range analysis.

Information regarding wildcards could, as future work, potentially be embedded into the type level information for the user-defined functions. For example, if enforced exhaustiveness is not desirable, one solution is to create a general wildcard pattern synonym to catch the employment of a wildcard. Listing 8.3 provides an example of this.

```
data ABC = A | B | C
data T   = T1 | T2 | T3

pattern WildCard :: TTUp t (NS Match (() : os))
pattern WildCard = TSnd (Z Yes)

pattern A_ :: TTup ABC (NS Match '[wild,(),o2,o3])
...

pattern T2_ :: TTup T (NS Match '[wild,o1,(),o3])
...

abcfoo :: TTup ABC (NS Match '[(),(),o2,o3]) -> Int
abcfoo = \case
  A_       -> 10
  WildCard -> 20

tfoo :: TTup T (NS Match '[(),o1,(),o3]) -> Int
tfoo = \case
  T2_      -> 10
  WildCard -> 20
```

Listing 8.3: Reflecting a concrete `WildCard` pattern to type level.

This type level reflection will integrate gracefully with the current state of the technique, and the wildcard will act as any other pattern. However, this kind of implementation strays further away from native case expressions as the usage of Haskell's `_` becomes redundant (but still non-exhaustive from Haskell's perspective). A more direct solution would be to try to reflect the correct type level information, without introducing a new representation

of wildcards. For example, if the _ is used inside the case expression, any constructor not matched will be explicitly represented at the type level.

```
abcfoo :: TTup T (NS Match '[(),(),o3]) -> Int
abcfoo = \case
  A_ -> 10
  _  -> 20

tfoo :: TTup T (NS Match '[(),o2,()]) -> Int
tfoo = \case
  T3_ -> 10
  _   -> 20
```

Listing 8.4: The usage of wildcard has made non matched patterns explicit in the type level list.

Listing 8.4 shows a theoretical example where patterns have been reflected as if explicit, even though they currently are not reflected if a wildcard occurs.

8.3 Improve generalization of pattern synonyms

Currently, our technique relies heavily on the usage of specified pattern synonyms. The type of these patterns is crucial for the technique to work, but they are tedious to write. And by generating them automatically, we depend on code-generating libraries such as Template Haskell. In that sense, the pattern synonyms are the least generalized part of our solution. We leave it as future work to explore the possibilities of improving these pattern synonyms to make them more general. It might be already that the core of our solution does not allow for generalized patterns, but there are still possible improvements. For example, the type level list may not necessarily be passed around in its entirety throughout the entire range analysis process. Exploring the possibilities of composing several `NS`s could make way for a solution where large type level lists are not threaded through the entire execution.

8.4 Branching and nested cases

The DSL design given in section 5 does not utilize any form of interval semantics, but purely focuses on encapsulating the technique. The DSL could be extended with interval semantics so that several range computations can be combined or sequenced as in Listing 8.5.

```
-- y is defined elsewhere, but has the range [3, 7]
query t =
    let x = some_computation t -- x has the range [10, 15, 20]
    in if x > 10 then x else y

some_computation :: Query T Int
some_computation = \case
    T1_ -> 10
    T2_ -> 15
    _   -> 20
```

Listing 8.5: Example programs where interval semantics could be useful.

The range of `query` is definitely dependent on some computation on `x`, but only conditionally dependent on `y`. Consider the following scenarios:

1. If `x` is 10, we only consider the else-branch of the continuation. The range of `query` is reduced to [3, 7].

2. If `x` is 15, we only consider the if-branch of the continuation. The range of the query is reduced to [15].

3. If `x` is 20, we once again only consider the if-branch of the continuation. The range of the query is reduced to [20].

We believe that it would be useful to define semantics such that while computing the range, it accounts for conditional branching inside of queries and report the actual range for the considered branch.

Also, since the type level list representation is constructed using units and type variables only, there is currently no support for nested case expressions. Although a nested case expression could be represented as a pre-defined n-ary tuple, it would reduce tediousness if nested cases could be used instead.

For future work, adding these features will allow for more expressive range computations, as well as removing the dependency on n-ary tuples to represent sequences of computations.

9

Related work

In this chapter, we bring up some related work which either inspired our solution, our are interesting in their own right. We also dedicate a portion of this chapter to other areas than differential privacy where our findings may be applicable.

9.1 Calculating sentivity

Since MWEM, and many other random noise differential privacy mechanism, are parametric in the sensitivity of queries, accurately estimating their sensitivity is of utter importance. As such, there exists plenty of previous work on calculating the sensitivity of functions/queries for the purpose of differential privacy applications.

