Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy (Extended Version)

Jason Reed* Benjamin Pierce Marco Gaboardi

Abstract

We want assurances that sensitive information will not be disclosed when aggregate data derived from a database is published. *Differential privacy* offers a strong statistical guarantee that the effect of the presence of any individual in a database will be negligible, even when an adversary has auxiliary knowledge. Much of the prior work in this area consists of proving algorithms to be differentially private one at a time; we propose to streamline this process with a functional language whose type system automatically guarantees differential privacy, allowing the programmer to write complex privacy-safe query programs in a flexible and compositional way.

The key novelty is the way our type system captures *function sensitivity*, a measure of how much a function can magnify the distance between similar inputs: well-typed programs not only can't go wrong, they *can't go too far* on nearby inputs. Moreover, by introducing a monad for random computations, we can show that the established definition of differential privacy falls out naturally as a special case of this soundness principle. We develop examples including known differentially private algorithms, privacy-aware variants of standard functional programming idioms, and compositionality principles for differential privacy.

1 Introduction

It's no secret that privacy is a problem. A wealth of information about individuals is accumulating in various databases — patient records, content and link graphs of social networking sites, book and movie ratings, ... — and there are many potentially good uses to which it could be put. But, as Netflix and others have learned [NS08] to their detriment, even when data collectors *try* to release only anonymized or aggregated results, it is easy to publish information that reveals much more than was intended, when cleverly combined with other data sources. An exciting new body of work on *differential privacy* [Dwo06, Dwo08, BLR08, NRS07, Dwo09, BDMN05, DMNS06] aims to address this problem by, first, replacing the informal goal of 'not violating privacy' with a technically precise and strong statistical guarantee, and then offering various mechanisms for achieving this guarantee. Essentially, a mechanism for publishing data is *differentially private* if any conclusion made from the published data is almost exactly as likely if any one individual's data is omitted from the database. Methods for achieving this guarantee can be attractively simple, usually involving taking the true answer to a query and adding enough random noise to blur the contributions of individuals.

For example, the query *"How many patients at this hospital are over the age of 40?"* is intuitively "almost safe"—safe because it aggregates many individuals' contributions together, and "almost" because, if an adversary happened to know the ages of every patient except John Doe, then answering this query would give them certain knowledge of a fact about John. The differential privacy methodology rests on the observation that, if we add a small amount of random noise to its result, we can still get a useful idea of the true answer to this query while obscuring the contribution of any single individual. By contrast, the query

*Jason Reed's contributions ended in November 2010 when he moved to Google.

"How many patients are over the age of 40 and also happen to be named John Doe?" is plainly problematic, since it is focused on an individual rather than an aggregate. Such a query cannot usefully be privatized: if we add enough noise to obscure any individual's contribution to the result, there won't be any signal left.

So far, most of the work in differential privacy concerns specific algorithms rather than general, compositional language features. Although there is already an impressive set of differentially private versions of particular algorithms [BDMN05, GLM+09], each new one requires its own separate proof. McSherry's Privacy Integrated Queries (PINQ) [McS09] are a good step toward more general principles: they allow for some relational algebra operations on database tables, as well as certain forms of composition of queries. But even these are relatively limited. We offer here a higher-order functional programming language whose type system directly embodies reasoning about differential privacy. In this language, we can *implement* McSherry's principles of sequential and parallel composition of differentially private computations, and many others besides, as higher-order functions. This provides a foundational explanation of why compositions of differentially private mechanisms succeed in the ways that they do.

The central idea in our type system also appears in PINQ and in many of the algorithm-by-algorithm proofs in the differential privacy literature: the *sensitivity* of query functions to quantitative differences in their input. Sensitivity is a sort of continuity property; a function of low sensitivity maps nearby inputs to nearby outputs. To give precise meaning to 'nearby,' we equip every type with a *metric* — a notion of distance — on its values.

Sensitivity matters for differential privacy because the amount of noise required to make a deterministic query differentially private is proportional to that query's sensitivity. The sensitivity of both queries discussed above is in fact 1: adding or removing one patient's records from the hospital database can only change the true value of the query by at most 1. This means that we should add the *same* amount of noise to *"How many patients at this hospital are over the age of 40?"* as to *"How many patients are over the age of 40, who also happen to be named John Doe?"* This may appear counter-intuitive, but actually it is just right: the privacy of single individuals is protected to exactly the same degree in both cases. Of course, the usefulness of the results differs: knowing the answer to the first query with, say, a typical error margin of ± 100 could still be valuable if there are thousands of patients in the hospital's records, whereas knowing the answer to the second query (which can only be zero or one) ± 100 is useless. (We might try making the second query more useful by scaling its answer up numerically: *"Is John Doe over 40? If yes, then 1,000, else 0."* But this query has a sensitivity of 1,000, not 1, and so 1,000 times as much noise must be added, blocking our sneaky attempt to violate privacy.)

To track function sensitivity, we give a *distance-aware* type system. This type system embodies two important connections between differential privacy and concepts from logic and type theory. First, reasoning about sensitivity itself strongly resembles *linear logic* [Gir87, Bar96], which has been widely applied in programming languages. The essential intuition about linear logic and linear type theories is that they treat assumptions as consumable resources. We will see that in our setting the *capability to sensitively depend on an input's value* behaves like a resource. This intuition recurs throughout the paper, and we sometimes refer to sensitivity to an input as if it is counting the number of "uses" of that input.

The other connection comes from the use of a *monad* to internalize the operation of adding random noise to query results. We include in the programming language a monad for random computations, similar to previously proposed stochastic calculi [RP02, PPT03]. Since every type has a metric in our setting, we are led to ask: what should the metric be for the monad? We find that, with the right choice of metric, the definition of differentially private functions falls out as a *special case* of the definition of function sensitivity for functions, when the function output happens to be monadic. This observation is very useful: while prior work treats differential privacy *mechanisms* and private *queries* as separate things, we see here that they can be unified in a single language. Our type system can express the privacy-safety of individual queries, as well as more complex query protocols (see Section 6) that repeatedly interact with a private database, adjusting which queries they perform depending on the responses they receive.

To briefly foreshadow what a query in our language looks like, suppose that we have the following

functions available:

$$\begin{aligned}
&\mathit{over_40} : \mathsf{row} \to \mathsf{bool} \\
&\mathit{size} : \mathsf{db} \multimap \mathbb{R} \\
&\mathit{filter} : (\mathsf{row} \to \mathsf{bool}) \to \mathsf{db} \multimap \mathsf{db} \\
&\mathit{add_noise} : !_\epsilon \mathbb{R} \multimap \bigcirc \mathbb{R}
\end{aligned}$$

The predicate *over_40* simply determines whether or not an individual database row indicates that patient is over the age of 40. The function *size* takes an entire database, and outputs how many rows it contains. Its type uses a special arrow $\multimap$, related to the linear logic function type of the same name, which expresses that the function has sensitivity of 1. The higher-order function *filter* takes a predicate on database rows and a database; it returns the subset of the rows in the database that satisfy the predicate. This filtering operation also has a sensitivity of 1 in its database argument, and again $\multimap$ is used in its type. Finally, the function *add_noise* is the differential privacy mechanism that takes a real number as input and returns a random computation (indicated by the monad $\bigcirc$) that adds in a bit of random noise (proportional to ϵ). This function has a sensitivity of ϵ (described by the scaling modality $!_\epsilon$), and this fact is intimately connected to privacy properties, as explained in Section 5.

With these in place, the query can be written as the program

$$\lambda d : !_\epsilon \mathsf{db}.\ \mathbf{let}\ !d' = d\ \mathbf{in}\ \mathit{add\_noise}\ !(\mathit{size}\ (\mathit{filter}\ \mathit{over\_40}\ d'))\ : !_\epsilon \mathsf{db} \multimap \bigcirc \mathbb{R}.$$

where some additional constructions are needed to deal with the privacy mechanism. As we explain in Section 5, its type indicates that it is a differentially private computation[1] taking a database and producing a real number. Its runtime behavior is to yield a privacy-preserving noised count of the number of patients in the hospital that are over 40.

We begin in Section 2 by describing a core type system that tracks function sensitivity. We state an informal version of the key *metric preservation theorem*, which says the execution of every well-typed function reflects the sensitivity that the type system assigns it. In Section 3 we state the standard safety properties of the type system, and give a formal statement and proof of the metric preservation theorem. Section 4 gives examples of programs that can be implemented in our language. Section 5 shows how to add the probability monad, and Section 6 develops further examples. The remaining sections discuss related work and offer concluding remarks.

2 A Type System for Function Sensitivity

2.1 Sensitivity

Our point of departure for designing a programming language for differential privacy is *function sensitivity*. A function is said to be *c-sensitive* (or *have sensitivity c*) if it can magnify distances between inputs by a factor of at most c. Since this definition depends on the input and output types of the function having a metric (a notion of distance) defined on them, we begin by discussing a special case of the definition for functions from $\mathbb{R}$ to $\mathbb{R}$, where we can use the familiar Euclidean metric $d_\mathbb{R}(x, y) = |x - y|$ on the real line. We can then formally define c-sensitivity for real-valued functions as follows.

2.1.1 Definition: A function $f : \mathbb{R} \to \mathbb{R}$ is said to be *c-sensitive* iff $d_\mathbb{R}(f(x), f(y)) \leq c \cdot d_\mathbb{R}(x, y)$ for all $x, y \in \mathbb{R}$.

A special case of this definition that comes up frequently is the case where $c = 1$. A 1-sensitive function is also called a *nonexpansive* function, since it keeps distances between input points the same or else makes them smaller. Some examples of 1-sensitive functions are

$$f_1(x) = x \qquad f_2(x) = -x \qquad f_3(x) = x/2$$

[1] More precisely its type indicates that it is a ϵ-differentially private computation.

$$f_4(x) = |x| \qquad f_5(x) = (x + |x|)/2$$

and some non-examples include: $f_6(x) = 2x$ and $f_7(x) = x^2$. The function f_6, while not 1-sensitive, *is* 2-sensitive. On the other hand, f_7 is not c-sensitive for any c.

2.1.2 Proposition: Every function that is c-sensitive is also c'-sensitive for every $c' \geq c$.

For example, f_3 is both $1/2$-sensitive and 1-sensitive.

So far we only have one type, $\mathbb{R}$, with an associated metric. We would like to introduce other base types, and type operators to build new types from old ones. We require that for every type τ that we discuss, there is a metric $d_\tau(x, y)$ for values $x, y \in \tau$. This requirement makes it possible to straightforwardly generalize the definition of c-sensitivity to arbitrary types.

2.1.3 Definition: A function $f : \tau_1 \to \tau_2$ is said to be *c-sensitive* iff $d_{\tau_2}(f(x), f(y)) \leq c \cdot d_{\tau_1}(x, y)$ for all $x, y \in \tau_1$.

The remainder of this subsection introduces several type operators, one after another, with examples of c-sensitive functions on the types that they express. We use suggestive programming-language terminology and notation, but emphasize that the discussion for now is essentially about pure mathematical functions — we do not yet worry about computational issues such as the possibility of nontermination. For example, we speak of values of a type in a way that should be understood as more or less synonymous with mere elements of a set — in Section 2.2 below, we will show how to actually speak formally about types and values.

First of all, when τ is a type with associated metric d_τ, let $!_r\tau$ be the type whose values are the same as those of τ, but with the metric 'scaled up' by a factor of r. That is, we define

$$d_{!_r\tau}(x, y) = r \cdot d_\tau(x, y).$$

One role of this type operator is to allow us to reduce the concept of c-sensitivity to 1-sensitivity. For we have

2.1.4 Proposition: A function f is a c-sensitive function in $\tau_1 \to \tau_2$ if and only f it is a 1-sensitive function in $!_c\tau_1 \to \tau_2$.

Proof: Let $x, y : \tau_1$ be given. Suppose $d_{\tau_1}(x, y) = r$. Then $d_{!_c\tau_1(x,y)} = cr$. For f to be c-sensitive as a function $\tau_1 \to \tau_2$ we must have $d_{\tau_2}(f(x), f(y)) \leq cr$, but this is exactly the same condition that must be satisfied for f to be a 1-sensitive function $!_c\tau_1 \to \tau_2$. □

We can see therefore that f_6 is a 1-sensitive function $!_2\mathbb{R} \to \mathbb{R}$, and also in fact a 1-sensitive function $\mathbb{R} \to !_{1/2}\mathbb{R}$. The symbol ! is borrowed from linear logic, where it indicates that a resource can be used an unlimited number of times. In our setting an input of type $!_r\tau$ is analogous to a resource that can be used at most r times. We can also speak of $!_\infty$, which scales up all non-zero distances to infinity, which is then like the original linear logic !, which allows unrestricted use.

Another way we can consider building up new metric-carrying types from existing ones is by forming products. If τ_1 and τ_2 are types with associated metrics d_{τ_1} and d_{τ_2}, then let $\tau_1 \otimes \tau_2$ be the type whose values are pairs (v_1, v_2) where $v_1 \in \tau_1$ and $v_2 \in \tau_2$. In the metric on this product type, we define the distance between two pairs to be the sum of the distances between each pair of components:

$$d_{\tau_1 \otimes \tau_2}((v_1, v_2), (v_1', v_2')) = d_{\tau_1}(v_1, v_1') + d_{\tau_2}(v_2, v_2')$$

With this type operator we can describe more arithmetic operations on real numbers. For instance,

$$f_8(x, y) = x + y \qquad f_9(x, y) = x - y$$

are 1-sensitive functions in $\mathbb{R} \otimes \mathbb{R} \to \mathbb{R}$, and

$$f_{10}(x, y) = (x, y) \qquad f_{11}(x, y) = (y, x) \qquad f_{12}(x, y) = (x + y, 0)$$

$$cswp(x,y) = \begin{cases} (x,y) & \text{if } x < y \\ (y,x) & \text{otherwise} \end{cases}$$

are 1-sensitive functions in $\mathbb{R} \otimes \mathbb{R} \to \mathbb{R} \otimes \mathbb{R}$. We will see the usefulness of *cswp* in particular below in Section 4.6. However,

$$f_{13}(x,y) = (x \cdot y, 0) \qquad f_{14}(x,y) = (x,x)$$

are not 1-sensitive functions in $\mathbb{R} \otimes \mathbb{R} \to \mathbb{R} \otimes \mathbb{R}$. The function f_{14} is of particular interest, since at no point do we ever risk multiplying x by a constant greater than 1 (as we do in, say, f_6 and f_{13}) and yet the fact that x is *used twice* means that variation of x in the input is effectively doubled in measurable variation of the output. This intuition about counting uses of variables is reflected in the connection between our type system and linear logic.

This metric is not the only one that we can assign to pairs. Just as linear logic has more than one conjunction, our type theory admits more than one product type. Another one that will prove useful is taking distance between pairs to be the *maximum* of the differences between their components instead the sum. Even though the underlying set of values is essentially the same, we regard choosing a different metric as creating a distinct type: the type $\tau_1 \mathbin{\&} \tau_2$ consists of pairs $\langle v_1, v_2 \rangle$, (written differently from pairs of type $\tau_1 \otimes \tau_2$ to further emphasize the difference) with the metric

$$d_{\tau_1 \& \tau_2}(\langle v_1, v_2 \rangle, \langle v_1', v_2 \rangle) = \max(d_{\tau_1}(v_1, v_1'), d_{\tau_2}(v_2, v_2')).$$

Now we can say that $f_{15}(x,y) = \langle x, x \rangle$ is a 1-sensitive function $\mathbb{R} \otimes \mathbb{R} \to \mathbb{R} \mathbin{\&} \mathbb{R}$. More generally, & lets us combine outputs of different c-sensitive functions even if they share dependency on common inputs.

2.1.5 Proposition: If $f : \tau \to \tau_1$ and $g : \tau \to \tau_2$ are c-sensitive functions, then $\lambda x.\langle f\ x, g\ x \rangle$ is a c-sensitive function in $\tau \to \tau_1 \mathbin{\&} \tau_2$.

Next we would like to capture the set of functions itself as a type, so that we can, for instance, talk about higher-order functions. Let us take $\tau_1 \multimap \tau_2$ to be the type whose values are 1-sensitive functions $f : \tau_1 \to \tau_2$. We have already established that the presence of $!_r$ means that having 1-sensitive functions suffices to express c-sensitive functions for all c, so we need not specially define an entire family of c-sensitive function type constructors: the type of c-sensitive functions from τ_1 to τ_2 is just $!_c\tau_1 \multimap \tau_2$. We define the metric for $\multimap$ as follows:

$$d_{\tau_1 \multimap \tau_2}(f, f') = \max_{x \in \tau_1} d_{\tau_2}(f(x), f'(x))$$

This is chosen to ensure that $\multimap$ and $\otimes$ have the expected currying/uncurrying behavior with respect to each other. We find in fact that

$$curry(f) = \lambda x.\lambda y.f\ (x,y)$$
$$uncurry(g) = \lambda(x,y).g\ x\ y$$

are 1-sensitive functions in $(\mathbb{R} \otimes \mathbb{R} \multimap \mathbb{R}) \to (\mathbb{R} \multimap \mathbb{R} \multimap \mathbb{R})$ and $(\mathbb{R} \multimap \mathbb{R} \multimap \mathbb{R}) \to (\mathbb{R} \otimes \mathbb{R} \multimap \mathbb{R})$, respectively.

We postulate several more type operators that are quite familiar from programming languages. The unit type 1 which has only one inhabitant $()$, has the metric $d_1((),()) = 0$. Given two types τ_1 and τ_2, we can form their disjoint union $\tau_1 + \tau_2$, whose values are either of the form $\mathbf{inj}_1\, v$ where $v \in \tau_1$, or $\mathbf{inj}_2\, v$ where $v \in \tau_2$. Its metric is

$$d_{\tau_1+\tau_2}(v, v') = \begin{cases} d_{\tau_1}(v_0, v_0') & \text{if } v = \mathbf{inj}_1\, v_0 \text{ and } v' = \mathbf{inj}_1\, v_0'; \\ d_{\tau_2}(v_0, v_0') & \text{if } v = \mathbf{inj}_2\, v_0 \text{ and } v' = \mathbf{inj}_2\, v_0'; \\ \infty & \text{otherwise.} \end{cases}$$

Note that this definition creates a type that is an *extremely* disjoint union of two components. Any distances between pairs of points within the same component take the distance that that component specifies, but distances from one component to the other are all infinite.

Notice what this means for the type bool in particular, which we define as usual as $1 + 1$. It is easy to write c-sensitive functions *from* bool to other types, for the infinite distance between the values true and

false gives us license to map them to any two values we like, no matter how far apart they are. However, it is conversely hard for a nontrivial function *to* bool to be c-sensitive. The function

$$gtzero(x) = \begin{cases} \text{true} & \text{if } x \geq 0; \\ \text{false} & \text{otherwise.} \end{cases}$$

which tests whether the input is greater than zero, is not c-sensitive for any finite c. This can be blamed, intuitively, on the discontinuity of *gtzero* at zero.

Finally, we include the ability to form (iso)recursive types $\mu\alpha.\tau$ whose values are of the form $\mathbf{fold}\, v$, where v is of the type $[\mu\alpha.\tau/\alpha]\tau$, and whose metric we would like to give as

$$d_{\mu\alpha.\tau}(\mathbf{fold}\, v, \mathbf{fold}\, v') = d_{[\mu\alpha.\tau/\alpha]\tau}(v, v').$$

This definition, however, is not well-founded, since it depends on a metric at possibly a more complex type, due to the substitution $[\mu\alpha.\tau/\alpha]\tau$. It will suffice as an intuition for our present informal discussion, since we only want to use it to talk about lists (rather than, say, types such as $\mu\alpha.\alpha$), but a formally correct treatment of the metric is given in Section 2.7.

With these pieces in place, we can introduce some useful and familiar types, such as the natural numbers $\mathbb{N} = \mu\alpha.1 + \alpha$, and the type of lists of real numbers, $\mathsf{listreal} = \mu\alpha.1 + \mathbb{R} \otimes \alpha$. (The reader is invited to consider also the alternative where $\otimes$ is replaced by &; we return to this choice below in Section 4.) The metric between lists that arises from the preceding definitions is as follows. Two lists of different lengths are at distance ∞ from each other; this comes from the definition of the metric on disjoint union types. For two lists $[x_1, \ldots, x_n]$ and $[y_1, \ldots, y_n]$ of the same length, we have

$$d_{\mathsf{listreal}}([x_1, \ldots, x_n], [y_1, \ldots, y_n]) = \sum_{i=1}^{n} |x_i - y_i|.$$

We now can make the claim that there is a 1-sensitive function $sort : \mathsf{listreal} \multimap \mathsf{listreal}$ that takes in a list of reals and outputs the sorted version of that same list. This fact may seem somewhat surprising, since a small variation in the input list can lead to an abrupt change in the permutation of the list that is produced. However, what we output is not the permutation itself, but merely the values of the sorted list; the apparent point of discontinuity where one value overtakes another is exactly where those two values are equal, and their exchange of positions in the output list is unobservable.

Of course, we would prefer not to rely on such informal arguments. This example is meant to serve as motivation for what we want to do in the sequel: design a rigorous type system to capture sensitivity of *programs*, so that we can see that the 1-sensitivity of sorting is a consequence of the fact that an implementation of a sorting program is well-typed.

2.2 Typing Judgment

Type safety for a programming language ordinarily guarantees that a well-typed open expression e of type τ is well-behaved during execution. 'Well-behaved' is usually taken to mean that e can accept any (appropriately typed) value for its free variables, and will evaluate to a value of type τ without becoming stuck or causing runtime errors: *Well-typed programs can't go wrong.* We mean to make a strictly stronger guarantee than this, namely a guarantee of c-sensitivity. It should be the case that if an expression is given *similar* input values for its free variables, the results of evaluation will also be suitably similar: *Well-typed programs can't go too far.*

We will take, as usual, a typing judgment $\Gamma \vdash e : \tau$ (expressing that e is a well-formed expression of type τ in a context Γ) but we add further structure to the contexts, describing for each variable how sensitive the expression is to it. By doing so we are essentially generalizing c-sensitivity to capture what it means for an expression to be sensitive to many inputs simultaneously — that is, to all of the variables in the context — rather than just one. Contexts Γ have the syntax

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, x :_r \tau$$

for $r \in \mathbb{R}^{>0} \cup \{\infty\}$. To have a hypothesis $x :_r \tau$ while constructing an expression e is to have permission to be r-sensitive to variation in the input x: the output of e is allowed to vary by a distance up to rs if the value substituted for x varies by distance s. We include the special value ∞ as an allowed value of r so that we can express ordinary (unconstrained by sensitivity) functions as well as c-sensitive functions. Algebraic operations involving ∞ are defined by setting $\infty \cdot r = \infty$ (except for $\infty \cdot 0 = 0$) and $\infty + r = \infty$. This means that to be ∞-sensitive is no constraint at all: if we consider the definition of sensitivity, then ∞-sensitivity permits any variation at all in the input to be blown up to arbitrary variation in the output.

A well-typed expression

$$x :_c \tau_1 \vdash e : \tau_2$$

is exactly a program that represents a c-sensitive computation. However, we can also consider more general programs

$$x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \vdash e : \tau$$

in which case we define the guarantee to be that, if each x_i varies by s_i, then the result of evaluating e only varies by $\sum_i r_i s_i$.

This intended meaning of the typing judgment must eventually be formulated as a theorem about the programming language as a whole, similar to standard preservation and progress theorems. In the notation we have developed so far, it can be stated approximately as follows:

2.2.1 Theorem [Metric Preservation]: Suppose $\Gamma \vdash e : \tau$. Let sequences of values $(v_i)_{1 \leq i \leq n}$ and $(v_i')_{1 \leq i \leq n}$ be given. Suppose for all $i \in 1, \ldots, n$ that we have

1. $\vdash v_i, v_i' : \tau_i$
2. $d_{\tau_i}(v_i, v_i') = s_i$
3. $x_i :_{r_i} \tau_i \in \Gamma$.

If the program $[v_1/x_1] \cdots [v_n/x_n]e$ evaluates to v, then there exists a v' such that $[v_1'/x_1] \cdots [v_n'/x_n]e$ evaluates to v', and

$$d_\tau(v, v') \leq \sum_i r_i s_i.$$

The notation $[v/x]e$ indicates (capture-avoiding) substitution of the value v for the variable x in expression e as usual. However, we have not yet formalized the definition of the metric for the programming language, which takes place below in Section 2.7. Subsequently, in Section 3, the metric preservation theorem is stated and proved. We proceed for now by setting up the usual apparatus of our programming language: its type structure, syntax, and its static and dynamic semantics.

2.3 Types

The complete syntax and formation rules for types are given in Figure 1. There are type variables α, (which appear in type variable contexts Ψ) base types b (drawn from a signature Σ), unit and void and sum types, metric-scaled types $!_r\tau$, and recursive types $\mu\alpha.\tau$. There are the two pair types $\otimes$ and &, which differ in their metrics. There are two kinds of function space, $\multimap$ and $\to$, where $\tau_1 \multimap \tau_2$ contains just 1-sensitive functions, while $\tau_1 \to \tau_2$ is the ordinary unrestricted function space, containing the functions that can be programmed without any sensitivity requirements on the argument.

2.4 Expressions

The syntax of expressions is straightforward; indeed, our language can be seen as essentially just a *refinement* type system layered over the static and dynamic semantics of an ordinary typed functional programming language. Almost all of the expression formers should be entirely familiar. One feature worth noting (which is also familiar from linear type systems) is that we distinguish two kinds of pairs: the one that arises from

$$\tau ::= \alpha \mid b \mid 1 \mid \mu\alpha.\tau \mid \tau + \tau \mid \tau \otimes \tau \mid \tau \,\&\, \tau \mid \tau \multimap \tau \mid \tau \to \tau \mid !_r\tau$$

$$\frac{}{\Psi, \alpha : \text{type} \vdash \alpha : \text{type}} \qquad \frac{\Psi, \alpha : \text{type} \vdash \tau : \text{type}}{\Psi \vdash \mu\alpha.\tau : \text{type}} \qquad \frac{}{\Psi \vdash 1 : \text{type}} \qquad \frac{b : \text{type} \in \Sigma}{\Psi \vdash b : \text{type}} \qquad \frac{\Psi \vdash \tau : \text{type} \qquad r \in \mathbb{R}^{>0} \cup \{\infty\}}{\Psi \vdash !_r\tau : \text{type}}$$

$$\frac{\Psi \vdash \tau_1 : \text{type} \qquad \Psi \vdash \tau_2 : \text{type} \qquad \star \in \{+, \&, \otimes, \multimap, \to\}}{\Psi \vdash \tau_1 \star \tau_2 : \text{type}}$$

Figure 1: Type Formation

$$\frac{r \geq 1}{\Gamma, x :_r \tau \vdash x : \tau}\, var \qquad \frac{}{\Gamma \vdash () : 1}\, 1I \qquad \frac{\Delta \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Delta + \Gamma \vdash (e_1, e_2) : \tau_1 \otimes \tau_2} \otimes I \qquad \frac{\Gamma \vdash e : \tau_1 \otimes \tau_2 \qquad \Delta, x :_r \tau_1, y :_r \tau_2 \vdash e' : \tau'}{\Delta + r\Gamma \vdash \mathbf{let}(x, y) = e \,\mathbf{in}\, e' : \tau'} \otimes E$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \,\&\, \tau_2} \,\&I \qquad \frac{\Gamma \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma \vdash \pi_i\ e : \tau_i} \,\&E \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \begin{array}{c}\Delta, x :_r \tau_1 \vdash e_1 : \tau' \\ \Delta, x :_r \tau_2 \vdash e_2 : \tau'\end{array}}{\Delta + r\Gamma \vdash \mathbf{case}\, e \,\mathbf{of}\, x.e_1 \mid x.e_2 : \tau'} + E \qquad \frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash \mathbf{inj}_i\, e : \tau_1 + \tau_2} + I$$

$$\frac{\Gamma, x :_1 \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \multimap \tau'} \multimap I \qquad \frac{\Delta \vdash e_1 : \tau \multimap \tau' \qquad \Gamma \vdash e_2 : \tau}{\Delta + \Gamma \vdash e_1\ e_2 : \tau'} \multimap E \qquad \frac{\Gamma, x :_\infty \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \to I \qquad \frac{\Delta \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Delta + \infty\Gamma \vdash e_1\ e_2 : \tau'} \to E$$

$$\frac{\Gamma \vdash e : \tau}{s\Gamma \vdash !e : !_s\tau}\, !I \qquad \frac{\Gamma \vdash e : !_s\tau \qquad \Delta, x :_{rs} \tau \vdash e' : \tau'}{\Delta + r\Gamma \vdash \mathbf{let}\, !x = e \,\mathbf{in}\, e' : \tau'} \,!E \qquad \frac{\Gamma \vdash e : [\mu\alpha.\tau/\alpha]\tau}{\Gamma \vdash \underset{\mu\alpha.\tau}{\mathbf{fold}}\, e : \tau} \,\mu I \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \underset{\mu\alpha.\tau}{\mathbf{unfold}}\, e : [\mu\alpha.\tau/\alpha]\tau} \,\mu E$$

Figure 2: Typing Rules

$\otimes$, which is eliminated by pattern-matching and written with (parentheses), and the one that arises from &, which is eliminated by projection and written with $\langle$angle brackets$\rangle$. The other is that for clarity we have explicit introduction and elimination forms (! and **let** !, respectively) for the type constructor $!_r$.

$$\begin{array}{rcl} e & ::= & x \mid c \mid () \mid \langle e, e \rangle \mid (e, e) \\ & & \mathbf{let}(x, y) = e \,\mathbf{in}\, e \mid \pi_i e \mid \lambda x.e \mid e\ e \mid \\ & & \mathbf{inj}_i\, e \mid (\mathbf{case}\, e \,\mathbf{of}\, x.e \mid x.e) \mid \\ & & !e \mid \mathbf{let}\ \ !x = e \,\mathbf{in}\, e \mid \\ & & \mathbf{unfold}_\tau\, e \mid \mathbf{fold}_\tau\, e \end{array}$$

Just as with base types, we allow for primitive constants c to be drawn from a signature Σ.

2.5 Typing Relation

To present the typing relation, we need a few algebraic operations on contexts. The notation $s\Gamma$ indicates pointwise scalar multiplication of all the sensitivity annotations in Γ by s :

$$\begin{array}{c} s\cdot = \cdot \\ s(\Gamma, x :_r \tau) = \Gamma, x :_{rs} \tau \end{array}$$

We can also define addition of two contexts (which may share some variables) by

$$\begin{array}{rcll} \cdot + \cdot & = & \cdot & \\ (\Gamma, x :_s \tau) + (\Delta, x :_r \tau) & = & (\Gamma + \Delta), x :_{r+s} \tau & \\ (\Gamma, x :_r \tau) + \Delta & = & (\Gamma + \Delta), x :_r \tau & (x \notin \Delta) \\ \Gamma + (\Delta, x :_r \tau) & = & (\Gamma + \Delta), x :_r \tau & (x \notin \Gamma) \end{array}$$

The typing relation is defined by the inference rules in Figure 2. Every occurrence of r and s in the typing rules is assumed to be drawn from $\mathbb{R}^{>0} \cup \{\infty\}$.

The rule *var* allows a variable from the context to be used as long as its annotation is at least 1, since the identity function is c-sensitive for any $c \geq 1$. Any other context Γ is allowed to appear in a use of *var*, because permission to depend on a variable is not an obligation to depend on it. (In this respect our type system is closer to affine logic than linear logic.)

In the rule $\otimes I$, consider the role of the contexts. Δ represents the variables that e_1 depends on, and captures quantitatively how sensitive it is to each one. Γ does the same for e_2. In the conclusion of the rule, we add together the sensitivities found in Γ and Δ, precisely because the distances in the type $\tau_1 \otimes \tau_2$ are measured by a sum of how much e_1 and e_2 vary. Compare this to $\& I$, where we merely require that the same context is provided in the conclusion as is used to type the two components of the pair.

We can see the action of the type constructor $!_r$ in its introduction rule. If we scale up the metric on the expression being constructed, then we must scale up the sensitivity of every variable in its context to compensate.

The closed-scope elimination rules for $\otimes$, $+$, and $!$ share a common pattern. The overall elimination has a choice as to how much it depends on the expression of the type being eliminated: this is written as the number r in all three rules. The cost of this choice is that context Γ that was used to build that expression must then be multiplied by r. The payoff is that the variable(s) that appear in the scope of the elimination (in the case of $\otimes E$, the two variables x and y, in $+E$ the xs one in each branch) come with permission for the body to be r-sensitive to them. In the case of $!E$, however, the variable appears with an annotation of rs rather than r, reflecting that the $!_s$ scaled the metric for that variable by a factor of s.

We note that $\multimap I$, since $\multimap$ is meant to capture 1-sensitive functions, appropriately creates a variable in the context with an annotation of 1. Compare this to $\to I$, which adds a hypothesis with annotation ∞, whose use is unrestricted. Conversely, in $\to E$, note that the context Γ used to construct the argument e_2 of the function is multiplied by ∞ in the conclusion. Because the function e_1 makes no guarantee how sensitive it is to its argument, we can in turn make no guarantee how much $e_1\ e_2$ depends on the variables in Γ. This plays the same role as requirements familiar in linear logic, that all variables used to form an argument to an unrestricted implication must themselves be unrestricted.

2.6 Evaluation

We give a small-step operational semantics for the language, which is almost entirely routine. Values, the subset of expressions that are allowed as finished results of evaluation, are defined as follows.

$$v \quad ::= \quad () \mid c \mid \langle v, v \rangle \mid (v, v) \mid \lambda x.e \mid \mathbf{inj}_i\, v \mid \mathbf{fold}_\tau\, v \mid !v$$

The judgment $e \mapsto e'$ says that e takes a step to e'. The complete set of evaluation rules is given in Figure 3.

2.7 Metric Relation

We come now to the task of providing a notion of distance — a metric — for all the types in our language. It is convenient to do this by establishing a metric *relation* $v \sim_r v'$ that means v and v' are no more than distance r apart, rather than a metric function that outputs the distance given two values.

In fact, we define three relations for $r \in \mathbb{R}^{>0}$

$$\begin{array}{l} \vdash v \sim_r v' : \tau \\ \vdash \sigma \sim_\gamma \sigma' : \Gamma^\circ \\ \Gamma \vdash e \approx_r e' : \tau \end{array}$$

The symbol σ here stands for a substitution $[v_1/x_1] \cdots [v_n/x_n]$ of values for variables, and γ for a variable-indexed vector of positive reals, which we write like a substitution, $[r_1/x_1] \cdots [r_n/x_n]$. The notation Γ° denotes Γ with all sensitivity annotations erased. These judgments are defined in Figure 4.

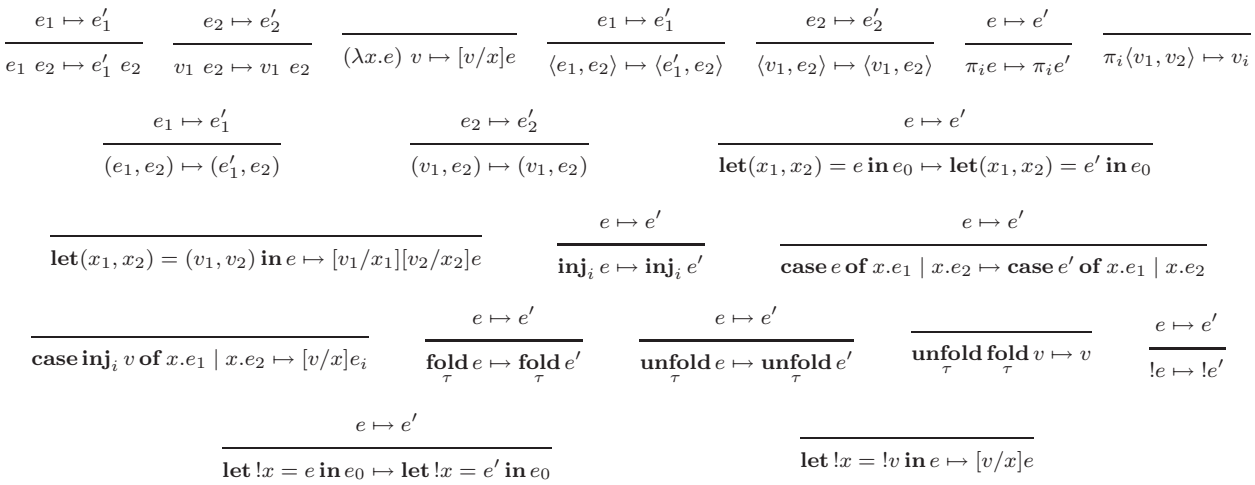

Figure 3: Evaluation Rules

We can summarize the meaning of the three metric judgments as follows. The basic judgment is $\vdash v \sim_r v' : \tau$, which asserts that v and v' are closed values of type τ no more than distance r apart from one another. The rules that assign distances to pairs of values are organized to follow the type system; there is one value metric rule for each way of forming a well-typed value. Of substitutions we can say $\vdash \sigma \sim_\gamma \sigma' : \Gamma^\circ$, which means that σ and σ' are both closed substitutions of well-typed values for all the variables in Γ, and the distance between each pair of values is reflected in the vector γ of real numbers. Finally, the metric on arbitrary (possibly open) expressions is given by the judgment $\Gamma \vdash e \approx_r e' : \tau$. This judgment is defined by a single rule, which requires that the two expressions be common substitution instances of a well-typed expression e_0. The distance between $e = \sigma e_0$ and $e' = \sigma' e_0$ is then $\gamma\Gamma$. This is essentially a dot product of the distance vector γ between σ and σ', and the sensitivities found in the context Γ used to type e_0. We define $\gamma\Gamma$ by

$$([r_1/x_1] \cdots [r_n/x_n])(x_1 :_{s_1} \tau_1, \ldots, x_n :_{s_n} \tau_n) = \sum_{i=1}^{n} r_i s_i.$$

It is worth noticing that in the metric judgments the meta-variables r and s are assumed to be drawn only from $\mathbb{R}^{>0}$. This differs from what happens in the case of the typing judgments, where r and s are assumed to be drawn from $\mathbb{R}^{>0} \cup \{\infty\}$. Two values that are not in the relation $\sim_r$ (for every r) are morally infinitely apart. For instance, this happens when two values in different components of a disjoint union are considered, i.e. $\forall r, \mathbf{inj}_1\, v \not\sim_r \mathbf{inj}_2\, v'$. In particular, note that in the metric scaling rule, the product rs should be in $\mathbb{R}^{>0}$. Analogously, in the metric for arbitrary expression we want $\gamma\Gamma$ to be defined only when $\gamma\Gamma \in \mathbb{R}^{>0}$.

The definition of the evaluation metric relation $\approx$ may seem circular: we want to be convinced that the sensitivity annotations in the type system constrain the metric behavior of programs, and yet we seem to be defining the metric in terms of the type system! However, this apparent circularity is ultimately benign. The heart of the matter is that the metric on expressions is *preserved* by the process of evaluation, and that at the end of the day, the metric on observable *values* is defined independently of the type system.

3 Metatheory

In this section we prove the formal correctness of the programming language described above. First of all, we can prove appropriate versions of the usual basic properties that we expect to hold of a well-formed typed programming language.

We say that $\Gamma \leq \Delta$ when there exists a Δ' such that $\Delta = \Delta' + \Gamma$.

$$\frac{\vdash v : \tau}{\vdash v \sim_0 v : \tau} \qquad \frac{\vdash v_1 \sim_{r_1} v_1' : \tau_1 \qquad \vdash v_2 \sim_{r_2} v_2' : \tau_2}{\vdash (v_1, v_2) \sim_{r_1+r_2} (v_1', v_2') : \tau_1 \otimes \tau_2} \qquad \frac{\vdash v_1 \sim_r v_1' : \tau_1 \qquad \vdash v_2 \sim_r v_2' : \tau_2}{\vdash \langle v_1, v_2\rangle \sim_r \langle v_1', v_2'\rangle : \tau_1 \,\&\, \tau_2} \qquad \frac{\vdash v \sim_r v' : \tau_i}{\vdash \mathbf{inj}_i\, v \sim_r \mathbf{inj}_i\, v' : \tau_1 + \tau_2}$$

$$\frac{x :_1 \tau \vdash e \approx_r e' : \tau_0}{\vdash \lambda x.e \sim_r \lambda x.e' : \tau \multimap \tau_0} \qquad \frac{x :_\infty \tau \vdash e \approx_r e' : \tau_0}{\vdash \lambda x.e \sim_r \lambda x.e' : \tau \to \tau_0} \qquad \frac{\vdash v \sim_r v' : \tau}{\vdash\, !v \sim_{rs} !v' : !_s\tau} \qquad \frac{\vdash v \sim_r v' : [\mu\alpha.\tau/\alpha]\tau}{\vdash \underset{\mu\alpha.\tau}{\mathbf{fold}}\, v \sim_r \underset{\mu\alpha.\tau}{\mathbf{fold}}\, v' : \tau}$$

$$\frac{\vdash \sigma \sim_\gamma \sigma' : \Gamma^\circ \qquad \Gamma, \Gamma_0 \vdash e_0 : \tau}{\Gamma_0 \vdash \sigma e_0 \approx_{\gamma\Gamma} \sigma' e_0 : \tau} \qquad \frac{\vdash v_1 \sim_{r_1} v_1' : \tau_1 \qquad \cdots \qquad \vdash v_n \sim_{r_n} v_n' : \tau_n}{\vdash [v_1/x_1]\cdots[v_n/x_n] \sim_{[r_1/x_1],\ldots,[r_n/x_n]} [v_1'/x_1]\cdots[v_n'/x_n] : (x_1 : \tau_1, \ldots, x_n : \tau_n)}$$

Figure 4: Metric Rules

3.1 Lemma [Weakening]: If $\Gamma \leq \Delta$, and $\Gamma \vdash e : \tau$, then $\Delta \vdash e : \tau$.

Proof: By straightforward induction on the typing derivation. □

3.2 Theorem [Substitution]: If $\Gamma \vdash e : \tau$ and $\Delta, x :_r \tau \vdash e' : \tau'$, then $\Delta + r\Gamma \vdash [e/x]e' : \tau'$.

Proof: By induction on the derivation of $\Delta, x :_r \tau \vdash e' : \tau'$. A critical case is when this derivation is an instance of the variable rule using the variable x:

$$\frac{r \geq 1}{\Delta, x :_r \tau \vdash x : \tau}\, var$$

In this case, $\tau = \tau'$. Our obligation is to show that $\Delta + r\Gamma \vdash e : \tau$. But it can easily be seen that $\Gamma \leq \Delta + r\Gamma$ when $r \geq 1$. Therefore we can apply the weakening lemma above to the assumption that $\Gamma \vdash e : \tau$.

The rest of the cases follow straightforwardly from application of the induction hypothesis to every premise of the relevant typing rule. Simple arithmetic reasoning is also required for the bookkeeping on context annotations. For instance, in the case of $e' = !e_0$, we consider a derivation

$$\frac{\Delta_0, x :_{r_0} \tau \vdash e_0 : \tau_0}{s\Delta_0, x :_{r_0 s} \tau \vdash !e_0 : !_s\tau_0}\, !I$$

where $\tau' = !_s\tau_0$ and $\Delta = s\Delta_0$ and $r = r_0 s$. In this case we apply the induction hypothesis to obtain $\Delta_0 + r_0\Gamma \vdash [e/x]e_0 : \tau_0$, and conclude (using the rule $!I$) that

$$s(\Delta_0 + r_0\Gamma) \vdash ![e/x]e_0 : !_s\tau_0$$

which is the same thing as

$$\Delta + r\Gamma \vdash [e/x]e' : \tau'$$

as required. □

3.3 Theorem [Preservation]: If $\vdash e : \tau$ and $e \mapsto e'$, then $\vdash e' : \tau$.

Proof: Direct proof by case analysis on the possible derivations of $e \mapsto e'$. We consider a typical case.

Case:

$$\frac{}{(\lambda x.e)\ v \mapsto [v/x]e}$$

By inversion on the well-typedness of $(\lambda x.e)\ v$ we either have a derivation of the form

$$\frac{\dfrac{x :_1 \tau \vdash e : \tau'}{\vdash \lambda x.e : \tau \multimap \tau'} \qquad \vdash v : \tau}{\vdash (\lambda x.e)\ v : \tau'}$$

or

$$\dfrac{\dfrac{x :_\infty \tau \vdash e : \tau'}{\vdash \lambda x.e : \tau \multimap \tau'} \qquad \vdash v : \tau}{\vdash (\lambda x.e)\ v : \tau'.}$$

In either case, the above substitution principle allows us to conclude that $[v/x]e : \tau'$.

□

3.4 Theorem [Progress]: If $\vdash e : \tau$, then either e is a value, or there exists $e' \mapsto e$.

Proof: By induction on the typing derivation. We again consider a typical case. Suppose e's typing derivation arises as

$$\dfrac{\Delta \vdash e_1 : \tau \multimap \tau' \qquad \Gamma \vdash e_2 : \tau}{\Delta + \Gamma \vdash e_1\ e_2 : \tau'} \multimap E$$

By induction hypothesis, either e_1 is a value or takes a step. If it takes a step, say, to e_1', then the application takes a step via

$$\dfrac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}$$

Otherwise it is a value, but it has type $\tau \multimap \tau'$, and by inversion on the typing rules must be of the form $\lambda x.e_0$. Again by induction e_2 is either a value or takes a step. If it takes a step, say, to e_2', then the application takes a step

$$\dfrac{e_2 \mapsto e_2'}{(\lambda x.e_0)\ e_2 \mapsto (\lambda x.e_0)\ e_2}$$

But if $e_2' = v$ is a value, then the application takes a step

$$\dfrac{}{(\lambda x.e_0)\ v \mapsto [v/x]e_0}$$

□

Note that the weakening lemma allows both making the context larger, and making the annotations numerically greater. The substitution property says that if we substitute e into a variable that is used r times, then Γ, the dependencies of e, must be multiplied by r in the result. The preservation and progress lemmas are entirely routine, but we still need to observe that the standard evaluation rules are compatible with our type system. Because evaluation takes place on closed terms, however, the context annotations we introduce are only trivially involved. They play an important role in the next section.

3.1 Metric Preservation Theorem

In this section we will show that parallel evaluation of a pair of expressions preserves (or decreases) the distance between them.

There are several lemmas required along the way. Our first milestone will be seeing that $\approx$ and $\sim$ coincide on those expressions that happen to be values. One direction of this is easy, namely seeing that $\sim$ implies $\approx$.

3.1.1 Lemma: If $\vdash v \sim_r v' : \tau$, then $\vdash v \approx_r v' : \tau$.

Proof: Use the expression metric rule directly.

$$\dfrac{\vdash [v/x] \sim_{[r/x]} [v'/x] : \Gamma \qquad \dfrac{}{x :_1 \tau \vdash x : \tau}\, var}{\vdash [v/x]x \approx_r [v'/x]x : \tau}$$

□

To accomplish the other direction, that $\approx$ implies $\sim$, we first prove a result that is like a strengthened version of the expression metric rule for substitutions that yield closed values.

3.1.2 Lemma: Suppose $\Gamma \vdash e : \tau$. If $\vdash \sigma \sim_\gamma \sigma' : \Gamma^\circ$, and σe is a value, then $\sigma' e$ must also be a value, and $\vdash \sigma e \sim_{\gamma\Gamma} \sigma' e : \tau$

Proof: By case analysis on the structure of e. One case arises from the possibility that e is a variable, and the remainder of cases correspond to value-formers in the language. There are no cases for elimination forms, because a substitution cannot turn an elimination into a value.

We show some representative cases:

Case: e is a variable $x : \tau_x \in \Gamma$. In this case, $\sigma(x) = v$ and $\sigma'(x) = v'$. Say $\gamma(x) = r$. We must show that $\vdash v \sim_r v' : \tau_x$, but this follows directly from inspection of $\sigma \sim_\gamma \sigma' : \Gamma^\circ$.

Case: $e = \lambda x.e_0$. We must show $\vdash \lambda x.\sigma e_0 \sim_{\gamma\Gamma} \lambda x.\sigma' e_0 : \tau \multimap \tau_0$, which in turn requires $x :_1 \tau \vdash \sigma e_0 \approx_{\gamma\Gamma} \sigma' e_0 : \tau_0$. But this follows immediately from the inference rule defining $\approx$.

Case: $e = (e_1, e_2)$. We must show $\vdash (\sigma e_1, \sigma e_2) \sim_{\gamma(\Gamma_1+\Gamma_2)} (\sigma' e_1, \sigma' e_2) : \tau_1 \otimes \tau_2$. But we get by the induction hypothesis that $\vdash \sigma e_i \sim_{\gamma\Gamma_i} \sigma' e_i : \tau_i$ for both i, so we just plug these facts into the value metric rule for $\otimes$. □

3.1.3 Corollary: If $\vdash v \approx_r v' : \tau$, then $\vdash v \sim_r v' : \tau$.

Proof: By inversion on the rule defining $\approx$, we know v, v', r must arise as $v = \sigma e_0$ and $v' = \sigma' e_0$ and $r = \gamma\Gamma$ so that

$$\frac{\vdash \sigma \sim_\gamma \sigma' : \Gamma^\circ \qquad \Gamma \vdash e_0 : \tau}{\vdash \sigma e_0 \approx_{\gamma\Gamma} \sigma' e_0 : \tau}$$

By the preceding lemma, we have $\vdash \sigma e_0 \sim_{\gamma\Gamma} \sigma' e_0 : \tau$, as required. □

Another important property of $\approx$ is that it respects substitution.

3.1.4 Lemma [Substitution into $\approx$]: The following rule is admissible:

$$\frac{\Delta \vdash e \approx_r e' : \tau \qquad \Delta_0, x :_s \tau \vdash e_0 \approx_{r_0} e_0' : \tau_0}{\Delta_0 + s\Delta \vdash [e/x]e_0 \approx_{r_0 + rs} [e'/x]e_0' : \tau_0}$$

Proof: By inversion on the rule defining $\approx$, there must exist $\hat{e}_0, \sigma_0, \sigma_0', \Gamma_0, \gamma_0$ such that such that $\sigma_0 \hat{e}_0 = e_0$, $\sigma_0' \hat{e}_0 = e_0'$, $\gamma_0\Gamma_0 = r_0$ and

$$\frac{\vdash \sigma_0 \sim_{\gamma_0} \sigma_0' : \Gamma_0^\circ \qquad \Gamma_0, \Delta_0, x :_s \tau \vdash \hat{e}_0 : \tau_0}{\Delta_0, x :_s \tau \vdash \sigma_0 \hat{e}_0 \approx_{\gamma_0\Gamma_0} \sigma_0' \hat{e}_0 : \tau_0}$$

as well as $\hat{e}, \sigma, \sigma', \Gamma, \gamma$ such that such that $\sigma\hat{e} = e$, $\sigma'\hat{e} = e'$, $\gamma\Gamma = r$ and

$$\frac{\vdash \sigma \sim_\gamma \sigma' : \Gamma^\circ \qquad \Gamma, \Delta \vdash \hat{e} : \tau}{\Delta \vdash \sigma\hat{e} \approx_{\gamma\Gamma} \sigma'\hat{e} : \tau}$$

By the substitution lemma on typing derivations that we have already shown, we find that

$$\Gamma_0, s\Gamma, \Delta_0 + s\Delta \vdash [\hat{e}/x]e_0 : \tau_0$$

and so we then directly build a derivation

$$\frac{\vdash (\sigma_0, \sigma) \sim_{(\gamma_0,\gamma)} (\sigma_0', \sigma') : (\Gamma_0^\circ, \Gamma^\circ) \qquad \Gamma_0, s\Gamma, \Delta_0 + s\Delta \vdash [\hat{e}/x]e_0 : \tau_0}{\Delta_0 + s\Delta \vdash [\sigma\hat{e}/x]\sigma_0 e_0 \approx_{\gamma_0\Gamma_0 + \gamma s\Gamma} [\sigma'\hat{e}/x]\sigma_0' e_0 : \tau_0}$$

□

Now we come to the central technical lemma of this section. The reader may wish to refer to the immediately following Theorem 3.1.6 to see how the statement of this lemma arises.

3.1.5 Lemma [Metric Compatibility]: $\Gamma \vdash e : \tau$ and $\vdash \sigma \sim_\gamma \sigma' : \Gamma^\circ$. If $\sigma e \mapsto e_f$, then then there exists e'_f such that $\sigma' e \mapsto e'_f$ and $\vdash e_f \approx_{\gamma\Gamma} e'_f : \tau$.

Proof: By induction on the typing derivation of e, with further case analysis on the evaluation step taken. The subscript f is used to denote the **f**uture state of evaluation. We show representative cases below.

Case: e's typing derivation is

$$\frac{\Gamma_L \vdash e_L : \tau_0 \multimap \tau \qquad \Gamma_R \vdash e_R : \tau_0}{\Gamma_L + \Gamma_R \vdash e_L\ e_R : \tau} \multimap E$$

By direct construction of derivations we find

$$\vdash \sigma e_L \approx_{\gamma\Gamma_L} \sigma' e_L : \tau_0 \multimap \tau$$

$$\vdash \sigma e_R \approx_{\gamma\Gamma_R} \sigma' e_R : \tau_0$$

Split cases on the evaluation rule used to show that $\sigma e \mapsto e_f$.

Subcase: $\dfrac{\sigma e_L \mapsto e_{Lf}}{\sigma e_L\ \sigma e_R \mapsto e_{Lf}\ \sigma e_R}$

By the induction hypothesis, there exists e'_{Lf} such that $\sigma' e_L \mapsto e'_{Lf}$ and $\vdash e_{Lf} \approx_{\gamma\Gamma_L} e'_{Lf} : \tau_0 \multimap \tau$. By appropriate substitutions into $x :_1 \tau_0 \multimap \tau, y :_1 \tau_0 \vdash x\ y \approx_0 x\ y : \tau$, we get that $\vdash e_{Lf}\ \sigma e_R \approx_{\gamma\Gamma_L+\gamma\Gamma_R} e'_{Lf}\ \sigma' e_R : \tau$, as required.

Subcase: $\dfrac{\sigma e_R \mapsto e_{Rf}}{\sigma e_L\ \sigma e_R \mapsto \sigma e_L\ e_{Rf}}$

Here it must be that σe_L is a value. Hence $\sigma' e_L$ must also be a value. By the induction hypothesis, there exists e'_{Rf} such that $\sigma' e_R \mapsto e'_{Rf}$ and $\vdash e_{Rf} \approx_{\gamma\Gamma_R} e'_{Rf} : \tau_0$. By appropriate substitutions into $x :_1 \tau_0 \multimap \tau, y :_1 \tau_0 \vdash x\ y \approx_0 x\ y : \tau$, we get that $\vdash \sigma e_L\ e_{Rf} \approx_{\gamma\Gamma_L+\gamma\Gamma_R} \sigma' e_{Lf}\ e'_{Rf} : \tau$, as required.

Subcase: $\dfrac{}{(\lambda x.e_0)\ (\sigma e_R) \mapsto [(\sigma e_R)/x]e_0}$

Here it must be that σe_R is a value, and $\sigma e_L = \lambda x.e_0$. Hence $\sigma' e_R$ is also a value, and $\sigma' e_L$ is of the form $\lambda x.e'_0$. By Corollary 3.1.3 (the coincidence of $\sim$ and $\approx$ on values) we know

$$\vdash \lambda x.e_0 \sim_{\gamma\Gamma_L} \lambda x.e'_0 : \tau_0 \multimap \tau$$

and by inversion we find

$$x :_1 \tau_0 \vdash e_0 \approx_{\gamma\Gamma_L} e'_0 : \tau$$

Plainly $(\lambda x.e'_0)\ (\sigma' e_R) \mapsto [(\sigma' e_R)/x]e'_0$, so what we must show is that

$$\vdash [(\sigma e_R)/x]e_0 \approx_{\gamma\Gamma_L+\gamma\Gamma_R} [(\sigma' e_R)/x]e'_0 : \tau$$

but this follows immediately by the substitution lemma for $\approx$.

Case:

$$\frac{\Gamma, x :_1 \tau_L \vdash e_0 : \tau_R}{\Gamma \vdash \lambda x.e_0 : \tau_L \multimap \tau_R} \to I$$

This case cannot arise. Since e is $\lambda x.e_0$, then no matter what σ is, σe is still a function value $\lambda x.\sigma e_0$ and cannot take a step.

Case: e's typing derivation is

$$\frac{\Gamma \vdash e_0 : \tau}{s\Gamma \vdash !e_0 : !_s\tau}\, !I$$

in this case the evaluation step must have been of the form

$$\frac{\sigma e_0 \mapsto e_{0f}}{!\sigma e_0 \mapsto !e_{0f}}$$

By induction hypothesis, there exists e'_{0f} such that $\vdash e_{0f} \approx_{\gamma\Gamma} e'_{0f}$ and $\sigma' e_0 \mapsto e'_{0f}$. Therefore $!\sigma e'_0 \mapsto !e'_{0f}$. We can form a derivation of $x :_s \tau \vdash !x : !_s\tau$, and from it a derivation of $x :_s \tau \vdash !x \approx_0 !x : !_s\tau$, and by substitution into $\approx$ we get $\vdash !e_{0f} \approx_{s\gamma\Gamma} !e'_{0f} : !_r\tau$, as required.

Case: e's typing derivation is

$$\frac{\Gamma \vdash e : !_s\tau_L \qquad \Delta, x :_{rs} \tau_L \vdash e_R : \tau_R}{\Delta + r\Gamma \vdash \mathbf{let}\, !x = e_L \,\mathbf{in}\, e_R : \tau_R}\, !E$$

Subcase: The evaluation step is

$$\frac{\sigma e_L \mapsto e_{Lf}}{\mathbf{let}\, !x = \sigma e_L \,\mathbf{in}\, \sigma e_0 \mapsto \mathbf{let}\, !x = e_{Lf} \,\mathbf{in}\, \sigma e_0}$$

By induction hypothesis, there exists e'_{Lf} such that we have $\sigma' e_L \mapsto e'_{Lf}$ and $\vdash e_{Lf} \approx_{\gamma\Gamma} e'_{Lf} : !_s\tau_L$. We can form a derivation

$$\frac{\vdash \sigma \sim_\gamma \sigma' : \Delta^\circ \qquad \Delta, y :_r \tau_L \vdash \mathbf{let}\, !x = y \,\mathbf{in}\, e_R : \tau_R}{y :_r \tau_L \vdash \mathbf{let}\, !x = y \,\mathbf{in}\, \sigma e_R \approx_{\gamma\Delta} \mathbf{let}\, !x = y \,\mathbf{in}\, \sigma' e_R : \tau_R}$$

(assuming without loss of generality that $\Gamma^\circ = \Delta^\circ$) and, by the substitution property for $\approx$, obtain

$$\vdash \mathbf{let}\, !x = e_{Lf} \,\mathbf{in}\, \sigma e_R \approx_{\gamma\Delta + r\gamma\Gamma} \mathbf{let}\, !x = e'_{Lf} \,\mathbf{in}\, \sigma' e_R : \tau_R$$

as required.

Subcase: The evaluation step is

$$\frac{}{\mathbf{let}\, !x = !v \,\mathbf{in}\, \sigma e_R \mapsto [v/x]\sigma e_R}$$

where $\sigma e_L = !v$ is a value. We can directly obtain:

$$\vdash \sigma e_L \approx_{\gamma\Gamma} \sigma' e_L : !_s\tau_L$$

$$x :_{rs} \tau_L \vdash \sigma e_R \approx_{\gamma\Delta} \sigma' e_R : \tau_R$$

But by Corollary 3.1.3 and inversion on the value metric rules, the former of these implies we must have had a derivation of

$$\frac{\vdash \sigma e_L \sim_q \sigma' e_L : \tau_L}{\vdash \sigma e_L \sim_{\gamma\Gamma} \sigma' e_L : !_s\tau_L}$$

for some number q such that $qs = \gamma\Gamma$. (We avoid saying $q = \gamma\Gamma/s$ since s might have been ∞.) So $\sigma' e_L$ is also a value, of the form $!v'$. Thus we have

$$\mathbf{let}\, !x = \sigma' e_L \,\mathbf{in}\, \sigma e_R \mapsto [v'/x]\sigma' e_R$$

and substitution leads to

$$\vdash [v/x]\sigma e_R \approx_{\gamma\Delta + qrs} [v'/x]\sigma' e_R$$

as required, since $\gamma\Delta + qrs = \gamma(\Delta + r\Gamma)$. □

We now obtain that the metric is preserved by each step of evaluation: if two expressions are close, and one takes a step, then the other must also take a step, and the two results remain as close.

3.1.6 Theorem [Metric Preservation]: If $\vdash e \approx_r e' : \tau$ and $e \mapsto e_f$, then there exists e'_f such that $e' \mapsto e'_f$ and $\vdash e_f \approx_r e'_f : \tau$.

Proof: By inversion on the rule defining $\approx$, we know that e, e', r must arise as $e = \sigma e_0$ and $e' = \sigma' e_0$ and $r = \gamma\Gamma$ such that

$$\frac{\vdash \sigma \sim_\gamma \sigma' : \Gamma^\circ \qquad \Gamma \vdash e_0 : \tau}{\vdash \sigma e_0 \approx_{\gamma\Gamma} \sigma' e_0 : \tau}$$

The proof is completed by appeal to the previous lemma. □

From this we can make a corresponsing statement about the complete evaluation of two expressions. Let $e \hookrightarrow v$ means that $e \mapsto \cdots \mapsto v$.

3.1.7 Theorem [Big-Step Metric Preservation]: If $\vdash e \approx_r e' : \tau$ and $e \hookrightarrow v$, then there exists v' such that $e' \hookrightarrow v'$ and $\vdash v \sim_r v' : \tau$.

Proof: By repeated application of Theorem 3.1.6, we find $\vdash v \approx_r v' : \tau$. But by Corollary 3.1.3, we know then also that $\vdash v \sim_r v' : \tau$. □

3.2 Primitive Operations

In the following sections, we will want to add typed primitive operations to make it possible to write realistic programs. In this section we schematically show what reasoning is required for each new primitive we want to add.

Suppose we want to add a 1-sensitive unary operation $\mathbf{f}$, whose input is of type τ, and whose return type is τ'. Operations of other arities and sensitivies can be achieved by judicious choice of τ. We sometimes will write suggestively that $\mathbf{f} : \tau \multimap \tau'$, even though this is somewhat an abuse of notation: $\mathbf{f}$ without its argument is not itself a well-formed expression. We assume that we have specified an underlying mapping (i.e., an ordinary mathematical function) f that takes a closed value of type τ, and yields a closed value of type τ'. Moreover we require f, as a mathematical function, is 1-sensitive:

Postulate Suppose we have $\vdash v : \tau$ and $\vdash v' : \tau$ and $\vdash v \sim_r v' : \tau$. Then $\vdash f(v) \sim_r f(v') : \tau$.

We extend the language with the expression

$$e \quad ::= \quad \cdots \mid \mathbf{f}(e)$$

and add a typing rule for it:

$$\frac{\Delta \vdash e : \tau}{\Delta \vdash \mathbf{f}(e) : \tau'}$$

as well as a pair of evaluation rules:

$$\frac{e \mapsto e'}{\mathbf{f}(e) \mapsto \mathbf{f}(e')} \qquad \frac{}{\mathbf{f}(v) \mapsto f(v)}$$

The cases introduced by this new expression former for the basic metatheoretic results (weakening, substitution, preservation, progress) are trivial to show. Since we introduce no new forms of values, many of the lemmas that work towards the metric preservation theorem are unchanged.

The new work we must do is confined to Lemma 3.1.5. The interesting case is for the evaluation step that encounters all values in the arguments of $\mathbf{f}$ and therefore actually invokes f. We have by assumption a derivation

$$\frac{\Delta \vdash e : \tau}{\Delta \vdash \mathbf{f}(e) : \tau'}$$

where by weakening we may assume without loss that $\Delta^\circ = \Gamma^\circ$. We also have substitutions $\vdash \sigma \sim_\gamma \sigma' : \Gamma^\circ$, and the evaluation step

$$\overline{\mathbf{f}(\sigma e) \mapsto f(\sigma e)}$$

together with the knowledge that σe is a value. By Lemma 3.1.2, $\sigma' e$ is also a value, such that $\vdash \sigma e \sim_{\gamma\Delta} \sigma' e : \tau'$. But from this we directly get

$$\overline{\mathbf{f}(\sigma' e) \mapsto f(\sigma' e)}$$

and by assumption of f, we know therefore that

$$\vdash f(\sigma e) \sim_{\gamma\Delta} f(\sigma' e) : \tau'$$

from which it follows that

$$\vdash f(\sigma e) \approx_{\gamma\Delta} f(\sigma' e) : \tau'$$

as required.

4 Examples

We now present some more complex examples of programs that can be written in our language. We continue to introduce new base types and new constants as they become relevant. For readability, we use syntactic sugar for case analysis and pattern matching *á la* ML.

4.1 Arithmetic Primitives

As a warm-up, add a base type $\mathbb{R}$ whose values are real numbers, with the metric rule

$$\frac{|r - r'| \leq k}{\vdash r \sim_k r' : \mathbb{R}}$$

For an example primitive operation, consider adding $\mathbf{plus} : \mathbb{R} \otimes \mathbb{R} \multimap \mathbb{R}$. By the result of the previous section, we must show that whenever we have $\vdash v \sim_k v' : \mathbb{R} \otimes \mathbb{R}$, that we also have $\vdash \mathbf{plus}(v) \sim_k \mathbf{plus}(v') : \mathbb{R}$. But every closed value v of type $\mathbb{R} \otimes \mathbb{R}$ must be a pair of real numbers (r_1, r_2), and likewise v' must be of the form (r_1', r_2'). Plugging in the intended semantics of the primitive as being addition of the underlying numbers, our proof obligation is to show from $\vdash (r_1, r_2) \sim_k (r_1', r_2') : \mathbb{R} \otimes \mathbb{R}$ that $\vdash (r_1 + r_2) \sim_k (r_1' + r_2') : \mathbb{R}$. By inversion of the tensor metric rule, we know that $\vdash r_1 \sim_{k_1} r_1' : \mathbb{R}$ and $\vdash r_2 \sim_{k_2} r_2' : \mathbb{R}$ for some k_1 and k_2 whose sum is k. But then by the triangle inequality we have $|(r_1 + r_2) - (r_1' + r_2')| \leq |r_1 - r_1'| + |r_2 - r_2'| \leq k_1 + k_2 = k$.

Note that if we prefer to have a curried addition function, we can easily write

$$\mathit{curriedPlus} = \lambda x.\lambda y.\mathbf{plus}(x, y)$$

so that

$$\vdash \mathit{curriedPlus} : \mathbb{R} \multimap \mathbb{R} \multimap \mathbb{R}$$

When we declare further primitives below, we tend to write them with curried function types without explicitly discussing the requisite wrapper. This is particularly more readable in the case of unrestricted functions: suppose that we wanted to add a multiplication primitive. Multiplication has no bounded sensitivity as a function from $\mathbb{R} \otimes \mathbb{R}$ to $\mathbb{R}$, but by using the ! type operator it can be added as a primitive $\mathbf{times} : !_\infty\mathbb{R} \otimes !_\infty\mathbb{R} \multimap \mathbb{R}$. Going through the same routine as above, we must show that $\vdash (r_1, r_2) \sim_k (r_1', r_2') : !_\infty\mathbb{R} \otimes !_\infty\mathbb{R}$ that implies $\vdash (r_1 + r_2) \sim_k (r_1' + r_2') : \mathbb{R}$, for every r_1, r_1', r_2, r_2', k.

However, because of the $!_\infty$s, the metric relation $\vdash (r_1, r_2) \sim_k (r_1', r_2') : !_\infty\mathbb{R} \otimes !_\infty\mathbb{R}$ only holds at all when $k = 0$ (and $r_1 = r_1'$ and $r_2 = r_2'$), and our obligation is trivialized. We can improve our programs'

appearance every time we call **times** by again currying, and hiding the expression-formers for ! inside a wrapper function:

$$\mathit{curriedTimes} = \lambda x.\lambda y.\mathbf{times}(!x, !y)$$

so that

$$\vdash \mathit{curriedTimes} : \mathbb{R} \to \mathbb{R} \to \mathbb{R}.$$

Again, when we introduce primitives below, we will freely give just the curried type without explicitly rehearsing this construction of wrapper functions.

Even if multiplication has no bounded sensitivity, it is sometimes useful having multiplication for a given positive real number. This allows one to scale both values and functions. For each $c \in \mathbb{R}^{>0}$ a primitive $\mathbf{scale}_c$ can be added as

$$\vdash \mathbf{scale}_c : !_c\mathbb{R} \to \mathbb{R}$$

These primitives will be useful in Section 6.

4.2 Fixpoint Combinator

Because we have general recursive types, we can simulate a fixpoint combinator in more or less the usual way: we just need to be a little careful about how sensitivity interacts with fixpoints.

Let $\tau_0 = \mu\alpha.\alpha \to (\tau \multimap \sigma)$. Then the expression

$$\begin{array}{rcl} Y_f & = & (\lambda x.\lambda a.f\ ((\mathbf{unfold}_{\tau_0}\ x)\ x)\ a) \\ & & \quad (\mathbf{fold}_{\tau_0}\ (\lambda x.\lambda a.f\ ((\mathbf{unfold}_{\tau_0}\ x)\ x)\ a)) \end{array}$$

satisfies

$$f :_\infty (\tau \multimap \sigma) \to (\tau \multimap \sigma) \vdash Y_f : \tau \multimap \sigma$$

This is the standard call-by-value fixed point operator (differing from the more familiar Y combinator by the two $\lambda a \cdots a$ eta-expansions). It is easy to check that the unfolding

$$Y_f\ v \mapsto^* f\ Y_f\ v$$

takes place if we let f be a function value $\lambda x.e$.

We could alternatively add a fixpoint operator $\mathbf{fix}f.e$ to the language directly, with the following typing rule:

$$\frac{\Gamma, f :_\infty \tau \multimap \sigma \vdash e : \tau \multimap \sigma}{\infty\Gamma \vdash \mathbf{fix}f.e : \tau \multimap \sigma}$$

This rule reflects the type we assigned to Y above: uses of **fix** can soundly be compiled away by defining $\mathbf{fix}f.e = Y_{\lambda f.e}$. The fact that f is added to the context annotated ∞ means that we are allowed to call the recursive function an unrestricted number of times within e. The context Γ must be multiplied by ∞ in the conclusion because we can't (because of the fixpoint), establish any bound on how sensitive the overall function is from just one call to it. In the rest of the examples, we write recursive functions in the usual high-level form, eliding the translation in terms of Y.

4.3 Lists

We can define the type of lists with elements in τ as follows:

$$\tau\ \mathsf{list} = \mu\alpha.1 + \tau \otimes \alpha$$

We write $[\,]$ for the nil value $\mathbf{fold}_{\tau\ \mathsf{list}}\,\mathbf{inj}_1()$ and $h :: tl$ for $\mathbf{fold}_{\tau\ \mathsf{list}}\,\mathbf{inj}_2(h, tl)$, and we use common list notations such as $[a, b, c]$ for $a :: b :: c :: [\,]$. Given this, it is straightforward to program *map* in the usual way.

$$\begin{array}{l} \mathit{map} : (\tau \multimap \sigma) \to (\tau\ \mathsf{list} \multimap \sigma\ \mathsf{list}) \\ \mathit{map}\ f\ [\,] = [\,] \\ \mathit{map}\ f\ (h :: tl) = (f\ h) :: \mathit{map}\ f\ tl \end{array}$$

The type assigned to *map* reflects that a nonexpansive function mapped over a list yields a nonexpansive function on lists. Every bound variable is used exactly once, with the exception of f; this is permissible since f appears in the context during the typechecking of *map* with an ∞ annotation.

Similarly, we can write the usual fold combinators over lists:

$$\begin{array}{l} \mathit{foldl} : (\tau \otimes \sigma \multimap \sigma) \to (\sigma \otimes \tau\ \mathsf{list}) \multimap \sigma \\ \mathit{foldl}\ f\ (\mathit{init}, [\,]) = \mathit{init} \\ \mathit{foldl}\ f\ (\mathit{init}, (h :: tl)) = \mathit{foldl}\ f\ (f(h, \mathit{init}), tl) \end{array}$$

$$\begin{array}{l} \mathit{foldr} : (\tau \otimes \sigma \multimap \sigma) \to (\sigma \otimes \tau\ \mathsf{list}) \multimap \sigma \\ \mathit{foldr}\ f\ (\mathit{init}, [\,]) = \mathit{init} \\ \mathit{foldr}\ f\ (\mathit{init}, (h :: tl)) = f\ (h, \mathit{foldr}\ f\ (\mathit{init}, tl)) \end{array}$$

Again, every bound variable is used once, except for f, which is provided as an unrestricted argument, making its repeated use acceptable. The fact that the initializer to the fold (of type σ) together with the list to be folded over (of type $\tau\ \mathsf{list}$) occur to the left of a $\multimap$ is essential, capturing the fact that variation in the initializer and in every list element can jointly affect the result.

Binary and iterated concatenation are also straightforwardly implemented:

$$\begin{array}{l} @ : \tau\ \mathsf{list} \otimes \tau\ \mathsf{list} \multimap \tau\ \mathsf{list} \\ @\ ([\,], x) = x \\ @\ (h :: tl, x) = h :: @\ (tl, x) \end{array}$$

$$\begin{array}{l} \mathit{concat} : \tau\ \mathsf{list}\ \mathsf{list} \multimap \tau\ \mathsf{list} \\ \mathit{concat}\ [\,] = [\,] \\ \mathit{concat}\ (h :: tl) = @\ (h, \mathit{concat}\ tl) \end{array}$$

If we define the natural numbers as usual by

$$\begin{array}{l} \mathsf{nat} = \mu\alpha.1 + \alpha \\ z = \mathbf{fold}_{\mathsf{nat}}\, \mathbf{inj}_1() \\ s\ x = \mathbf{fold}_{\mathsf{nat}}\, \mathbf{inj}_2\, x \end{array}$$

then we can implement a function that finds the length of a list as follows:

$$\begin{array}{l} \mathit{length} : \tau\ \mathsf{list} \multimap \mathsf{nat} \\ \mathit{length}\ [\,] = z \\ \mathit{length}\ (h :: tl) = s\ (\mathit{length}\ tl) \end{array}$$

However, this implementation is less than ideal, for it 'consumes' the entire list in producing its answer, leaving further computations unable to depend on it. We can instead write

$$\begin{array}{l} \mathit{length} : \tau\ \mathsf{list} \multimap \tau\ \mathsf{list} \otimes \mathsf{nat} \\ \mathit{length}\ [\,] = ([\,], z) \\ \mathit{length}\ (h :: tl) = \mathbf{let}(tl', \ell) = \mathit{length}\ tl\ \mathbf{in}(h :: tl', s\ \ell) \end{array}$$

which deconstructs the list enough to determine its length, but builds up and returns a fresh copy that can be used for further processing. Consider why this function is well-typed: as it decomposes the input list into h and tl, the *value* of h is only used once, by including it in the output. Also, tl is only used once, as it is passed to the recursive call, which is able to return a reconstructed copy tl', which is then included in the output. At no point is any data duplicated, but only consumed and reconstructed.

The definition on lists permits to distribute the metric-scale modality over list elements. Indeed, for each $c \in \mathbb{R}$ we have a function:

$$\begin{array}{l} \mathit{ldistribute} : !_c(\tau\ \mathsf{list}) \multimap !_c \tau\ \mathsf{list} \\ \mathit{ldistribute}\ !([\,]) = [\,] \\ \mathit{ldistribute}\ !(h :: tl) = !h :: (\mathit{ldistribute}\ tl) \end{array}$$

These functions will be useful in Section 6.

4.4 &-lists

Another definition of lists uses & instead of $\otimes$: we can say $\tau\,\mathsf{alist} = \mu\alpha.1 + \tau\ \&\ \alpha$. (the 'a' in alist is for 'ampersand'). To distinguish these lists visually from the earlier definition, we write *Nil* for $\mathbf{fold}_{\tau\,\mathsf{alist}}\,\mathbf{inj}_1()$ and *Cons* p for $\mathbf{fold}_{\tau\,\mathsf{list}}\,\mathbf{inj}_2\,p$.

Recall that & is eliminated by projection rather than pattern-matching. This forces certain programs over lists to be implemented in different ways. We can still implement *map* for this kind of list without much trouble.

$$\begin{array}{l} \mathit{amap} : (\tau \multimap \sigma) \to (\tau\,\mathsf{alist} \multimap \sigma\,\mathsf{alist}) \\ \mathit{amap}\ f\ \mathit{Nil} = \mathit{Nil} \\ \mathit{amap}\ f\ (\mathit{Cons}\ p) = \mathit{Cons}\langle f\ (\pi_1 p), \mathit{map}\ f\ (\pi_2 p)\rangle \end{array}$$

This function is well-typed (despite the apparent double use of p in the last line!) because the $\&I$ rule allows the two components of an &-pair to use the same context. This makes sense, because the eventual fate of an &-pair is to have one or the other of its components be projected out.

The *fold* operations are more interesting. Consider a naïve implementation of *foldl* for alist

$$\begin{array}{l} \mathit{afoldl} : (\tau\ \&\ \sigma \multimap \sigma) \to (\sigma\ \&\ \tau\,\mathsf{alist}) \multimap \sigma\,\mathsf{alist} \\ \mathit{afoldl}\ f\ p = \mathbf{case}\,\pi_2 p\,\mathbf{of}\,x.\ \pi_1 p \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid x.\ \mathit{afoldl}\ f\ \langle f\langle \pi_1 x, \pi_1 p\rangle, \pi_2 x\rangle \end{array}$$

where we have replaced $\otimes$ with & everywhere in *foldl*'s type to get the type of *afoldl*. This program is *not* well-typed, because $\pi_1 p$ is still used in each branch of the case despite the fact that $\pi_2 p$ is case-analyzed. The $+E$ rule sums together these uses, so the result has sensitivity 2, while *afoldl* is supposed to be only 1-sensitive to its argument of type $\sigma\ \&\ \tau\,\mathsf{alist}$.

We would like to case-analyze the structure of the second component of that pair, the $\tau\,\mathsf{alist}$, without effectively consuming the first component. The existing type system does not permit this, but we can soundly add a primitive[2]

$$\mathit{analyze} : \sigma\ \&\ (\tau_1 + \tau_2) \multimap (\sigma\ \&\ \tau_1) + (\sigma\ \&\ \tau_2)$$

that gives us the extra bit that we need. The operational behavior of *analyze* is simple: given a pair value $\langle v, \mathbf{inj}_i\, v'\rangle$ with $v : \sigma$ and $v' : \tau_i$, it returns $\mathbf{inj}_i\,\langle v, v'\rangle$. With this primitive, a well-typed implementation of *afoldl* can be given as follows:

$$\begin{array}{l} \mathit{unf} : (\sigma\ \&\ \tau\,\mathsf{alist}) \multimap (\sigma\ \&\ (1 + \tau\ \&\ \tau\,\mathsf{alist})) \\ \mathit{unf}\ p = \langle \pi_1 p, \mathbf{unfold}_{\tau\,\mathsf{alist}}\,\pi_2 p\rangle \end{array}$$

$$\begin{array}{l} \mathit{afoldl} : (\tau\ \&\ \sigma \multimap \sigma) \to (\sigma\ \&\ \tau\,\mathsf{alist}) \multimap \sigma\,\mathsf{alist} \\ \mathit{afoldl}\ f\ p = \mathbf{case}\ \mathit{analyze}\ (\mathit{unf}\ p)\ \mathbf{of} \\ \quad x : (\sigma\ \&\ 1).\ \pi_1 x \\ \mid x : (\sigma\ \&\ (\tau\ \&\ \tau\,\mathsf{alist})).\ \mathit{afoldl}\ f\ \langle f\langle \pi_1\pi_2 x, \pi_1 x\rangle, \pi_2\pi_2 x\rangle \end{array}$$

4.5 Sets

Another useful collection type is finite sets. We posit that $\tau\,\mathsf{set}$ is a type for any type τ, whose values are finite sets $\{v_1, \ldots, v_n\}$ of closed values of type τ. The metric on $\tau\,\mathsf{set}$ is the Hamming metric, defined by the rule

$$\frac{\|S_1 \mathbin{\triangle} S_2\| \le r}{\vdash S_1 \sim_r S_2 : \tau\,\mathsf{set}}$$

[2]The reader may note that this primitive is exactly the well-known distributivity property that the BI, the logic of bunched implications [OP99], notably satisfies in contrast with linear logic. We conjecture that a type system based on BI might also be suitable for distance-sensitive computations, but we leave this to future work, because of uncertainties about the decidability of typechecking and BI's lack of exponentials, that is, operators such as !, which are important for interactions between distance-sensitive and -insensitive parts of a program.

where $\triangle$ indicates symmetric difference of sets, and $\|S\|$ the cardinality of the set S. In short, the distance between two sets is the number of elements that are in one set but not in the other.

Note that there is no obvious way to implement this type of sets in terms of the list types just presented, because of the metric; two sets of different size are a finite distance from one another, but two lists (as implemented above) of different size are infinitely far apart.

Some basic set primitives that can be include

$$\begin{array}{l} \mathit{size} : \tau\ \mathsf{set} \multimap \mathbb{R} \\ \cap, \cup, \setminus : \tau\ \mathsf{set} \otimes \tau\ \mathsf{set} \multimap \tau\ \mathsf{set} \end{array}$$

where *size* returns the cardinality of a set, $\cap$ returns the intersection of two sets, $\cup$ their union, and $\setminus$ the difference. It is easy to check that these satisfy the postulate in Section 3.2. For example, *size* is 1-sensitive because the insertion or deletion of any element from the set input only results in a numeric change of 1 in the output. Notably, for the last three primitives above, we could *not* have given them the type $\tau\ \mathsf{set}\ \&\ \tau\ \mathsf{set} \multimap \tau\ \mathsf{set}$. For an informal counterexample, consider $\{b\} \cup \{c, d\} = \{b, c, d\}$ and $\{a\} \cup \{c, d, e\} = \{a, c, d, e\}$. We have $\{b\} \sim_2 \{a\}$ and $\{c, d\} \sim_1 \{c, d, e\}$ on the two inputs to $\cup$, but we can only get $\{b, c, d\} \sim_3 \{a, c, d, e\}$, a greater distance than $2 = \max(2, 1)$.

We would also like to add primitives to manipulate sets. In particular we would like to add primitives forstandard higher-order functional operations like filter and map. We might suppose we could include the following operations:

$$\begin{array}{l} \mathit{setfilter} : (\tau \to \mathsf{bool}) \to \tau\ \mathsf{set} \multimap \tau\ \mathsf{set} \\ \mathit{setmap} : (\sigma \to \tau) \to \sigma\ \mathsf{set} \multimap \tau\ \mathsf{set} \end{array}$$

However, as written, these types are not correct for the intended behavior. The problem has to do with the possibility of nontermination in the non sensitive functions passed in as an argument.

Consider the function

$$f =_{def} \lambda x.\, \mathbf{if}\ x = 0\ \mathbf{then}\ \mathit{loop}()\ \mathbf{else}\ \mathbf{true} : \mathbb{N} \to \mathsf{bool}$$

(where *loop* is a function that goes into an infinite loop) and the two programs

$$f\ \{1, 2, 3\} \qquad \text{and} \qquad f\ \{0, 1, 2, 3\}.$$

We have $\vdash \{1, 2, 3\} \sim_1 \{0, 1, 2, 3\} : \mathbb{N}\,\mathsf{set}$, and yet we would expect to have *setfilter* $f\ \{1, 2, 3\} \hookrightarrow \{1, 2, 3\}$ while *setfilter* $f\ \{0, 1, 2, 3\}$ should diverge.

It is worth noticing the difference with the list case. Remember that map on lists has been defined with type

$$(\tau \multimap \sigma) \to (\tau\ \mathsf{list} \multimap \sigma\ \mathsf{list})$$

So, in the list case the function f defined above cannot be passed as argument. One can wonder why this difference? Or even one can think to restrict map and filter also in the set case to functions which are 1-sensitive. In the case of sets where elements cannot be accessed directly, 1-sensitive functions are of little interest because these are functions that cannot really distinghuish between elements. While can be accepted on list where there is a direct access to the elements, in the case of sets one want to use functions as map and filter to obtain set manipulations through an (indirect) access to the elements. So, one should be able to map or filter using non sensitive functions.

This problem can be solved by imposing limits on the execution of the functional argument. We add instead

$$\begin{array}{l} \mathit{setfilter} : (\tau \to \mathsf{bool}) \to \mathbb{N} \to \tau\ \mathsf{set} \multimap \tau\ \mathsf{set} \\ \mathit{setmap} : (\sigma \to \tau) \to \mathbb{N} \to \tau \to \sigma\ \mathsf{set} \multimap \tau\ \mathsf{set} \end{array}$$

where the argument of type $\mathbb{N}$ is the number of steps we allow the function to run before (in the case of *setfilter*) eliminating the element from the filtered set or (in the case of *setmap*) returning instead a default element of type τ, provided as an argument to *setmap*.

Formally, we specify the underlying mathematical functions of these as

$$\mathit{setfilter}(f, \ell, \{v_1, \ldots, v_n\}) = \{v_i \mid \exists v. f\ v_i \mapsto^k \mathbf{true} \wedge k \leq \ell \wedge i \in 1 \ldots n\}$$

$$\mathit{mapone}(f, \ell, v_{\mathsf{def}}, v) = \begin{cases} v' & \text{if } f\ v \mapsto^k v' \text{ and } k \leq \ell; \\ v_{\mathsf{def}} & \text{otherwise.} \end{cases}$$

$$\mathit{setmap}(f, \ell, v_{\mathsf{def}}, \{v_1, \ldots, v_n\}) = \{\mathit{mapone}(f, \ell, v_{\mathsf{def}}, v_i) \mid i \in 1 \ldots n\}$$

It is then easy to show that these constitute (since they have finite bounds) computable and well-defined functions, and that they satisfy the criterion in Section 3.2. Requiring the programmer to provide time bounds may be considered unpleasant; an entirely reasonable alternative (which we do not pursue here) is to work with a language (or language fragment) that is statically guaranteed to terminate.

In fact *size* can be construed as special case of a more basic summation primitive:

$$\mathit{sum} : \mathbb{R}\,\mathsf{set} \multimap \mathbb{R}$$

$$\mathit{sum}(S) = \sum_{s \in S} \mathit{clip}(s)$$

where $\mathit{clip}(x)$ returns x clipped to the interval $[-1, 1]$ if necessary. This clipping is required for *sum* to be 1-sensitive in its set argument. Otherwise, an individual set element could affect the sum by an unbounded amount. We can then define $\mathit{size}\ S = \mathit{sum}\ (\mathit{setmap}\ (\lambda x.1)\ 1\ 0\ S)$.

Another useful operation on sets is converting a set of sum-typed elements into two sets of the underlying types. a primitive

$$\mathit{setdist} : (\tau_1 + \tau_2)\,\mathsf{set} \multimap \tau_1\,\mathsf{set} \otimes \tau_2\,\mathsf{set}$$

with underlying function

$$\mathit{setdist}(\{\mathbf{inj}_{k_1}\ v_1, \ldots, \mathbf{inj}_{k_n}\ v_n\}) = (\{v_i \mid k_i = 1\}, \{v_i \mid k_i = 2\}).$$

Note carefully that it *is* in fact acceptable to give the output a $\otimes$ type. If the set input to *setdist* changes by the insertion or deletion of one element, then only one of the two output sets is affected.

From this we can implement a higher-order function for splitting a set into two sets according to a provided predicate. Specifically, we can write the program

$$\begin{array}{l} \mathit{setsplit} : (\tau \to \mathsf{bool}) \to \mathbb{N} \to \tau \to \tau\,\mathsf{set} \multimap \tau\,\mathsf{set} \otimes \tau\,\mathsf{set} \\ \mathit{setsplit} = \lambda f.\lambda \ell.\lambda v_{\mathsf{def}}.\lambda S.\mathit{setdist}\ (\mathit{setmap} \\ \quad (\lambda x.\, \mathbf{if}\ f\ x\ \mathbf{then}\ \mathbf{inj}_1\ x\ \mathbf{else}\ \mathbf{inj}_2\ x)\ \ell\ (\mathbf{inj}_1\ v_{\mathsf{def}})\ S) \end{array}$$

By using *setsplit* repeatedly, we can write programs that, given a set of points in $\mathbb{R}$, computes a *histogram*, a list of counts indicating how many points are in each of many intervals. For a simple example, suppose our histogram bins are the intervals $(-\infty, 0], (0, 10], \ldots, (90, 100], (100, \infty)$.

$$\begin{array}{l} \mathit{hist}' : \mathbb{R} \to \mathbb{R}\,\mathsf{set} \multimap (\mathbb{R}\,\mathsf{set})\,\mathsf{list} \\ \mathit{hist}'\ c\ s = \mathbf{if}\ c \geq 101\ \mathbf{then}\ [s]\ \ \mathbf{else} \\ \qquad\qquad \mathbf{let}(y, n) = \mathit{setsplit}\ (\lambda z.c \geq z)\ 2\ 0\ s\ \ \mathbf{in} \\ \qquad\qquad\quad y :: \mathit{hist}'\ (c + 10)\ n \end{array}$$

$$\begin{array}{l} \mathit{hist} : \mathbb{R}\,\mathsf{set} \multimap \mathbb{R}\,\mathsf{list} \\ \mathit{hist}\ s = \mathit{map}\ \mathit{size}\ (\mathit{hist}'\ 0\ s) \end{array}$$

Note that it suffices here to use ordinary distance-insensitive arithmetic operations $\geq : \mathbb{R} \to \mathbb{R} \to \mathsf{bool}$ and $+ : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$. To see why these can be included, refer to the discussion in Section 4.1 of a comparable distance-insensitive multiplication operation. We see in the next section that comparison operators like $\geq$ cannot be so straightforwardly generalized to be distance sensitive.

4.6 Sorting

In this section we show how to code up a distance-sensitive sorting algorithm. Ordinarily, the basis of sorting functions is a comparison operator such as $\geq_\tau : \tau \times \tau \to \mathsf{bool}$. However, we cannot take $\geq_\mathbb{R} : \mathbb{R} \otimes \mathbb{R} \multimap \mathsf{bool}$ as a primitive, because $\geq$ is not 1-sensitive in either of its arguments: it has a glaring discontinuity. (Compare the example of *gtzero* in Section 2.1) Although $(0, \epsilon)$ and $(\epsilon, 0)$ are nearby values in $\mathbb{R} \otimes \mathbb{R}$ if ϵ is small (they are just 2ϵ apart), nonetheless $\geq_\mathbb{R}$ returns false for one and true for the other, values of bool that are by definition infinitely far apart.

Because of this we instead take as a primitive the conditional swap function $\mathit{cswp} : \mathbb{R} \otimes \mathbb{R} \multimap \mathbb{R} \otimes \mathbb{R}$ defined in Section 2.1, which takes in a pair, and outputs the same pair, swapped if necessary so that the first component is no larger than the second. We are therefore essentially concerned with *sorting networks*, [Bat68] with *cswp* being the comparator. With the comparator, we can easily implement a version of insertion sort.

$$
\begin{aligned}
&\mathit{insert} : \mathbb{R} \multimap \mathbb{R}\,\mathsf{list} \multimap \mathbb{R}\,\mathsf{list}\\
&\mathit{insert}\ x\ [\,] = [x]\\
&\mathit{insert}\ x\ (h :: tl) = \mathbf{let}(a, b) = \mathit{cswp}\ (x, h)\,\mathbf{in}\\
&\qquad\qquad\qquad\qquad a :: (\mathit{insert}\ b\ tl)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{sort} : \mathbb{R}\,\mathsf{list} \multimap \mathbb{R}\,\mathsf{list}\\
&\mathit{sort}\ [\,] = [\,]\\
&\mathit{sort}\ (h :: tl) = \mathit{insert}\ h\ (\mathit{sort}\ tl)
\end{aligned}
$$

Of course, the execution time of this sort is $\Theta(n^2)$. It is an open question whether any of the typical $\Theta(n \log n)$ sorting algorithms (merge sort, quick sort, heap sort) can be implemented in our language, but we can implement bitonic sort [Bat68], which is $\Theta(n(\log n)^2)$, and we conjecture that one can implement the log-depth (and therefore $\Theta(n \log n)$ time) sorting network due to Ajtai, Komlós, and Szemerédi [AKS83].

More generally, we say a *comparator* for a type τ is a function $\mathit{cswp}_\tau : \tau \otimes \tau \multimap \tau \otimes \tau$. We may ask the question: can we derive from cswp_τ comparators for types based on τ? For sum types, and the comparator that induces the usual sum-ordering on $\tau + \sigma$ where every value in τ is considered to be greater than every value of σ, the answer is yes.

$$
\begin{aligned}
&\mathit{cswp}_{\tau+\sigma} : (\tau + \sigma) \otimes (\tau + \sigma) \multimap (\tau + \sigma) \otimes (\tau + \sigma)\\
&\mathit{cswp}_{\tau+\sigma}(\mathbf{inj}_1\, x, \mathbf{inj}_1\, y) = \mathbf{let}(x', y') = \mathit{cswp}_\tau(x, y)\,\mathbf{in}\\
&\qquad\qquad (\mathbf{inj}_1\, x', \mathbf{inj}_1\, y')\\
&\mathit{cswp}_{\tau+\sigma}(\mathbf{inj}_2\, x, \mathbf{inj}_2\, y) = \mathbf{let}(x', y') = \mathit{cswp}_\sigma(x, y)\,\mathbf{in}\\
&\qquad\qquad (\mathbf{inj}_2\, x', \mathbf{inj}_2\, y')\\
&\mathit{cswp}_{\tau+\sigma}(\mathbf{inj}_1\, x, \mathbf{inj}_2\, y) = (\mathbf{inj}_2\, y, \mathbf{inj}_1\, x)\\
&\mathit{cswp}_{\tau+\sigma}(\mathbf{inj}_2\, x, \mathbf{inj}_1\, y) = (\mathbf{inj}_2\, x, \mathbf{inj}_1\, y)
\end{aligned}
$$

However, for $\otimes$ the answer seems to be no; lexicographic sorting of pairs is *not* 1-sensitive. Although there does exist a function

$$
\begin{aligned}
&\mathit{cswp}_{\tau\otimes\sigma} : (\tau \otimes \sigma) \otimes (\tau \otimes \sigma) \multimap (\tau \otimes \sigma) \otimes (\tau \otimes \sigma)\\
&\mathit{cswp}_{\tau\otimes\sigma}((x_1, y_1), (x_2, y_2)) =\\
&\quad \mathbf{let}(x_1', x_2') = \mathit{cswp}_\tau(x_1, x_2)\,\mathbf{in}\\
&\quad \mathbf{let}(y_1', y_2') = \mathit{cswp}_\sigma(y_1, y_2)\,\mathbf{in}\\
&\qquad ((x_1', y_1'), (x_2', y_2'))
\end{aligned}
$$

it does not serve as a sensible comparator operation for any ordering on $\tau \otimes \sigma$. For consider $\tau = \sigma = \mathbb{R}$, and observe that $\mathit{cswp}_{\mathbb{R}\otimes\mathbb{R}}((0, 5), (10, 1)) = ((0, 1), (10, 5))$, where neither $(0, 1)$ nor $(10, 5)$ were pairs that occurred in the input at all.

4.7 Finite Maps

Related to sets are finite maps from σ to τ, which we write as the type $\sigma \rightharpoonup \tau$. A finite map f from σ to τ is an unordered set of mappings $s \mapsto t$ where $s : \sigma$ and $t : \tau$, subject to the constraint that each key s has at most one value t associated with it: if $s \mapsto t \in f$ and $s \mapsto t' \in f$, then $t = t'$. One can think of finite maps as SQL databases where one column is distinguished as the primary key.

This type has essentially the same metric as the metric for sets, namely

$$\frac{\|D_1 \mathbin{\triangle} D_2\| \leq r}{\vdash D_1 \sim_r D_2 : \sigma \rightharpoonup \tau}$$

By isolating the primary key, we can support some familiar relational algebra operations:

$$\begin{aligned}
&\mathit{fmsize} : (\sigma \rightharpoonup \tau) \multimap \mathbb{R} \\
&\mathit{fmfilter} : (\sigma \to \tau \to \mathsf{bool}) \to \mathbb{N} \to (\sigma \rightharpoonup \tau) \multimap (\sigma \rightharpoonup \tau) \\
&\mathit{mapval} : (\tau_1 \to \tau_2) \to \mathbb{N} \to \tau_2 \to (\sigma \rightharpoonup \tau_1) \multimap (\sigma \rightharpoonup \tau_2) \\
&\mathit{join} : (\sigma \rightharpoonup \tau_1) \otimes (\sigma \rightharpoonup \tau_2) \multimap (\sigma \rightharpoonup (\tau_1 \otimes \tau_2))
\end{aligned}$$

The size and filter functions work similar to the corresponding operations on sets, and we can map over values. Similar to the discussion for higher-order operations for sets, we include a time limit and (in the case of *mapval*) a default value to use when the time limit is exceeded. The join operation takes two maps $(i, s_i)_{i \in I_1}$ and $(i, s'_i)_{i \in I_2}$, and outputs the map $(i, (s_1, s_2))_{i \in I_1 \cap I_2}$. This operation is 1-sensitive in the pair of input maps, but only because we have identified a unique primary key for both of them! For comparison, the cartesian product $\times$ on sets — the operation that join is ordinarily derived from in relational algebra — is *not* c-sensitive for any finite c, for we can see that $(\{x\} \cup X) \times Y$, when $x \notin X$, has $\|Y\|$ many additional elements compared to $X \times Y$, and $\|Y\|$ is not bounded. McSherry also noted this issue with unrestricted joins, and deals with it in a similar way in PINQ [McS09].

Finally, we are also able to support a form of GroupBy aggregation, in the form of a primitive

$$\mathit{group} : !_2(\sigma \otimes \tau)\,\mathsf{set} \multimap (\sigma \rightharpoonup (\tau\,\mathsf{set}))$$

which takes a set S of key-value pairs (s, t), where the keys are type σ and the values of type τ, and returns a finite map which maps values $s \in \sigma$ to the set of $t \in \tau$ such that $(s, t) \in S$. This function is 2-sensitive (thus the $!_2$) in the set argument, because the addition or removal of a single set element may *change* one element in the output map: it takes two steps to represent such a change as the removal of the old mapping, and the insertion of the new one. For example, $\vdash \{(a, b), (a, c), (a, d)\} \sim_1 \{(a, b), (a, c)\} : (\sigma \otimes \tau)\,\mathsf{set}$, and $\vdash \{a \mapsto \{b, c, d\}\} \sim_2 \{a \mapsto \{b, c\}\} : \sigma \rightharpoonup (\tau\,\mathsf{set})$.

5 A Calculus for Differential Privacy

We now describe how to apply the above type system to expressing *differentially private* computations. There are two ways to do this. One is to leverage the fact that our type system captures sensitivity, and use standard results about obtaining differential privacy by adding noise to c-sensitive functions. Since Theorem 2.2.1 guarantees that every well-typed expression $b :_c \mathsf{db} \vdash e : \mathbb{R}$ (for a type db of databases) is a c-sensitive function $\mathsf{db} \to \mathbb{R}$, we can apply Proposition 5.1.3 below to obtain a differentially private function by adding the appropriate amount of noise to the function's result. But we can do better. In this section, we show how adding a probability monad to the type theory allows us to directly capture differential privacy *within* our language.

5.1 Background

First, we need a few technical preliminaries from the differential privacy literature [Dwo06].

The definition of differential privacy is a property of randomized functions that take as input a *database*, and return a result, typically a real number.

For the sake of the current discussion, we take a database to be a set of 'rows', one for each user whose privacy we mean to protect. The type of one user's data—that is, of one row of the database—is written row. For example, row might be the type of a single patient's complete medical record. The type of databases is then $\mathsf{db} = \mathsf{row\,set}$; we use the letter b for elements of this type. Differential privacy is parametrized by a number ϵ, which controls how strong the privacy guarantee is: the smaller ϵ is, the more privacy is guaranteed. It is reasonable also to think about ϵ as a measure rather of *how much privacy can be lost* by allowing a query to take place. We assume from now on that we have fixed ϵ to some particular appropriate value.

Informally, a function is differentially private if it behaves statistically similarly on similar databases, so that any individual's presence in the database has a statistically negligible effect. Databases b and b' are considered *similar*, written $b \sim b'$ if they differ by at most one row—in other words if $d_{\mathsf{db}}(b, b') \leq 1$. The standard definition [DMNS06] of differential privacy for functions from databases to real numbers is as follows:

5.1.1 Definition: A random function $q : \mathsf{db} \to \mathbb{R}$ is ϵ-differentially private if for all $S \subseteq \mathbb{R}$, and for all databases b, b' with $b \sim_1 b'$, we have $Pr[q(b) \in S] \;\leq\; e^{\epsilon} Pr[q(b') \in S]$.

We see that for a differentially private function, when its input database has one row added or deleted, there can only be a very small multiplicative difference (e^{ϵ}) in the probability of *any* outcome S. For example, suppose an individual is concerned about their data being included in a query to a hospital's database; perhaps that the result of that query might cause them to be denied health insurance. If we require that query to be 0.1-differentially private (i.e., if ϵ is set to 0.1), then they can be reassured that the chance of them being denied health care can only increase by about 10%. (Note that this is a 10% increase *relative* to what the probability would have been without the patient's participation in the database. If the probability without the patient's data being included was 5%, then including the data raises it at most to 5.5%, not to 15%!)

It is straightforward to generalize this definition to other types, by using the distance between two inputs instead of the database similarity condition. We say:

5.1.2 Definition: A random function $q : \tau \to \sigma$ is ϵ-differentially private if for all sets S of closed values of type σ, and for all $v, v' : \tau$ such that $\vdash v \sim_r v' : \tau$, we have $Pr[q(v) \in S] \;\leq\; e^{\epsilon r} Pr[q(v') \in S]$.

For the time being, however, we continue considering only the special case of functions $\mathsf{db} \to \mathbb{R}$.

One way to achieve differential privacy is via the *Laplace mechanism*. We suppose we have a deterministic database query, a function $f : \mathsf{db} \to \mathbb{R}$ of known sensitivity, and we produce a differentially private function by adding *Laplace-distributed noise* to the result of f. The Laplace distribution $\mathcal{L}_k$ is parametrized by k—intuitively, a measure of the spread, or 'amount', of noise to be added. It has the probability density function $Pr[x] = \frac{1}{2k} e^{-|x|/k}$. The Laplace distribution is symmetric and centered around zero, and its probabilities fall off exponentially as one moves away from zero. It is a reasonable noise distribution, which is unlikely to yield values extremely far from zero. The intended behavior of the Laplace mechanism is captured by the following result:

5.1.3 Proposition [[DMNS06]]: Suppose $f : \mathsf{db} \to \mathbb{R}$ is c-sensitive. Define the random function $q : \mathsf{db} \to \mathbb{R}$ by $q = \lambda b.f(b) + N$, where N is a random variable distributed according to $\mathcal{L}_{c/\epsilon}$. Then q is ϵ-differentially private.

That is, the amount of noise required to make a c-sensitive function ϵ-private is c/ϵ. Stronger privacy requirements (smaller ϵ) and more sensitive functions (larger c) both require more noise.

Note that we must impose a global limit on how many queries can be asked of the same database: if we could ask the same query over and over again, we could eventually learn the true value of f with high probability despite the noise. If we exhaust the "privacy budget" for a given database, the database must be destroyed. This resource-consumption aspect of differentially private queries was the initial intuition that guided us to the use of ideas from linear logic in design of the type system.

5.2 The Probability Monad

Since the definition of differential privacy is probabilistic by nature, we now want to accomodate probabilistic features in our programming language. To do this we turn to Ramsey and Pfeffer's stochastic lambda calculus [RP02] for inspiration, and treat finite probability distributions over a type τ in a first-class way, letting them be the expressions that inhabit a monadic type $\bigcirc\tau$. We will see below in Section 5.3 that this suffices for accounting for the apparently continuous probability distributions usually involved in differential privacy, for we can easily universally quantify over all sound finite approximations to the continuous Laplace-distribution-based noising operator.

We extend the syntax of the language as follows:

$$\begin{array}{rrl}
\text{Types } \tau & ::= & \cdots \mid \mathcal{P} \mid \bigcirc\tau \\
\text{Prob. Vectors } p & ::= & (r_1, \ldots, r_n) \\
\text{Expressions } e & ::= & p \mid \mathbf{return}\, e \mid \{e, (e_1, \ldots, e_n)\} \mid \mathbf{let}\, \bigcirc x = e\, \mathbf{in}\, e \\
\text{Values } v & ::= & p \mid \mathbf{return}\, v \mid \{v, (e_1, \ldots, e_n)\} \\
\text{Evaluation States } s & ::= & \mathbf{do}\, e \mid \{p, (s_1, \ldots, s_n)\} \\
\text{Final States } f & ::= & \mathbf{do}\, \mathbf{return}\, v \mid \{p, (f_1, \ldots, f_n)\}
\end{array}$$

We add a type $\mathcal{P}$ for probability vectors, and a monadic type $\bigcirc\tau$ for probability distributions over τ. Probability vectors p are simply lists of numbers in the interval $[0, 1]$ that sum to 1. We single them out as a separate syntactic concept (as opposed to mixing in probabilities with expressions) so that we can focus on how the metric interacts with them separately from the expressions they label.

Expressions are extended by probability vectors, terms of the shape $\{e, (e_1, \ldots, e_n)\}$ and by monadic constructions. An expression $\{e, (e_1, \ldots, e_n)\}$ can be thought of as a thunk that has not yet actually interrogated any random number generator. Concerning monadic constructions, we add a monadic return: the expression $\mathbf{return}\, e$ can be interpreted as the distribution that deterministically always yields e, as well as monadic sequencing: the expression $\mathbf{let}\, \bigcirc x = e\, \mathbf{in}\, e'$ can be interpreted as drawing a sample x from the random computation e, and then continuing with the computation e'.

An evaluation state can be of the shape $\mathbf{do}\, e$ or of the shape $\{(r_1, \ldots, r_n), (s_1, \ldots, s_n)\}$, that is a 'superposition' of n different states, such that the probability of obtaining s_i is r_i. These superpositions can be nested. For an example, the state

$$\{(0.2, 0.8), (\mathbf{do}\, \mathbf{return}\, 1, \{(0.5, 0.5), (\mathbf{do}\, \mathbf{return}\, 2, \mathbf{do}\, \mathbf{return}\, 3)\})\}$$

is a final state, which represents a 0.2 probability of yielding the value 1, and a 0.4 ($= 0.8 \cdot 0.5$) probability of yielding either the value 2 or 3, respectively.

There are new typing and metric judgments for final states (which are analogous to values) and general evaluation states (which are respectively analogous to expressions):

$$\begin{array}{ll}
\Gamma \vdash f : \tau & \vdash f \sim_r f' : \tau \\
\Gamma \vdash s : \tau & \Gamma \vdash s \approx_r s' : \tau
\end{array}$$

We add the following inference rules to the language.

Typing:

$$\frac{\Gamma \vdash e : \tau}{\infty\Gamma \vdash \mathbf{return}\, e : \bigcirc\tau}\bigcirc I \qquad \frac{\Delta \vdash e : \bigcirc\tau \qquad \Gamma, x :_\infty \tau \vdash e' : \bigcirc\tau'}{\Delta + \Gamma \vdash \mathbf{let}\, \bigcirc x = e\, \mathbf{in}\, e' : \bigcirc\tau'}\bigcirc E$$

$$\frac{\Delta \vdash e : \mathcal{P} \qquad \Gamma \vdash e_i : \bigcirc\tau \quad (\forall i)}{\Delta + \Gamma \vdash \{e, (e_1, \ldots, e_n)\} : \bigcirc\tau}\{\} \qquad \frac{}{\Gamma \vdash (r_1, \ldots, r_n) : \mathcal{P}}$$

$$\frac{\Gamma \vdash e : \bigcirc\tau}{\Gamma \vdash \mathbf{do}\, e : \tau} \qquad \frac{\Delta \vdash p : \mathcal{P} \qquad \Gamma \vdash s_i : \tau \quad (\forall i)}{\Delta + \Gamma \vdash \{p, (s_1, \ldots, s_n)\} : \tau}$$

Metric:

$$\frac{\vdash v \sim_r v' : \tau}{\vdash \mathbf{return}\, v \sim_{\infty r} \mathbf{return}\, v' : \bigcirc\tau}$$

$$\frac{\vdash p \sim_r p' : \mathcal{P} \qquad \vdash e_i \approx_s e_i' : \bigcirc\tau \quad (\forall i)}{\vdash \{p, (e_1, \ldots, e_n)\} \sim_{r+s} \{p', (e_1', \ldots, e_n')\} : \bigcirc\tau}$$

$$\frac{|\ln(r_i/r_i')| \leq s \qquad (\forall i)}{\vdash (r_1, \ldots, r_n) \sim_s (r_1', \ldots, r_n') : \mathcal{P}}$$

$$\frac{\vdash \mathbf{return}\, v \sim_r \mathbf{return}\, v' : \bigcirc\tau}{\vdash \mathbf{do}\, \mathbf{return}\, v \sim_r \mathbf{do}\, \mathbf{return}\, v' : \tau}$$

$$\frac{\vdash p \sim_r p' : \mathcal{P} \qquad \vdash f_i \sim_s f_i' : \tau \quad (\forall i)}{\vdash \{p, (f_1, \ldots, f_n)\} \sim_{r+s} \{p', (f_1', \ldots, f_n')\} : \tau}$$

$$\frac{\vdash \sigma \sim_\gamma \sigma' : \Gamma \qquad \Gamma, \Gamma_0 \vdash s_0 : \tau}{\Gamma_0 \vdash \sigma s_0 \approx_{\gamma\Gamma} \sigma' s_0 : \tau}$$

Evaluation:

$$\frac{e \mapsto e'}{\mathbf{return}\, e \mapsto \mathbf{return}\, e'}$$

$$\frac{e \mapsto e'}{\{e, (e_1, \ldots, e_n)\} \mapsto \{e', (e_1, \ldots, e_n)\}}$$

$$\frac{e \mapsto e'}{\mathbf{let} \bigcirc x = e\, \mathbf{in}\, e_0 \mapsto \mathbf{let} \bigcirc x = e'\, \mathbf{in}\, e_0}$$

$$\frac{}{\mathbf{let} \bigcirc x = \mathbf{return}\, v\, \mathbf{in}\, e' \mapsto [v/x]e'}$$

$$\frac{}{\mathbf{let} \bigcirc x = \{p, (e_i)_{i\in 1\ldots n}\}\, \mathbf{in}\, e' \mapsto \{p, (\mathbf{let} \bigcirc x = e_i\, \mathbf{in}\, e')_{i\in 1\ldots n}\}}$$

$$\frac{e \mapsto e'}{\mathbf{do}\, e \mapsto \mathbf{do}\, e'} \qquad \frac{}{\mathbf{do}\ \{p, (e_i)_{i\in 1\ldots n}\} \mapsto \{p, (\mathbf{do}\, e_i)_{i\in 1\ldots n}\}}$$

$$\frac{\emptyset \neq I \subseteq 1\ldots n \qquad s_i \mapsto s_i' \qquad s_j = s_j' = f_j \qquad (\forall i \in I, j \notin I)}{\{p, (s_i)_{i\in 1\ldots n}\} \mapsto \{p, (s_i')_{i\in 1\ldots n}\}}$$

We revisit the results shown above. The results split in a mostly predictable way into the existing statement about expressions, and a corresponding statement about evaluation states.

5.2.1 Theorem [Progress]:

1. Suppose $\Gamma \vdash e : \tau$. Either $e \mapsto e'$, or e is a value.
2. Suppose $\Gamma \vdash s : \tau$. Either $s \mapsto s'$, or s is a final state.

Proof: By induction on the typing derivation. □

We show again that $\approx$ and $\sim$ coincide on values.

5.2.2 Lemma: Suppose $\vdash \sigma \sim_\gamma \sigma' : \Gamma^\circ$.

1. If $\Gamma \vdash e : \tau$ and σe is a value, then $\sigma' e$ is a value, and $\vdash \sigma e \sim_{\gamma\Gamma} \sigma' e : \tau$
2. If $\Gamma \vdash s : \tau$ and σs is final, then $\sigma' s$ is final, and $\vdash \sigma s \sim_{\gamma\Gamma} \sigma' s : \tau$

Proof: By induction on the typing derivation of e or s. Some representative cases:

Case:

$$\frac{}{\Gamma, x :_r \bigcirc\tau \vdash x : \bigcirc\tau}$$
$$\Gamma, x :_r \bigcirc\tau \vdash \mathbf{do}\, x : \tau$$

The state s is $\mathbf{do}\, x$. In this case, we must have $\sigma(x) = \mathbf{return}\, v$ for σs to be final. But $\sigma \sim \sigma'$ requires that $\sigma'(x) = \mathbf{return}\, v'$ for some v' such that $\mathbf{return}\, v \sim_r \mathbf{return}\, v'$. But then $\mathbf{do}\,\mathbf{return}\, v \sim_r \mathbf{do}\,\mathbf{return}\, v'$ and we are done.

Case:

$$\frac{\dfrac{\Gamma \vdash e : \tau}{\infty\Gamma \vdash \mathbf{return}\, e : \bigcirc\tau}\bigcirc I}{\infty\Gamma \vdash \mathbf{do}\,\mathbf{return}\, e : \tau}$$

The state s is $\mathbf{do}\,\mathbf{return}\, e$. In this case, we must have that σe is a value for σs to be final. By induction hypothesis, $\sigma' e$ is also a value, with $\vdash \sigma e \sim_{\gamma\Gamma} \sigma' e : \tau$. Hence $\sigma(\mathbf{do}\,\mathbf{return}\, e) \sim_{\infty\gamma\Gamma} \sigma'(\mathbf{do}\,\mathbf{return}\, e)$, as required, since $\infty(\gamma\Gamma) = \gamma(\infty\Gamma)$.

Case:

$$\frac{\Delta \vdash e : \mathcal{P} \qquad \Gamma \vdash s_i : \tau \quad (\forall i)}{\Delta + \Gamma \vdash \{e, (s_1, \ldots, s_n)\} : \tau}\,\{\}$$

By assumption σe is a value, and σs_i is final for all i. By repeated use of the induction hypothesis $\sigma' e$ is a value, and $\sigma' s_i$ is final for all i, and we have $\vdash \sigma e \sim_{\gamma\Delta} \sigma' e : \mathcal{P}$, and $\vdash \sigma s_i \sim_{\gamma\Gamma} \sigma' s_i : \tau$ for all i. Therefore we have that $\vdash \sigma\{e, (s_1, \ldots, s_n)\} \sim_{\gamma(\Delta+\Gamma)} \sigma'\{e, (s_1, \ldots, s_n)\} : \tau$, as required.

□

5.2.3 Corollary:

1. If $\vdash v \approx_r v' : \tau$, then $\vdash v \sim_r v' : \tau$.
2. If $\vdash f \approx_r f' : \tau$, then $\vdash f \sim_r f' : \tau$.

Proof: By inversion on the rule defining $\approx$, we know f, f', r must arise as $f = \sigma s_0$ and $f' = \sigma' s_0$ and $r = \gamma\Gamma$ so that

$$\frac{\vdash \sigma \sim_\gamma \sigma' : \Gamma^\circ \qquad \Gamma \vdash s_0 : \tau}{\vdash \sigma s_0 \approx_{\gamma\Gamma} \sigma' s_0 : \tau}$$

By the preceding lemma, we have $\vdash \sigma s_0 \sim_{\gamma\Gamma} \sigma' s_0 : \tau$, as required. □

5.2.4 Lemma [Substitution into $\approx$]: The following rules are admissible:

$$\frac{\Delta \vdash e \approx_r e' : \tau \qquad \Delta_0, x :_s \tau \vdash e_0 \approx_{r_0} e_0' : \tau_0}{\Delta_0 + s\Delta \vdash [e/x]e_0 \approx_{r_0 + rs} [e'/x]e_0' : \tau_0}$$

$$\frac{\Delta \vdash s \approx_r s' : \tau \qquad \Delta_0, x :_s \tau \vdash s_0 \approx_{r_0} s_0' : \tau_0}{\Delta_0 + s\Delta \vdash [s/x]s_0 \approx_{r_0 + rs} [s'/x]s_0' : \tau_0}$$

Proof: Essentially the same proof as before. □

5.2.5 Lemma: Suppose $\vdash \sigma \sim_\gamma \sigma' : \Gamma$.

1. If $\Gamma \vdash e : \tau$ and $\sigma e \mapsto e_f$, then $\exists e_f'.\ \sigma' e \mapsto e_f'$ and $\vdash e_f \approx_{\gamma\Gamma} e_f' : \tau$.
2. If $\Gamma \vdash s : \tau$ and $\sigma s \mapsto s_f$, then $\exists s_f'.\ \sigma' s \mapsto s_f'$ and $\vdash s_f \approx_{\gamma\Gamma} s_f' : \tau$.

Proof: By induction on the typing derivation, with further case analysis on the step taken.

Case:

$$\frac{\Delta \vdash e_L : \bigcirc\tau_1 \qquad \Gamma, x :_\infty \tau_1 \vdash e_R : \bigcirc\tau_2}{\Delta + \Gamma \vdash \mathbf{let}\,\bigcirc x = e_L\,\mathbf{in}\,e_R : \bigcirc\tau_2}\,\bigcirc E$$

Subcase:

$$\frac{\sigma e_L \mapsto e_f}{\mathbf{let}\,\bigcirc x = \sigma e_L\,\mathbf{in}\,\sigma e_R \mapsto \mathbf{let}\,\bigcirc x = e_f\,\mathbf{in}\,\sigma e_R}$$

By induction hypothesis, $\sigma' e \mapsto e'_f$ with $\vdash e_f \approx_{\gamma\Delta} e'_f : \bigcirc\tau_1$. We therefore find that

$$\frac{\sigma' e_L \mapsto e'_f}{\mathbf{let}\,\bigcirc x = \sigma e_L\,\mathbf{in}\,\sigma' e_R \mapsto \mathbf{let}\,\bigcirc x = e'_f\,\mathbf{in}\,\sigma' e_R}$$

By the definition of $\approx$ on the well-typedness of e_R, we know

$$y :_\infty \tau_1 \vdash (\mathbf{let}\,\bigcirc x = y\,\mathbf{in}\,\sigma e_R) \approx_{\gamma\Gamma} (\mathbf{let}\,\bigcirc x = y\,\mathbf{in}\,\sigma' e_R)$$

and so substituting into this the fact that $\vdash e_f \approx_{\gamma\Delta} e'_f : \bigcirc\tau_1$ yields

$$\vdash (\mathbf{let}\,\bigcirc x = e_f\,\mathbf{in}\,\sigma e_R) \approx_{\gamma(\Delta+\Gamma)} (\mathbf{let}\,\bigcirc x = e'_f\,\mathbf{in}\,\sigma' e_R)$$

as required.

Subcase:

$$\frac{}{\mathbf{let}\,\bigcirc x = \mathbf{return}\,v\,\mathbf{in}\,\sigma e_R \mapsto [v/x]\sigma e_R}$$

Here we know that σe_L is a value $\mathbf{return}\,v$. By Lemma 5.2.2, $\sigma' e_L$ must be of the form $\mathbf{return}\,v'$, with $\vdash \mathbf{return}\,v \sim_{\gamma\Delta} \mathbf{return}\,v' : \bigcirc\tau_1$. By inversion, $\gamma\Delta = \infty r$, and $\vdash v \sim_r v' : \tau_1$. We can extend σ and σ' to the substitutions $[v/x]\sigma$ and $[v'/x]\sigma'$ and note that

$$\vdash \sigma[v/x] \sim_{(\gamma, x:r)} \sigma'[v'/x] : \Gamma^\circ, x : \tau_1$$

We can see directly that

$$\frac{}{\mathbf{let}\,\bigcirc x = \mathbf{return}\,v'\,\mathbf{in}\,\sigma' e_R \mapsto [v'/x]\sigma' e_R}$$

and furthermore that

$$\vdash [v/x]\sigma e_R \approx_{\infty r + \gamma\Gamma} [v'/x]\sigma' e_R$$

which is what we need, since $\infty r = \gamma\Delta$.

Subcase:

$$\frac{}{\mathbf{let}\,\bigcirc x = \{p, (e_i)_{i\in 1\ldots n}\}\,\mathbf{in}\,\sigma e_R \mapsto \{p, (\mathbf{let}\,\bigcirc x = e_i\,\mathbf{in}\,\sigma e_R)_{i\in 1\ldots n}\}}$$

Here we know that σe_L is a value $\{p, (e_i)_{i\in 1\ldots n}\}$. By Lemma 5.2.2, $\sigma' e_L$ must be of the form $\{p', (e'_i)_{i\in 1\ldots n}\}$, with

$$\frac{\vdash p \sim_r p' : \mathcal{P} \qquad \vdash e_i \approx_s e'_i : \bigcirc\tau_1 \quad (\forall i)}{\vdash \{p, (e_1, \ldots, e_n)\} \sim_{r+s} \{p', (e'_1, \ldots, e'_n)\} : \bigcirc\tau_1}$$

and $r + s = \gamma\Delta$.

We can easily check that the other expression takes a step, namely

$$\frac{}{\mathbf{let}\,\bigcirc x = \{p', (e'_i)_{i\in 1\ldots n}\}\,\mathbf{in}\,\sigma' e_R \mapsto \{p', (\mathbf{let}\,\bigcirc x = e'_i\,\mathbf{in}\,\sigma' e_R)_{i\in 1\ldots n}\}}$$

and so we must show that

$$\vdash \{p, (\mathbf{let}\bigcirc x = e_i \,\mathbf{in}\, \sigma e_R)_{i\in 1\ldots n}\} \approx \{p', (\mathbf{let}\bigcirc x = e'_i \,\mathbf{in}\, \sigma' e_R)_{i\in 1\ldots n}\} : \bigcirc\tau_2$$

This can be done by fist observing that

$$y : \mathcal{P}, z_1, \ldots, z_n : \bigcirc\tau_1 \vdash \{y, (\mathbf{let}\bigcirc x = z_i \,\mathbf{in}\, \sigma e_R)_{i\in 1\ldots n}\}$$
$$\approx \{y, (\mathbf{let}\bigcirc x = z_i \,\mathbf{in}\, \sigma' e_R)_{i\in 1\ldots n}\} : \bigcirc\tau_2$$

and then carrying out repeated substitutions.

□

The central metric preservation theorem is now

5.2.6 Theorem [Metric Preservation]:

1. If $\vdash e \approx_r e' : \tau$ and $e \mapsto e_f$, then $\exists e'_f.\ e' \mapsto e'_f$ and $\vdash e_f \approx_r e'_f : \tau$.
2. If $\vdash s \approx_r s' : \tau$ and $s \mapsto s_f$, then $\exists s'_f.\ s' \mapsto s'_f$ and $\vdash s_f \approx_r s'_f : \tau$.

Proof: By appeal to the previous lemma. □

The metric on probability distributions was carefully chosen so that the above theorem implies that the definition of differential privacy can be encoded directly in the type system. We first formally define the probability $Pr_f[v]$ that a final state f yields a value v. It is defined recursively on f:

$$Pr_{\mathbf{do\,return}\,v'}[v] = \begin{cases} 1 & \text{if } v = v'; \\ 0 & \text{otherwise.} \end{cases} \qquad Pr_{\{(p_1,\ldots,p_n),(f_1,\ldots,f_n)\}}[v] = \sum_{i=1}^{n} p_i\, Pr_{f_i}[v]$$

The metric on final states gives corresponds to the relation on probability distributions involved in the definition of differential privacy.

5.2.7 Lemma: Let $\vdash s_f : \tau$ and $\vdash s'_f : \tau$ be two closed final states such that $s_f \sim_r s'_f : \tau$. Then for every value $\vdash v : \tau$:

$$Pr_{s_f}[v] \leq e^r Pr_{s'_f}[v]$$

Proof: By induction on the structure of s_f with further case analysis on $s_f \sim_r s'_f : \tau$.

Case $s_f = \mathbf{do\,return}\, v'$.

By definition of $s_f \sim_r s'_f : \tau$ we must have $s'_f = \mathbf{do\,return}\, v''$ with $v' = v''$ and $r = 0$. So the conclusion follows easily.

Case $s_f = \{(p_1, \ldots, p_n), (f_1, \ldots, f_n)\}$.

By definition of $s_f \sim_r s'_f : \tau$ we must have $s'_f = \{(p'_1, \ldots, p'_n), (f'_1, \ldots, f'_n)\}$ with $r = s + t$ where $(p_1, \ldots, p_n) \sim_s (p'_1, \ldots, p'_n)$ and $\forall i : f_i \sim_t f'_i : \tau_i$. The former implies that $\forall j : |\ln(p_j/p'_j)| \leq s$, while the latter by induction hypothesis implies that for every i and for every value $\vdash v : \tau_i$: $Pr_{s_{f_i}}[v] \leq e^t Pr_{s_{f'_i}}[v]$. So, we have

$$Pr_{s_f}[v] = \sum_{i=1}^{n} p_i\, Pr_{f_i}[v] \leq \sum_{i=1}^{n} p_i\, e^t Pr_{s_{f'_i}}[v] \leq \sum_{i=1}^{n} e^s p'_i\, e^t Pr_{s_{f'_i}}[v] \leq e^{s+t} \sum_{i=1}^{n} p'_i\, Pr_{s_{f'_i}}[v] = e^{s+t}\, Pr_{s'_f}[v]$$

from which the conclusion follows.

□

Finally, we can show that the definition of differential privacy is encoded in the type system.

5.2.8 Corollary: The execution of any closed program e such that

$$\vdash e : !_\epsilon \tau \multimap \bigcirc \sigma$$

is an ϵ-differentially private function from τ to σ. That is, for all closed vlaues $v, v' : \tau$ such that $\vdash v \sim_r v' : \tau$, and all closed values $w : \sigma$, we have that if $\mathbf{do}(e\ v) \mapsto s_f$ and $\mathbf{do}(e\ v) \mapsto s'_f$:

$$Pr_{s_f}[w] \leq e^{r\epsilon} Pr_{s'_f}[w]$$

Proof: By using the fact that $\vdash [v/x] \sim_{[r/x]} [v'/x] : (x : \tau)$ we have that $\vdash [v/x]\,\mathbf{do}(e\ x) \approx_{r\epsilon} [v'/x]\,\mathbf{do}(e\ x) : \sigma$. By Metric Preservation Theorem 5.2.6 we obtain $s_f \approx_{r\epsilon} s'_f : \tau$, and so by Corollary 5.2.3 and Lemma 5.2.7 we can conclude. □

The above corollary shows that in order to ensure that the execution of a program e corresponds to an ϵ-differentially private random function from values in τ to values in σ, it is sufficient to check that the program e is typable as:

$$\vdash e : !_\epsilon \tau \multimap \bigcirc \sigma$$

So, well typed programs cannot disclose information about individuals.

5.3 Finite Approximations and the Laplace Primitive

In order to be able to program differentially private algorithms as described in Proposition 5.1.3 we need now a new primitive adding noise distributed according to Laplace distribution. In particular it is sufficient to add a primitive computing the function *Lap* adding noise distributed according to $\mathcal{L}_1$. Since however such a primitive is not in general computable, we will show here that is sound to add as primitive any of its finite approximations.

Let $N = (n_i)_{i \in 1 \ldots n}$ be a finite sequence of disjoint intervals whose union is $\mathbb{R}$. Let $V = (v_i)_{i \in 1 \ldots n}$ be a finite sequence of points in $\mathbb{R}$, such that $v_i \in n_i$ for every i. For example, N could be the intervals $(-\infty, 0), [0, 1), [1, \infty)$ and V could be $-1, 1/2, 1$. We can, however, choose N to be as refined as we want — for instance, having one interval surround every number representable in IEEE floating point — and the result below will still hold.

We then define a primitive $\mathbf{Lap}_{NV} : \mathbb{R} \multimap \bigcirc\mathbb{R}$ with the underlying value function[3]

$$Lap_{NV}(v) = \{(p_1, \ldots, p_n), (v_1, \ldots, v_n)\}$$

where

$$p_i = \int_{N_i} e^{v-x} dx$$

To satisfy the metric preservation theorem, we need to establish that we can get from $\vdash v \sim_k v' : \mathbb{R}$ to $\vdash Lap_{NV}(v) \sim_k Lap_{NV}(v') : \bigcirc\mathbb{R}$. In other words, we must prove the following lemma:

5.3.1 Lemma: If $|v - v'| \leq k$, then

$$\left| \ln \left(\int_{N_i} e^{v-x} dx \Big/ \int_{N_i} e^{v'-x} dx \right) \right| \leq k$$

[3]Technically, in order to respect the typing, the application of the $\mathbf{Lap}_{NV}$ primitive to a value v must yield an expression $\{(p_1, \ldots, p_n), (\mathbf{return}\, v_1, \ldots, \mathbf{return}\, v_n)\}$.

Proof: To say that $|a| \leq k$ (for positive k) is the same as saying that both $a \leq k$ and $-a \leq k$. Our assumption is therefore tantamount to $v - v' \leq k$ and $v' - v \leq k$. Very easily from these we can get both of

$$e^{v-x} \leq e^k e^{v'-x}$$

$$e^{v'-x} \leq e^k e^{v-x}$$

for every x. Integrating both sides of both inequalities over the interval N_i yields

$$\int_{N_i} e^{v-x} dx \leq e^k \int_{N_i} e^{v'-x} dx$$

$$\int_{N_i} e^{v'-x} dx \leq e^k \int_{N_i} e^{v-x} dx$$

which, by rearranging, yields both of

$$\ln \left(\int_{N_i} e^{v-x} dx \Big/ \int_{N_i} e^{v'-x} dx \right) \leq k$$

$$\ln \left(\int_{N_i} e^{v'-x} dx \Big/ \int_{N_i} e^{v-x} dx \right) \leq k$$

from which follows the required conclusion. □

We conclude that it is sound to include any finite approximation to the *Lap* primitive, and leave it up to potential language implementers to determine the most appropriate level of accuracy.

