Distance Makes the Types Grow Stronger

A Calculus for Differential Privacy

Jason Reed   Benjamin C. Pierce

University of Pennsylvania

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.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—specialized application languages

General Terms Languages

Keywords Differential Privacy, Type Systems

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 [26] 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* [6, 7, 12–15, 27] 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 *"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 [6, 18], each new one requires its own separate proof. McSherry's Privacy Integrated Queries (PINQ) [25] 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* [4, 16], 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 [29, 30]. 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 5) 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:

$$over\_40 : \mathsf{row} \to \mathsf{bool}$$
$$size : \mathsf{db} \multimap \mathbb{R}$$
$$filter : (\mathsf{row} \to \mathsf{bool}) \to \mathsf{db} \multimap \mathsf{db}$$
$$add\_noise : \mathbb{R} \multimap \bigcirc \mathbb{R}$$

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. This function also has a sensitivity

of 1, and this fact is intimately connected to privacy properties, as explained in Section 4.

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

$$\lambda d : \mathsf{db}.\ add\_noise\ (filter\ over\_40\ d)\ : \mathsf{db} \multimap \bigcirc \mathbb{R}.$$

As we explain in Section 4, its type indicates that it is a differentially private computation 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. Section 3 gives examples of programs that can be implemented in our language. Section 4 shows how to add the probability monad, and Section 5 develops further examples. In Section 6 we state the standard safety properties of the type system, give a formal statement of the metric preservation theorem, and sketch its proof. 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.

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

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

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

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.

PROPOSITION 2.1. *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.

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

PROPOSITION 2.2. *A function f is a c-sensitive function in $\tau_1 \rightarrow \tau_2$ if and only if it is a 1-sensitive function in $!_c\tau_1 \rightarrow \tau_2$.*

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 $\tau_1 \rightarrow \tau_2$ 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 $!_c\tau_1 \rightarrow \tau_2$. ∎

We can see therefore that f_6 is a 1-sensitive function $!_2\mathbb{R} \rightarrow \mathbb{R}$, and also in fact a 1-sensitive function $\mathbb{R} \rightarrow !_{1/2}\mathbb{R}$. 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 $\mathbb{R} \otimes \mathbb{R} \rightarrow \mathbb{R}$, and

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

$$f_{12}(x, y) = (x + y, 0) \qquad cswp(x, y) = \begin{cases} (x, y) & \text{if } x < y \\ (y, x) & \text{otherwise} \end{cases}$$

are 1-sensitive functions in $\mathbb{R} \otimes \mathbb{R} \rightarrow \mathbb{R} \otimes \mathbb{R}$. We will see the usefulness of $cswp$ in particular below in Section 3.6. However,

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

are not 1-sensitive functions in $\mathbb{R} \otimes \mathbb{R} \rightarrow \mathbb{R} \otimes \mathbb{R}$. 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 $\tau_1 \ \& \ \tau_2$ 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 $\mathbb{R} \otimes \mathbb{R} \rightarrow \mathbb{R} \ \& \ \mathbb{R}$. More generally, & lets us combine outputs of different c-sensitive functions even if they share dependency on common inputs.

PROPOSITION 2.3. *If $f : \tau \rightarrow \tau_1$ and $g : \tau \rightarrow \tau_2$ are c-sensitive, then $\lambda x.\langle f\ x, g\ x \rangle$ is a c-sensitive function in $\tau \rightarrow \tau_1 \ \& \ \tau_2$.*

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 $f : \tau_1 \rightarrow \tau_2$. 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 $(\mathbb{R} \otimes \mathbb{R} \multimap \mathbb{R}) \rightarrow (\mathbb{R} \multimap \mathbb{R} \multimap \mathbb{R})$ and $(\mathbb{R} \multimap \mathbb{R} \multimap \mathbb{R}) \rightarrow (\mathbb{R} \otimes \mathbb{R} \multimap \mathbb{R})$, 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 licenses us 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 $gtzero : \mathbb{R} \rightarrow$ bool, which returns true when 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 6.1.

With these pieces in place, we can introduce a 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 3.) 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 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. So let us turn next to designing 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 result of evaluation will also be suitably close—i.e., *Well-typed programs can't go too far.* To this end, we 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 the contexts. 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 rs if the value substituted for x varies by 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 the guarantee is that, if each x_i varies by s_i, then the result of evaluating e only varies by $\sum_i r_i s_i$. More carefully, we state the following metric preservation theorem for the type system, which is of central importance. The notation $[v/x]e$ indicates substitution of the value v for the variable x in expression e as usual.

THEOREM 2.4 (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*

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

We give a more precise version of this result in Section 6.

2.3 Types

The complete syntax and formation rules for types are given in Figure 1. Essentially all of these types have already been mentioned in Section 2.1. 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 $\rightarrow$, where $\tau_1 \multimap \tau_2$ contains just 1-sensitive functions, while $\tau_1 \rightarrow \tau_2$ is the ordinary unrestricted function space, containing the functions that can be programmed without any sensitivity requirements on the argument. As in linear logic, there is an encoding of $\tau_1 \rightarrow \tau_2$, in our case as $!_\infty\tau_1 \multimap \tau_2$, but it is convenient to have the built-in type constructor $\rightarrow$ to avoid having to frequently introduce and eliminate $!$-typed expressions.

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 $\otimes$, 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 for the type constructor $!_r$.

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

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

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

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

$$\frac{}{\Psi, \alpha : \text{type} \vdash \alpha : \text{type}} \qquad \frac{\Psi, \alpha : \text{type} \vdash \tau : \text{type}}{\Psi \vdash \mu\alpha.\tau : \text{type}} \qquad \frac{}{\Psi \vdash 1 : \text{type}}$$

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

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

Figure 1. Type Formation

$$\frac{r \geq 1}{\Gamma, x :_r \tau \vdash x : \tau}\, var \qquad \frac{\Delta \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Delta + \Gamma \vdash (e_1, e_2) : \tau_1 \otimes \tau_2}\, \otimes I$$

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \,\&\, \tau_2}\, \&I \qquad \frac{\Gamma \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma \vdash \pi_i e : \tau_i}\, \&E$$

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

$$\frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash \mathbf{inj}_i\, e : \tau_1 + \tau_2}\, +I \qquad \frac{\Gamma, x :_1 \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \multimap \tau'}\, \multimap I$$

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

$$\frac{\Delta \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Delta + \infty\Gamma \vdash e_1\, e_2 : \tau'}\, \to E \qquad \frac{\Gamma \vdash e : \tau}{s\Gamma \vdash !e : !_s\tau}\, !I$$

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

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \underset{\mu\alpha.\tau}{\mathbf{unfold}}\, e : [\mu\alpha.\tau/\alpha]\tau}\, \mu E$$

Figure 2. Typing Rules

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 $\mathbb{R}^{>0} \cup \{\infty\}$. Type-checking is decidable; see Section 6 and the appendix[1] for more details. In short, the only novelty is that lower bounds on the annotations in the context are inferred top-down from the leaves to the root of the derivation tree.

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 $c \geq 1$ (cf. Proposition 2.1). 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.)

$$\frac{}{\lambda x.e \hookrightarrow \lambda x.e} \qquad \frac{e_1 \hookrightarrow \lambda x.e \qquad e_2 \hookrightarrow v \qquad [v/x]e \hookrightarrow v'}{e_1\, e_2 \hookrightarrow v'} \qquad \frac{}{() \hookrightarrow ()}$$

$$\frac{e_1 \hookrightarrow v_1 \qquad e_2 \hookrightarrow v_2}{\langle e_1, e_2 \rangle \hookrightarrow \langle v_1, v_2 \rangle} \qquad \frac{e_1 \hookrightarrow v_1 \qquad e_2 \hookrightarrow v_2}{(e_1, e_2) \hookrightarrow (v_1, v_2)}$$

$$\frac{e \hookrightarrow (v_1, v_2) \qquad [v_1/x][v_2/y]e' \hookrightarrow v'}{\mathbf{let}(x, y) = e \,\mathbf{in}\, e' \hookrightarrow v'} \qquad \frac{e \hookrightarrow \langle v_1, v_2 \rangle}{\pi_i e \hookrightarrow v_i}$$

$$\frac{e \hookrightarrow v}{\mathbf{inj}_i\, e \hookrightarrow \mathbf{inj}_i\, v} \qquad \frac{e \hookrightarrow \mathbf{inj}_i\, v \qquad [v/x]e_i \hookrightarrow v'}{\mathbf{case}\, e \,\mathbf{of}\, x.e_1 \mid x.e_2 \hookrightarrow v'} \qquad \frac{e \hookrightarrow v}{\underset{\tau}{\mathbf{fold}}\, e \hookrightarrow \underset{\tau}{\mathbf{fold}}\, v}$$

$$\frac{e \hookrightarrow \underset{\tau}{\mathbf{fold}}\, v}{\underset{\tau}{\mathbf{unfold}}\, e \hookrightarrow v} \qquad \frac{e \hookrightarrow v}{!e \hookrightarrow !v} \qquad \frac{e \hookrightarrow !v \qquad [v/x]e' \hookrightarrow e'}{\mathbf{let}\, !x = e \,\mathbf{in}\, e' \hookrightarrow v'}$$

Figure 3. Evaluation Rules

In the rule $\otimes I$, consider the role of the contexts. Γ represents the variables that e_1 depends on, and captures quantitatively how sensitive it is to each one. Δ 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 the argument to an unrestricted implication cannot depend on linear resources.

2.6 Evaluation

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

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

The judgment $e \hookrightarrow v$ says that e evaluates to v. The complete set of evaluation rules is given in Figure 3.

3. Examples

We now present some more sophisticated examples of programs that can be written in this 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.

3.1 Fixpoint Combinator

Because we have general recursive types, we can simulate a fixpoint combinator in pretty much 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

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

has type $((\tau \multimap \sigma) \to (\tau \multimap \sigma)) \to (\tau \multimap \sigma)$. 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 rule

$$
\frac{f\ (Y\ f)\ v \hookrightarrow v_0}{Y\ f\ v \hookrightarrow v_0}
$$

is admissible whenever f is 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 $\mathbf{fix}$ 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.

3.2 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.

$$
\begin{aligned}
& map : (\tau \multimap \sigma) \to (\tau\ \mathsf{list} \multimap \sigma\ \mathsf{list}) \\
& map\ f\ [\ ] = [\ ] \\
& map\ f\ (h :: tl) = (f\ h) :: map\ f\ tl
\end{aligned}
$$

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:

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

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

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:

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

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

If we define the natural numbers as usual by

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

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

$$
\begin{aligned}
& length : \tau\ \mathsf{list} \multimap \mathsf{nat} \\
& length\ [\ ] = z \\
& length\ (h :: tl) = s\ (length\ tl)
\end{aligned}
$$

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

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

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.

3.3 &-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.

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

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

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

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 $\sigma \& \tau$ alist.

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

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

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:

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

$afoldl : (\tau \& \sigma \multimap \sigma) \rightarrow (\sigma \& \tau\, \mathsf{alist}) \multimap \sigma\, \mathsf{alist}$
$afoldl\, f\, p = \mathbf{case}\, analyze\, (unf\, p)\, \mathbf{of}$
  $x : (\sigma \& 1).\, \pi_1 x$
  $\mid x : (\sigma \& (\tau \& \tau\, \mathsf{alist})).\, afoldl\, f\, \langle f\langle \pi_1\pi_2 x, \pi_1 x\rangle, \pi_2\pi_2 x\rangle$

3.4 Sets

Another useful collection type is finite sets. We posit that τ set is a type for any type τ, with the metric on it being the Hamming metric

$$d_{\tau\, \mathsf{set}}(S_1, S_2) = ||S_1 \triangle S_2||$$

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

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

Primitives that can be added for this type include

$size : \tau\, \mathsf{set} \multimap \mathbb{R}$
$setfilter : (\tau \rightarrow \mathsf{bool}) \rightarrow \tau\, \mathsf{set} \multimap \tau\, \mathsf{set}$
$setmap : (\sigma \rightarrow \tau) \rightarrow \tau \rightarrow \sigma\, \mathsf{set} \multimap \tau\, \mathsf{set}$
$\cap, \cup, \backslash : \tau\, \mathsf{set} \otimes \tau\, \mathsf{set} \multimap \tau\, \mathsf{set}$
$split : (\tau \rightarrow \mathsf{bool}) \rightarrow \tau\, \mathsf{set} \multimap \tau\, \mathsf{set} \otimes \tau\, \mathsf{set}$

where *size* returns the cardinality of a set, $\cap$ returns the intersection of two sets, $\cup$ their union, and $\backslash$ the difference. Notably, for these last three primitives, we could *not* have given them the type $\tau\, \mathsf{set}\, \& $ $\tau\, \mathsf{set} \multimap \tau\, \mathsf{set}$. To see why, consider $\{b\} \cup \{c, d\} = \{b, c, d\}$ and $\{a\} \cup \{c, d, e\} = \{a, c, d, e\}$. We have $d(\{b\}, \{a\}) = 2$ and $d(\{c, d\}, \{c, d, e\}) = 1$ on the two inputs to $\cup$, but on the output $d(\{b, c, d\}, \{a, c, d, e\}) = 3$, and 3 is strictly larger than $\max(2, 1)$. The functions *setfilter* and *setmap* work mostly as expected, but with a proviso concerning termination below in Section 3.5.

We note that *size* is a special case of a more basic summation primitive:

$$sum : (\tau \rightarrow \mathbb{R}) \rightarrow \tau\, \mathsf{set} \multimap \mathbb{R}$$

---

[2] The reader may note that this primitive is exactly the well-known distributivity property that the BI, the logic of bunched implications [28], 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.

The expression *sum f S* returns $\sum_{s \in S} clip(f(s))$, where $clip(x)$ 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 $size\, S = sum\, (\lambda x.1)\, S$.

The operation *split* takes a predicate on τ, and a set; it yields two sets, one containing the elements of the original set that satisfy the predicate and the other containing all the elements that don't. Notice that *split* is 1-sensitive in its set argument; this is because if an element is added to or removed from that set, it can only affect one of the two output sets, not both.

By using *split* 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 $(-\infty, 0], (0, 10], \ldots, (90, 100], (100, \infty)$.

$hist' : \mathbb{R} \rightarrow \mathbb{R}\, \mathsf{set} \multimap \mathbb{R}\, \mathsf{set}\, \mathsf{list}$
$hist'\, c\, s = \mathbf{if}\, c \geq 101\, \mathbf{then}\, [s]\, \mathbf{else}$
      $\mathbf{let}(y, n) = split(\lambda z.c \geq z)\, \mathbf{in}$
        $y :: hist'\, (c + 10)\, n$

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

Here we are also assuming the use of ordinary distance-insensitive arithmetic operations such as $\geq\, :\, \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathsf{bool}$ and $+ : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}$. We see in the next section that comparison operators like $\geq$ cannot be so straightforwardly generalized to be distance sensitive.

3.5 Higher-Order Set Operations and Termination

A few comments are in order on the termination of the higher-order functions *setfilter*, *setmap*, and *setsplit*. Consider the expression *setfilter f s* for s of type τ set and an arbitrary function $f : \tau \rightarrow \mathsf{bool}$. If f diverges on some particular input $v : \tau$, then the presence or absence of v in the set s can make *setfilter f s* diverge or terminate. This runs afoul of the claim of Theorem 2.4 that two metrically similar computations should together evaluate to metrically nearby values.

One way of avoiding this problem is to adopt primitives for which 2.4 can still be proved: we can ensure *dynamically* that the function argument (*setfilter*, *setmap*, and *setsplit*) terminates by imposing a time limit on the number of steps it can run over each element of the set. Whenever a function exceeds its time limit while operating on a set element x, it is left out of the filter or of the current split as appropriate, and in the case of *setmap*, a default element of type τ is used.

An alternative is to weaken Theorem 2.4 to state that if two computations over metrically related inputs *do* both terminate, then their outputs are metrically related. This weakened result is considerably less desirable for our intended application to differential privacy, however.

A final option is to statically ensure the termination of the function argument. This seems to combine the best features of both of the other choices, at the price of a more complex program analysis.

3.6 Sorting

What about distance-sensitive sorting? Ordinarily, the basis of sorting functions is a comparison operator such as $\geq_\tau : \tau \times \tau \rightarrow \mathsf{bool}$. 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 $cswp : \mathbb{R} \otimes \mathbb{R} \multimap \mathbb{R} \otimes \mathbb{R}$ 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*, [5] with *cswp* being the comparator. With the comparator, we can easily implement a version of insertion sort.

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

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

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 [5], 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 [2].

3.7 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 tuples (s, 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 $(s, t) \in f$ and $(s, t') \in f$, 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, $d_{\sigma \rightharpoonup \tau}(S_1, S_2) = ||S_1 \triangle S_2||$. By isolating the primary key, we can support some familiar relational algebra operations:

$$fmsize : (\sigma \rightharpoonup \tau) \multimap \mathbb{R}$$
$$fmfilter : (\sigma \rightarrow \tau \rightarrow \mathsf{bool}) \rightarrow (\sigma \rightharpoonup \tau) \multimap (\sigma \rightharpoonup \tau)$$
$$mapval : (\tau_1 \rightarrow \tau_2) \rightarrow (\sigma \rightharpoonup \tau_1) \multimap (\sigma \rightharpoonup \tau_2)$$
$$join : (\sigma \rightharpoonup \tau_1) \otimes (\sigma \rightharpoonup \tau_2) \multimap (\sigma \rightharpoonup (\tau_1 \otimes \tau_2))$$

The size and filter functions work similar to the corresponding operations on sets, and there are now two different map operators, one that operates on keys and one on values. 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 $(\{x\} \cup X) \times Y$ has $|Y|$ many more elements than $X \times Y$. McSherry also noted this issue with unrestricted joins, and deals with it in a similar way in PINQ [25].

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

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

which takes a *key extraction* function $f : \tau \rightarrow \sigma$, and a set S of values of type τ, and returns a finite map which maps values $y \in \sigma$ to the set of $s \in S$ such that $f(s) = y$. 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.

4. 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 2.4 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 $\mathsf{db} \rightarrow \mathbb{R}$, we can apply Proposition 4.1 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.

4.1 Background

First, we need a few technical preliminaries from the differential privacy literature [14].

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 perhaps just as well 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 [15] of differential privacy for functions from databases to real numbers is as follows:

Definition A random function $q : \mathsf{db} \rightarrow \mathbb{R}$ is ϵ-differentially private if for all $S \subseteq \mathbb{R}$, and for all databases b, b' with $b \sim b'$, 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:

Definition A random function $q : \tau \rightarrow \sigma$ is ϵ-differentially private if for all $S \subseteq \sigma$, and for all $v, v' : \tau$, have $Pr[q(v) \in S] \leq e^{\epsilon d_\tau(v, v')} Pr[q(v') \in S]$.

Although we will use this general definition below in Lemma 4.2, for the time being we continue considering only functions $\mathsf{db} \rightarrow \mathbb{R}$.

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:

PROPOSITION 4.1 ([15]). *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 data-consuming aspect of differentially private queries was the initial intuition that guided us to the linear-logic-inspired design of the type system.

4.2 The Probability Monad

We now show how to extend our language with a monad of random computations. Formally, the required extensions to the syntax are:

$$
\begin{array}{llll}
\text{Types } \tau & ::= & \cdots \mid \bigcirc\tau \\
\text{Expressions } e & ::= & \cdots \mid \mathbf{return}\, x \mid \mathbf{let}\,\bigcirc x = e\,\mathbf{in}\,e' \\
\text{Values } v & ::= & \cdots \mid \delta
\end{array}
$$

We add $\bigcirc\tau$, the type of random computations over τ. Expressions now include a monadic return, which deterministically always yields x, 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'. We postpone discussing the typing rules until after we have established what the metric on $\bigcirc\tau$ is, and for that we need to understand what its values are.

For simplicity, we follow Ramsey and Pfeiffer [30] in taking a rather *denotational* approach, and think of values of type $\bigcirc\tau$ as literally being mathematical probability distributions. A more strictly syntactic presentation (in terms of, say, pseudo-random number generators) certainly is also possible, but is needlessly technical for our present discussion. In what follows, a probability distribution δ is written as $(p_i, v_i)_{i \in I}$, a multiset of probability-value pairs. We write $\delta(v)$ for the probability $((p_i, v_i)_{i \in I})(v) = \sum_{\{i \mid v_i = v\}} p_i$ of observing v in the distribution δ.

The metric on probability distributions is carefully chosen to allow our type system to speak about differential privacy. Recall that we have assumed ϵ to be fixed, and define:

$$
d_{\bigcirc\tau}(\delta_1, \delta_2) = \frac{1}{\epsilon}\left(\max_{x \in \tau}\left|\ln\left(\frac{\delta_1(x)}{\delta_2(x)}\right)\right|\right)
$$

The definition measures how *multiplicatively* far apart two distributions are in the worst case, as is required by differential privacy. We can then easily see by unrolling definitions that

LEMMA 4.2. *A 1-sensitive function $\tau \to \bigcirc\sigma$ is the same thing as an ϵ-differentially private random function $\tau \to \sigma$.*

The typing rules for the monad are as follows:

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

The introduction rule multiplies the context by infinity, because nearby inputs (perhaps surprisingly!) do not lead to nearby *deterministic* probability distributions. Even if t and t' are close, say $d_\tau(t, t') = \epsilon$, still $\mathbf{return}\, t$ has a 100% chance — and $\mathbf{return}\, t'$ has a 0% chance — of yielding t. The elimination rule adds together the influence Δ that e may have over the final output distribution to the influence Γ that e' has, and provides the variable x *unrestrictedly* (with annotation ∞) to e', because once a differentially private query is made, the published result can be used in any way at all.

We add the following cases to the operational semantics:

$$
\frac{e \hookrightarrow v}{\mathbf{return}\, e \hookrightarrow (1, v)}
$$

$$
\frac{e_1 \hookrightarrow (p_i, v_i)_{i \in I} \qquad \forall i \in I.\, [v_i/x]e_2 \hookrightarrow (q_{ij}, w_{ij})_{j \in J_i}}{\mathbf{let}\,\bigcirc x = e_1\,\mathbf{in}\,e_2 \hookrightarrow (p_i q_{ij}, w_{ij})_{i \in I, j \in J_i}}
$$

We see that $\mathbf{return}$ creates the trivial distribution that always yields v. Monadic sequencing considers all possible values v_i that e could evaluate to, and then subsequently all the values that e' could evaluate to, assuming that it received the sample v_i. The probabilities of these two steps are multiplied, and appropriately aggregated together.

Combining the type system's metric preservation property with Lemma 4.2, we find that typing guarantees differential privacy:

COROLLARY 4.3. *The execution of any closed program e such that $\vdash e : !_n\tau \multimap \bigcirc\sigma$ is an $(n\epsilon)$-differentially private function from τ to σ.*

5. Differential Privacy Examples

Easy examples of ϵ-differentially private computations come from applying the Laplace mechanism at the end of a deterministic computation. We can add a primitive function

$$
add\_noise : \mathbb{R} \multimap \bigcirc\mathbb{R}
$$

which adds Laplace noise $\mathcal{L}_{1/\epsilon}$ to its input. According to Proposition 4.1, this is exactly the right amount of noise to add to a 1-sensitive function to make it ϵ-differentially private.

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.

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

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

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 3.4. We can first of all write the following program.

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

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:

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

$$hist\_query : \mathsf{row}\,\mathsf{set} \multimap \bigcirc(\mathbb{R}\,\mathsf{list})$$
$$hist\_query\,b = seq\,(hist\_query'\,b)$$

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 [15]. 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[3] of McSherry's principle of sequential composition [25].

LEMMA 5.1 (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:

$$sc : (\tau_1 \multimap \bigcirc \tau_2) \to (\tau_1 \multimap \tau_2 \to \bigcirc \tau_3) \to (!_2\tau_1 \multimap \bigcirc \tau_3)$$
$$sc\,f_1\,f_2\,t_1 = \mathbf{let}\,!t_1' = t_1\,\mathbf{in}\,\mathbf{let}\,\bigcirc t_2 = f_1\,t_1'\,\mathbf{in}\,f_2\,t_1'\,t_2$$

It takes two 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:

LEMMA 5.2 (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 $\otimes$.

$$pc : (\tau_1 \multimap \bigcirc \tau_2) \to (\sigma_1 \multimap \bigcirc \sigma_2) \to (\tau_1 \otimes \sigma_1) \multimap \bigcirc(\tau_2 \otimes \sigma_2)$$
$$pc\,f\,g\,(t, s) = \mathbf{let}\,\bigcirc t' = f\,t\,\mathbf{in}\,\mathbf{let}\,\bigcirc s' = g\,s\,\mathbf{in}\,\mathbf{return}(t', s')$$

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 *split* returns a $\otimes$-pair of two sets.

For a final, slightly more complex example, let us consider the privacy-preserving implementation of k-means by Blum et al. [6]. 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.

---

[3] 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.

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:

$$assign : \mathsf{pt}\,\mathsf{list} \to \mathsf{pt}\,\mathsf{set} \multimap (\mathsf{pt} \otimes \mathsf{int})\,\mathsf{set}$$
$$partition : (\mathsf{pt} \otimes \mathsf{int})\,\mathsf{set} \multimap \mathsf{pt}\,\mathsf{set}\,\mathsf{list}$$
$$totx, toty : \mathsf{pt}\,\mathsf{set} \multimap \mathbb{R}$$
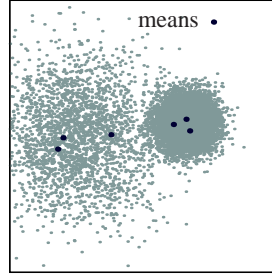$$zip : \tau\,\mathsf{list} \to \sigma\,\mathsf{list} \to (\tau \otimes \sigma)\,\mathsf{list}$$

Figure 4. k-Means Output

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 *split*. 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:

$$iterate : !_3\mathsf{pt}\,\mathsf{set} \multimap \mathbb{R}\,\mathsf{list} \to \bigcirc(\mathbb{R}\,\mathsf{list})$$
$$iterate\,b\,ms = \mathbf{let}\,!b' = b\,\mathbf{in}$$
$$\mathbf{let}$$
$$b'' = partition\,(assign\,ms\,b')$$
$$tx = map\,(add\_noise \circ totx)\,b''$$
$$ty = map\,(add\_noise \circ toty)\,b''$$
$$t = map\,(add\_noise \circ size)\,b''$$
$$stats = zip\,(zip\,(tx, ty), t)$$
$$\mathbf{in}$$
$$seq\,(map\,avg\,stats)$$

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*:

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

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:

$$two\_iters : !_6\mathsf{pt}\,\mathsf{set} \multimap \mathbb{R}\,\mathsf{list} \to \bigcirc(\mathbb{R}\,\mathsf{list})$$
$$two\_iters\,b\,ms = \mathbf{let}\,!b' = b\,\mathbf{in}\,iterate\,!b'\,(iterate\,!b'\,ms)$$

This function is 6ϵ-differentially private. Figure 4 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.

6. Metatheory

In this section we address 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.

LEMMA 6.1 (Weakening). *If $\Gamma \vdash e : \tau$, then $\Gamma + \Delta \vdash e : \tau$.*

$$\frac{\forall v : \tau_1.[v/x]e_1 \sim_r [v/x]e_2 : \tau_2}{\lambda x.e_1 \sim_r \lambda x.e_2 : \tau_1 \to \tau_2} \qquad \frac{\forall v : \tau_1.[v/x]e_1 \sim_r [v/x]e_2 : \tau_2}{\lambda x.e_1 \sim_r \lambda x.e_2 : \tau_1 \multimap \tau_2}$$

$$\frac{v_1 \sim_{r_1} v_1' : \tau_1 \qquad v_2 \sim_{r_2} v_2' : \tau_2}{(v_1, v_2) \sim_{r_1 + r_2} (v_1', v_2') : \tau_1 \otimes \tau_2} \qquad \frac{v \sim_r v' : \tau}{!v \sim_{rs} !v' : !_s \tau}$$

$$\frac{v_1 \sim_r v_1' : \tau_1 \qquad v_2 \sim_r v_2' : \tau_2}{\langle v_1, v_2 \rangle \sim_r \langle v_1', v_2' \rangle : \tau_1 \& \tau_2} \qquad () \sim_r () : 1$$

$$\frac{v \sim_r v' : [\mu \alpha.\tau/\alpha]\tau}{\mathbf{fold}\, v \sim_r \mathbf{fold}\, v' : \mu\alpha.\tau} \qquad \frac{v \sim_r v' : \tau_i}{\mathbf{inj}_i\, v \sim_r \mathbf{inj}_i\, v' : \tau_1 + \tau_2}$$

$$\frac{\forall v_1 : \tau.e_1 \hookrightarrow v_1 \Rightarrow \exists v_2.e_2 \hookrightarrow v_2 \wedge v_1 \sim_r v_2 : \tau}{e_1 \sim_r e_2 : \tau}$$

$$\frac{\forall v \in \tau.\delta_1(v) \le e^{r\epsilon}\delta_2(v)}{\delta_1 \sim_r \delta_2 : \bigcirc\tau}$$

Figure 5. Metric Relation

THEOREM 6.2 (Substitution). *If* $\Gamma \vdash e : \tau$ *and* $\Delta, x :_r \tau \vdash e' : \tau'$, *then* $\Delta + r\Gamma \vdash [e/x]e' : \tau'$.

THEOREM 6.3 (Preservation). *If* $\vdash e : \tau$ *and* $e \hookrightarrow v$, *then* $\vdash v : \tau$.

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 lemma is routine; if we had presented the operational semantics in a small-step style, a progress theorem would also be easy to show.

6.1 Defining the Metric

Up to now, the metrics on types have been dealt with somewhat informally; in particular, our 'definition' of distance for recursive types was not well founded. We now describe a formal definition. It is convenient to treat the metric not as a function, but rather as a *relation* on values and expressions. The relation $v \sim_r v' : \tau$ (resp. $e \sim_r e' : \tau$) means that values v and v' (expressions e and e') of type τ are at a distance of no more than r apart from each other. The metric on expressions is defined by evaluation: if the values that result from evaluation of the two expressions are no farther than r apart, then the two expressions are considered to be no farther than r apart. This relation is defined coinductively on the rules in Figure 5. By this we mean that we define $v \sim_r v' : \tau$ to be the greatest relation consistent with the given rules. A relation is said to be consistent with a set of inference rules if for any relational fact that holds, there exists an inference rule whose conclusion is that fact, and all premises of that rule belong to the relation. Intuitively, this means that we allow infinitely deep inference trees. Note that $\sim_r$ never appears negatively (i.e., negated or to the left of an implication) in the premise of any rule, so we can see that closure under the rules is a property preserved by arbitrary union of relations, and therefore the definition is well-formed.

6.2 Metric Preservation Theorem

Now we can state the central novel property that our type system guarantees. We introduce some notation to make the statement more compact. Suppose $\Gamma = x :_{s_1} \tau_1, \dots x :_{s_n} \tau_n$. A substitution σ for Γ is a list of individual substitutions of values for variables

in Γ, written $[v_1/x_1] \cdots [v_n/x_n]$. A *distance vector* γ is a list $r_1, \dots, r_n$ such that every r_i is in $\mathbb{R}^{\ge 0} \cup \infty$. We say $\sigma \sim_\gamma \sigma' : \Gamma$ when, for every $[v_i/x_i] \in \sigma$ and $[v_i'/x_i] \in \sigma'$, we have $v_i \sim_{r_i} v_i' : \tau_i$. In this case we think of σ and σ' as being 'γ apart': the distance vector γ tracks the distance between each corresponding pair of values. We define the *dot product* of a distance vector and a context as follows: if γ is $r_1, \dots, r_n$, and Γ is as above, then $\gamma \cdot \Gamma = \sum_{i=1}^n r_i s_i$.

THEOREM 6.4 (Metric Preservation). *Suppose* $\Gamma \vdash e : \tau$. *Suppose* σ, σ' *are two substitutions for* Γ *such that* $\sigma \sim_\gamma \sigma' : \Gamma$. *Then we have* $\sigma e \sim_{\gamma \cdot \Gamma} \sigma' e : \tau$.

A straightforward proof attempt of this theorem fails. If we try to split cases by the typing derivation of e, a problem arises at the case where $e = e_1 e_2$. The induction hypothesis will tell us that σe_1 is close to $\sigma' e_1$, and that σe_2 is close to $\sigma' e_2$. But the definition of the metric at function types (whether $\to$ or $\multimap$ — the problem arises for both of them) only quantifies over one value — how then can we reason about both σe_2 and $\sigma' e_2$? This problem is solved by using a *step-indexed metric logical relation* [1, 3] which represents a stronger induction hypothesis, but which agrees with the metric. We defer further details of this argument to the appendix.

7. Related Work

The seminal paper on differential privacy is [15]; 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 [24], and techniques have been developed to reduce the amount of noise required for repeated queries, such as the median mechanism [31]. Dwork [13] gives a useful survey of recent results.

Girard's linear logic [16] 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 [20] 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 [19]), where weakening is permitted. The idea of counting, as we do, multiple uses of the same resource was explored by Wright [32], 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 [11], and Machanavajjhala et al.'s comparison of different notions of privacy with respect to real-world census data [22].

Quantitative Information Flow [21, 23] 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 [8] 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 [17]. Chaudhuri et al. [10] 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* [24] 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. [18] 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. [9] 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.

Acknowledgments

References

[1] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *of Lecture Notes in Computer Science*, volume 3924, pages 69–83, 2006.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in c log n parallel steps. *Combinatorica*, 3(1):1–19, March 1983. ISSN 0209-9683.

[3] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001. ISSN 0164-0925.

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

[5] 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.

[6] A. Blum, C. Dwork, F. McSherry, and K. 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.

[7] A. Blum, K. Ligett, and A. 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.

[8] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In J. Bussche and V. 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.

[9] T.-H. H. Chan, E. Shi, and D. Song. Private and continual release of statistics. Cryptology ePrint Archive, Report 2010/076, 2010. http://eprint.iacr.org/.

[10] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. *SIGPLAN Not.*, 45(1):57–70, 2010. ISSN 0362-1340.

[11] N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

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

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

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

[15] C. Dwork, F. Mcsherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, 2006.

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

[17] T. J. Green, G. Karvounarakis, and V. 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.

[18] A. Gupta, K. Ligett, F. McSherry, A. Roth, and K. Talwar. Differentially private combinatorial optimization. Nov 2009.

[19] J. Ketonen. A decidable fragment of predicate calculus. *Theoretical Computer Science*, 32(3):297–307, 1984. ISSN 03043975.

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

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

[22] A. Machanavajjhala, D. Kifer, J. Abowd, J. Gehrke, and L. 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.

[23] S. McCamant and M. 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.

[24] F. McSherry and K. 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.

[25] F. 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.

[26] A. Narayanan and V. 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. ISBN 978-0-7695-3168-7. doi: http://dx.doi.org/10.1109/SP.2008.33.

[27] K. Nissim, S. Raskhodnikova, and A. 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.

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

[29] S. Park, F. Pfenning, and S. 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.

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

[31] A. Roth and T. Roughgarden. The median mechanism: Interactive and efficient privacy with multiple queries, 2010. To appear in STOC 2010.

[32] D. Wright and C. 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.

$$\frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e} \qquad \frac{e_1 \overset{\ell}{\hookrightarrow} \lambda x.e \qquad e_2 \overset{m}{\hookrightarrow} v \qquad [v/x]e \overset{p}{\hookrightarrow} v'}{e_1\, e_2 \overset{\ell+m+p}{\hookrightarrow} v'} \qquad \frac{}{() \overset{0}{\hookrightarrow} ()}$$

$$\frac{e_1 \overset{\ell}{\hookrightarrow} v_1 \qquad e_2 \overset{m}{\hookrightarrow} v_2}{\langle e_1, e_2 \rangle \overset{\ell+m}{\hookrightarrow} \langle v_1, v_2 \rangle} \qquad \frac{e_1 \overset{\ell}{\hookrightarrow} v_1 \qquad e_2 \overset{m}{\hookrightarrow} v_2}{(e_1, e_2) \overset{\ell+m}{\hookrightarrow} (v_1, v_2)}$$

$$\frac{e \overset{\ell}{\hookrightarrow} (v_1, v_2) \qquad [v_1/x][v_2/y]e' \overset{m}{\hookrightarrow} v'}{\mathbf{let}(x, y) = e \,\mathbf{in}\, e' \overset{\ell+m}{\hookrightarrow} v'} \qquad \frac{e \overset{\ell}{\hookrightarrow} \langle v_1, v_2 \rangle}{\pi_i e \overset{\ell}{\hookrightarrow} v_i}$$

$$\frac{e \overset{\ell}{\hookrightarrow} v}{\mathbf{inj}_i\, e \overset{\ell}{\hookrightarrow} \mathbf{inj}_i\, v} \qquad \frac{e \overset{\ell}{\hookrightarrow} \mathbf{inj}_i\, v \qquad [v/x]e_i \overset{m}{\hookrightarrow} v'}{\mathbf{case}\, e \,\mathbf{of}\, x.e_1 \mid x.e_2 \overset{\ell+m}{\hookrightarrow} v'} \qquad \frac{e \overset{\ell}{\hookrightarrow} v}{\mathbf{fold}\, e \overset{\ell}{\underset{\tau}{\hookrightarrow}} \mathbf{fold}\, v}$$

$$\frac{e \overset{\ell}{\hookrightarrow} \underset{\tau}{\mathbf{fold}}\, v}{\mathbf{unfold}\, e \overset{\ell+1}{\underset{\tau}{\hookrightarrow}} v} \qquad \frac{e \overset{\ell}{\hookrightarrow} v}{!e \overset{\ell}{\hookrightarrow} !v} \qquad \frac{e \overset{\ell}{\hookrightarrow} !v \qquad [v/x]e' \overset{m}{\hookrightarrow} e'}{\mathbf{let}\, !x = e \,\mathbf{in}\, e' \overset{\ell+m}{\hookrightarrow} v'}$$

$$\frac{e_1 \overset{\ell}{\hookrightarrow} (p_i, v_i)_{i \in I} \qquad \forall i \in I.\ [v_i/x]e_2 \overset{m_i}{\hookrightarrow} (q_{ij}, w_{ij})_{j \in J_i}}{\mathbf{let}\, \bigcirc x = e_1 \,\mathbf{in}\, e_2 \overset{\ell + \min_i m_i}{\hookrightarrow} (p_i q_{ij}, w_{ij})_{i \in I, j \in J_i}}$$

$$\frac{e \overset{\ell}{\hookrightarrow} v}{\mathbf{return}\, e \overset{\ell}{\hookrightarrow} (1, v)}$$

Figure 6. Step-Indexed Evaluation Rules

A. Appendix

In this section we sketch in somewhat more detail the proof of the central novel soundness property of our type system.

At a high level, the proof works in two steps. First, we relate the metric as defined above to a *step-indexed metric logical relation* which, as we will see, determines the same metric, but in a way that constitutes a stronger induction hypothesis. Subsequently, we can prove a metric preservation theorem directly on the logical relation. The fact that the logical relation is *step-indexed* means that we imagine we have a finite budget of 'computation steps' with which to discriminate between similar expressions. This notion of 'computation step' is made precise in the following section.

A.1 Step-Indexed Evaluation

The purpose of step-indexing (at least for our purposes) is to accommodate the presence of nontermination in the language, which in turn can be blamed on the presence of recursive types. Because of this, we consider as computation steps the β-reductions of a **fold** against an **unfold**; that is, from a small-step point of view, we care about reductions of the form

$$\mathbf{unfold}\, \mathbf{fold}\, v \mapsto v \qquad (\dagger)$$

But we can count such reductions in big-step style without much difficulty. We give a step-indexed refinement of the existing operational semantics: $e \overset{\ell}{\hookrightarrow} v$ means that e evaluates to v, and during that evaluation, the number of reductions of the form $(\dagger)$ is ℓ. The rules are given in Figure.**??**. Observe that the rule that evaluates a **unfold** adds one to the count, and all other rules simply add together the counts from their subderivations, if any.

A.2 Step-Indexed Metric Logical Relation

Now we introduce the step-indexed metric logical relation. The relation $v_1 \sim_r^k v_2 : \tau$ is conceptually a variant of $v_1 \sim_r v_2 : \tau$. It means approximately the following: that after k computation steps, (in the sense we have just described) there is still no evidence to refute the possibility that values v_1 and v_2 are at least as close as distance r. It is defined by the rules in Figure **??**. This relation is connected to the metric by the fact that $v_1 \sim_r v_2 : \tau$ holds if $v_1 \sim_r^k v_2 : \tau$ holds for all k; this is proved in Section **??**.

A basic property of the logical relation, which helps form an intuition for it, is the fact that it is preserved by decreasing k, and by increasing r. Formally, we have:

LEMMA A.1 (Monotonicity). *Suppose $v_1 \sim_r^k v_2 : \tau$.*
If $k' \leq k$, then $v_1 \sim_r^{k'} v_2 : \tau$.
If $r' \geq r$, then $v_1 \sim_{r'}^{k} v_2 : \tau$.

Proof By induction. ∎

A.3 Fundamental Lemma of Logical Relations

The usual fundamental lemma to show for a logical relation is a form of reflexivity: that every expression is related to itself, assuming everything in its context is related to itself. We must generalize this to account for the metric, but the required lemma is essentially identical to the metric preservation lemma we have already discussed. The only novelty is that both the premise and conclusion are indexed by a step-index k.

LEMMA A.2 (Fundamental Lemma). *Let a well-typed expression $\Gamma \vdash e : \tau$ be given. Suppose σ, σ' are two substitutions for Γ such that $\sigma \sim_\gamma^k \sigma' : \Gamma$. Then we have $\sigma e \sim_{\gamma \cdot \Gamma}^k \sigma' e : \tau$.*

Proof By induction first on the number of steps k, then on the typing derivation of e. We split cases on the typing derivation of e. We show a couple of illustrative cases as examples.

Case:
$$\frac{\Gamma \vdash e_1 : \tau \multimap \tau' \qquad \Delta \vdash e_2 : \tau}{\Gamma + \Delta \vdash e_1\, e_2 : \tau'} \multimap E$$
In this case, we want to show:

$$\forall v. \forall j < k.\ \sigma(e_1\, e_2) \overset{j}{\hookrightarrow} v \Rightarrow \exists v'. \sigma'(e_1\, e_2) \hookrightarrow v'$$

$$\wedge \qquad v \sim_{\gamma \cdot (\Gamma + \Delta)}^{k-j} v' : \tau$$

Let v and $j < k$ be given, and assume $\sigma(e_1\, e_2) \overset{j}{\hookrightarrow} v$. This means we have a derivation

$$\frac{\sigma e_1 \overset{\ell}{\hookrightarrow} \lambda x.e_0 \qquad \sigma e_2 \overset{m}{\hookrightarrow} v_2 \qquad [v_2/x]e_0 \overset{p}{\hookrightarrow} v}{\sigma(e_1\, e_2) \overset{\ell+m+p}{\hookrightarrow} v}$$

such that $\ell + m + p = j$.
By the induction hypothesis on $\Delta \vdash e_2 : \tau$, we know that

$$\forall v_*. \forall j_* < k.\ \sigma e_2 \overset{j_*}{\hookrightarrow} v_* \Rightarrow \exists v_2'. \sigma' e_2 \hookrightarrow v_2'$$

$$\wedge \qquad v_* \sim_{\gamma \cdot \Delta}^{k-j_*} v_2' : \tau$$

so pick $v_* = v_2$ and $j_* = m$, and use the fact that $\sigma e_2 \overset{m}{\hookrightarrow} v_2$ to obtain v_2'. What we know about v_2' at present is that $\sigma' e_2 \hookrightarrow v_2'$ and $v_2 \sim_{\gamma \cdot \Delta}^{k-m} v_2' : \tau$. Using monotonicity, this latter fact becomes:
$$v_2 \sim_{\gamma \cdot \Delta}^{k-\ell-m} v_2' : \tau \qquad (*)$$
By the induction hypothesis on $\Gamma \vdash e_1 : \tau \multimap \tau'$, we know that

$$\forall v_*. \forall j_* < k.\ \sigma e_1 \overset{j_*}{\hookrightarrow} v_* \Rightarrow \exists v_\bullet. \sigma' e_1 \hookrightarrow v_\bullet$$

$$\wedge \qquad v_* \sim_{\gamma \cdot \Gamma}^{k-j_*} v_\bullet : \tau \multimap \tau'$$

so pick $v_* = \lambda x.e_0$ and $j_* = \ell$ and use the fact that $\sigma e_1 \overset{\ell}{\hookrightarrow} \lambda x.e_0$ to obtain $v_\bullet$. By inversion on the rules defining $\sim$, we

$$\frac{\forall s{:}\mathbb{R}^{\geq 0} \cup \{\infty\}.\forall j \leq k.\forall v_1, v_2 : \tau_1.v_1 \sim_s^j v_2 : \tau_1 \Rightarrow [v_1/x]e_1 \sim_{r+s}^j [v_2/x]e_2 : \tau_2}{\lambda x.e_1 \sim_r^k \lambda x.e_2 : \tau_1 \multimap \tau_2}$$

$$\frac{v_1 \sim_{r_1}^k v_1' : \tau_1 \qquad v_2 \sim_{r_2}^k v_2' : \tau_2}{(v_1, v_2) \sim_{r_1+r_2}^k (v_1', v_2') : \tau_1 \otimes \tau_2}$$

$$\frac{\forall s{:}\mathbb{R}^{\geq 0} \cup \{\infty\}.\forall j \leq k.\forall v_1, v_2 : \tau_1.v_1 \sim_s^j v_2 : \tau_1 \Rightarrow [v_1/x]e_1 \sim_{r+\infty s}^j [v_2/x]e_2 : \tau_2}{\lambda x.e_1 \sim_r^k \lambda x.e_2 : \tau_1 \rightarrow \tau_2}$$

$$\frac{v_1 \sim_r^k v_1' : \tau_1 \qquad v_2 \sim_r^k v_2' : \tau_2}{\langle v_1, v_2 \rangle \sim_r^k \langle v_1', v_2' \rangle : \tau_1 \& \tau_2}$$

$$\frac{}{() \sim_r^k () : 1} \qquad \frac{v \sim_r^k v' : [\mu\alpha.\tau/\alpha]\tau}{\mathbf{fold}\, v \sim_r^{k+1} \mathbf{fold}\, v' : \mu\alpha.\tau} \qquad \frac{}{\mathbf{fold}\, v \sim_r^0 \mathbf{fold}\, v' : \mu\alpha.\tau} \qquad \frac{v \sim_r^k v' : \tau_i}{\mathbf{inj}_i\, v \sim_r^k \mathbf{inj}_i\, v' : \tau_1 + \tau_2} \qquad \frac{v \sim_r^k v' : \tau}{!v \sim_{rs}^k !v' : !_s\tau}$$

$$\frac{\forall v_1.\forall j < k.e_1 \overset{j}{\hookrightarrow} v_1 \Rightarrow \exists v_2.e_2 \hookrightarrow v_2 \wedge v_1 \sim_r^{k-j} v_2 : \tau}{e_1 \sim_r^k e_2 : \tau}$$

$$\frac{\forall i.x_i : \tau_i \in \Gamma \wedge v_i \sim_{r_i}^k v_i' : \tau_i}{[v_1/x_1]\cdots[v_n/x_n] \sim_{r_1,\ldots,r_n}^k [v_1'/x_1]\cdots[v_n'/x_n] : \Gamma}$$

$$\frac{\forall v \in \tau.\delta_1(v) \leq e^{r\epsilon}\delta_2(v)}{\delta_1 \sim_r^k \delta_2 : \bigcirc\tau}$$

Figure 7. Step-Indexed Metric Logical Relation

have that $v_\bullet$ must be of the form $\lambda x.e_0'$, and so what we know about it is that $\sigma'e_1 \hookrightarrow \lambda x.e_0'$ and

$$\lambda x.e_0 \sim_{\gamma\cdot\Gamma}^{k-\ell} \lambda x.e_0' : \tau \multimap \tau'$$

By inversion on this, we have

$$\forall s.\forall j_* \leq k - \ell.\, \forall v_2, v_2' : \tau'.\quad v_2 \sim_s^{j_*} v_2' : \tau \Rightarrow$$

$$[v_2/x]e_0 \sim_{\gamma\cdot\Gamma+s}^{j_*} [v_2'/x]e_0' : \tau'$$

so choose $j_* = k - \ell - m$ and $s = \gamma \cdot \Delta$ and use $(*)$ to see that

$$[v_2/x]e_0 \sim_{\gamma\cdot\Gamma+\gamma\cdot\Delta}^{k-\ell-m} [v_2'/x]e_0' : \tau'$$

By inversion on this, we have

$$\forall v.\forall j_* < k - \ell - m.\, [v_2/x]e_0 \overset{j_*}{\hookrightarrow} v \Rightarrow \exists v'.[v_2'/x]e_0' \hookrightarrow v'$$

$$\wedge \quad v \sim_{\gamma\cdot\Gamma+\gamma\cdot\Delta}^{k-\ell-m-j_*} v' : \tau'$$

so choose $j_* = p$ and apply the known fact that $[v_2/x]e_0 \overset{p}{\hookrightarrow} v$, to obtain the required v' such that

$$v \sim_{\gamma\cdot(\Gamma+\Delta)}^{k-j} v' : \tau$$

by observing that $\gamma \cdot (\Gamma + \Delta) = \gamma \cdot \Gamma + \gamma \cdot \Delta$. Note also that we have established enough facts about evaluation to derive

$$\frac{\sigma e_1 \hookrightarrow \lambda x.e_0' \qquad \sigma e_2 \hookrightarrow v_2' \qquad [v_2'/x]e_0' \hookrightarrow v'}{\sigma'(e_1\, e_2) \hookrightarrow v'}$$

Case:

$$\frac{\Gamma, x :_1 \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \multimap \tau'} \multimap I$$

Let σ, σ' be given, and assume $\sigma \sim_\gamma^k \sigma' : \Gamma$. We must show

$$\lambda x.\sigma e \sim_{\gamma\cdot\Gamma}^k \lambda x.\sigma'e : \tau \multimap \tau'$$

which means showing that

$$\forall s.\forall j \leq k.\forall v_1, v_2 : \tau.\quad v_1 \sim_s^j v_2 : \tau \Rightarrow$$

$$[v_1/x]\sigma e \sim_{\gamma\Gamma+s}^j [v_2/x]\sigma'e : \tau'$$

by the definition of the logical relation at $\tau \multimap \tau'$. But this follows immediately from the induction hypothesis applied to the derivation of $\Gamma, x :_1 \tau \vdash e : \tau'$ and the substitutions $[v_1/x]\sigma$ and $[v_2/x]\sigma'$.

∎

As a corollary, we obtain a more familiar result, that every expression is related to itself at distance zero.

COROLLARY A.3. *If $\vdash e : \tau$, then for any k we have $e \sim_0^k e : \tau$.*

A.4 Relating the Metric to the Logical Relation

To see that the metric coincides with the logical relation, we must first show that the metric satisfies a variant of the triangle inequality familiar from the study of metric spaces.

LEMMA A.4 (Triangle Inequality). *For any closed, well-typed values $v, v', v'' : \tau$,*
If $v \sim_r v' : \tau$ and $v' \sim_s v'' : \tau$, then $v \sim_{r+s} v'' : \tau$.

Proof By induction on the derivation. ∎

With this in place, we can show the soundness and completeness of the logical relation with respect to the metric. We assume tacitly in both of the following results that v, v' are closed, well-typed values of type τ.

LEMMA A.5. *If $v \sim_r^k v' : \tau$ for all k, then $v \sim_r v' : \tau$.*

LEMMA A.6. *If $v \sim_r v' : \tau$, then $v \sim_r^k v' : \tau$ for all k.*