Airavat[25] was an early example of a distributed MapReduce[26] implementation which kept track of each computation's sensitivity. It could thus put an upper bound on the privacy loss across an entire computation, which may consist of many composed mappers and reducers. The caveat is that some user need to manually assign a range of possible values that every mapper produce, as this information is not derived from the mappers themselves.

There exists multiple, but distinct, papers [27, 1] which show another approach at computing the sensitivity of functions by obtaining proofs of sensitivity at the type level. The solution of [27] is similar to our work in the sense that it is DSL based, but differs vastly in the sense that that entire sensitivity computation is present at the type level. Both [27, 1] present combinators which embed the notion of composition from the start. Composition is something that needs to be solved separately for our mechanism, which is why it is listed as future work.

While our technique is closely related to functional languages and programming language

techniques, there are papers such as [28] that covers differential privacy and sensitivity computations in the context of a database and its formal query language, which indeed is a natural setting for differential privacy. The authors of [28] discuss privacy concerns related to database operations and propose "a method to compute a bound on the sensitivity of a query in relational algebra in a compositional way".

9.2 Case analysis in other domains

We are optimistic that the core technique that lifts explicitly matched patterns to the type level could be proven useful in other domains than differential privacy. Two projects that influenced this work the most are code-emitting Haskell EDSLs: Haski, and Accelerate [12, 6]. Both papers aim to embed Haskell's pattern matching facility in their language, with the ultimate goal of providing programmers with as many features of native Haskell as possible.

They were both able to do it but in different ways. Accelerate built a DSL specifically for pattern matching, which they then use internally in the larger Accelerate project. Within the pattern matching DSL, every pattern is captured as Haskell data, which can be applied to the case expression to generate all outputs. Haski also provides an embedded facility for lifting case-expressions to the EDSL. To analyse this expression, Haski applies the Trick. It generates the entire input domain and sequentially applies it to the case-expression.

The fortunate effect observed in Accelerate was that embedded case-expressions now could be compiled to LLVM-IR switch-statements. Compare this with compiling branching code to a chain of if-then-else expressions, which was the case previously. Being able to compile embedded case-expressions to corresponding switch-statements is something that Haski is also able to do. However, the caveat is that the compiled switch-statements exhaustively cover all inputs in the domain of the case-expression, rather than just the possibly few relevant inputs to the case-expression. The consequence is that every case-expression on any sizeable enumeration type blows up in size in the generated code, regardless if the programmer did explicitly match the entire input domain or not. Generating less code is not only desirable for the sake of keeping the size down, but is also beneficial for increasing the verifiability of the generated code. The argument is that it is easier for a human to verify that the code generation is correct if the correspondence between the case-expression in Haskell and the generated code is as close to 1:1 as possible. It is easier to verify that a case-expression that use n explicit matches was compiled correctly if the generated switch-statement only acts on n distinct cases.

Possibly, Haski and similar projects could benefit from our approach, but what we want to highlight in this section is the broader picture. With our core technique, we have a

facility for accurately producing the *minimal set* of distinct outcomes of pattern matches on enumeration types. If The Trick, as presented by N. Jones et al., is a shotgun, our technique behaves more like a sniper rifle.

Bibliography

[1] Jason Reed and Benjamin C Pierce. "Distance makes the types grow stronger: A calculus for differential privacy". In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. 2010, pp. 157–168.

[2] Moritz Hardt, Katrina Ligett, and Frank McSherry. "A simple and practical algorithm for differentially private data release". In: *Advances in neural information processing systems* 25 (2012).

[3] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. "Smooth sensitivity and sampling in private data analysis". In: *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. 2007, pp. 75–84.

[4] Simon Marlow et al. "Haskell 2010 language report". In: (2010).

[5] Tim Sheard and Simon Peyton Jones. "Template meta-programming for Haskell". In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. 2002, pp. 1–16.

[6] Trevor L McDonell, Joshua D Meredith, and Gabriele Keller. "Embedded Pattern Matching". In: *arXiv preprint arXiv:2108.13114* (2021).

[7] Josef Svenningsson and Emil Axelsson. "Combining deep and shallow embedding of domain-specific languages". In: *Computer Languages, Systems & Structures* 44 (2015), pp. 143–165. URL: `http://www.cse.chalmers.se/~josefs/publications/deepshallow.pdf`.

[8] Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Jan. 1993. ISBN: 0-13-020249-5.

[9] Edsko de Vries and Andres Löh. "True Sums of Products". In: *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*. WGP '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 83–94. ISBN: 9781450330428. DOI: `10.1145/2633628.2633634`. URL: `https://doi.org/10.1145/2633628.2633634`.