6 Differential Privacy Examples

Easy examples of ϵ-differentially private computations come from applying the Laplace mechanism at the end of a deterministic computation. We want to have a function

$$add\_noise : !_\epsilon \mathbb{R} \multimap \bigcirc \mathbb{R}$$

which adds Laplace noise $\mathcal{L}_{1/\epsilon}$ to its input. According to Proposition 5.1.3, this is exactly the right amount of noise to add to a 1-sensitive function to make it ϵ-differentially private. In order to program (an approximation of) the *add_noise* function, we can fix N and V and use the $\mathbf{Lap}_{NV} : \mathbb{R} \multimap \bigcirc \mathbb{R}$ primitive introduced in the previous section. So, we have

$$add\_noise = \lambda x.\, \mathbf{let}\, \bigcirc y = \mathbf{Lap}_{NV}(\mathbf{scale}_\epsilon x)\, \mathbf{in}\ \mathbf{return}\ (\mathbf{times}(y, !\epsilon))$$

For a concrete example, suppose that we have a function $age : \mathsf{row} \to \mathsf{int}$. We can then straightforwardly implement the over-40 count query from the introduction.

$$\begin{array}{l} over\_40 : \mathsf{row} \to \mathsf{bool}. \\ over\_40\ r = age\ r > 40. \end{array}$$

$$\begin{array}{l} count\_query : !_\epsilon(\mathsf{row\ set}) \multimap \bigcirc \mathbb{R} \\ count\_query\ b = \mathbf{let}\, !x = b\, \mathbf{in}\, add\_noise\ !(setfilter\ over\_40\ x) \end{array}$$

Notice that we are able to use convenient higher-order functional programming idioms without any difficulty. The function *over_40* is also an example of how 'ordinary programming' can safely be mixed in with distance-sensitive programs. Since the type of *over_40* uses $\to$ rather than $\multimap$, it makes no promise about sensitivity, and it is able to use 'discontinuous' operations like numeric comparison $>$.

Other deterministic queries can be turned into differentially private functions in a similar way. For example, consider the histogram function $hist : \mathbb{R}\,\mathsf{set} \multimap \mathbb{R}\,\mathsf{list}$ from Section 4.5. We can first of all write the following program.

$$\begin{array}{l} hist\_query' : !_\epsilon(\mathsf{row\,set}) \multimap (\bigcirc\mathbb{R})\,\mathsf{list} \\ hist\_query'\ b = \mathbf{let}!b' = b\,\mathbf{in}\ map\ add\_noise\ (ldistribute\ !(hist\ (setmap\ age\ b')) \end{array}$$

This takes a database, finds the age of every individual, and computes a histogram of the ages. Then we prescribe that each item in the output list — every bucket in the histogram — should be independently noised. This yields a list of random computations, while what we ultimately want is a random computation returning a list. But we can use monadic sequencing to get exactly this:

$$\begin{array}{l} seq : (\bigcirc\mathbb{R})\,\mathsf{list} \multimap \bigcirc(\mathbb{R}\,\mathsf{list}) \\ seq\ [] = \mathbf{return}\ [] \\ seq\ (h :: tl) = \mathbf{let}\bigcirc h' = h\,\mathbf{in} \\ \qquad\qquad\qquad \mathbf{let}\bigcirc tl' = seq\ tl\,\mathbf{in} \\ \qquad\qquad\qquad\quad \mathbf{return}(h' :: tl') \end{array}$$

$$\begin{array}{l} hist\_query : !_\epsilon(\mathsf{row\,set}) \multimap \bigcirc(\mathbb{R}\,\mathsf{list}) \\ hist\_query\ b = seq\ (hist\_query'\ b) \end{array}$$

In the differential privacy literature, there are explicit definitions of both the meaning of sensitivity and the process of safely adding enough noise to lists of real numbers [DMNS06]. By contrast, we have shown how to *derive* these concepts from the primitive metric type $\mathbb{R}$ and the type operators μ, 1, $+$, $\otimes$, and $\bigcirc$.

We can also derive more complex combinators on differentially private computations, merely by programming with the monad. We consider first a simple version[4] of McSherry's principle of sequential composition [McS09].

6.1 Lemma [Sequential Composition]: Let f_1 and f_2 be two ϵ-differentially private queries, where f_2 is allowed to depend on the output of f_1. Then the result of performing both queries is 2ϵ-differentially private.

In short, the privacy losses of consecutive queries are added together. This principle can be embodied as the following higher-order function:

$$\begin{array}{l} sc : (!_\epsilon\tau_1 \multimap \bigcirc\tau_2) \to (!_\epsilon\tau_1 \multimap \tau_2 \to \bigcirc\tau_3) \to (!_{2\epsilon}\tau_1 \multimap \bigcirc\tau_3) \\ sc\ f_1\ f_2\ t_1 = \mathbf{let}\ \ !t_1' = t_1\,\mathbf{in}\,\mathbf{let}\bigcirc t_2 = f_1\ !t_1'\,\mathbf{in}\,f_2\ !t_1'\ t_2 \end{array}$$

Its arguments are the functions f_1 and f_2, which are both ϵ-differentially private in a data source of type τ_1 (and f_2 additionally has unrestricted access to the τ_2 result of f_1), and returns a 2ϵ-differentially private computation.

McSherry also identifies a principle of parallel composition:

6.2 Lemma [Parallel Composition]: Let f_1 and f_2 be two ϵ-differentially private queries, which depend on disjoint data. Then the result of performing both queries is ϵ-differentially private.

This can be coded up by interpreting "disjoint" with $\otimes$.

$$\begin{array}{l} pc : (!_\epsilon\tau_1 \multimap \bigcirc\tau_2) \to (!_\epsilon\sigma_1 \multimap \bigcirc\sigma_2) \to !_\epsilon(\tau_1 \otimes \sigma_1) \multimap \bigcirc(\tau_2 \otimes \sigma_2) \\ pc\ f\ g\ b = \mathbf{let}\,!x = b\,\mathbf{in}\ \ \mathbf{let}(t,s) = x\,\mathbf{let}\bigcirc t' = f\ !t\,\mathbf{in}\,\mathbf{let}\bigcirc s' = g\ !s\,\mathbf{in}\,\mathbf{return}(t',s') \end{array}$$

In McSherry's work, what is literally meant by "disjoint" is disjoint subsets of a database construed as a set of records. This is also possible to treat in our setting, since we have already seen that *setsplit* returns a $\otimes$-pair of two sets.

[4] McSherry actually states a stronger principle, where there are k different queries, all of different privacy levels. This can also be implemented in our language.

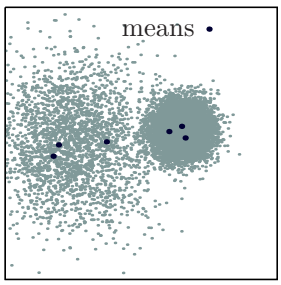

Figure 5: k-Means Output

For a final, slightly more complex example, let us consider the privacy-preserving implementation of k-means by Blum et al. [BDMN05]. Recall that k-means is a simple clustering algorithm, which works as follows. We assume we have a large set of data points in some space (say $\mathbb{R}^n$), and we want to find k 'centers' around which they cluster. We initialize k provisional 'centers' to random points in the space, and iteratively try to improve these guesses. One iteration consists of grouping each data point with the center it is closest to, then taking the next round's set of k centers to be the mean of each group.

We sketch how this program can be implemented, taking data points to be of the type $\mathsf{pt} = \mathbb{R} \otimes \mathbb{R}$. The following helper functions are used:

$$
\begin{aligned}
&\mathit{assign} : \mathsf{pt\,list} \to \mathsf{pt\,set} \multimap (\mathsf{pt} \otimes \mathsf{int})\,\mathsf{set} \\
&\mathit{partition} : (\mathsf{pt} \otimes \mathsf{int})\,\mathsf{set} \multimap \mathsf{pt\,set\,list} \\
&\mathit{totx}, \mathit{toty} : !_\epsilon(\mathsf{pt\,set}) \multimap !_\epsilon\mathbb{R} \\
&\mathit{size}' : !_\epsilon(\mathsf{pt\,set}) \multimap !_\epsilon\mathbb{R} \\
&\mathit{zip} : \tau\,\mathsf{list} \to \sigma\,\mathsf{list} \to (\tau \otimes \sigma)\,\mathsf{list}
\end{aligned}
$$

These can be written with the primitives we have described; *assign* takes a list of centers and the dataset, and returns a version of the dataset where each point is labelled by the index of the center it's closest to. Then *partition* divides this up into a list of sets, using *setsplit*. The functions *totx* and *toty* compute the sum of the first and second coordinates, respectively, of each point in a set. This can be accomplished with *sum*. Finally, *zip* is the usual zipping operation that combines two lists into a list of pairs. With these, we can write a function that performs one iteration of private k-means:

$$
\begin{aligned}
&\mathit{iterate} : !_{3\epsilon}\mathsf{pt\,set} \multimap \mathbb{R}\,\mathsf{list} \to \bigcirc(\mathbb{R}\,\mathsf{list}) \\
&\mathit{iterate}\ b\ ms = \mathbf{let}\ !b' = b\,\mathbf{in} \\
&\quad \mathbf{let} \\
&\qquad b'' = \mathit{ldistribute}\ !(\mathit{partition}\ (\mathit{assign}\ ms\ b')) \\
&\qquad \mathit{tx} = \mathit{map}\ (\mathit{add\_noise} \circ \mathit{totx})\ b'' \\
&\qquad \mathit{ty} = \mathit{map}\ (\mathit{add\_noise} \circ \mathit{toty})\ b'' \\
&\qquad t = \mathit{map}\ (\mathit{add\_noise} \circ \mathit{size}')\ b'' \\
&\qquad \mathit{stats} = \mathit{zip}\ (\mathit{zip}\ (\mathit{tx}, \mathit{ty}), t) \\
&\quad \mathbf{in} \\
&\qquad \mathit{seq}\ (\mathit{map}\ \mathit{avg}\ \mathit{stats})
\end{aligned}
$$

It works by asking for noisy sums of the x-coordinate total, y-coordinate total, and total population of each cluster. These data are then combined via the function *avg*:

$$
\begin{aligned}
&\mathit{avg} : ((\bigcirc\mathbb{R} \otimes \bigcirc\mathbb{R}) \otimes \bigcirc\mathbb{R}) \multimap \bigcirc(\mathbb{R} \otimes \mathbb{R}) \\
&\mathit{avg}\ ((x, y), t) = \mathbf{let} \bigcirc x' = x\,\mathbf{in}\,\mathbf{let} \bigcirc y' = y\,\mathbf{in} \\
&\qquad\qquad\qquad\qquad \mathbf{let} \bigcirc t' = t\,\mathbf{in}\,\mathbf{return}\ (x'/t', y'/t')
\end{aligned}
$$

We can read off from the type that one iteration of k-means is 3ϵ-differentially private. This type arises from the 3-way replication of the variable b''. We can use monadic sequencing to do more than one iteration:

$$\begin{aligned}&\mathit{two\_iters} : !_{6\epsilon}\mathsf{pt}\,\mathsf{set} \multimap \mathbb{R}\,\mathsf{list} \to \bigcirc(\mathbb{R}\,\mathsf{list})\\&\mathit{two\_iters}\ b\ ms = \mathbf{let}\ \ !b' = b\,\mathbf{in}\,\mathit{iterate}\ !b'\ (\mathit{iterate}\ !b'\ ms)\end{aligned}$$

This function is 6ϵ-differentially private. Figure 5 shows the result of three independent runs of this code, with $k = 2$, $6\epsilon = 0.05$, and 12,500 points of synthetic data. We see that it usually manages to come reasonably close to the true center of the two clusters. We have also developed appropriate additional primitives and programming techniques to make it possible (as one would certainly hope!) to choose the number of iterations not statically but at runtime, but space reasons prevent us from discussing them here.

7 Related Work

The seminal paper on differential privacy is [DMNS06]; it introduces the fundamental definition and the Laplace mechanism. More general mechanisms for directly noising types other than $\mathbb{R}$ also exist, such as the exponential mechanism [MT07], and techniques have been developed to reduce the amount of noise required for repeated queries, such as the median mechanism [RR10]. Dwork [Dwo08] gives a useful survey of recent results.

Girard's linear logic [Gir87] was a turning point in a long and fruitful history of investigation of *substructural logics*, which lack structural properties such as unrestricted weakening and contraction. A key feature of linear logic compared to earlier substructural logics [Lam58] is its ! operator, which bridges linear and ordinary reasoning. Our type system takes its structure from the *affine* variant of linear logic (also related to Ketonen's Direct Logic [Ket84]), where weakening is permitted. The idea of counting, as we do, multiple uses of the same resource was explored by Wright [WBF93], but only integral numbers of uses were considered.

The study of database privacy and statistical databases more generally has a long history. Recent work includes Dalvi, Ré, and Suciu's study of probabilistic database management systems [DRS09], and Machanavajjhala et al.'s comparison of different notions of privacy with respect to real-world census data [MKA+08].

Quantitative Information Flow [Low02, ME08] is, like our work, concerned with how much one piece of a program can affect another, but measures this in terms of how many bits of entropy leak during one execution. Provenance analysis [BKWC01] in databases tracks the input data actually used to compute a query's output, and is also capable of detecting that the same piece of data was used multiple times to produce a given answer [GKT07]. Chaudhuri et al. [CGL10] also study automatic program analyses that establish continuity (in the traditional topological sense) of numerical programs. Our approach differs in two important ways. First, we consider the stronger property of c-sensitivity, which is essential for differential privacy applications. Second, we achieve our results with a logically motivated type system, rather than a program analysis.

8 Conclusion

We have presented a typed functional programming language that guarantees differential privacy. It is expressive enough to encode examples both from the differential privacy community and from functional programming practice. Its type system shows how differential privacy arises conceptually from the combination of sensitivity analysis and monadic encapsulation of random computations.

There remains a rich frontier of differentially private mechanisms and algorithms that are known, but which are described and proven correct individually. We expect that the exponential mechanism should be easy to incorporate into our language, as a higher-order primitive which directly converts McSherry and Talwar's notion of *quality functions* [MT07] into probability distributions. The median mechanism, whose analysis is considerably more complicated, is likely to be more of a challenge. The private combinatorial

optimization algorithms developed by Gupta et al. [GLM+09] use different definitions of differential privacy which have an additive error term; we conjecture this could be captured by varying the notion of sensitivity to include additive slack. We believe that streaming private counter of Chan et al. [CSS10] admits an easy implementation by coding up stream types in the usual way. We hope to show in future work how these, and other algorithms can be programmed in a uniform, privacy-safe language.

9 *

References

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in c log n parallel steps. *Combinatorica*, 3(1):1–19, March 1983.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, University of Edinburgh, 1996.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, New York, NY, USA, 1968. ACM.

[BDMN05] Avrim Blum, Cynthia Dwork, Frank McSherry, and Kobbi Nissim. Practical privacy: the sulq framework. In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 128–138, New York, NY, USA, 2005. ACM.

[BKWC01] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In Jan Bussche and Victor Vianu, editors, *Database Theory ICDT 2001*, volume 1973 of *Lecture Notes in Computer Science*, chapter 20, pages 316–330. Springer Berlin Heidelberg, Berlin, Heidelberg, October 2001.

[BLR08] Avrim Blum, Katrina Ligett, and Aaron Roth. A learning theory approach to non-interactive database privacy. In *STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 609–618, New York, NY, USA, 2008. ACM.

[CGL10] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. *SIGPLAN Not.*, 45(1):57–70, 2010.

[CSS10] T-H. Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. Cryptology ePrint Archive, Report 2010/076, 2010. Available at http://eprint.iacr.org/.

[DMNS06] Cynthia Dwork, Frank Mcsherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, 2006.

[DRS09] Nilesh Dalvi, Christopher Ré, and Dan Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

[Dwo06] Cynthia Dwork. Differential privacy. In *Proceedings of ICALP (Part*, volume 2, pages 1–12, 2006.

[Dwo08] C. Dwork. Differential privacy: A survey of results. *5th International Conference on Theory and Applications of Models of Computation*, pages 1–19, 2008.

[Dwo09] Cynthia Dwork. The differential privacy frontier (extended abstract). In *Theory of Cryptography*, Lecture Notes in Computer Science, chapter 29, pages 496–502. 2009.

[Gir87] J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[GKT07] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, New York, NY, USA, 2007. ACM.

[GLM+09] Anupam Gupta, Katrina Ligett, Frank McSherry, Aaron Roth, and Kunal Talwar. Differentially private combinatorial optimization. Nov 2009.

[Ket84] J. Ketonen. A decidable fragment of predicate calculus. *Theoretical Computer Science*, 32(3):297–307, 1984.

[Lam58] Joachim Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65(3):154–170, 1958.

[Low02] Gavin Lowe. Quantifying information flow. In *In Proc. IEEE Computer Security Foundations Workshop*, pages 18–31, 2002.

[McS09] Frank D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 19–30, New York, NY, USA, 2009. ACM.

[ME08] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 193–205, New York, NY, USA, 2008. ACM.

[MKA+08] Ashwin Machanavajjhala, Daniel Kifer, John Abowd, Johannes Gehrke, and Lars Vilhuber. Privacy: Theory meets practice on the map. In *ICDE '08: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 277–286, Washington, DC, USA, 2008. IEEE Computer Society.

[MT07] Frank McSherry and Kunal Talwar. Mechanism design via differential privacy. In *FOCS '07: Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, pages 94–103, Washington, DC, USA, 2007. IEEE Computer Society.

[NRS07] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 75–84, New York, NY, USA, 2007. ACM.

[NS08] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 111–125, Washington, DC, USA, 2008. IEEE Computer Society.

[OP99] P.W. O'Hearn and D.J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[PPT03] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A monadic probabilistic language. In *In Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 38–49. ACM Press, 2003.

[RP02] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *In 29th ACM POPL*, pages 154–165. ACM Press, 2002.

[RR10] Aaron Roth and Tim Roughgarden. The median mechanism: Interactive and efficient privacy with multiple queries, 2010. To appear in STOC 2010.

[WBF93] D.A. Wright and C.A. Baker-Finch. Usage Analysis with Natural Reduction Types. In *Proceedings of the Third International Workshop on Static Analysis*, pages 254–266. Springer-Verlag London, UK, 1993.