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

[11] GHC Team. *6.8.8. Instance declarations and resolution*. 2020. URL: `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/instances.html` (visited on 05/18/2022).

[12] Nachiappan Valliappan et al. "Towards Secure IoT Programming in Haskell". In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. Haskell 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 136–150. ISBN: 9781450380508. DOI: `10.1145/3406088.3409027`. URL: `https://doi.org/10.1145/3406088.3409027`.

[13] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: `http://archive.ics.uci.edu/ml/datasets/adult`.

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

[15] Cynthia Dwork, Aaron Roth, et al. "The algorithmic foundations of differential privacy." In: *Found. Trends Theor. Comput. Sci.* 9.3-4 (2014), pp. 211–407.

[16] Edward Kmett. "intervals: Basic interval arithmetic". In: *Available online https://hackage.haskell.org/pac* (2021).

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

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

[19] Swedish transportation administration. `https://www.transportstyrelsen.se/sv/vagtrafik/statistik/Fordonsstatistik/2016/fordonsstatistik-december/`.

[20] Rachel Cummings, Gabriel Kaptchuk, and Elissa M Redmiles. "" I need a better description": An Investigation Into User Expectations For Differential Privacy". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 3037–3052.

[21] MARY ANNE SMART, DHRUV SOOD, and KRISTEN VACCARO. "Understanding Risks of Privacy Theater with Differential Privacy". In: (2022).

[22] Apple Inc. *Apple Differential Privacy Technical Overview*. URL: `https://www.apple.com/privacy/docs/Differential_Privacy_Overview.pdf`.

[23] Jun Tang et al. "Privacy loss in apple's implementation of differential privacy on macos 10.12". In: *arXiv preprint arXiv:1709.02753* (2017).

[24] Ken Thompson. “Reflections on trusting trust”. In: *Communications of the ACM* 27.8 (1984), pp. 761–763.

[25] Indrajit Roy et al. “Airavat: Security and privacy for MapReduce.” In: *NSDI*. Vol. 10. 2010, pp. 297–312.

[26] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[27] Elisabet Lobo-Vesga. “Let's not Make a Fuzz about it”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2021, pp. 114–116. DOI: 10.1109/ICSE-Companion52605.2021.00051.

[28] Catuscia Palamidessi and Marco Stronati. “Differential privacy for relational algebra: Improving the sensitivity bounds via constraint systems”. In: *arXiv preprint arXiv:1207.0872* (2012).

A

Type class instance resolution

Haskell supports ad hoc polymorphism, also known as overloading, via type classes. Overloading allow for different algorithms to share the same function name. Which algorithm that is ultimately chosen is based on the final type of the overloaded function.

The step where the compiler figures out which instance to pick is called *instance resolution*. By default, there need not be any ambiguity in which instance to pick where applicable. If ambiguity exists, GHC will fail to compile the program. Sometimes ambiguity is inevitable, and then the compiler extension `OverlappingInstances` provides escape hatches. Type class instances can be marked with one of the following four pragmas to guide the compiler where ambiguity would otherwise arise: `{-# OVERLAPPING #-}`, `{-# OVERLAPPABLE #-}`, `{-# OVERLAPS #-}`, or `{-# INCOHERENT #-}`. These escape hatches should be used with caution, since it may result in different instances being chosen in different parts of the program.

The algorithm for instance resolution can be summarized in 7 steps. The following description comes from the GHC team's writeup on GHC's instance resolution algorithm [11].

1. Find all instances I that *match* the target constraint; that is, the target constraint is a substitution instance of I. These instance declarations are the *candidates*.

2. If no candidates remain, the search **fails**.

3. Eliminate any candidate I_x for which there is another candidate I_y such that both of the following hold:

   - I_y is strictly more specific than I_x. That is, I_y is a substitution instance of I_x

but not vice versa.

- Either I_x is *{-# OVERLAPPABLE #-}*, or I_y is *{-# OVERLAPPING #-}*.

4. If all the remaining candidates are *{-# INCOHERENT #-}*, the search **succeeds**, returning an arbitrary surviving candidate.

5. If more than one non-incoherent candidate remains, the search **fails**.

6. Otherwise there is exactly one non-incoherent candidate; call it the *prime candidate*.

7. Now find all instances, or in-scope given constraints, that unify with the target constraint, but do not match it. Such non-candidate instances might match when the target constraint is further instantiated.

   - If all of them are incoherent top-level instances, the search succeeds, returning the prime candidate.

   - Otherwise the search fails.

For completeness, we provide one trivial example and one ambiguous example of instance resolution. In the ambiguous example, it is necessary to use *{-# INCOHERENT #-}* whereas in the trivial example, no tricks are needed.

A.1 Trivial example

The operator `(<>)` is defined in the `Semigroup` type class, and has many associated implementations. In the program

```
"Hello, " <> "World!"
```

the correct `Semigroup` instance is chosen to be `Semigroup String` after type inference. Thus, `(<>) :: a -> a -> a` is specialized to `(<>) :: String -> String -> String` in this context.

A.2 Overlapping instances example

Given the following type class, instances and constraint, we will step through the above algorithm in 8 steps.

```haskell
data Match a = Yes | No
-- * Type class and instances.
class Evidence t where
  evidence :: Match t

instance Evidence () where                        -- (A)
  evidence = Yes

instance {-# INCOHERENT #-} Evidence a where  -- (B)
  evidence = No

-- * Function providing the constraint.
constraint :: Match () -> Int
constraint = \case
  Yes -> 10

-- * Algorithm using the Evidence type class.
apply :: forall a r . Evidence a => (Match a -> r) -> r
apply f = f (evidence @a)

-- ghci> apply constraint
-- 10
```

Listing A.1: Toy example of overlapping instances.

Without marking the general instance `Evidence a` as *{-# INCOHERENT #-}*, the call `apply constraint` would result in a compile time error. This is because GHC infer that the type `()` can match the head of both `instance Evidence ()` and `instance Evidence a`. By marking the general instance as *{-# INCOHERENT #-}*, the instance resolution succeeds in the following way:

1. (1)[1] GHC finds all instances that match the constraint `Evidence ()`:

   (a) `instance Evidence ()`

   (b) `instance Evidence a`

   These both instances A and B are now **candidates**.

[1] (n) correspond to step n of the above algorithm

2. (2) Two candidates remain, and so the search continues.

3. (3) For candidate A, we check both conditions from step 3 of the algorithm:

   (a) B **is not** strictly more specific than A.

   (b) The first condition failed, so we do not check this one.

   The first condition does not hold, and so A is not eliminated.

4. (3) For candidate B, we also check both conditions from step 3 of the algorithm:

   (a) A **is** strictly more specific than B.

   (b) B is not `{-# OVERLAPPABLE #-}` **and** A is not `{-# OVERLAPPING #-}`.

   The second condition does not hold, and so B is not eliminated.

5. (4) Not all remaining candidates are incoherent. The search continues.

6. (5) Exactly one non-incoherent candidate remains (A). The search continues.

7. (6) As A is the only non-incoherent candidate remaining, we call it the *prime candidate*.

8. (7) Candidate A *match*[2] the constraint `()`, and is therefore not considered. B is the only instance which can unify with the target constraint `()`, but does not match it. Because B is also an incoherent instance, we return the prime candidate A.

Note that the potential overlap of declaring both instances `Evidence a` and `Evidence ()` is okay. It only becomes an error if any constraint leaves GHC with more than one potential instance. This would happen if `Evidence a` was *not* marked with `{-# INCOHERENT #-}`, and we call `apply constraint`.

It is also not a complete solution to put `{-# OVERLAPPING #-}` on the specific `instance Evidence ()`, as one might deduce by looking at Listing A.1 while inspecting step 3 of the algorithm. Problem arise if we try to use the function `apply` with other constraints:

[2] See step 1 of the above algorithm.

```
-- * Function providing the constraint.
constraint2 :: Match a -> Int
constraint2 = \case
  No -> 10

-- ghci> apply constraint2
-- error:
    • Overlapping instances for Evidence o10
        arising from a use of ‘apply’
      Matching instances:
        instance Evidence a
        instance [overlapping] Evidence ()
      The choice depends on the instantiation of ‘o10’
    • In the expression: apply constraint2
```

Listing A.2: Toy example of overlapping instances.

Which is why `{-# INCOHERENT #-} instance Evidence a` is the only valid pragma in the above example.

B

Breakdown of Adult data set

Attribute	Type	Values
Age	Continuous	
Workclass	Discrete	Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
Fnlwgt	Continuous	
Education	Discrete	Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool.
Education-num	Continuous	
Marital-status	Discrete	Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse.
Occupation	Discrete	Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
Relationship	Discrete	Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
Race	Discrete	White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
Sex	Discrete	Female, Male.
Capital-gain	Continuous	
Capital-loss	Continuous	
Hours-per-week	Continuous	
Country	Discrete	United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands

C

List of used language extensions

KindSignatures
TypeApplications
DataKinds
LambdaCase
TemplateHaskell
GADTs
PatternSynonyms
MultiParamTypeClasses
ScopedTypeVariables
FlexibleInstances
FlexibleContexts

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS