Contextual Linear Types for Differential Privacy

MATÍAS TORO, Computer Science Department (DCC), University of Chile, Chile
DAVID DARAIS, Galois, Inc., USA
CHIKE ABUAH, Amazon, USA
JOSEPH P. NEAR, Computer Science Department, University of Vermont, USA
DAMIÁN ÁRQUEZ, FEDERICO OLMEDO, and ÉRIC TANTER, Computer Science Department (DCC), University of Chile & IMFD, Chile

Language support for differentially private programming is both crucial and delicate. While elaborate program logics can be very expressive, type-system-based approaches using linear types tend to be more lightweight and amenable to automatic checking and inference, and in particular in the presence of higher-order programming. Since the seminal design of Fuzz, which is restricted to ϵ-differential privacy in its original design, significant progress has been made to support more advanced variants of differential privacy, like (ϵ, δ)-differential privacy. However, supporting these advanced privacy variants while also supporting higher-order programming in full has proven to be challenging. We present Jazz, a language and type system that uses linear types and latent contextual effects to support both advanced variants of differential privacy and higher-order programming. Latent contextual effects allow delaying the payment of effects for connectives such as products, sums, and functions, yielding advantages in terms of precision of the analysis and annotation burden upon elimination, as well as modularity. We formalize the core of Jazz, prove it sound for privacy via a logical relation for metric preservation, and illustrate its expressive power through a number of case studies drawn from the recent differential privacy literature.

CCS Concepts: • **Security and privacy** → **Logic and verification**; • **Theory of computation** → *Linear logic*; **Type structures**; **Program semantics;**

Additional Key Words and Phrases: Type systems, differential privacy

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

This work is partially funded by ANID FONDECYT Projects 11181208, 1190058, 3200583, ANID Millennium Science Initiative Program code ICN17_002, and NSF award CCF-2119939.
David Darais work done in part while at University of Vermont.
The publication was written prior to Chike Abuah joining Amazon.
Authors' addresses: M. Toro, Computer Science Department (DCC), University of Chile, Santiago, Chile; email: mtoro@dcc.uchile.cl; D. Darais, Galois, Inc., Portland; email: darais@galois.com; C. Abuah, Amazon; email: abuahchu@gmail.com; J. P. Near, Computer Science Department, University of Vermont, Burlington; email: jnear@uvm.edu; D. Árquez, F. Olmedo, and É. Tanter, Computer Science Department (DCC), University of Chile & IMFD, Santiago, Chile; emails: darquez@dcc.uchile.cl, folmedo@dcc.uchile.cl, etanter@dcc.uchile.cl.

0164-0925/2023/05-ART8 $15.00
https://doi.org/10.1145/3589207

1 INTRODUCTION

Note. This article uses colorblind-friendly colors in notation to convey information and is best consumed using an electronic device or color printer.

Over the past decade, differential privacy [32] has become the de facto gold standard in protecting the privacy of individuals when processing sensitive data. In contrast to traditional approaches like de-identification, differential privacy provides a formal, composable privacy guarantee. Differentially private algorithms typically protect privacy by selecting from a handful of basic *mechanisms* to perturb their outputs. For example, the *Laplace mechanism* can be used to add noise to the population count of a city to prevent an adversary from successfully guessing whether or not a particular individual lives in that city. Most programming-language-based approaches to differential privacy are applied to verifying either the *implementation* of a mechanism, such as the Exponential mechanism, or the *composition* of multiple uses of mechanisms, such as computing a histogram using the Laplace mechanism (multiple times) as a primitive.

There are two challenges when writing differentially private programs. First, noise must be added to the right values in the program to achieve *some* guarantee of privacy; this includes the final output of the program, as well as many intermediate program values. Second, the *correct amount of noise* must be added in those places to achieve the *desired amount* of privacy. In the differential privacy framework, privacy is a quantitative feature—more noise gives more privacy. Adding too little noise is as ineffective as adding no noise at all, and adding too much noise renders the result of the computation useless. Programmers must therefore ensure they have added *enough noise*, *in the right places*, and that the noise is *minimal*—a daunting task.

Since differential privacy is a probabilistic, multi-run (hyper [25]) property, it is not straightforward to develop test cases for differentially private algorithms. Consequently, differentially private algorithms are usually developed by experts in the field, and these experts produce manual proofs of privacy for each new algorithm. This reliance on experts is limiting. First, there is a practical need for developing privacy-preserving applications without access to an expert in differential privacy. Even still, experts are not perfect: For example, several incorrect versions of the Sparse Vector Technique [31, 32] have appeared in published papers [41], despite being authored and peer-reviewed by experts in differential privacy.

Due to these challenges, verifying differential privacy in programs via typechecking has received considerable attention. The first such approach, Fuzz [49], uses linear types to verify pure ϵ-differential privacy. Fuzz and its successor DFuzz [34] have a number of attractive properties, including support for automation and higher-order programming. Fuzz was the first to use linear types to bound *function sensitivity*: how much a function's output changes given a change to its input. Sensitivity is then used to determine the (minimal) amount of noise required to achieve privacy. Fuzz uses the same sensitivity type system to also track privacy, which is advantageous due to its simplicity, but as a consequence is unable to support advanced variants of differential privacy, like (ϵ, δ). A recent approach [29] extends the terminating fragment of Fuzz using graded comonadic liftings to support advanced variants such as (ϵ, δ)-differential privacy. In the following, we call this extended language Fuzz$^{\epsilon\delta}$. Another approach, HOARe^2, uses relational refinement types to encode differential privacy [14] and improves on Fuzz-like systems in its ability to support advanced variants. In general, type-based approaches such as Fuzz and HOARe^2 are used to verify programs that compose mechanisms, and not the implementations of mechanisms.

An alternative set of approaches use program logics [15–17, 50] to verify both the implementations of mechanisms and simple forms of composing mechanisms, while also supporting advanced variants such as (ϵ, δ)-differential privacy [32], zero-concentrated differential privacy [21], and Rényi differential privacy [44]. However, these benefits come at the expense of automation and support for higher-order programming.

The Duet language [46] and type system strikes a new balance in this space by building on the designs of Fuzz and DFuzz. Like Fuzz, Duet supports automation and higher-order programming, and like Fuzz$^{\epsilon\delta}$, HOARe^2, and recently developed program logics, Duet supports advanced variants of differential privacy. Like all type-based approaches, Duet cannot be used to verify implementations of mechanisms, however, even when verifying programs that compose mechanisms there is still room to improve: Duet is not expressive enough to support higher-order programming in full generality—something Fuzz, DFuzz, Fuzz$^{\epsilon\delta}$, and HOARe^2 are each able to achieve.

This article presents Jazz, the successor to Duet that significantly improves upon its design. Jazz is a linear type system with support for *latent contextual effects* for function sensitivity and differential privacy; this combination supports advanced privacy variants (such as Duet, Fuzz$^{\epsilon\delta}$, and HOARe^2), automation (such as Duet and DFuzz), and fully general higher-order programming (such as Fuzz/DFuzz, Fuzz$^{\epsilon\delta}$, and HOARe^2). Like Duet, the Jazz language is built from two mutually embedded sublanguages—one for sensitivity and one for privacy—which allows it to support advanced variants of differential privacy automatically through typechecking. Also like Duet (and Fuzz/DFuzz, Fuzz$^{\epsilon\delta}$, and HOARe^2), Jazz is designed for verifying the composition of mechanisms and not their direct implementation.

The key insight of Jazz is the incorporation of *latent contextual effects* into a linear type system. A *latent* effect is one that is *deferred* or *delayed*; rather than accounting for the effect immediately, it is tracked and accounted for later. A *contextual* effect is one that tracks effect information for each variable in the context, including closure variables used in higher-order function bodies. Technically, this is similar to the *open closure types* introduced by Scherer and Hoffmann [51], specialized to the tracking of sensitivity and privacy and generalized to positive type constructors such as sums and products. In addition to supporting higher-order programming in the presence of advanced privacy variants, these latent contextual effects also can yield advantages in terms of precision of the analysis, annotation burden, and modularity.

The challenge of higher-order programming. Consider the n-iteration loop combinator in Fuzz, loop_n, which has type $\tau \to (\tau \to \bigcirc\tau) \multimap_n \bigcirc\tau$. This type describes a two-argument function that takes some value of type τ as the first argument, a function as second argument (which accepts and returns values of type τ), and returns a final value of type τ. The modality $\bigcirc$ in the return type for the function argument and final return type indicates that the function is probabilistic (due to the use of differential privacy mechanisms) and when appearing in the codomain of a linear arrow $\multimap$ indicates that the function satisfies differential privacy.

Both function sensitivity and differential privacy are two-run (hyper)properties of a function output w.r.t. some particular input. For example, a function of body $2x + 3y$ is 2-sensitive in x and 3-sensitive in y, meaning that if e.g., input x varies by at most d and y is held constant, then the function output varies at most by $2d$. When a closure is created, the closure captures sensitivities as well as values, so the sensitivity of the closure $\lambda x.\ 2x + 3y$ would be "3 in y." The situation is analogous when tracking privacy and creating closures that capture privacy costs. Looking back to the type of loop_n in Fuzz, the second argument will be a closure whose captured environment tracks a privacy cost for each closure variable. The interpretation of the linear function type $\multimap_n$ is to *scale* the privacy effects in the closure environment of the looping function of type $\tau \to \bigcirc\tau$ by n. We call this scaling *implicit* and *pervasive* in Fuzz, because it occurs at every let-binding and function call. In the original Fuzz language, such scaling is only sound and precise for pure ϵ-differential privacy, and as a consequence of this pervasive scaling, Fuzz could not be instantiated to advanced differential privacy variants, until recently, where Fuzz$^{\epsilon\delta}$ now supports advanced variants such as (ϵ, δ)-differential privacy through a path metric construction.

The Duet language prohibits this pervasive scaling in its type system to support advanced differential privacy variants, but as a consequence it cannot initially derive a type for loop_n, and

instead it must define a custom typing rule for loop_n. The issue is that Duet prohibits *all* scaling of privacy quantities. However, scaling *is* allowable (i.e., sound) in special restricted instances when using advanced variants. The challenge is then to disallow implicit pervasive scaling while allowing explicit restricted scaling. Because no type can be written for loop_n in Duet, it (and many other higher-order functions) must be given explicit typing rules. This poses a significant restriction on higher-order programming, for instance, loop_n cannot be lambda-abstracted in Duet.

Jazz directly solves the challenge of encoding the explicit, restricted scaling that is required to support both advanced privacy variants and higher-order programming. In Jazz, the type of the n-iteration construct is: $\text{loop}_n : \tau \to (\tau \xrightarrow{\rceil\Sigma\lceil^{\epsilon,\,\delta}} \tau) \xrightarrow{\rceil\Sigma\lceil^{n\epsilon,\,n\delta}} \tau$. In this type, the privacy effect on the closure is given an explicit representation notated $\rceil\Sigma\lceil^{\epsilon,\,\delta}$, which means "$(\epsilon,\delta)$-privacy for variables in the closure environment Σ." This effect is *latent,* because the effect is not "paid for" until (and each time) the function is called, and it is *contextual* because it includes a privacy effect (which may be "zero") for each free variable in the context. This effect is then explicitly scaled by n, the number of loop iterations, in the final effect of applying the function.[1] More powerful looping combinators such as advanced composition can also be encoded with these latent contextual effects; such combinators cannot be described in any prior linear type system—including Fuzz and DFuzz.

Contributions. Jazz supports writing ***higher-order programs****, and* ***automatically*** *verifying that such programs satisfy* ***advanced variants*** *of differential privacy.* The novel features of Jazz—linear types with latent contextual effects—are crucial for practical differentially private programming. We illustrate this expressive power by showing how to encode numerous mechanisms and tools for differential privacy as Jazz primitives, including the Laplace, Gaussian, and Exponential mechanisms, advanced composition, and privacy amplification by subsampling. We also demonstrate the use of Jazz to verify larger algorithms in two case studies: the MWEM algorithm [38] and a recently proposed differentially private machine learning algorithm based on gradient descent with adaptive gradient clipping [53]. Note that these examples are expressible in Duet only by adding new core typing rules for each primitive used, which strictly speaking requires re-proving the metatheory of the extended language. In contrast, Jazz subsumes Duet and supports all these examples without having to add new typing rules, and with a much smaller core language. Finally, Jazz is amenable to reasonably efficient automated typechecking; we have implemented a typechecker for the language that can verify privacy costs for our case studies in milliseconds.

We prove the type soundness of Jazz using a step-indexed logical relation over a mixed big-step/denotational semantics with embedded discrete probability distributions as **probability mass functions (PMFs)**.[2]

In summary, the contributions of this article are:

- Jazz, a practical, higher-order, general purpose programming language for writing differentially private programs, which supports advanced variants of differential privacy.
- A novel linear type system for Jazz that includes latent contextual effects, allowing to delay the payment of effects of connectives such as product, sums, and functions, until actually eliminated; e.g., if the second element of a pair is never used, then it does not contribute to the effect of the program.
- A formalization and proof of type soundness of λ_{J}, the core language of Jazz, based on a proof technique with step-indexed logical relations.

[1] In addition to the color red, we notate the arrow in privacy function types with a double head $\twoheadrightarrow$ to further visually distinguish them from sensitivity function arrows $\to$. We describe details such as the definition of the $\rceil\_\lceil-$ notation later.
[2] We restrict ourselves to discrete distributions, because considering continuous distributions would complicate the language semantics (continuous distributions interact badly with higher-order functions), and our main focus here is on the type system.

- A prototype implementation of the Jazz typechecker, together with a library of primitives for differential privacy, and case studies that demonstrate the expressive power and practicality of Jazz.

We first briefly introduce some key concepts of differential privacy (Section 2) and then give an overview of key design choices and benefits of contextual linear types in Jazz (Section 3). Jazz is a two-language design, and what follows is a presentation of each sub-language in two multi-section arcs. First, we present the sensitivity-only language design (Section 4) and metatheory (Section 5). This language does not include differential privacy operations in the language or privacy quantities in types. Next, building on this sensitivity core, we present the full privacy language design and metatheory (Sections 6 and 7). Finally, we discuss implementation details including gaps between the actual implementation of Jazz and its formal model (Section 8), present a few case studies in Jazz (Section 9), discuss related work (Section 10), and conclude (Section 11). An extended version with proofs of all the results stated in the article can be found in the companion technical report [54].

2 A DIFFERENTIAL PRIVACY PRIMER

Differential privacy is a mathematical definition of what it means for a computation over sensitive data to preserve privacy [32]. It interprets privacy as a form of plausible deniability and relies on the use of randomization to achieve it. Informally, a randomized algorithm is differentially private if the probability that it outputs a particular value remains almost the same with or without a single individual's data used as part of the input. Formally, the definition is parameterized by two *privacy parameters* ϵ and δ that specify to what extent two probabilities are "almost the same," and by a *distance metric* over the algorithm's (sensitive) input whose role we discuss shortly.

Definition 2.1 (Differential Privacy). Given a randomized algorithm (or *mechanism*) $\mathcal{M} \in A \to B$ and a distance metric $\mathfrak{D}_A \in A \times A \to \mathbb{R}$, the algorithm $\mathcal{M}$ satisfies (ϵ, δ)-differential privacy if for all $x, x' \in A$ such that $\mathfrak{D}_A(x, x') \leq 1$ and all possible sets $S \subseteq B$ of outcomes, $\Pr[\mathcal{M}(x) \in S] \leq e^\epsilon \Pr[\mathcal{M}(x') \in S] + \delta$.

The parameter ϵ quantifies the adversary ability to distinguish two neighboring inputs upon observing the corresponding algorithm outputs. It represents the privacy guarantee provided by the algorithm—the smaller, the less information is leaked about its input. However, the parameter δ represents a *failure* probability: With probability at most δ, the algorithm is allowed to violate privacy altogether. In combination, ϵ and δ are typically understood as the "privacy cost" incurred by publicly releasing the algorithm output, associated to a given sensitive input. The case where $\delta = 0$ is called *pure* (or *pure* ϵ-) differential privacy, and the case where $\delta > 0$ is called *approximate* differential privacy. Several other recently proposed variants of the definition build on the advantages of (ϵ, δ)-differential privacy while eliminating the potential for failure; these include **Rényi differential privacy (RDP)** [44], **zero-concentrated differential privacy (zCDP)** [21], and **truncated concentrated differential privacy (tCDP)** [20].

Two algorithm inputs are said to be *neighbors* if the distance between them is bounded by 1 (i.e., $\mathfrak{D}(x, x') \leq 1$). For the formal definition to match our informal statement, the distance metric $\mathfrak{D}_A$ should ensure that neighboring inputs differ by at most one individual's data. Formalizing this notion depends heavily on the domain, so different definitions of $\mathfrak{D}$ are used in different domains. When considering a relational database table represented as a bag of tuples, one commonly used definition for $\mathfrak{D}_{\text{DB}}$ is symmetric difference [42]: $\mathfrak{D}_{\text{DB}}(x, x') = |(x - x') \cup (x' - x)|$. Under this definition, $\mathfrak{D}_{\text{DB}}(x, x') = 1$ for tables that differ in one *row*; if the data contributed by each individual is bounded to a single row, then this is a good approximation of neighboring inputs.

The definition of differential privacy implies two key properties: post-processing and composition. Post-processing means that the output of a differentially private mechanism *stays* differentially private, no matter what additional processing is applied. Composition allows bounding the privacy cost of multiple computations over the same underlaying data: Running an (ϵ_1, δ_1)-differentially private mechanism followed by an (ϵ_2, δ_2)-differentially private mechanism satisfies $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$-differential privacy. The privacy parameters ϵ and δ are often called the *privacy cost* because of the additive nature of composition.

Basic mechanisms, composition, and scaling. Differential privacy mechanisms typically add noise to the output of a deterministic function scaled to the function's *sensitivity* [32]. A function $f \in A \to B$ with distance metrics $\mathfrak{D}_A$ and $\mathfrak{D}_B$ is called *s-sensitive* if $\mathfrak{D}_A(x, y) \leq d \implies \mathfrak{D}_B(f(x), f(y)) \leq sd$ for every $d \in \mathbb{R}$ and every $x, y \in A$. Two commonly used mechanisms are the *Laplace* [32] and the *Gaussian* [7, 32] mechanisms. Given an s-sensitive function $f \in A \to \mathbb{R}$, the Laplace mechanism releases $f(x) + \text{Lap}(\frac{s}{\epsilon})$, where $\text{Lap}(b)$ denotes a random sample from the Laplace distribution centered at 0 with scale b; it satisfies ϵ-differential privacy. The Gaussian mechanism releases $f(x) + \mathcal{N}(\frac{2s^2 \ln(1.25/\delta)}{\epsilon^2})$, where $\mathcal{N}(\sigma^2)$ denotes a random sample from the Gaussian distribution centered at 0 with variance σ^2 and $\epsilon, \delta \in (0, 1)$; it satisfies (ϵ, δ)-differential privacy. While the original Gaussian mechanism requires $\epsilon < 1$, Balle and Wang [7] introduce a variant—called *analytic* Gaussian mechanism—that drops the requirement that $\epsilon < 1$.

For implementation purposes, naive floating-point truncations of the real-valued Laplace distribution lead to fatal privacy breaches [43]. Canonne et al. [22] have shown that the Laplacian and Gaussian mechanisms can both be discretized while still providing formal privacy guarantees.

Advanced composition [32] yields tighter bounds on privacy cost for many iterative algorithms but requires (ϵ, δ)-differential privacy. For $\epsilon, \delta, \delta' \geq 0$, the class of (ϵ, δ)-differentially private mechanisms satisfies $(\epsilon', k\delta + \delta')$-differential privacy under *k-fold adaptive composition* (e.g., a loop with k iterations) where $\epsilon' = k\epsilon(e^\epsilon - 1) + \epsilon\sqrt{2k \ln(1/\delta')}$. Advanced composition is especially useful for iterative algorithms that perform many differentially private steps in sequence (e.g., iterative machine learning algorithms).

Differential privacy is stated in terms of neighboring inputs, i.e., inputs x and x' such that $\mathfrak{D}_A(x, x') \leq 1$. When $\mathfrak{D}_A(x, x') > 1$, an ϵ-differentially private mechanism provides $\mathfrak{D}_A(x, x') \cdot \epsilon$-differential privacy. Distances larger than one are typically interpreted as groups of individuals, e.g., $\mathfrak{D}_A(x, x') = k$ represents a change to k individual's input data. Therefore, an ϵ-differentially private mechanism provides $k\epsilon$-differential *group privacy* [32] for groups of size k. A similar property holds for (ϵ, δ)-differential privacy and the more recently developed advanced variants, but the scaling of privacy cost is *nonlinear* for all of these variants. For example, on inputs at distance k, an algorithm satisfying (ϵ, δ)-differential privacy yields outputs that are only $(k\epsilon, k\delta e^{(k-1)\epsilon})$-close–instead of $(k\epsilon, k\delta)$-close. This nonlinearity makes it difficult to apply techniques based on linear types (which generally internalize linear scaling of costs [34, 49]) for these variants of differential privacy.

Verification Techniques for Differential Privacy. A number of techniques have been proposed for verifying that a program satisfies differential privacy, including approaches based on linear logic [29, 34, 46, 49, 60], couplings and program logics [5, 12, 13, 15–17, 50], and randomness alignments [55, 59]. Our work is most closely related to Fuzz [49] and its descendants DFuzz [34], Duet [46], and Fuzz$^{\epsilon\delta}$ [29], which are based on linear type systems. In particular, these approaches focus heavily on fully automated verification of differential privacy properties through typechecking and are typically less expressive than program logics, which by contrast support significantly less (or no) automation. We defer a more complete discussion of related work to Section 10.

3 OVERVIEW OF JAZZ

Jazz builds on the linear type system of Fuzz [49] and the two-language design of Duet [46] by introducing *latent contextual effects*. This section introduces and motivates the design of Jazz's two languages—one for sensitivity and one for privacy—using simple examples. The design of each language is described in Sections 4 and 6, respectively, and each metatheory is described in Sections 5 and 7.

3.1 A Two-language Design

Jazz follows Duet in being structured as two mutually embedded sublanguages, one for sensitivity and one for privacy. In a nutshell, this is because supporting *scaling* of both sensitivity and privacy in a uniform and tight fashion is sound only for ϵ-differential privacy, and *not* for (ϵ, δ)-differential privacy, which has nonlinear group privacy, as discussed in Section 2.

Let us elaborate on this. Recall that the framework of differential privacy builds on randomization to achieve privacy, and the randomization is typically calibrated according to the sensitivity of the function whose output one wants to protect. Therefore, in a language to describe differentially private computations, we can mostly distinguish two classes of functions: On the one hand, randomized (effectful) functions that are annotated with privacy information, and on the other hand, pure functions that are annotated with sensitivity information. However, when composing functions, the way their information is combined highly depends on the class of the composed functions.

For example, when composing two functions from the sensitivity fragment, their information is naturally combined via scaling. Consider, for instance, a 3-sensitive function f; it is not hard to see that the expression

$$f(x + x)$$

is $(2 \cdot 3)$-sensitive in x. The same scaling pattern remains valid when composing a function from the sensitivity fragment with a function from the privacy fragment, provided the latter satisfies pure differential privacy: If g is an, e.g., ϵ-differentially private function, then the computation

$$g(x + x)$$

is $(2 \cdot \epsilon)$-differentially private in x. This uniform scaling behavior, which besides being sound is also tight, lies at the heart of Fuzz/DFuzz design. In fact, in Fuzz/DFuzz, both classes of functions live within the same space, following the same typing rules. To enable this uniform treatment, the languages rely on the two fundamental ingredients: (i) the presence of a monad/modality to encode randomization and (ii) the association of a metric space to each type, which in the case of monadic types, is tailored to *encode* differential privacy. Said otherwise, in Fuzz/DFuzz, differential privacy is encoded as a sensitivity claim about functions (with a monadic return type).

Unfortunately, this linear scaling—pervasive in Fuzz/DFuzz—is no longer sound when composing a function from the sensitivity fragment with a function from the privacy fragment that satisfies *approximate*—rather than *pure*—differential privacy. Returning to the previous example, if g is instead (ϵ, δ)-differentially private, then $g(x + x)$ does *not* satisfy $(2\epsilon, 2\delta)$-differential privacy in x, but only $(2\epsilon, 2\delta e^{\epsilon})$-differential privacy as dictated by the group privacy property for approximate differential privacy (see Section 2 and Dwork and Roth [32, Section 2.3]). In effect, this is why approximate differential privacy lies out of the scope of Fuzz/DFuzz "uniform" design—although recently, de Amorim et al. [29] have shown that using different metrics allows approximate differential privacy functions to be linear, extending a terminating subset of Fuzz to support (ϵ, δ)-differentially privacy via a path metric construction.

In view of this, Duet introduced a two-language design, separating a sensitivity sublanguage in which scaling remains *implicit* (internalized by the typing rules) and *pervasive* (modeling function composition) and a privacy sublanguage in which scaling is *explicit and restricted* (ad hoc typing rules are needed, e.g., to model some privacy combinators). Jazz builds upon this approach and significantly improves both sublanguages, thanks to latent contextual effects, as we illustrate next.

Coloring convention. As noted in the introduction, this article uses colorblind-friendly colors in notation to convey information and is best consumed using an electronic device or color printer. Jazz consists of two mutually embedded sublanguages, and each language is given its own color. Furthermore, these two languages share the same language of types, so we use a third color for the shared fragment. Consequently, we use three color schemes throughout the article: (1) blue for general math notation and the type system shared between languages; (2) green for the sensitivity language; and (3) red for the privacy language. We have carefully chosen the schemes to be distinguishable (as much as possible) for persons with various forms of color blindness.[3]

3.2 Sensitivity

In the sensitivity sublanguage of Jazz, the identity function is written $\lambda^{\mathsf{s}} x.\ x$. We write sensitivity lambdas as λ^{s} to more easily distinguish them from privacy lambdas, written λ^{p} (described later). The identity function is 1-sensitive in its argument x: If x changes by d, then the function's output also changes by d. Similarly to the identity function, the doubling function $\lambda^{\mathsf{s}} x.\ x + x$ is 2-sensitive in x: If x changes by d, then the function's output changes by $2d$.

Fuzz extends the notion of sensitivity to multi-argument functions by assigning a sensitivity to each argument. For example, the curried function $\lambda^{\mathsf{s}} x.\ \lambda^{\mathsf{s}} y.\ x + x + y$ is 2-sensitive in x and 1-sensitive in y. If x changes by d_x and y changes by d_y, then the function's output changes by $2d_x + 1d_y$.

In general, the sensitivity of a function can be written as a linear combination of the changes in its inputs. In Jazz, we express function sensitivities as linear formulas over the function's input variables, using the variable name itself as a placeholder for the change in that input. Jazz's type system gives the following types for the three examples we have seen so far:

$$(\lambda^{\mathsf{s}} x.\ x) : (x : \mathbb{R}) \xrightarrow{x} \mathbb{R},$$
$$(\lambda^{\mathsf{s}} x.\ x + x) : (x : \mathbb{R}) \xrightarrow{2x} \mathbb{R},$$
$$(\lambda^{\mathsf{s}} x.\ \lambda^{\mathsf{s}} y.\ x + x + y) : (x : \mathbb{R}) \xrightarrow{0x} (y : \mathbb{R}) \xrightarrow{2x+y} \mathbb{R}.$$

The linear formulas written above function arrows represent the *sensitivity effect* of the corresponding function. The general form of sensitivity function types is $(x : \tau_1) \xrightarrow{\Sigma} \tau_2$, where Σ is the *sensitivity effect* of the function, expressed as a linear formula. Note that x is in scope for Σ and τ_2. Importantly, occurrences of x in Σ and τ_2 represent the *sensitivity* of the variable x, rather than its value, so Jazz supports *sensitivity-dependent* types. Also, we usually drop "null" effects over function arrows such as $\xrightarrow{0x}$ above and instead just write $\rightarrow$.

As usual, Jazz accommodates higher-order functions by scaling sensitivities. For example, applying a 2-sensitive function twice yields a 4-sensitive function:

$$(\lambda^{\mathsf{s}} f.\ \lambda^{\mathsf{s}} y.\ f\ y + f\ y)\ (\lambda^{\mathsf{s}} x.\ x + x) : (y : \mathbb{R}) \xrightarrow{4y} \mathbb{R}.$$

[3]We chose colors following the 24-Color Palette from http://mkweb.bcgsc.ca/colorblind. E.g., to persons with deuteranopia (the most common form of color blindness), colorschemes ■/■/■ appear as ■/■/■, respectively.

In addition to function types, other type connectives in Jazz such as sums and products also carry sensitivity effects, such as $\tau_1 \ {}^{\Sigma_1}\!\oplus^{\Sigma_2}\ \tau_2$ for sums, $\tau_1 \ {}^{\Sigma_1}\!\otimes^{\Sigma_2}\ \tau_2$ for multiplicative products, and $\tau_1 \ {}^{\Sigma_1}\!\&^{\Sigma_2}\ \tau_2$ for additive products. These connectives are extensions of the linear type connectives $\tau_1 \oplus \tau_2$, $\tau_1 \otimes \tau_2$ and $\tau_1 \ \& \ \tau_2$ from Fuzz, augmented with latent sensitivity effects.

We say that sensitivity effects in Jazz are *latent,* because they only contribute to the sensitivity of an expression when the type connective is actually eliminated. For instance, in the third example above—a curried function—the sensitivity effect on the first argument is delayed until the second argument is received. If a second argument is *never* received, then the sensitivity effect on the first argument can be ignored. Likewise, the annotations Σ_1 and Σ_2 in the type $\tau_1 \ {}^{\Sigma_1}\!\otimes^{\Sigma_2}\ \tau_2$ encode the latent sensitivity cost for each component of the connective: for τ_1 (the left) and τ_2 (the right), respectively. In contrast to Fuzz, creating a pair in Jazz can have no immediate sensitivity cost: Only projecting out of a pair has a cost in sensitivity, depending on which component is projected. Additionally, we say that sensitivity effects are *contextual* because Σ can refer to variables in scope.

3.3 Privacy

To encode differential privacy, Jazz makes use of *privacy functions* (notated $\twoheadrightarrow$) rather than of *sensitivity functions* (notated $\to$) as in the previous examples. As such, privacy functions are annotated with *privacy*—rather than *sensitivity*—effects. The type of a function from τ_1 to τ_2 that is (ϵ,δ)-differentially private in its argument is as follows:

$$(x : \tau_1 \cdot d) \xrightarrow{(\epsilon,\ \delta)x} \tau_2.$$

Semantically, this type describes functions f, where if $\mathfrak{D}_{\tau_1}(x, x') \leq d$, then $f(x)$ and $f(x')$ yield distributions that are "(ϵ,δ)-close" according to the definition of (ϵ, δ)-differential privacy.

The annotation d is necessary to support (and unique to) advanced variants of differential privacy like (ϵ, δ)-differential privacy. In the pure ϵ-differential privacy framework, it is common to first establish the property for $d = 1$, that is, if $\mathfrak{D}_{\tau_1}(x, x') \leq 1$ then $f(x)$ and $f(x')$ are ϵ-close. Once established, this property then implies that if $\mathfrak{D}_{\tau_1}(x, x') \leq d$, then $f(x)$ and $f(x')$ are $d\epsilon$-close, for any d. However, this linear scaling does *not* carry over to advanced variants like (ϵ, δ)-differential privacy. As a consequence, d must be specified directly as a parameter and cannot be recovered by scaling the property instantiated to $d = 1$. We refer to this distance—d—as the *relational distance*, since it pertains to the (two-run) relational property of differential privacy, and specifically, the distance between inputs x and x' for each of the two executions $f(x)$ and $f(x')$. We also use this terminology in the context of sensitivity, e.g., an s-sensitive function is one that upon inputs within relational distance d returns outputs within relational distance sd.

As explained in Section 2, differential privacy is usually achieved by the use of mechanisms such as the Laplace (for ϵ-differential privacy) or the Gaussian mechanism (for (ϵ, δ)-differential privacy). In Jazz, the primitive function implementing the Laplace mechanism has the following type:

$$\mathrm{laplace} : \forall(\hat{d} : \mathbb{R})\ (\hat{\epsilon} : \mathbb{R}).\ (d : \mathbb{R}[\hat{d}]) \to (\epsilon : \mathbb{R}[\hat{\epsilon}]) \to (x : \mathbb{R} \cdot \hat{d}) \xrightarrow{\infty(d+\epsilon)+(\hat{\epsilon},0)x} \mathbb{R}.$$

There are three logical parameters to the laplace function: d is the relational distance (explained above) used in the statement of privacy satisfied by the function, ϵ is the privacy level we want to enforce, and x is the value we (want to protect and) are adding noise to. When *executing* laplace, the amount of noise added is $\mathrm{Lap}(\frac{d}{\epsilon})$, which depends on both d and ϵ, so they must be *runtime values*. Also, when *typechecking* laplace, the amount of privacy obtained depends on ϵ, and the distance d must also be tracked to enforce that the computation feeding laplace with its argument x produces values within relational distance no larger than d. Because the values of d and ϵ are required for both runtime execution and typechecking, we require a form of dependent types.

To support dependent types, we use a *singletons* approach—a technique initially developed for dependently typed programming in Haskell [33, 39] and which we borrow directly from DFuzz in the context of supporting parameterized differentially private functions [34]. In this approach, each dependent argument has two representations—one for the type and term level, respectively. In the type of laplace, $\hat{d}$ is the type-level representation of term-level variable d, and likewise for $\hat{\epsilon}$ and ϵ. (We further discuss singletons and their implementation in Section 8.)

The final argument $x : \mathbb{R}{\cdot}\hat{d}$ will have Laplace noise added to it and then returned as the result of the laplace function. The annotation "$\cdot\hat{d}$" in the type of x places a *precondition* on the *computation* used to supply the value to protect: Its output must have relational distance no larger than $\hat{d}$. After all, the noise added is only guaranteed to give ϵ-differential privacy for values that result from computations with relational distance $\hat{d}$.

The final privacy effect for the function is $\infty(d+\epsilon)+(\hat{\epsilon},0)x$, indicating that no privacy promises are made for the values d and ϵ and that privacy is promised for input x with cost $(\hat{\epsilon},0)$; we write $\infty(d+\epsilon)$ as shorthand for $\infty d + \infty\epsilon$.

Like Fuzz, DFuzz, and Duet, Jazz extends the notion of differential privacy from single-argument to multi-argument functions, assigning each argument a privacy cost (e.g., the privacy effect $\infty(d+\epsilon)+(\hat{\epsilon},0)x$ for the laplace function describes privacy costs for d, ϵ, and x). This approach is formalized in Section 7.3. By convention, most differentially private programs expect a single input to contain the sensitive data, and the privacy cost assigned to this argument is most important in ensuring privacy. The costs associated with the other arguments are typically infinite, indicating that the program does not preserve privacy for these inputs.

We can give a similar type to the gauss function, which provides (ϵ,δ)-differential privacy by adding Gaussian noise drawn from $\mathcal{N}(\frac{2d^2\ln(1.25/\delta)}{\epsilon^2})$:

$$\text{gauss} : \forall(\hat{d} : \mathbb{R})\ (\hat{\epsilon} : \mathbb{R})\ (\hat{\delta} : \mathbb{R}).\ (d : \mathbb{R}[\hat{d}]) \to (\epsilon : \mathbb{R}[\hat{\epsilon}]) \to (\delta : \mathbb{R}[\hat{\delta}]) \to (x : \mathbb{R}{\cdot}\hat{d}) \xrightarrow{\infty(d+\epsilon+\delta)+(\hat{\epsilon},\hat{\delta})x} \mathbb{R}.$$

In Jazz, privacy primitives are used in the privacy sublanguage. For example, the following privacy-sublanguage expression partially applies the Gaussian mechanism to values for d, ϵ and δ, resulting in a privacy function that satisfies $(1.5, 10^{-5})$-differential privacy for any input at relational distance 4:

$$\text{gauss } 4\ 1.5\ 10^{-5} : (x : \mathbb{R}{\cdot}4) \xrightarrow{(1.5,\,10^{-5})x} \mathbb{R}.$$

Note that we omit the instantiation of forall-quantified type variables $\hat{d}$, $\hat{\epsilon}$, and $\hat{\delta}$ to type-level constants 4, 1.5, and 10^{-5}, as they can be inferred from the value-level arguments d, ϵ, and δ.

The privacy sublanguage also contains monadic *bind* (notated $\leftarrow$) and *return* constructs for composing differentially private computations. Privacy functions $\lambda^{\mathrm{p}}(x{\cdot}d).\ e$ are created in the sensitivity sublanguage (because function creation is "pure"), although the function body e lives in the privacy sublanguage. The annotation d is the relational distance explained previously for privacy function types. For example, the following function computes two differentially private results and adds them together:

$$\begin{array}{l} \lambda^{\mathrm{p}}(x{\cdot}1).\ r_1 \leftarrow \text{gauss } 1\ 1.5\ 0.001\ x; \\ \qquad\quad r_2 \leftarrow \text{gauss } 2\ 0.5\ 0.001\ (x+x); \\ \qquad\quad \texttt{return}\ (r_1 + r_2) \end{array} \quad : \quad (x : \mathbb{R}{\cdot}1) \xrightarrow{(2.0,0.002)x} \mathbb{R}$$

The *bind* operator encodes the sequential composition property of differential privacy (Section 2), adding up the ϵ and δ values of subcomputations. The *return* operator encodes the post-processing property of differential privacy. The relational distance parameter of 1 is in general inferrable during typechecking; we include it as a visible term-level parameter for presentation purposes.

Table 1. Datatype Abstractions Provided by Systems Based on Linear Types [49]

Datatype	Intro.	Elimination	Distance
Multiplicative product ($\otimes$)	$\langle e_1, e_2\rangle$	$\texttt{let}\ x_1, x_2 = e\ \texttt{in}\ e'$	$\mathfrak{D}_{\tau_1\otimes\tau_2}(\langle e_{11}, e_{12}\rangle, \langle e_{21}, e_{22}\rangle) \triangleq \mathfrak{D}_{\tau_1}(e_{11}, e_{21}) + \mathfrak{D}_{\tau_2}(e_{12}, e_{22})$
Additive product ($\&$)	(e_1, e_2)	(1) $\texttt{fst}\ e$ (2) $\texttt{snd}\ e$	$\mathfrak{D}_{\tau_1\&\tau_2}((e_{11}, e_{12}), (e_{21}, e_{22})) \triangleq \max(\mathfrak{D}_{\tau_1}(e_{11}, e_{21}), \mathfrak{D}_{\tau_2}(e_{12}, e_{22}))$	
Sum ($\oplus$)	(1) $\texttt{inl}\ e$ (2) $\texttt{inr}\ e$	$\texttt{case}\ e\ \texttt{of}$ $\{x_1 \Rightarrow e_1\}$ $\{x_2 \Rightarrow e_2\}$	$\mathfrak{D}_{\tau_1\oplus\tau_2}(\texttt{inl}\ e_{11}, \texttt{inl}\ e_{21}) \triangleq \mathfrak{D}_{\tau_1}(e_{11}, e_{21})$ $\mathfrak{D}_{\tau_1\oplus\tau_2}(\texttt{inr}\ e_{12}, \texttt{inr}\ e_{22}) \triangleq \mathfrak{D}_{\tau_2}(e_{12}, e_{22})$ $\mathfrak{D}_{\tau_1\oplus\tau_2}(\texttt{inl}\ e_{11}, \texttt{inr}\ e_{22}) \triangleq \mathfrak{D}_{\tau_1\oplus\tau_2}(\texttt{inr}\ e_{12}, \texttt{inl}\ e_{21}) \triangleq \infty$

For defining the distance associated to the datatypes (table last column), we assume that $e_{11}, e_{21} : \tau_1$ and $e_{12}, e_{22} : \tau_2$.

Beyond Duet. The privacy sublanguage of Jazz briefly introduced here lifts a number of important limitations of the privacy sublanguage of Duet. We sketch two of these here and postpone further comparison to later sections.

First, to avoid scaling in the privacy sublanguage, Duet requires the arguments to privacy functions to have a maximum sensitivity or relational distance of 1. This limitation makes it impossible to give general types to the gauss and laplace functions as we have just shown in Jazz. As a result, Duet includes a dedicated *type rule* for each basic differential privacy mechanism, where each rule is parametric in the sensitivity or relational distance of the argument. Jazz's addition of relational distance annotation "$\cdot d$" in the types of function arguments eliminates the need for special type rules, and mechanisms can instead be encoded as primitives with an axiomatized type. The primary benefit of this is that the metatheory need not be extended each time a new mechanism is considered.

Second, while pervasive scaling is generally undesirable for privacy costs, some constructs such as advanced composition rely on the ability to scale privacy costs in controlled ways that are supported by theorems specific to that privacy model. Because Duet's privacy language disallows scaling entirely, these constructs are impossible to encode as functions and must also be given special typing rules. The latent privacy effects in Jazz allow constructs like advanced composition to be given regular function types. Overall, the Jazz design makes differential privacy by typing in the presence of higher-order programming possible for advanced differential privacy variants. The following sections dive into these benefits by focusing first on the sensitivity sublanguage (Section 4) and then the privacy sublanguage (Section 6). Sections 5 and 7 develop the metatheory of each respective sublanguage.

4 DESIGN OF JAZZ'S SENSITIVITY TYPE SYSTEM

Jazz builds upon prior approaches to encoding differential privacy using linear types. In this section, we first overview some limitations of these approaches related to the *tracking of sensitivities* and then discuss how they can be addressed by Jazz. In this section, we color expressions and metavariables green as they pertain to the sensitivity fragment of Jazz.

4.1 Linear Products and Sums

Existing approaches based on linear types [34, 46, 49] provide elementary datatype abstractions to programmers such as pairs (products) and tagged unions (sums). However, some of the sensitivity analysis they implement for these datatypes can lead to overly imprecise—or even unsound—approximations in some circumstances.

We now briefly overview these datatype abstractions; a summary is provided in Table 1.

Linear products. Because existing systems are based on intuitionistic linear logic, two product types emerge: *multiplicative* products $\otimes$ and *additive* products $\&$. Multiplicative pairs encode two resources, both of which can be used. Additive pairs encode two resources, but in contrast to multiplicative pairs, only one of them can be used at a time—a computation may use either their left or right component, but not both. This constraint is reflected on the management of type environments in their typing rules. Consider, for instance, the multiplicative product $\langle x, x\rangle$ and the additive product (x, x). Fuzz generates the following type derivations for the pairs:

$$\frac{x :_1 \mathbb{R} \vdash x : \mathbb{R} \qquad x :_1 \mathbb{R} \vdash x : \mathbb{R}}{x :_2 \mathbb{R} \vdash \langle x, x\rangle : \mathbb{R} \otimes \mathbb{R}}\ \otimes\mathrm{I} \qquad \frac{x :_1 \mathbb{R} \vdash x : \mathbb{R} \qquad x :_1 \mathbb{R} \vdash x : \mathbb{R}}{x :_1 \mathbb{R} \vdash (x, x) : \mathbb{R} \,\&\, \mathbb{R}}\ \&\mathrm{I},$$

where judgment $x :_s \tau \vdash e : \tau'$ denotes that expression e is an s-sensitive computation on x (and has type τ' assuming that x has type τ). The type derivation on the left (for multiplicative products) adds (variablewise) the environments of both components ($x :_2 \mathbb{R} = x :_1 \mathbb{R} + x :_1 \mathbb{R}$), reporting a sensitivity of 2 in x. However, the type derivation on the right (for additive products) calculates the maximum (variablewise) between the environments of both components ($x :_1 \mathbb{R} = max(x :_1 \mathbb{R}, x :_1 \mathbb{R})$), reporting a sensitivity of 1 in x. The elimination rules also follow these principles: While a multiplicative pair is destructed via pattern matching giving access to both its components, an additive product is destructed via projection operators that give access to a single component.

When applied to sensitivity analysis, these type connectives no longer encode *accessibility* of a pair of resources; rather, they encode an abstraction of the *sensitivities* of each component of the pair. The sensitivity for the whole pair is coarse and either tracks the sum of sensitivities of each component (in the case of multiplicative products) or their maximum (in the case of additive products), as reflected in the last column of Table 1.

Linear sums. Rather than a simultaneous occurrence of resources, sums encode an *alternative* occurrence of resources. Sums are introduced via `inl` and `inr` constructors and destructed via a `case` expression with one branch for each of the constructors.

In the context of sensitivity analysis, the sensitivity of a sum `in(l/r)` e encodes *both* the sensitivities of the contained expression e, as well as the sensitivities for the *direction* of the injection (left or right). For example, `inl` $(x + x)$ is 2-sensitive in x, however, `if` $y \leq 10$ `then inl` x `else inr` x is ∞-sensitive in y because a change in y could change the direction of the injection.

As usual, these systems leverage sums to encode Boolean values, e.g., the Boolean type is encoded as $\mathbb{B} \triangleq$ `unit` $\oplus$ `unit`, where `unit` represents the unit *type*, inhabited by unit *value* `tt`. Under this encoding, an `if–then–else` expression becomes syntactic sugar for a `case` expression. Also note that Boolean values true $\triangleq$ `inl tt` and false $\triangleq$ `inr tt` are at distance ∞ from each other in this encoding. This observation will be particularly relevant in some of the forthcoming examples.

4.2 Limitations of Prior Sensitivity Linear Type Systems

Fuzz [49] is the first work to leverage linear (or affine) types for reasoning about program sensitivity. Since its introduction, other systems based on linear types were developed to address different limitations of Fuzz. These primarily comprise DFuzz [34], which allows value-dependent sensitivities and privacy costs, and Duet [46], which allows advanced variants of differential privacy.

Being based on the same underlying sensitivity analysis, all these systems suffer from common limitations related to the sensitivity tracking for products and sums. Through a series of minimal—yet instructive—examples, we now discuss the limitations we have identified.

Limitations related to linear products. In Fuzz-like systems, each product and sum type introduces an approximation for the sensitivity analysis they underpin. When using pair types, this

approximation forces the programmer to predict how each pair will be used in later parts of the program and select the right one to achieve precision: If only one component of the pair is used, then the additive product will give perfect precision; conversely, if both components of the pair are used with the same sensitivity, then the multiplicative product will give perfect precision. This is limiting for abstraction, e.g., a library author must commit to one product type, and clients of the library may turn out to require the other.

Imprecision issues remain even if functions can be inlined: (1) the optimal product choice may be influenced by the dynamic control flow of the program, which cannot be predicted statically in general; and (2) for multiplicative products in particular, if both components of the pair are used with different sensitivities in the body of the pattern match, then the sensitivity estimation may give imprecise results. To illustrate these limitations, consider the following examples as seen by Fuzz:

Example 4.1 (Dynamic Control). The program below contains a branch on a Boolean variable, which determines the usage pattern of an additive pair: While one branch uses one component of the pair, the other branch uses both.

```
// variant using an additive pair (·, ·)
let p = (2 ∗ x, x) in
if b then 3 ∗ fst p
     else 2 ∗ (fst p + snd p)
```

First, observe that the program is semantically equivalent to $6 * x$, which is 6-sensitive in x. For the sensitivity analysis à la Fuzz, the pair p is assigned 2-sensitivity in x (the max of each side). The `if` rule pessimistically takes the maximum between the sensitivities of each branch. This maximum sensitivity is attained by the `else`-branch and gives $8 = 2 \cdot (\underline{2} + \underline{2})$, where the underlined $\underline{2}$ corresponds to the sensitivity of pair p in variable x.

Now assume that we rewrite the program using a multiplicative—rather than additive—pair:

```
// variant using a multiplicative pair ⟨·, ·⟩
let x1, x2 = ⟨2 ∗ x, x⟩ in
if b then 3 ∗ x1
     else 2 ∗ (x1 + x2).
```

In this case, the pair is considered 3-sensitive (rather than 2-sensitive) in x, an estimate that is obtained by *adding* the sensitivities of its two components, instead of taking their maximum. To obtain the overall program sensitivity, the pair sensitivity is scaled by the maximum sensitivity of the two branches in either component of the pair; this maximum is attained by the `then`-branch and gives 3 (since the `else`-branch has sensitivity 2 in both pair components). Overall, this gives an even worse sensitivity in variable x of $9 = 3 \cdot 3$.

In summary, following Fuzz-like analysis, there is no choice of product connective that yields the precise sensitivity bound in x of 6. □

Example 4.2 (Imprecise Scaling). This example shows how imprecision can arise when components of a pair are scaled before introduction and then in an asymmetric way after elimination. We only show the multiplicative pair variant.

```
let x1, x2 = ⟨2 ∗ x, y⟩ in
x1 + 2 ∗ x2
```

The above program is semantically equivalent to $2{*}x{+}2{*}y$, which is 2-sensitive in x and 2-sensitive in y. However, the type-based analysis yields a sensitivity bound of 4 in x, doubling its actual value. The analysis proceeds roughly as follows: The left component of the pair is 2-sensitive in x, and the right component is 1-sensitive in y. As hinted in the previous example, for multiplicative pairs,

Fuzz-like systems *sum* the sensitivities of each component to yield the sensitivity of the whole, so the resulting pair is 2-sensitive in x and 1-sensitive in y; note that $\langle 2*x+y, 0\rangle$, $\langle 0, 2*x+y\rangle$ or even $\langle x, x+y\rangle$ would also result in the exact same sensitivity analysis. The effect of eliminating the pair via pattern matching is to scale the pair sensitivity by the maximum sensitivity of the body $(x_1 + 2 * x_2)$ in the pattern variables (x_1 and x_2), 2 in this case. The result is a final sensitivity of $4 = 2 \cdot 2$ in x and $2 = 2 \cdot 1$ in y, which is precise for y, but imprecise for x.

If the program is converted to instead use additive pairs, then the sensitivity of the pair construction is 2 in x and 1 in y (the pointwise max from of each side), and the sensitivity of the whole expression is 6 in x and 3 in y—strictly worse than the analysis when using multiplicative pairs.

We could fix this program in Fuzz, just like in the previous example, by rewriting the program to use the scaling operator: either $\texttt{let}\ x_1, x_2 = \langle !2*x, y\rangle\ \texttt{in let}\ x_2' = x_2\ \texttt{in}\ x_1 + 2*x_2$ or $\texttt{let}\ x_1, x_2 = \langle 2 * x, !y\rangle\ \texttt{in let}\ x_2' = x_2\ \texttt{in}\ x_1 + 2 * x_2$. This may be considered as an annotation burden for programmers, because (1) the programmer must know beforehand that the analysis is imprecise (which might be hard for long and complex programs), and (2) the programmer must manually know where to apply scaling to achieve better precision. Also, this process relies on an algorithmic version of the type system of Fuzz, which is not trivial to achieve [27]. Finally, note that scaling in Fuzz is restricted to non-zero sensitivities. This means that a program such as $\texttt{let}\ x_1, x_2 = \langle 2 * x, !y\rangle\ \texttt{in}\ x_1$ would be pessimistically considered to be 2-sensitive in x and 2-sensitive in y, although the program is really 2-sensitive in x and 0-sensitive in y. □

Limitations related to linear sums. In addition to imprecision with the product types, Fuzz-like systems exhibit imprecision with sum types. In these systems, the sensitivity analysis for a sum introduction is straightforward: The sensitivity of $\texttt{in(l/r)}\ e$ is simply the sensitivity of e. The sensitivity analysis for a sum elimination via expression $\texttt{case}\ e\ \texttt{of}\ \{x_1 \Rightarrow e_1\}\{x_2 \Rightarrow e_2\}$ is, however, more involved. First, it computes the sensitivity of e_i in binder x_i for $i = 1, 2$ and retains the greatest, say, r. The sensitivity of the overall case expression in some variable, say, x, is then computed as the sum between (1) the max sensitivity of e_i in x for $i = 1, 2$, and (2) the sensitivity of e in x, scaled by r. This brings both unsound and imprecise estimations.

Example 4.3 (Discontinuous Predicate). An unsound corner case of the above sensitivity analysis arises, for example, for the program:

$$\texttt{if}\ (x \leq 10)\ \texttt{then true else false}$$

The program, which desugars to $\texttt{case}\ (x \leq 10)\ \{x_1 \Rightarrow \text{true}\}\{x_2 \Rightarrow \text{false}\}$, is *semantically* ∞-sensitive in x because changing x by, say, 1, could change the result from true to false, which are infinitely far apart values. Intuitively, we can attribute this to the discontinuity of the program at $x = 10$. As for DFuzz and derivative systems like Duet (which support null sensitivities), they derive a sensitivity of 0 in x. To illustrate this, let us consider the corresponding type derivation in Duet:

$$\frac{x :_\infty \mathbb{R} \vdash x \leq 10 : \mathbb{B} \qquad x :_0 \mathbb{R}, x_1 :_0 \mathbb{R} \vdash \text{true} : \mathbb{B} \qquad x :_0 \mathbb{R}, x_2 :_0 \mathbb{R} \vdash \text{false} : \mathbb{B}}{x :_0 \mathbb{R} \vdash \texttt{case}\ (x \leq 10)\ \{x_1 \Rightarrow \text{true}\}\{x_2 \Rightarrow \text{false}\} : \mathbb{B}}\ \uplus\text{-E}\ .$$

The reported sensitivity environment is $0{\cdot}x :_\infty \mathbb{R} + x :_0 \mathbb{R} = x :_{(0\cdot\infty+0)} \mathbb{R} = x :_0 \mathbb{R}$: The left summand $0{\cdot}\infty$ originates from the fact that branches are 0-sensitive in their binders, and expression $x \leq 10$ is ∞-sensitive in x, and the right summand 0 originates from the fact that both branches are 0-sensitive in x. Since the product operation (for sensitivities) adopted by DFuzz regards $0{\cdot}\infty = 0$, the analysis wrongly infers an overall sensitivity of 0 in x. □

Although DFuzz and derivative systems do not account for this corner case and are, therefore, unsound, this soundness problem is not present in Fuzz, as its type system is constrained to non-null sensitivities (therefore, leaving the program out of its scope). Follow-up work such as Reference [26] and Fuzz$^{\epsilon\delta}$ introduce rules that recover the analysis soundness by interpreting $\infty{\cdot}0 = 0{\cdot}\infty = \infty$ rather than $\infty{\cdot}0 = 0$, but this leads to imprecision elsewhere in the system. For example, with this fix the program $\texttt{let}\ y = x \leq 10\ \texttt{in}\ 1$ reports sensitivity ∞ in x despite the term being equivalent to the constant 1:

$$\frac{x :_{\infty} \mathbb{R} \vdash x \leq 10 : \mathbb{B} \qquad x :_{0} \mathbb{R}, y :_{0} \mathbb{B} \vdash 1 : \mathbb{R}}{x :_{0\cdot\infty} \mathbb{R} \vdash \texttt{let}\ y = x \leq 10\ \texttt{in}\ 1 : \mathbb{R}}\ .$$

A more recent work [28] defines a non-commutative multiplication operator where $0{\cdot}\infty = \infty$ but $\infty{\cdot}0 = 0$. In doing so, it addresses the soundness problem for case expressions, and even though not supporting let-like operations, it could be extended to do so in a precise manner (e.g., $x :_{\infty\cdot 0} \mathbb{R} \vdash \texttt{let}\ y = x \leq 10\ \texttt{in}\ 1 : \mathbb{R}$). This multiplication operator is, however, awkward to manipulate and not amenable to automation due to lack of support for non-commutative ring theories in SMT solvers. Even still, imprecisions continue to arise in this design, as we will see in the forthcoming Example 4.4.

Besides the corner case described above leading to unsound estimates, *imprecise* estimates can also arise when eliminating sums. Imprecision arises because, loosely speaking, the analysis approximates the sensitivity of a sum elimination via a case expression as the maximum sensitivity of its branches. As illustrated by the following example, this analysis can dismiss significant information.

Example 4.4 (Conflated Branches). Consider the following program:

$$\begin{array}{l}\texttt{let}\ a = \texttt{if}\ b\ \texttt{then}\ \texttt{inl}\ (x * x)\ \texttt{else}\ \texttt{inr}\ x\ \texttt{in}\\ \texttt{case}\ a\ \texttt{of}\ \{x_1 \Rightarrow 0\}\{x_2 \Rightarrow x_2\}.\end{array}$$

A sum is created as either the left injection of an expression that is ∞-sensitive in x (since $x * x$ is so) or the right injection of an expression 1-sensitive in x. In Fuzz-like systems, such a sum is conservatively deemed ∞-sensitive in x. The sum is then eliminated with a constant left branch, and a right branch that is 1-sensitive in its binder. The ground truth for the program is that it is 1-sensitive in x, as the left injection ∞-sensitive in x is eliminated to a constant. However, the usual linear typing discipline does not match the sensitivities of each injection with the case-branch that each injection would see, reporting an imprecise final sensitivity of ∞ in x. □

4.3 Latent Contextual Effects for Precise Sensitivity Tracking in Jazz

Jazz adopts a novel approach to sensitivity tracking for product and sum types, which can address the previous limitations without the need to rely on scaling of types. The key insight is to *delay* the tracking of sensitivities whenever possible and to *split* it into two separate analyses: one for each side of the product or sum. Technically, the main idea is to encode latent sensitivity effects at the type-connective level. For instance, for multiplicative pairs Jazz has type $\tau_1 \ {}^{\Sigma_1}\!\otimes^{\Sigma_2}\ \tau_2$, where Σ_1 and Σ_2 denote the latent sensitivity effects of each of the pair components. This is in contrast to the Fuzz type $\tau_1 \otimes \tau_2$, which pays for all of its sensitivity effects upfront when the pair is created.

Precise products. Consider the three related multiplicative pair constructions:

$$e_1 \triangleq \langle 2 * x + y, 0\rangle \qquad e_2 \triangleq \langle 0, 2 * x + y\rangle \qquad e_3 \triangleq \langle x, x + y\rangle.$$

For the purpose of sensitivity analysis, Fuzz is unable to distinguish them, as it derives the very same type judgment for all three, namely

$$x :_2 \mathbb{R}, y :_1 \mathbb{R} \vdash e : \mathbb{R} \otimes \mathbb{R} \qquad \text{for } e \in \{e_1, e_2, e_3\}.$$

The type judgment says that the pairs are 2-sensitive in x and 1-sensitive in y (the subscript annotations in the type environment) but does not say how this sensitivity effect is distributed between the pair components. In other words, Fuzz treats pairs as a whole. In contrast, Jazz can derive three different type judgments, precisely capturing the sensitivity of each pair component:

$$x : \mathbb{R}, y : \mathbb{R} \vdash e_1 : \mathbb{R} \,{}^{2x+y}\!\otimes \mathbb{R} \qquad x : \mathbb{R}, y : \mathbb{R} \vdash e_2 : \mathbb{R} \otimes^{2x+y} \mathbb{R} \qquad x : \mathbb{R}, y : \mathbb{R} \vdash e_3 : \mathbb{R} \,{}^{x}\!\otimes^{x+y} \mathbb{R}.$$

Recall from Section 3.2 that in Jazz, we use linear formulas to denote sensitivity effects and therefore, in, e.g., the first type judgment above, $2x+y$ refers to the sensitivity effect $\Sigma \triangleq \{x \mapsto 2, y \mapsto 1\}$, meaning 2-sensitive in x and 1-sensitive in y. Moreover, we elide null sensitivity effects like $0x+0y$. This fine-grained tracking of the sensitivity of each pair component allows, in turn, deferring the payment of the pair sensitivity effect to the precise point where the pair is used, i.e., eliminated, and therefore paying only for what (and how it) is used. For example, if pair e_3 is used in a context where only its first component is referred, we pay for sensitivity effect x. Fuzz, in contrast, would always pay $2x + y$.

Let us discuss the benefits that this fine-grained tracking brings to Examples 4.1 and 4.2. Consider first the program from Example 4.1, more concretely, the variant with additive pairs. The sensitivity of the `then`-branch is calculated as $6x$ from scaling by 3 the (latent) sensitivity effect $2x$ of the left component of pair p. Likewise, the sensitivity of the `else`-branch is calculated also as $6x$ from scaling by 2 the sum of (latent) sensitivity effects $2x$ and x of the respective left and right component of the pair. As a result, Jazz reports the precise sensitivity of $6x$ for the whole program. An analogous fine-grained tracking for the program from Example 4.2 gives also precise sensitivity $2x + 2y$.

Precise sums. The use of latent sensitivity effects yields tighter sensitivity bounds also for sums. However, the handling of sums impose an additional technical challenge related to the impossibility of delaying sensitivity effects. To illustrate this phenomenon, consider expressions:

$$e_4 \triangleq \texttt{inl}\ (x * x) \qquad e_5 \triangleq \texttt{inr}\ (x * x) \qquad e_6 \triangleq \texttt{if}\ (x \leq 10)\ \texttt{then inl 1 else inr 1}.$$

All three expressions are ∞-sensitive in x. Fuzz sensitivity analysis conflates the three expressions to the same type, and some Fuzz derivative systems with support for 0-sensitivities, such as DFuzz, derive an unsound type (w.r.t. the embodied sensitivity analysis) for e_6: $x :_0 \mathbb{R} \vdash e_6 : \mathbb{R} \oplus \mathbb{R}$.

$$x :_\infty \mathbb{R} \vdash e_4 : \mathbb{R} \oplus \mathbb{R} \qquad x :_\infty \mathbb{R} \vdash e_5 : \mathbb{R} \oplus \mathbb{R} \qquad x :_\infty \mathbb{R} \vdash e_6 : \mathbb{R} \oplus \mathbb{R}$$

Jazz derives instead:

$$x : \mathbb{R} \vdash e_4 : \mathbb{R} \,{}^{\infty x}\!\oplus \mathbb{R} \qquad x : \mathbb{R} \vdash e_5 : \mathbb{R} \oplus^{\infty x} \mathbb{R} \qquad x : \mathbb{R} \vdash e_6 : \mathbb{R} \oplus \mathbb{R}.$$

The types of e_4 and e_5 encode a latent sensitivity effect for each side of the sum. In contrast, the type of e_6 is not able to represent its ∞-sensitivity in x as a latent effect, because x influences which injection is used to create the sum itself, not the value inside the injection. Instead, the effect must be paid for *eagerly* in the so-called *ambient* sensitivity effect (which was elided in previous examples). Therefore, type judgments in Jazz have shape $\Gamma \vdash e : \tau \ ; \Sigma$, where Σ represents the ambient sensitivity effect and Γ is a "traditional" environment, mapping variables to types. Thus, expression e_6 is formally typed as:

$$x : \mathbb{R} \vdash e_6 : \mathbb{R} \oplus \mathbb{R} \ ; \infty x$$

Table 2. Comparison of Sensitivity Type-system: Fuzz and DFuzz-like Type Systems vs. Jazz

Fuzz(*)		DFuzz-like TS(**)		Jazz	
Reported Sensitivity	**Bound Quality**	**Reported Sensitivity**	**Bound Quality**	**Reported Sensitivity**	**Bound Quality**
Example 4.1 (additive)					
$8x$	loose	$8x$	loose	$6x$	tight
Example 4.2 (multiplicative)					
$2x + 2y$	tight	$4x + 2y$	loose (in x)	$2x + 2y$	tight
Example 4.3					
∞x	tight	$0x$	unsound	∞x	tight
Example 4.4					
$\infty x + b$	loose (in x)	$\infty x + 0b$	loose (in x) unsound in b	$x + b$	tight
Example 4.5					
p	tight	p	tight	$2p$ p	loose (latent) tight (prepay)

(*): sensitivities strictly greater than 0, programs transformed using scaling.
(**): sensitivities can be greater or equal to 0, no scaling allowed.

with ambient sensitivity effect ∞x. e_4 is typed as $x : \mathbb{R} \vdash e_4 : \mathbb{R} \,{}^{\infty x}\!\oplus \mathbb{R} \;;\; \varnothing$, i.e., with an empty ambient sensitivity effect, and analogously for e_5.

To showcase the benefits of this design, let us re-examine Example 4.4. In Jazz, the type for a is $\mathbb{R} \,{}^{\infty x}\!\oplus^{x} \mathbb{R}$ with ambient effect b.[4] To compute the sensitivity of the `case`-expression over a, we join—by taking the variable-wise maximum—the ambient effect of a, namely, b, with the "global" sensitivity effect of the second branch, namely, $[b + x/x_2]x_2$—the first branch is dismissed because it has no ambient effect. To compute the purported sensitivity effect of the second branch, we take its ambient effect x_2 and replace every occurrence of the branch binder, also x_2, with the effect $b + x$ of the right component of a, computed as the sum between its ambient effect b and its latent effect x. This yields an overall sensitivity of $b + x = b \sqcup [b + x/x_2]x_2$ for the `case`-expression.

Consider now Example 4.3. The guard $x \leq 10$ of the conditional expression has type $\texttt{unit} \oplus \texttt{unit}$ with ambient effect ∞x. Since the branches are constant and have no ambient effect, they do not contribute to the sensitivity of the conditional. Jazz analysis then concludes that the sensitivity of the conditional reduces to the ambient sensitivity of the guard, namely, ∞x, recovering soundness (and precision).

Jazz recovers soundness and precision for all four examples discussed in Section 4.2, as summarized in Table 2. With this observation, we conclude our motivation for the design of the Jazz sensitivity type system based on latent contextual effects.

Example 4.5 (Prepayment of Effects). We remark that the use of latent contextual effects does not always yield better precision than eager (Fuzz-like) systems. Consider the following program:

$$\texttt{let}\ y_1, y_2 = (\texttt{let}\ x_1, x_2\ = p\ \texttt{in}\ \langle x_1, x_2\rangle)\ \texttt{in}\ y_1 + y_2.$$

Using latent effects, the subexpression $\langle x_1, x_2\rangle$ has type $\mathbb{R} \,{}^{x_1}\!\otimes^{x_2} \mathbb{R}; \varnothing$. Thus, the subexpression $\texttt{let}\ x_1, x_2\ = p\ \texttt{in}\ \langle x_1, x_2\rangle$ has type $\mathbb{R} \,{}^{p}\!\otimes^{p} \mathbb{R}; \varnothing$, i.e., it represents a pure expression where the cost of accessing either of its component is p. The ambient effect of the whole expression is the sum

[4] At first sight, one might think that a is ∞-sensitive in b because a change in b may flip the direction of the returned injection. However, any change on the value of b necessarily results in an infinite variation, since true and false are ∞ far apart. Therefore, the induced variation on the value of a is trivially bounded by ∞, scaled by 1, turning a 1-sensitive in b.

of the cost of accessing the pair ($\varnothing$), plus the cost of accessing y_1 (p), plus the cost of accessing y_2 (p), yielding effect $2p$. In Fuzz, the same program reports sensitivity p, yielding better precision than Jazz. To recover Fuzz's precision, Jazz allows effects of products, sums, and functions to be paid for eagerly by combining contextual and linear effects: Parts of the sensitivity effect of each component of the product can contribute to the ambient effect of the product. For instance, consider environment $\Gamma = x : \mathbb{R}, y : \mathbb{R}$. Jazz can produce the following type derivations for expression $e = \langle x, y \rangle$:

$$\Gamma \vdash e : \mathbb{R}\ {}^{x}\!\otimes^{y} \mathbb{R}; \varnothing \qquad \Gamma \vdash e_2 : \mathbb{R} \otimes^{y} \mathbb{R}; x \qquad \Gamma \vdash e_3 : \mathbb{R}\ {}^{x}\!\otimes \mathbb{R}; y \qquad \Gamma \vdash e_3 : \mathbb{R} \otimes \mathbb{R}; x + y.$$

In the first type derivation, the effect of both components are latent, and thus the ambient effect is empty. In the second (respectively, third) type derivation, the latent effect of the type is the ambient effect of the right (respectively, left) component, and the ambient effect of the product is the ambient effect of the left (respectively, right) component. In the last type derivation, the latent effect of the type is empty, everything is paid upfront, coinciding with Fuzz-like type systems. Going back to the example, if we prepay the effects of the subexpression $\langle x_1, x_2 \rangle$, then the product has type $\mathbb{R}\ {}^{\varnothing}\!\otimes^{\varnothing} \mathbb{R}; x_1 + x_2$. Now the subexpression $\texttt{let}\ x_1, x_2\ = p\ \texttt{in}\ \langle x_1, x_2 \rangle$ has type $\mathbb{R}\ {}^{\varnothing}\!\otimes^{\varnothing} \mathbb{R}; p$, because using multiplicative products, we only pay for p proportional to the maximum sensitivity between x_1 and x_2, i.e., $(1 \sqcup 1)p$. The ambient effect of the whole expression is the sum of the cost of accessing the pair (p), plus the cost of accessing y_1 and y_2 ($\varnothing$), yielding the tight ambient effect p.

As a final remark, note that, contrary to Fuzz, Jazz does not currently support recursive types; such functions are required to be primitives, as illustrated in Section 8.

The following section presents the formal development of latent contextual effects for sensitivity typing and includes a step-by-step type derivation for all four examples.

5 SAX: JAZZ'S SENSITIVITY TYPE SYSTEM, FORMALLY

In this section, we present a core sensitivity sublanguage of Jazz, called Sax, for which we develop the sensitivity metatheory. In particular, we prove the type soundness property known as *sensitivity metric preservation* [49]. The core subset of Jazz that extends Sax with privacy is presented in later sections.

5.1 Syntax and Type System

The Sax type system is technically a type-and-effect system [36]. It supports real numbers, functions, sums, and products. As Sax only deals with ambient effects, all metavariables and keywords are typeset in green.

Syntax. Figure 1 presents the syntax of Sax. Expressions e are mostly standard and include: real number r, addition $e + e$, multiplication $e * e$, comparison $e \leq e$, variable x, sensitivity lambda $\lambda^{s}(x : \tau).\ e$, application $e\ e$, unit value $\texttt{tt}$, sum constructors $\texttt{inl}^{\tau_2}\ e$ and $\texttt{inr}^{\tau_1}\ e$, and the sum destructor $\texttt{case}\ e\ \texttt{of}\ \{x \Rightarrow e\}\ \{x \Rightarrow e\}$.

Sax also supports two linear products types: additive and multiplicative. With additive products, the sensitivity of a pair may be approximated as the *max* of the sensitivities of each side; this sensitivity is paid for every projection. With multiplicative products, the sensitivity of a pair may be approximated as the *sum* of the sensitivities of each side; this sensitivity is paid for every tuple pattern match, scaled by the sensitivities of pattern variables in the body. We write additive product constructions (e, e) and destructions $\texttt{fst}\ e$ and $\texttt{snd}\ e$, and multiplicative product constructions $\langle e, e \rangle$ and destructions $\texttt{let}\ x, x = e\ \texttt{in}\ e$.

Finally, an expression e can be an ascription $e :: \tau$, or a derived expression such as a Boolean b, a conditional $\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$, or a let expression $\texttt{let}\ x = e\ \texttt{in}\ e$. Booleans are encoded as

$$
\begin{array}{lll}
r \in \mathbb{R} & & \\
b \in \mathbb{B} & & \\
x \in \mathrm{var} & & \\
e \in \mathit{sexpr} ::= & r \mid e + e \mid e * e \mid e \leq e & \text{real numbers} \\
& \mid\ x \mid \lambda^{s}(x : \tau).\ e \mid e\ e & \text{variables, functions, applications} \\
& \mid\ \mathtt{tt} & \text{unit} \\
& \mid\ \mathtt{inl}^{\tau_2}\ e \mid \mathtt{inr}^{\tau_1}\ e \mid \mathtt{case}\ e\ \mathtt{of}\ \{x \Rightarrow e\}\ \{x \Rightarrow e\} & \text{sums} \\
& \mid\ (e, e) \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e & \text{add. products} \\
& \mid\ \langle e, e \rangle \mid \mathtt{let}\ x, x = e\ \mathtt{in}\ e & \text{mult. products} \\
& \mid\ e :: \tau & \text{ascription} \\
s \in \mathit{sens} & \triangleq\ \mathbb{R}^{\infty}_{\geq 0} & \text{sensitivities} \\
\Sigma \in \mathit{senv} & \triangleq\ \mathrm{var} \rightharpoonup \mathit{sens} ::= sx + \ldots + sx & \text{sensitivity environments} \\
\tau \in \mathrm{type} & ::= \mathbb{R} \mid \mathtt{unit} \mid (x : \tau) \xrightarrow{\Sigma} \tau & \\
& \mid\ \tau\ {}^{\Sigma}\oplus^{\Sigma}\ \tau \mid \tau\ {}^{\Sigma}\&^{\Sigma}\ \tau \mid \tau\ {}^{\Sigma}\otimes^{\Sigma}\ \tau & \text{types} \\
\Gamma \in \mathrm{tenv} & \triangleq\ \mathrm{var} \rightharpoonup \mathrm{type} ::= \{x : \tau, \ldots, x : \tau\} & \text{type environments}
\end{array}
$$

Fig. 1. Sax: Syntax.

$\mathtt{true} \triangleq \mathtt{inl}\ \mathtt{tt}$, $\mathtt{false} \triangleq \mathtt{inr}\ \mathtt{tt}$, and $\mathbb{B}$ as $\mathtt{unit}^{\varnothing} \oplus^{\varnothing} \mathtt{unit}$, conditionals as $\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \triangleq \mathtt{case}\ e_1\ \mathtt{of}\ \{x \Rightarrow e_2\}\ \{y \Rightarrow e_3\}$, and let expressions as $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \triangleq (\lambda^{s}(x : \tau_1).e_2)\ e_1$.

A sensitivity s is either a non-negative real number or the symbol ∞, which represents an unbounded sensitivity; we notate this set $\mathbb{R}^{\infty}_{\geq 0} \triangleq \mathbb{R}_{\geq 0} \uplus \{\infty\}$. A sensitivity environment Σ is a mapping from variables to their sensitivities. For convenience, we write sensitivity environments as first-order polynomials, e.g., $\Sigma = 1x + 2y$ corresponds to an environment Σ such that $\Sigma(x) = 1$ and $\Sigma(y) = 2$. A type τ is either the real number type $\mathbb{R}$, the Boolean type $\mathbb{B}$, the unit type $\mathtt{unit}$, a function type $(x : \tau) \xrightarrow{\Sigma} \tau$, a sum type $\tau\ {}^{\Sigma}\oplus^{\Sigma}\ \tau$, an additive product type $\tau\ {}^{\Sigma}\&^{\Sigma}\ \tau$, or a multiplicative product type $\tau\ {}^{\Sigma}\otimes^{\Sigma}\ \tau$. The sensitivity environment annotation Σ is called the *latent contextual sensitivity effect* (also called latent effect when clear from the context) and represents a *delayed* effect that emerges when a term of said type is eliminated. The latent effect Σ of a function of type $(x : \tau) \xrightarrow{\Sigma} \tau$ corresponds to the effects of applying the function, i.e., a static approximation of the sensitivity of each variable used in its body. The sensitivity environment Σ_1 (respectively, Σ_2) in $\tau_1\ {}^{\Sigma_1}\oplus^{\Sigma_2}\ \tau_2$ corresponds to the latent effect of the injected value using $\mathtt{inl}$ (respectively, $\mathtt{inr}$). And similarly, Σ_1 and Σ_2 in $\tau_1\ {}^{\Sigma_1}\&^{\Sigma_2}\ \tau_2$ or $\tau_1\ {}^{\Sigma_1}\otimes^{\Sigma_2}\ \tau_2$ correspond to the latent effect of accessing the first and second components of the pair, respectively. Finally, a type environment Γ is, as usual, a mapping from variables to types.

Type system. The Sax type system is presented in Figure 2. The judgment $\Gamma \vdash e : \tau\ ;\ \Sigma$ says that the term e has type τ and ambient sensitivity effect Σ (or ambient effect when clear from the context) under type environment Γ. The ambient effect Σ represents an upper bound (conservative approximation) of the real sensitivity of e after executing the program. The use of a sensitivity environment Σ is different from Duet, where sensitivities are tracked in Γ and presented as a necessary condition to typecheck the expression. In other words, in SaxΣ is used to infer sensitivities, whereas in Duet Γ is used to check sensitivities.

- Rules RLIT and UNIT are standard and report no effect, as no variable is accessed. These two rules present no novelty with respect to Duet.
- Rule VAR is mostly standard; it reports an ambient effect $1x$.

$$\textsc{rlit}\ \frac{}{\Gamma \vdash r : \mathbb{R} \,;\, \varnothing} \qquad \textsc{var}\ \frac{\Gamma\,(x) = \tau}{\Gamma \vdash x : \tau \,;\, x} \qquad \textsc{plus}\ \frac{\Gamma \vdash e_1 : \mathbb{R} \,;\, \Sigma_1 \qquad \Gamma \vdash e_2 : \mathbb{R} \,;\, \Sigma_2}{\Gamma \vdash e_1 + e_2 : \mathbb{R} \,;\, \Sigma_1 + \Sigma_2}$$

$$\textsc{times}\ \frac{\Gamma \vdash e_1 : \mathbb{R} \,;\, \Sigma_1 \qquad \Gamma \vdash e_2 : \mathbb{R} \,;\, \Sigma_2 \qquad e_1 \neq r \qquad e_2 \neq r}{\Gamma \vdash e_1 * e_2 : \mathbb{R} \,;\, \infty(\Sigma_1 + \Sigma_2)} \qquad \textsc{l-scale}\ \frac{\Gamma \vdash e : \mathbb{R} \,;\, \Sigma}{\Gamma \vdash r * e : \mathbb{R} \,;\, r\Sigma} \qquad \textsc{r-scale}\ \frac{\Gamma \vdash e : \mathbb{R} \,;\, \Sigma}{\Gamma \vdash e * r : \mathbb{R} \,;\, r\Sigma}$$

$$\textsc{leq}\ \frac{\Gamma \vdash e_1 : \mathbb{R} \,;\, \Sigma_1 \qquad \Gamma \vdash e_2 : \mathbb{R} \,;\, \Sigma_2}{\Gamma \vdash e_1 \leq e_2 : \mathbb{B} \,;\, \infty(\Sigma_1 + \Sigma_2)} \qquad \textsc{lam}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \,;\, \Sigma + \Sigma'}{\Gamma \vdash \lambda^{s}(x : \tau_1).\ e : (x : \tau_1) \xrightarrow{\Sigma} \tau_2 \,;\, \Sigma'}$$

$$\textsc{app}\ \frac{\Gamma \vdash e_1 : (x : \tau_1) \xrightarrow{\Sigma} \tau_2 \,;\, \Sigma_1 \qquad \Gamma \vdash e_2 : \tau_1 \,;\, \Sigma_2}{\Gamma \vdash e_1\ e_2 : [\Sigma_2/x]\tau_2 \,;\, \Sigma_1 + [\Sigma_2/x]\Sigma} \qquad \textsc{unit}\ \frac{}{\Gamma \vdash \mathtt{tt} : \mathtt{unit} \,;\, \varnothing}$$

$$\textsc{inl}\ \frac{\Gamma \vdash e : \tau_1 \,;\, \Sigma + \Sigma'}{\Gamma \vdash \mathtt{inl}^{\tau_2}\ e : \tau_1\ {}^{\Sigma}\!\oplus^{\varnothing}\ \tau_2 \,;\, \Sigma'} \qquad \textsc{inr}\ \frac{\Gamma \vdash e : \tau_2 \,;\, \Sigma + \Sigma'}{\Gamma \vdash \mathtt{inr}^{\tau_1}\ e : \tau_1\ {}^{\varnothing}\!\oplus^{\Sigma}\ \tau_2 \,;\, \Sigma'}$$

$$\textsc{case}\ \frac{\Gamma \vdash e_1 : \tau_{11}\ {}^{\Sigma_{11}}\!\oplus^{\Sigma_{12}}\ \tau_{12} \,;\, \Sigma_1 \qquad \Gamma, x : \tau_{11} \vdash e_2 : \tau_2 \,;\, \Sigma_2 \qquad \Gamma, y : \tau_{12} \vdash e_3 : \tau_3 \,;\, \Sigma_3}{\begin{array}{l}\Gamma \vdash \mathtt{case}\ e_1\ \mathtt{of}\ \{x \Rightarrow e_2\}\ \{y \Rightarrow e_3\} : \\ \quad [\Sigma_1 + \Sigma_{11}/x]\tau_2 \sqcup [\Sigma_1 + \Sigma_{12}/y]\tau_3 \,;\, \Sigma_1 \sqcup ([\Sigma_1 + \Sigma_{11}/x]\Sigma_2 \sqcup ([\Sigma_1 + \Sigma_{12}/y]\Sigma_3))\end{array}}$$

$$\textsc{pair}\ \frac{\Gamma \vdash e_1 : \tau_1 \,;\, \Sigma_1 + \Sigma_1' \qquad \Gamma \vdash e_2 : \tau_2 \,;\, \Sigma_2 + \Sigma_2'}{\Gamma \vdash\ (e_1, e_2) : \tau_1\ {}^{\Sigma_1}\!\&^{\Sigma_2}\ \tau_2 \,;\, \Sigma_1' \sqcup \Sigma_2'} \qquad \textsc{proj1}\ \frac{\Gamma \vdash e : \tau_1\ {}^{\Sigma_1}\!\&^{\Sigma_2}\ \tau_2 \,;\, \Sigma}{\Gamma \vdash \mathtt{fst}\ e : \tau_1 \,;\, \Sigma + \Sigma_1}$$

$$\textsc{proj2}\ \frac{\Gamma \vdash e : \tau_1\ {}^{\Sigma_1}\!\&^{\Sigma_2}\ \tau_2 \,;\, \Sigma}{\Gamma \vdash \mathtt{snd}\ e : \tau_2 \,;\, \Sigma + \Sigma_2} \qquad \textsc{tup}\ \frac{\Gamma \vdash e_1 : \tau_1 \,;\, \Sigma_1 + \Sigma_1' \qquad \Gamma \vdash e_2 : \tau_2 \,;\, \Sigma_2 + \Sigma_2'}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1\ {}^{\Sigma_1}\!\otimes^{\Sigma_2}\ \tau_2 \,;\, \Sigma_1' + \Sigma_2'}$$

$$\textsc{untup}\ \frac{\Gamma \vdash e_1 : \tau_{11}\ {}^{\Sigma_{11}}\!\otimes^{\Sigma_{12}}\ \tau_{12} \,;\, \Sigma_1 \qquad \Gamma, x_1 : \tau_{11}, x_2 : \tau_{12} \vdash e_2 : \tau_2 \,;\, s_1 x_1 + s_2 x_2 + \Sigma_2}{\Gamma \vdash \mathtt{let}\ x_1, x_2 = e_1\ \mathtt{in}\ e_2 : [\Sigma_1 + \Sigma_{11}/x_1][\Sigma_1 + \Sigma_{12}/x_2]\tau_2 \,;\, (s_1 \sqcup s_2)\Sigma_1 + s_1\Sigma_{11} + s_2\Sigma_{12} + \Sigma_2}$$

$$\textsc{ascr}\ \frac{\Gamma \vdash e : \tau \,;\, \Sigma \qquad \tau <: \tau'}{\Gamma \vdash (e :: \tau') : \tau' \,;\, \Sigma}$$

Fig. 2. Sax: Type system.

For example,

$$\textsc{var}\ \frac{(x : \mathbb{R})(x) = \mathbb{R}}{x : \mathbb{R} \vdash x : \mathbb{R} \,;\, x}\ .$$

- Rule PLUS computes the resulting ambient effect as the addition of the ambient effects of both subterms. To add sensitivity environments, we use the $+$ operator, which is simply defined as the addition of polynomials, e.g., $(1x + 2y) + (3x) = 4x + 2y$.
  For example, in the following type derivation

$$\frac{x : \mathbb{R} \vdash x : \mathbb{R} \,;\, x \qquad x : \mathbb{R} \vdash x : \mathbb{R} \,;\, x}{x : \mathbb{R} \vdash x + x : \mathbb{R} \,;\, 2x}\ \text{PLUS},$$

  we write $2x$ instead of $x + x$.
- Rules TIMES and LEQ are similar to PLUS, but the resulting sensitivity effect is scaled by infinity because (1) the sensitivity of a multiplication when neither side is a constant is unbounded, and (2) the distance between distinct Boolean values is deemed infinite, as explained in Section 4.2. Scaling a sensitivity environment Σ by sensitivity s, written $s\Sigma$, produces a new sensitivity environment in which each sensitivity in Σ is multiplied by s. For multiplication, we assume that $0s = s0 = 0$ for all $s \in \mathbb{R}^{\infty}_{\geq 0}$ and we deem $\infty s = s\infty = \infty$ for $s \neq 0$.
  Rules L-SCALE and R-SCALE address the overapproximation yielded by rule TIMES when one of the factors is a real number. For instance, for program $0.5 * x$ rule L-SCALE reports a (precise) sensitivity of $0.5x$, whereas rules TIMES would report ∞x.
- Rule LAM typechecks sensitivity functions and is novel with respect to DUET. The type of the function is annotated with a latent effect Σ, computed as a subset of the effect of its body. On a fully latent discipline, the whole body effect is left as latent and the ambient effect Σ' of the function is empty. However, full eagerness of effects, as in DUET, is achieved when the latent effect Σ is empty and the full effect of the body is paid upon construction.
  Since the splitting of $\Sigma + \Sigma'$ is non-deterministic, a lambda expression can be given many types, ranging from fully latent to fully eager disciplines. We show this behavior later when explaining rules PAIR and TUP. The implementation addresses this issue through the use of additional type annotations. Without loss of generality, in this article, we assume the fully latent derivation for all lambdas unless stated otherwise. The same applies to other language constructs that exhibit this kind of non-deterministic prepayment of latent effects.
  For example, consider program $\lambda^{s}(x : \mathbb{R}).\ x + x$ and its type derivation:

$$\frac{x : \mathbb{R} \vdash x + x : \mathbb{R} \,;\, 2x}{\varnothing \vdash \lambda^{s}(x : \mathbb{R}).\ x + x : (x : \mathbb{R}) \xrightarrow{2x} \mathbb{R} \,;\, \varnothing}\ \text{LAM}$$

  The ambient effect of the program is empty (values are pure) but its latent effect is $2x$, the ambient effect of its body.
- Rule APP deals with function application. Unlike DUET, as variable x may be free in τ_2 (e.g., τ_2 can be a function type whose latent effect includes x), the resulting type replaces x with the ambient effect Σ_2 of its argument using the sensitivity environment substitution operator defined in Figure 3. For instance, consider type $(x : \mathbb{R}) \xrightarrow{\varnothing} (z : \mathbb{R}) \xrightarrow{2x+y+z} \mathbb{R}$. After application, if $\Sigma_2 = 3y$, then the resulting type would be $[3y/x]((z : \mathbb{R}) \xrightarrow{2x+y+z} \mathbb{R}) = ((z : \mathbb{R}) \xrightarrow{6y+y+z} \mathbb{R}) = ((z : \mathbb{R}) \xrightarrow{7y+y} \mathbb{R})$. The ambient effect of an application is computed as the ambient effect of the function Σ_1 plus its latent effect; but as x is free, we substitute it by Σ_2, e.g., if $\Sigma = \Sigma' + sx$, then $[\Sigma_2/x](\Sigma_1 + \Sigma' + sx) = \Sigma_1 + (\Sigma' + s\Sigma_2)$. This is different from DUET, as where the latent effect of the function Σ is paid when the function is created.

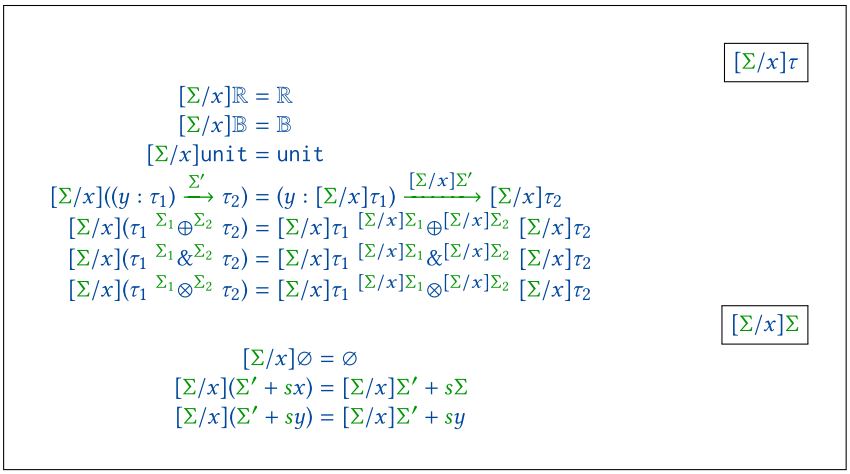

Fig. 3. Sax: Auxiliary definitions of the static semantics (selected rules).

For instance, consider the open program $(\lambda^{s}(x : \mathbb{R}).\ 2 * x + y)\ (3 * y)$ and the following type derivation:

$$\dfrac{y : \mathbb{R} \vdash \lambda^{s}(x : \mathbb{R}).\ 2 * x + y : (x : \mathbb{R}) \xrightarrow{2x+y} \mathbb{R}\ ;\ \varnothing \qquad y : \mathbb{R} \vdash 3 * y : \mathbb{R}; 3y}{y : \mathbb{R} \vdash (\lambda^{s}(x : \mathbb{R}).\ 2 * x + y)\ (3 * y) : \mathbb{R}\ ;\ 7y}\ \text{APP}\ .$$

The resulting ambient effect cannot depend on x (otherwise it would be free), therefore, it is computed as the substitution of x by the ambient effect of the argument $[3y/x](2x + y) = 7y$.

- Contrary to previous work [34, 49], and in particular Duet, Rule inl does not necessarily report the effect of its body. The payment of effects for the subexpression (or a subset of it) can be *delayed* and eventually paid only if the sum is accessed or used. The term is tagged with type τ_2 to aid type inference. The resulting type is just a sum type where the latent effect of the left type is Σ, a subset of the ambient effect of its subterm, and the latent effect of the right type is empty (as it will never be used/accessed, so we choose the tighter ambient effect). Non-determinism is addressed similarly to rule lam. Rule inr is defined similarly.
  For instance, consider the type derivations of expressions $e_4 = \texttt{inl}\ (x * x)$ and $e_5 = \texttt{inr}\ (x * x)$ of Section 4.3:

$$\dfrac{x : \mathbb{R} \vdash (x * x) : \mathbb{R}\ ;\ \infty x}{\varnothing \vdash \texttt{inl}^{\mathbb{R}}\ (x * x) : \mathbb{R}\ {}^{\infty x}\oplus^{\varnothing}\ \mathbb{R}\ ;\ \varnothing}\ \text{INL} \qquad \dfrac{x : \mathbb{R} \vdash x * x : \mathbb{R}\ ;\ \infty x}{\varnothing \vdash \texttt{inr}^{\mathbb{R}}\ (x * x) : \mathbb{R}\ {}^{\varnothing}\oplus^{\infty x}\ \mathbb{R}\ ;\ \varnothing}\ \text{INR}\ .$$

  For expression e_4, the latent effect of the left type is ∞x and of the right type is empty (it is the tighter upper bound, as the right component cannot be accessed). An analogous argument is used for e_5.
- Rule case is more involved. The resulting type of the case is just the least upper bound (join) of the branch types τ_2 and τ_3. The join operator is defined in Figure 4. Note that similarly to rule app, τ_2 and τ_3 may have x and y as free variables, respectively, thus, we replace those variables with the ambient effects of using the sum term e_1: $\Sigma_1 + \Sigma_{11}$ and $\Sigma_1 + \Sigma_{12}$, respectively. The resulting ambient effect is computed as follows: We join the cost of reducing e_1: Σ_1, with the join of the cost of taking each branch. This is different from Duet, where Σ_1 is **added** to the cost of taking each branch, leading to a looser bound. Similarly to types τ_2 and τ_3, ambient effects Σ_2 and Σ_3

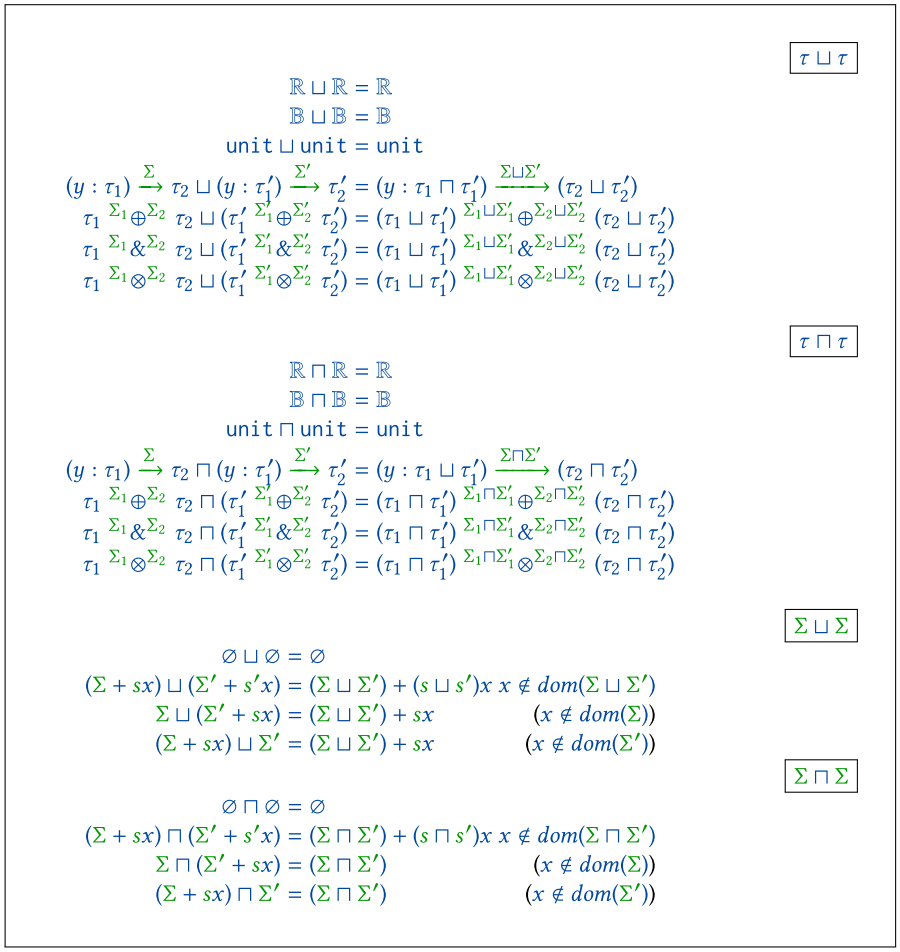

Fig. 4. Sax: Join and Meet of types and sensitivity environments.

may have x and y free, so we substitute them away from the effects. Note that we use the join between Σ_1 and the cost of the branches (instead of the addition for instance), otherwise the result would be less precise when the branches use x or y.
For instance, the type derivation of Example 4.4 is described below:

$$\frac{\Gamma \vdash e : \mathbb{R} \,{}^{\infty x}\!\oplus^{x}\, \mathbb{R}; b \qquad \Gamma, x_1 : \mathbb{R} \vdash 0 : \mathbb{R}; \varnothing \qquad \Gamma, x_2 : \mathbb{R} \vdash x_2 : \mathbb{R}; x_2}{\Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\ \{x_1 \Rightarrow 0\}\{x_2 \Rightarrow x_2\} : \mathbb{R}; b \sqcup (0b + 0(\infty x)) \sqcup (1b + 1(x))}\ \textsc{case}\ ,$$

where $\Gamma = b : \mathbb{B}, x : \mathbb{R}$, and $e = \mathtt{if}\ b\ \mathtt{then}\ \{\mathtt{inl}\ (x * x)\}\ \mathtt{else}\ \{\mathtt{inr}\ x\}$. As $0\infty = 0$, the resulting ambient effect is $b \sqcup (0b + 0(\infty x)) \sqcup (1b + 1(x)) = b \sqcup (0b + 0x) \sqcup (b + x) = b + x$, where previous work reported ∞ on x.
Notice that if we change the program to $\mathtt{case}\ e\ \mathtt{of}\ \{x_1 \Rightarrow 0\}\{x_2 \Rightarrow 1\}$, then the resulting ambient effect is $b \sqcup (0b + 0(\infty x)) \sqcup (0b + 0(x)) = b$, i.e., the payment is not zero but b, the cost of reducing expression e to a value.

Example 4.3 is desugared and typechecked as follows:

$$\textsc{case}\ \frac{\Gamma \vdash x \leq 10 : \mathbb{B}; \infty x \qquad \Gamma, x_1 : \mathbb{R} \vdash \mathsf{true} : \mathbb{R}; \varnothing \qquad \Gamma, x_2 : \mathbb{R} \vdash \mathsf{false} : \mathbb{R}; \varnothing}{\Gamma \vdash \mathsf{case}\ x \leq 10\ \mathsf{of}\ \{x_1 \Rightarrow \mathsf{true}\}\{x_2 \Rightarrow \mathsf{false}\} : \mathbb{B}; \infty x \sqcup (0(\infty x) + 0(\varnothing)) \sqcup (0(\infty x) + 0(\varnothing))}\ ,$$

where $\mathbb{B} = \mathsf{unit}\ ^{\varnothing}\oplus^{\varnothing}\ \mathsf{unit}$. As $\infty x \sqcup (0(\infty x) + 0(\varnothing)) \sqcup (0(\infty x) + 0(\varnothing)) = \infty x \sqcup 0x = \infty x$, the expression is ∞-sensitive in x.

- Rules PAIR and TUP are novel and non-deterministic: the ambient effects are computed using subsets of the ambient effect of each component. If the ambient effects of the left component is $\Sigma_1 + \Sigma_1'$ and of the right component is $\Sigma_2 + \Sigma_2'$ (for some $\Sigma_1, \Sigma_1', \Sigma_2, \Sigma_2'$), then the latent effects of using the left component is Σ_1, and for the right component is Σ_2. For additive products, the ambient effect is the maximum between Σ_1' and Σ_2', and for multiplicative products, the sum between Σ_1' and Σ_2'.
  For instance, let us consider examples $e_1 = (2 * x + y, 0)$, $e_2 = (0, 2 * x + y)$, and $e_3 = (x, x + y)$ from Section 4.3. We present next "lazy" type derivations for each of the examples:

$$\textsc{pair}\ \frac{\Gamma \vdash 2 * x + y : \mathbb{R}\ ;\ 2x + y \qquad \Gamma \vdash 0 : \mathbb{R}\ ;\ \varnothing}{\Gamma \vdash (2 * x + y, 0) : \mathbb{R}\ ^{2x+y}\&^{\varnothing}\ \mathbb{R}\ ;\ \varnothing} \qquad \textsc{pair}\ \frac{\Gamma \vdash 0 : \mathbb{R}\ ;\ \varnothing \qquad \Gamma \vdash 2 * x + y : \mathbb{R}\ ;\ 2x + y}{\Gamma \vdash (0, 2 * x + y) : \mathbb{R}\ ^{\varnothing}\&^{2x+y}\ \mathbb{R}\ ;\ \varnothing}$$

$$\textsc{pair}\ \frac{\Gamma \vdash x : \mathbb{R}\ ;\ x \qquad \Gamma \vdash x + y : \mathbb{R}\ ;\ x + y}{\Gamma \vdash (x, 2 * x + y) : \mathbb{R}\ ^{x}\&^{x+y}\ \mathbb{R}\ ;\ \varnothing}$$

where $\Gamma = x : \mathbb{R}, y : \mathbb{R}$.
Now, let us consider examples $(2x, x)$ and $\langle 2x, x \rangle$. Here are six possible type derivations of paying eagerly for effects:

$$\textsc{pair}\ \frac{\Gamma \vdash 2x : \mathbb{R}\ ;\ \varnothing + 2x \qquad \Gamma \vdash x : \mathbb{R}\ ;\ x + \varnothing}{\Gamma \vdash (2x, x) : \mathbb{R}\ \&^{x}\ \mathbb{R}\ ;\ 2x} \qquad \textsc{tup}\ \frac{\Gamma \vdash 2x : \mathbb{R}\ ;\ \varnothing + 2x \qquad \Gamma \vdash x : \mathbb{R}\ ;\ x + \varnothing}{\Gamma \vdash \langle 2x, x \rangle : \mathbb{R}\ \otimes^{x}\ \mathbb{R}\ ;\ 2x}$$

$$\textsc{pair}\ \frac{\Gamma \vdash 2x : \mathbb{R}\ ;\ x + x \qquad \Gamma \vdash x : \mathbb{R}\ ;\ x + \varnothing}{\Gamma \vdash (2x, x) : \mathbb{R}\ ^{x}\&^{x}\ \mathbb{R}\ ;\ x} \qquad \textsc{tup}\ \frac{\Gamma \vdash 2x : \mathbb{R}\ ;\ x + x \qquad \Gamma \vdash x : \mathbb{R}\ ;\ x + \varnothing}{\Gamma \vdash \langle 2x, x \rangle : \mathbb{R}\ ^{x}\otimes^{x}\ \mathbb{R}\ ;\ x}$$

$$\textsc{pair}\ \frac{\Gamma \vdash 2x : \mathbb{R}\ ;\ \varnothing + 2x \qquad \Gamma \vdash x : \mathbb{R}\ ;\ \varnothing + x}{\Gamma \vdash (2x, x) : \mathbb{R}\ \&\ \mathbb{R}\ ;\ 2x} \qquad \textsc{tup}\ \frac{\Gamma \vdash 2x : \mathbb{R}\ ;\ \varnothing + 2x \qquad \Gamma \vdash x : \mathbb{R}\ ;\ \varnothing + x}{\Gamma \vdash \langle 2x, x \rangle : \mathbb{R} \otimes \mathbb{R}\ ;\ 3x}$$

Note that the difference between the two form of products is only present when effects are paid eagerly for both components.

- Rules PROJ1 and PROJ2 typecheck the deconstruction of an additive product. The ambient effect is computed as the cost of reducing the product (Σ) plus the cost of accessing either the first or the second component correspondingly (Σ_1 or Σ_2). This differs from DUET, where, and conservatively, the cost of accessing both components are paid when the pair is created. In SAX, we only paid for the component we are accessing.

For instance, let us consider the first projections of last examples:

$$\frac{\Gamma \vdash (2 * x + y, 0) : \mathbb{R}\ {}^{2x+y}\&^{\varnothing}\ \mathbb{R}\ ;\ \varnothing}{\Gamma \vdash \mathtt{fst}\ (2 * x + y, 0) : \mathbb{R}\ ;\ 2x + y}\ \textsc{proj1} \qquad \frac{\Gamma \vdash (0, 2 * x + y) : \mathbb{R}\ {}^{\varnothing}\&^{2x+y}\ \mathbb{R}\ ;\ \varnothing}{\Gamma \vdash \mathtt{fst}\ (0, 2 * x + y) : \mathbb{R}\ ;\ \varnothing}\ \textsc{proj1}$$

$$\frac{\Gamma \vdash (x, 2 * x + y) : \mathbb{R}\ {}^{x}\&^{x+y}\ \mathbb{R}\ ;\ \varnothing}{\Gamma \vdash \mathtt{fst}\ (x, 2 * x + y) : \mathbb{R}\ ;\ x}\ \textsc{proj1}\ .$$

Contrary to previous work, the ambient effects of all three projections are different, as they capture precisely the variables accessed on the corresponding component.

- Rule UNTUP typechecks the deconstruction of a multiplicative product and is a little more involved. To compute the ambient effect, we start by paying for Σ_2, the ambient effect of subexpression e_2. We also want to pay Σ_{11} and Σ_{12}, the cost of accessing the left and the right components, respectively, proportionally to the sensitivity of the left and right variables x_1 and x_2 in e_2, i.e., $s_1\Sigma_{11} + s_2\Sigma_{12}$. But, we also have to pay for Σ_1, the ambient effect of e_1. We could pay $(s_1 + s_2)\Sigma_1$, but that would be an unnecessary over-approximation. For instance, program $\mathtt{let}\ x_1, x_2\ =\ p\ \mathtt{in}\ x_1 + x_2$ would pay twice for p (the ambient effect of e_1), even though the whole pair is used only once. Instead, we want to pay proportional to the the maximum sensitivity between x_1 and x_2, i.e., $(s_1 \sqcup s_2)\Sigma_1$. Finally, the ambient effect of the let expression is $(s_1 \sqcup s_2)\Sigma_1 + s_1\Sigma_{11} + s_2\Sigma_{12} + \Sigma_2$.
  For instance, let us consider the typing derivation of Example 4.2:

$$\frac{\Gamma \vdash (2 * x, y) : \mathbb{R}\ {}^{2x}\otimes^{y}\ \mathbb{R}\ ;\ \varnothing \qquad \Gamma, x_1 : \mathbb{R}, x_2 : \mathbb{R} \vdash x_1 + 2 * x_2 : \mathbb{R}\ ;\ x_1 + 2x_2}{\Gamma \vdash \mathtt{let}\ x_1, x_2 = (2 * x, y)\ \mathtt{in}\ x_1 + 2 * x_2 : \mathbb{R}\ ;\ 1(2x) + 2(y)}\ \textsc{untup}\ ,$$

  where $\Gamma = x : \mathbb{R}, y : \mathbb{R}$. The resulting ambient effect is $1(2x) + 2(y) = 2x + 2y$.
  Now consider $\Gamma = p : \mathbb{R}\ {}^{\varnothing}\otimes^{\varnothing}\ \mathbb{R}$, and the following typing derivation:

$$\frac{\dfrac{\Gamma \vdash p : \mathbb{R}\ {}^{\varnothing}\otimes^{\varnothing}\ \mathbb{R}\ ;\ p \qquad \Gamma, x_1 : \mathbb{R}, x_2 : \mathbb{R} \vdash \langle x_1, x_2\rangle : \mathbb{R}\ {}^{\Sigma_1}\otimes^{\Sigma_2}\ \mathbb{R}\ ;\ \Sigma_3}{\Gamma \vdash \mathtt{let}\ x_1, x_2 = p\ \mathtt{in}\ \langle x_1, x_2\rangle : \mathbb{R}\ {}^{\Sigma'_1}\otimes^{\Sigma'_2}\ \mathbb{R}\ ;\ \Sigma'_3}\ \textsc{untup} \qquad \Gamma, y_1 : \mathbb{R}, y_2 : \mathbb{R} \vdash y_1 + y_2 : \mathbb{R}\ ;\ y_1 + y_2}{\Gamma \vdash \mathtt{let}\ y_1, y_2 = (\mathtt{let}\ x_1, x_2 = p\ \mathtt{in}\ \langle x_1, x_2\rangle)\ \mathtt{in}\ y_1 + y_2 : \mathbb{R}\ ;\ \Sigma'_3 + \Sigma'_1 + \Sigma'_2}\ \textsc{untup}\ .$$

  We can typecheck subexpression $\mathtt{let}\ x_1, x_2 = p\ \mathtt{in}\ (x_1, x_2)$ in different ways. If we do not prepay effects, then $\Sigma_1 = x_1, \Sigma_2 = x_2$, and $\Sigma_3 = \varnothing$. Thus, $\Sigma'_1 = \Sigma'_2 = p$, and $\Sigma'_3 = \varnothing$. Finally the ambient effect of the program is $2p$.
  If we prepay the accesses of x_1 and x_2, then $\Sigma_1 = \varnothing, \Sigma_2 = \varnothing$, and $\Sigma_2 = x_1 + x_2$. Thus, $\Sigma'_1 = \Sigma'_2 = \varnothing$, and $\Sigma'_3 = (1 \sqcup 1)p = p$. Finally the ambient effect of the program is p.
- Rules for Booleans, conditionals, and let expressions are derived rules from sums, case, and application rules, respectively, and can be found in Figure 5.
- Finally, Rule ASCR is the only rule that supports the use of subtyping and takes the role of *checking* whether the subexpression is subtype of a given type.

Subtyping for types and sensitivity environments is presented in Figure 6 and is mostly standard. We only allow subtyping for the sensitivity parts of types. A sensitivity environment is subtype of another if their sensitivities are less than or equal to the other for each variable. For instance, $(x : \mathbb{R}) \xrightarrow{x+y} \mathbb{R} <: (x : \mathbb{R}) \xrightarrow{x+2y+3z} \mathbb{R}$, because $x + y <: x + 2y + 3z$ ($x \le x$, $y \le 2y$, and $0z \le 3z$).

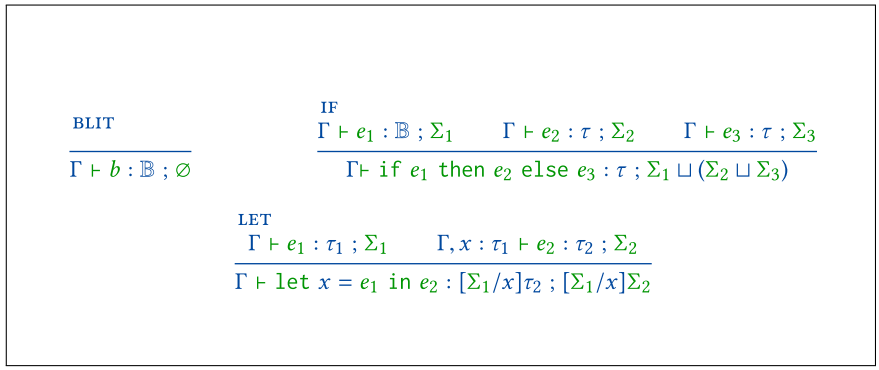

Fig. 5. Sax: Derived type rules.

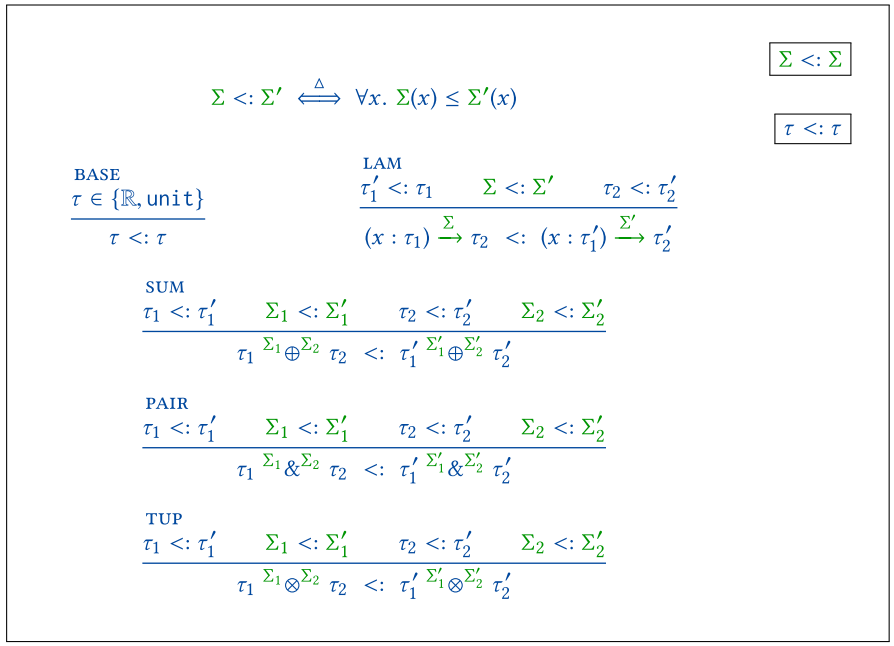

Fig. 6. Sax: Subtyping.

5.2 Type Safety

Type safety is established relative to the runtime semantics of Sax. We adopt a big-step semantics with explicit substitutions. Concretely, we use $\gamma \vdash e \Downarrow v$ to represent that *configuration* $\gamma \vdash e$ —formed by expression e and value environment γ mapping variables to values— reduces to value v after some number of steps. Reduction rules are rather standard and can be found in Appendix B, Figure 27.

To establish Sax type safety, we employ simple unary logical relations, called the *type safety logical relations*, that characterize well-typed, non-stuck execution. This relation is well defined, because it is defined by induction over the structure of types. The type safety result itself is derived as a corollary of the fundamental property of the type safety logical relations.

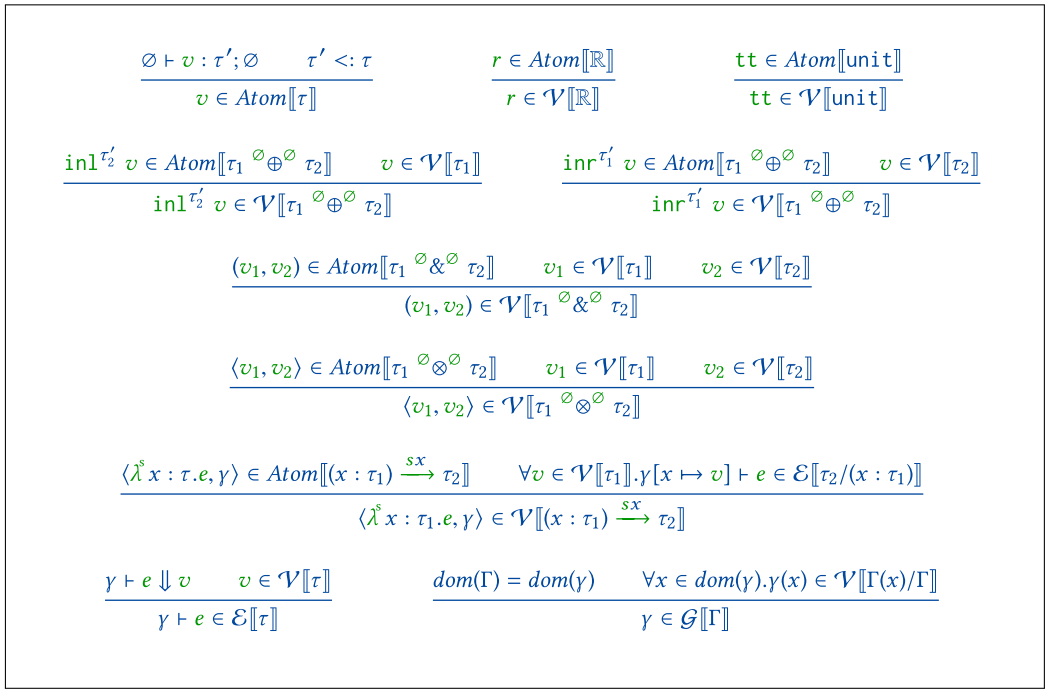

Fig. 7. Sax: Type safety logical relations.

The type safety logical relations is defined in Figure 7. For simplicity, we only present the cases for real numbers, variables, functions, and sums. The other cases are similar and straightforward. The unary logical relations are split into mutually recursive value relations $\mathcal{V}$, computation relation $\mathcal{E}$, and environment relation $\mathcal{G}$, and defined as follows:

- Any value is in $Atom[\![\tau]\!]$ if the value typechecks to some $\tau' <: \tau$ under an empty type environment.
- A real number is in the value relation at type $\mathbb{R}$ if the number is in $Atom[\![\mathbb{R}]\!]$.
- Similarly, a unit value $\mathtt{tt}$ is always related at type $\mathtt{unit}$.
- An $\mathtt{inl}$(respectively, $\mathtt{inr}$) value is in the value relation at $\tau_1\ {}^{\varnothing}\oplus^{\varnothing}\ \tau_2$ if the value is in $Atom[\![\tau_1\ {}^{\varnothing}\oplus^{\varnothing}\ \tau_2]\!]$ and the underlying value v is in the value relation at τ_1 (respectively, τ_2).
- A closure is in the value relation at type $(x : \tau_1) \xrightarrow{sx} \tau_2$ if it satisfies $Atom[\![(x : \tau_1) \xrightarrow{sx} \tau_2]\!]$, and given any value v in the value relation at argument type τ_1, the extended configuration $\gamma[x \mapsto v] \vdash e$ is in the computation relation at type $\tau_2/(x : \tau_1)$. We use the $./\Gamma$ operator to remove variables from a type and is defined as follows:

$$\tau/\Gamma = [\varnothing/x_1, ..., \varnothing/x_n]\tau, \forall x_i \in dom(\Gamma).$$

- A configuration is in the computation relation at type τ if the configuration reduces to some value v, which is itself in the value relation at type τ.
- Finally, a value environment γ is in the environment relation at Γ if the domains of γ and Γ are the same, and for each variable in the domain of γ the underlying value $\gamma(x)$ is in the value relation at type $\Gamma(x)/\Gamma$ (we use the $./\Gamma$ operator to emphasize that the type is closed).

As usual, the fundamental property of the type safety logical relation states that well-typed open terms are in the relation closed by an adequate environment γ:

$$
\begin{aligned}
(r_1, r_2) \in \mathcal{V}_d[\![\mathbb{R}]\!] &\overset{\Delta}{\iff} |r_1 - r_2| \le d \\
(v_1, v_2) \in \mathcal{V}_d[\![\mathtt{unit}]\!] &\overset{\Delta}{\iff} v_1 = \mathtt{tt} \wedge v_2 = \mathtt{tt} \\
(\mathtt{inl}\ v_1, \mathtt{inl}\ v_2) \in \mathcal{V}_d[\![\sigma_1\ {}^{d_1}\!\oplus^{d_2}\ \sigma_2]\!] &\overset{\Delta}{\iff} (v_1, v_2) \in \mathcal{V}_{d+d_1}[\![\sigma_1]\!] \\
(\mathtt{inr}\ v_1, \mathtt{inr}\ v_2) \in \mathcal{V}_d[\![\sigma_1\ {}^{d_1}\!\oplus^{d_2}\ \sigma_2]\!] &\overset{\Delta}{\iff} (v_1, v_2) \in \mathcal{V}_{d+d_2}[\![\sigma_2]\!] \\
((v_{11}, v_{12}), (v_{21}, v_{22})) \in \mathcal{V}_d[\![\sigma_1\ {}^{d_1}\!\&^{d_2}\ \sigma_2]\!] &\overset{\Delta}{\iff} \exists d_1', d_2', d = d_1' \sqcup d_2' \wedge \\
&\quad (v_{11}, v_{21}) \in \mathcal{V}_{d_1+d_1'}[\![\sigma_1]\!] \wedge (v_{12}, v_{22}) \in \mathcal{V}_{d_2+d_2'}[\![\sigma_2]\!] \\
(\langle v_{11}, v_{12}\rangle, \langle v_{21}, v_{22}\rangle) \in \mathcal{V}_d[\![\sigma_1\ {}^{d_1}\!\otimes^{d_2}\ \sigma_2]\!] &\overset{\Delta}{\iff} \exists d_1', d_2', d = d_1' + d_2' \wedge \\
&\quad (v_{11}, v_{21}) \in \mathcal{V}_{d_1+d_1'}[\![\sigma_1]\!] \wedge (v_{12}, v_{22}) \in \mathcal{V}_{d_2+d_2'}[\![\sigma_2]\!] \\
(v_1, v_2) \in \mathcal{V}_d[\![(x : \sigma_1) \xrightarrow{\Delta\cdot\Sigma+sx} \sigma_2]\!] &\overset{\Delta}{\iff} \forall d', v_1', v_2', \gamma_1, \gamma_2, (v_1', v_2') \in \mathcal{V}_{d'}[\![\sigma_1]\!] \implies \\
&\quad (\gamma_1 \vdash v_1\ v_1', \gamma_2 \vdash v_2\ v_2') \in \mathcal{E}_{d+\Delta\cdot\Sigma+sd'}[\![d'x(\sigma_2)]\!] \\
(\gamma_1 \vdash e_1, \gamma_2 \vdash e_2) \in \mathcal{E}_d[\![\sigma]\!] &\overset{\Delta}{\iff} \forall v_1, (d < \infty \wedge \gamma_1 \vdash e_1 \Downarrow v_1) \implies \\
&\quad \exists v_2, \gamma_2 \vdash e_2 \Downarrow v_2 \wedge (v_1, v_2) \in \mathcal{V}_d[\![\sigma]\!] \\
(\gamma_1, \gamma_2) \in \mathcal{G}_\Delta[\![\Gamma]\!] &\overset{\Delta}{\iff} dom(\gamma_1) = dom(\gamma_2) = dom(\Gamma) \wedge \\
&\quad \forall x \in dom(\Gamma).\ (\gamma_1(x), \gamma_2(x)) \in \mathcal{V}_{\Delta(x)}[\![\Delta(\Gamma(x))]\!]
\end{aligned}
$$

Fig. 8. Sax: Logical relations for metric preservation.

Proposition 5.1 (Fundamental Property of the Type Safety Logical Relation). *Let* $\Gamma \vdash e : \tau; \Sigma$, *and* $\gamma \in \mathcal{G}[\![\Gamma]\!]$. *Then* $\gamma \vdash e \in \mathcal{E}[\![\tau/\Gamma]\!]$.

Type safety for closed terms follows immediately as a corollary:

Corollary 5.1 (Type Safety and Normalization of Sax). *Let* $\vdash e : \tau; \varnothing$, *then* $\vdash e \Downarrow v$ *for some* v *and* τ', *such that* $\vdash v : \tau'; \varnothing$ *and* $\tau' <: \tau$.

5.3 Type Soundness

This section establishes the *type soundness* of Sax, stated in terms of a *metric preservation* result. Loosely speaking, metric preservation captures the maximum variation of an open term when it is closed under two different (but related) environments.

Logical relations. To establish this soundness result, we make use of logical relations [4, 6]. In particular, we define (mutually recursive) logical relations for sensitivity values, computations, and environments; see Figure 8. The logical relations for values ($\mathcal{V}_d[\![\sigma]\!]$) and computations ($\mathcal{E}_d[\![\sigma]\!]$) are indexed by a relational distance $d \in \mathbb{R}^\infty_{\ge 0}$ and a so-called *relational distance type* σ, which is a regular type where sensitivity environments are enriched with a constant $d \in \mathbb{R}^\infty_{\ge 0}$ denoting the distance induced by pair of substitutions. Formally, the syntax of relational distance types is defined as follows:

$$\sigma \ = \ \mathbb{R} \mid \mathbb{B} \mid \mathtt{unit} \mid (x : \sigma) \xrightarrow{\Sigma+d} \sigma \mid \sigma\ {}^{\Sigma+d}\!\oplus^{\Sigma+d}\ \sigma \mid \sigma\ {}^{\Sigma+d}\!\&^{\Sigma+d}\ \sigma \mid \sigma\ {}^{\Sigma+d}\!\otimes^{\Sigma+d}\ \sigma.$$

Notice that the logical relations do not mention sensitivity environments Σ, because they are defined over closed terms and values. Nevertheless, relational distance types σ do mention sensitivity environments Σ. We use a combination of sensitivity environments and relational distances ($\Sigma + d$), because function types introduce binders that cannot be substituted until application. For instance, consider type $(x : \mathbb{R}) \xrightarrow{x+y} (\mathbb{R}\ {}^{x+3z}\!\oplus^{2x+2y}\ \mathbb{R})$ and two pair of substitutions for y and z, at distance 2 and 1, respectively, e.g., $\gamma_1 = y \mapsto 1, z \mapsto 1$ and $\gamma_2 = y \mapsto 3, z \mapsto 2$, where $|\gamma_1(y) - \gamma_2(y)| \le 2$ and $|\gamma_1(z) - \gamma_2(z)| \le 1$. The corresponding relational distance type

after substitution is $(x : \mathbb{R}) \xrightarrow{x+2} (\mathbb{R} \ {}^{x+3}\!\oplus^{2x+4} \ \mathbb{R})$. For notation simplicity, in the rest of the section, we name relational distance types as types when the accompanying relational distances can be inferred from the context. Also, we omit the environment notations when they are empty. However, the logical relation for environments ($\mathcal{G}_\Delta[\![\Gamma]\!]$) is indexed by a *relational distance environment* Δ, mapping variables to relational distances in $\mathbb{R}^\infty_{\geq 0}$ and a type environment Γ. We use $(v_1, v_2) \in \mathcal{V}_d[\![\sigma]\!]$ to denote that value v_1 is related to value v_2 at type σ and relational distance d and likewise for expressions (i.e., computations) and environments.

To define logical relations, we also make use of *relational distance instantiations*, which have shape $\Delta{\cdot}(\Sigma + d)$ and act by replacing free variables in sensitivity environment Σ with the distances provided by distance environment Δ. Relational distance instantiations only close variables defined in Δ and are formally defined as:

$$\begin{aligned}
\Delta{\cdot}\varnothing &= 0 \\
\Delta{\cdot}(\Sigma + d) &= \Delta{\cdot}\Sigma + d \\
\Delta{\cdot}(\Sigma + sx) &= \Delta{\cdot}\Sigma + sd \quad \text{if } \Delta(x) = d \\
\Delta{\cdot}(\Sigma + sx) &= \Delta{\cdot}\Sigma + sx \quad \text{otherwise.}
\end{aligned}$$

Furthermore, to close a type under a sensitivity environment, we use the relational distance type instantiation operator $\Delta(\sigma)$ (note that a τ is also an σ assuming that the "default" relational distance d is 0) defined below:

$$\begin{aligned}
\Delta(\mathbb{R}) &= \mathbb{R} \\
\Delta(\mathbb{B}) &= \mathbb{B} \\
\Delta(\texttt{unit}) &= \texttt{unit} \\
\Delta((x : \sigma_1) \xrightarrow{\Sigma'+d} \sigma_2) &= (x : \Delta(\sigma_1)) \xrightarrow{\Delta\cdot\Sigma'+d} \Delta(\sigma_2) \\
\Delta(\sigma_1 \ {}^{\Sigma_1+d_1}\!\oplus^{\Sigma_2+d_2} \ \sigma_2) &= \Delta(\sigma_1) \ {}^{\Delta\cdot\Sigma_1+d_1}\!\oplus^{\Delta\cdot\Sigma_2+d_2} \ \Delta(\sigma_2) \\
\Delta(\sigma_1 \ {}^{\Sigma_1+d_1}\!\&^{\Sigma_2+d_2} \ \sigma_2) &= \Delta(\sigma_1) \ {}^{\Delta\cdot\Sigma_1+d_1}\!\&^{\Delta\cdot\Sigma_2+d_2} \ \Delta(\sigma_2) \\
\Delta(\sigma_1 \ {}^{\Sigma_1+d_1}\!\otimes^{\Sigma_2+d_2} \ \sigma_2) &= \Delta(\sigma_1) \ {}^{\Delta\cdot\Sigma_1+d_1}\!\otimes^{\Delta\cdot\Sigma_2+d_2} \ \Delta(\sigma_2).
\end{aligned}$$

Now that we have all the prerequisites, we briefly go through the definition of the logical relations (in Figure 8)[5]:

- Two real numbers are related at type $\mathbb{R}$ and distance d, if and only if the absolute difference between both numbers is at most d. For instance, $(1, 3) \in \mathcal{V}_2[\![\mathbb{R}]\!]$ and $(3, 1) \in \mathcal{V}_2[\![\mathbb{R}]\!]$, as the logical relations are reflexive.
- Unit value `tt` is always related to itself at type `unit` under any distance.
- Two `inl` (respectively, `inr`) values are related at $\sigma_1 \ {}^{d_1}\!\oplus^{d_2} \ \sigma_2$ and distance d if the underlying values are related at type σ_1 (respectively, σ_2) and distance $d + d_1$ (respectively, $d + d_2$). The intuition is that d can be treated as the distance between two computations that reduce to the given sums, and d_1 can be treated as the distance between the underlying values; thus, the total cost is the addition of both distances. For instance, for any d and σ, we have $(\texttt{inl}\ 1, \texttt{inl}\ 3) \in \mathcal{V}_0[\![\mathbb{R} \ {}^{2}\!\oplus^{d} \ \sigma]\!]$, because they are at immediate distance zero (both are `inl`) and latent distance 2; instead of delaying the distance, one also has $(\texttt{inl}\ 1, \texttt{inl}\ 3) \in \mathcal{V}_2[\![\mathbb{R} \ {}^{0}\!\oplus^{d} \ \sigma]\!]$, i.e., both values are at distance 2 with zero latent distance between their content.
- Two additive (respectively, multiplicative) products are related at type $\sigma_1 \ {}^{d_1}\!\&^{d_2} \ \sigma_2$ (respectively, $\sigma_1 \ {}^{d_1}\!\otimes^{d_2} \ \sigma_2$) and distance $d + d'_1 \sqcup d'_2$ (respectively, $d + d'_1 + d'_2$) if both first components are related at type σ_1 and distance $d + d_1 + d'_1$ and both second components are related at type σ_2 and

[5]For simplicity, we use "distance" instead of "relational distance."

distance $d + d_2 + d_2'$. For instance, $((\texttt{inl}\ 1, 4), (\texttt{inl}\ 3, 5)) \in \mathcal{V}_0[\![(\mathbb{R}\ {}^2\oplus^0\ \sigma)\ {}^0\&^1\ \mathbb{R}]\!]$ are at distance 0 and $(\texttt{inl}\ 1, \texttt{inl}\ 3) \in \mathcal{V}_0[\![\mathbb{R}\ {}^2\oplus^0\ \sigma]\!]$ and $(4, 5) \in \mathcal{V}_1[\![\mathbb{R}]\!]$.

- Two sensitivity closures are related if, given related inputs, they produce related computations. In more detail, first the environments have to be related at some Γ and distance environment Δ. Note that Δ has to be the same environment that closes the latent effect of the function $\Delta{\cdot}\Sigma + s'x$ and the one that closes the input type ($\sigma_1 = \Delta(\tau_1)$). Second, inputs v_1' and v_2' have to be related at argument type σ_1 and any distance d'. Finally, the body of the functions in environments extended with inputs v_1' and v_2' have to be related computations at type $d'x(\sigma_2)$ and distance $d + \Delta{\cdot}\Sigma + sd'$. Note that, as the variable x is out of scope after the application, we replace any instance of x with the distance of the inputs d' using the distance type instantiation operator. The new distance at which both computations are now related is computed as the addition of the distance of the values d and the closed latent effect $d'x(\Delta{\cdot}\Sigma + sx) = \Delta{\cdot}\Sigma + sd'$. For instance, $(\langle \lambda^s x : \mathbb{R}.x + y), y \mapsto 1\rangle, \langle \lambda^s x : \mathbb{R}.x + y), y \mapsto 3\rangle \in \mathcal{V}_0[\![(x : \mathbb{R}) \xrightarrow{2+1x} \mathbb{R}]\!]$, as in this case $\Delta = 2y$, $\Sigma = 1y$, and $\Delta{\cdot}\Sigma = 2y{\cdot}1y = 2$.
- Two sensitivity configurations are related computations at type σ and distance d, noted $(\gamma_1 \vdash e_1, \gamma_2 \vdash e_2) \in \mathcal{E}_d[\![\sigma]\!]$, when the distance is infinite, or if the first configuration reduces to a value, then the second configuration also reduces to a value, and these values are related at type σ and distance d.
- Finally, value environment γ_1 is related to value environment γ_2 at type environment Γ and distance environment Δ, written $(\gamma_1, \gamma_2) \in \mathcal{G}_\Delta[\![\Gamma]\!]$, if they both map each variable x in the type environment to values related at their corresponding type (closed with Δ) and at distance $\Delta(x)$.

Sensitivity Metric Preservation. Armed with these logical relations, we can establish the notion of type soundness and prove the fundamental property—well-typed terms are related with themselves—which corresponds to metric preservation [49]. As usual, we state this property appealing to open terms, where free variables indicate input parameters, which are then closed by related value environments.

Theorem 5.2 (Sensitivity Metric Preservation). *If $\Gamma \vdash e : \tau\ ;\ \Sigma$, then for any distance environment Δ with $dom(\Gamma) \subseteq dom(\Delta)$ and any pair of value environments $(\gamma_1, \gamma_2) \in \mathcal{G}_\Delta[\![\Gamma]\!]$, it holds that $(\gamma_1 \vdash e, \gamma_2 \vdash e) \in \mathcal{E}_{\Delta\cdot\Sigma}[\![\Delta(\tau)]\!]$.*

In other words, if a sensitivity term is well-typed, then for any valid distance environment Δ (that "fits" Γ) and any two value environments γ_1, γ_2 related at Γ and Δ, configurations $\gamma_1 \vdash e, \gamma_2 \vdash e$ represent related computations at type $\Delta(\tau)$ (closing all free variables) and distance $\Delta{\cdot}\Sigma$. Note that, since $dom(\Sigma) \subseteq dom(\Gamma) \subseteq dom(\Delta)$, we have $\Delta{\cdot}\Sigma \in \mathbb{R}^\infty_{\geq 0}$.

From the above theorem it is easy to derive a corollary that only characterizes closed terms:

Corrolary 5.2.1 (FP for Closed Sensitivity Terms). *If $\varnothing \vdash e : \tau\ ;\ \varnothing$, then $(\varnothing \vdash e, \varnothing \vdash e) \in \mathcal{E}_0[\![\tau]\!]$.*

As a direct consequence of Theorem 5.2, we can also establish the sensitivity type soundness at base types:

Theorem 5.3 (Sensitivity Type Soundness at Base Types). *If $\varnothing \vdash e : (x : \mathbb{R}) \xrightarrow{sx} \mathbb{R}\ ;\ \varnothing$, $|r_1 - r_2| \leq d$, $\varnothing \vdash e\ r_1 \Downarrow r_1'$, $\varnothing \vdash e\ r_2 \Downarrow r_2'$, then $|r_1' - r_2'| \leq sd$.*

Let us illustrate metric preservation by revisiting some examples. Consider Example 4.2:

$$x : \mathbb{R}, y : \mathbb{R} \vdash \texttt{let}\ x_1, x_2 = \langle 2 * x, y\rangle\ \texttt{in}\ x_1 + 2 * x_2 : \mathbb{R}; 2x + 2y.$$

If we know that in two different executions x may differ in at most 1, and y in at most 3, i.e., $\Delta = 1x + 3y$, then the result will differ in at most $\Delta{\cdot}(2x + 2y) = 1{\cdot}2 + 3{\cdot}2 = 8$. For instance, if in one

execution x is bound to 0 and y to 4, then the result will be 8. In a second execution, if x is bound to 1 and y to 6, then the result will be 14. Comparing both results, we get $|8 - 14| = 6 \leq 8$. Finally, in a third execution, if x is bound to 1 and y to 7, then the result will be 16. Comparing with the first execution, we have $|8 - 16| = 8 \leq 8$, and with the second $|14 - 16| = 2 \leq 8$.

Now consider Example 4.3:

$$x : \mathbb{R} \vdash \texttt{case}\ (x \leq 10)\ \{x_1 \Rightarrow \text{true}\}\{x_2 \Rightarrow \text{false}\} : \mathbb{B}; \infty x.$$

In this case, if x varies in two different executions $(\Delta(x) > 0)$, then the outcome will differ in at most $\Delta(x) \cdot \infty = \infty$. For instance, if in one execution x is bound to 0 the result will be true, and if in a second execution x is bound to 1, then the result is also going to be true, and true is at distance zero with respect to itself, and $0 \leq \infty$. If in a third execution x is bound to 11, then the result will be false, and false is at distance infinity with respect to true. Now, if we now that x is constant across multiple executions $(\Delta(x) = 0)$, then we know from metric preservation that the result will differ in $0 \cdot \infty = 0$, i.e., the result will be constant.

6 DESIGN OF JAZZ'S PRIVACY TYPE SYSTEM

In this section, we review the limitations of prior approaches related to the tracking of privacy and then discuss how they are addressed by Jazz. In this section, we color expressions and metavariables *red* as they pertain to the privacy fragment of Jazz.

6.1 Privacy Closures

Consider a family of looping combinators parameterized by the number of loop iterations n, e.g., where $\text{loop}_3\ x\ f\ =\ f\ (f\ (f\ x))$. In Fuzz, loop_n would have the type $\tau \to (\tau \to \bigcirc\tau) \multimap_n \bigcirc\tau$. In this type, regular arrows $\to$ mean no sensitivity is tracked for the argument. The linear arrow $\multimap_n$ means the result is n-sensitive (where n is the number of loop iterations) in the *closure variables* of the supplied function of type $(\tau \to \bigcirc\tau)$. This allows for instantiating loop with a closure capturing a sensitive variable, like db. So, $\text{loop}_n\ 0\ (\lambda x.\ x + \texttt{laplace}_\epsilon\ db)$ will give $n\epsilon$ differential privacy for db by scaling ϵ—the privacy cost of closure variable db—by the loop iteration n. When supporting advanced variants of differential privacy like (ϵ, δ), a different metric must be chosen to recover this kind of scaling; otherwise, this argument only holds for pure ϵ-differential privacy.

In Duet, to support (ϵ, δ)-differential privacy (and disallow problematic scaling), privacy closures immediately report unbounded privacy (∞) for any captured variables in privacy lambdas. The *principle* of loop's type above is justified in Duet, but not via a scaling argument, and instead via a primitive type *rule*—it cannot be expressed as a *type*. This is problematic for two reasons: First, it is not possible to extend Duet's implementation with new looping primitives by adding terms with axiomatically justified types, leading to a bloated set of core typing rules, and second, it is not possible to lambda abstract looping combinators, e.g., to chain or compose them in helper functions.

To see the root cause for the limitation in Duet, we show the type rules for looping (advanced composition) and function introduction (from Reference [46]):

Duet: Loop *(Advanced Composition)*

$$\frac{\Gamma_1 \vdash e_1 : \tau \qquad \boxed{\rceil\Gamma_2\lceil^{\epsilon,\delta}}\ , x :_\infty \tau \vdash e_2 : \tau}{\rceil\Gamma_1\lceil^\infty + \boxed{\rceil\Gamma_2\lceil^{2\epsilon\sqrt{2n\ln\frac{1}{\delta'}},\delta' + n\delta}} \vdash \text{loop}_n^{\delta'}\ e_1\ \{x \Rightarrow e_2\} : \tau}$$

Duet: Privacy-Fun-I *(1-ary)*

$$\frac{\boxed{\Gamma}\ , x :_{\epsilon,\delta} \tau_1 \vdash e : \tau_2}{\boxed{\rceil\Gamma\lceil^\infty} \vdash \lambda^{\text{p}}(x : \tau_2).\ e : \tau_1 @ (\epsilon, \delta) \multimap^* \tau_2}\ .$$

In the rule for advanced composition shown above (left), e_1 is the initial value for the looping state of type τ, and e_2 is the loop body that updates the looping state $\tau \to \tau$ and may mention

closure variables in Γ_2. Parameter δ' is a meta-parameter for the advanced composition formula—this parameter is unique to looping in (ϵ, δ)-differential privacy. The notation $\rceil\Gamma_2\lceil^{\epsilon,\,\delta}$ means there must exist some privacy cost ϵ and δ which upper-bounds any individual cost for each of these closure variables. The privacy cost of the whole loop is calculated based on this upper bound for closure variables with the formula $2\epsilon\sqrt{2n\ln(1/\delta')}, \delta' + n\delta$. An attempt to turn `loop` into a primitive (or abstract over `loop`, e.g., eta-expand via lambda abstraction) fails, because privacy types in Duet do not track privacy effects for closure variables; instead, they are just thrown away. In the rule for function introduction shown above (right), the function type $\tau_1@(\epsilon, \delta) \multimap^* \tau_2$ is a probabilistic function from elements in τ_1 to elements in τ_2, which satisfies (ϵ, δ)-differential privacy in its argument. Notice the closure environment Γ above the line that is bumped to ∞ in $\rceil\Gamma\lceil^{\infty}$ below the line. This has the effect of tossing out privacy bounds for anything with non-zero privacy in Γ, i.e., any closure variables that are used in the function's definition. Privacy is only tracked for the function parameter x (or possibly multiple parameters; privacy functions in Duet are n-ary).

A deeper limitation in Duet is that the iterated 1-ary function space does not generalize to support encoding n-ary functions (i.e., currification is not supported). For this reason, n-ary functions are primitive in Duet. Implementing n-ary from 1-ary functions is computationally possible in Duet, but results in discarding bounds on privacy effects. For example, the Duet term $\lambda^{\mathrm{p}}(x : \tau_1).\ \texttt{return}\ (\lambda^{\mathrm{p}}(y : \tau_2).\ f(x, y))$ in a context where $f : (\tau_1@(\epsilon_1, \delta_1), \tau_2@(\epsilon_2, \delta_2)) \multimap^* \tau_3$ has type $(\tau_1@\infty) \multimap^* (\tau_2@(\epsilon_2, \delta_2)) \multimap^* \tau_3$, i.e., the privacy bounds (ϵ_1, δ_1) for the first argument τ_1 get discarded due to the Duet: Privacy-Fun-I rule.

Privacy Closures in Jazz. In Jazz, both privacy and sensitivity effects are delayed and attached to type-level connectives, including for privacy functions. Whereas, in Duet, privacy functions are written $(\tau_1@p_1, \ldots, \tau_n@p_n) \multimap^* \tau$, privacy functions in Jazz are written simply $(x : \tau_1) \xrightarrow{\Sigma} \tau_2$ where Σ is a latent contextual effect that can mention x. A type can now be given to loop (a named constant, analogous to the `loop` *primitive* from Duet) in Jazz, and abstracting over loop is possible due to the function introduction rule, also shown below.

$$\mathrm{loop}_n^{\delta'} : \tau \to (\tau \xrightarrow{\boxed{\rceil\Sigma\lceil^{\epsilon,\delta}}} \tau) \xrightarrow{\boxed{\rceil\Sigma\lceil^{\epsilon',\delta''}}} \tau \quad \textit{where} \quad \epsilon', \delta'' \triangleq 2\epsilon\sqrt{2n\ln(1/\delta')}, \delta' + n\delta$$

Jazz: Privacy-Fun-I

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2\ ;\ \boxed{\Sigma}}{\Gamma \vdash \lambda^{\mathrm{p}}(x : \tau_1 \cdot d).\ e : (x : \tau_1 \cdot d) \xrightarrow{\boxed{\Sigma}} \tau_2\ ;\ \varnothing}$$

N-ary functions are now recoverable from 1-ary ones using latent contextual effects in closures. The relational distance d defaults to 1 when omitted. The encoding of lambda-abstracted gauss then follows the usual approach of nested lambda abstractions, but with sensitivity lambdas on the outside with a single privacy lambda on the inside. A 3-ary abstraction of the Gaussian mechanism applied to the sum of three arguments is as follows:

$$(\lambda^{\mathrm{s}}(x : \mathbb{R}\cdot 1).\ \lambda^{\mathrm{s}}(y : \mathbb{R}\cdot 1).\ \lambda^{\mathrm{p}}(z : \mathbb{R}\cdot 1).\ \mathrm{gauss}\ \underbrace{3}_{\text{sensitivity sum of } (x+y+z)}\ \overbrace{\epsilon\ \delta}^{\text{desired privacy}}\ (x + y + z)) : (x : \mathbb{R}\cdot 1) \to (y : \mathbb{R}\cdot 1) \to (z : \mathbb{R}\cdot 1) \xrightarrow{(\epsilon,\delta)x \sqcup (\epsilon,\delta)y \sqcup (\epsilon,\delta)z} \mathbb{R}.$$

Notice here that the latent contextual effect is computed using a syntactic *join* operator $(\epsilon, \delta)x \sqcup (\epsilon, \delta)y \sqcup (\epsilon, \delta)z$, which computes the pointwise maximum, instead of the sum $((\epsilon, \delta)x + (\epsilon, \delta)y + (\epsilon, \delta)z)$. One of the novelties of Jazz is that we can reason about two executions where more than one input is at relational distance greater than 0. In particular, if x, y and z are at relational distance 1, i.e., the argument of gauss 3 ϵ δ is at relational distance 3, then using addition would yield an over-approximated latent privacy of $(3\epsilon, 3\delta)$, while using the join, we obtain a latent privacy of (ϵ, δ), as desired.

Abstracting Privacy Mechanisms. Even with support for privacy closures, there are still challenges in supporting lambda abstraction around privacy mechanisms in full generality. In Fuzz, the type assigned to the family of Laplace differential privacy mechanisms parameterized by ϵ is $\mathtt{laplace}_\epsilon : \mathbb{R} \multimap_\epsilon \bigcirc\mathbb{R}$ for achieved privacy ϵ. This mechanism does not need a dedicated type rule in the core calculus—it can be axiomatized as a primitive with the right type—and lambda-abstracting this primitive is natural via eta-expansion $\lambda(x : \mathbb{R}).\ \mathtt{laplace}_\epsilon\ x$ resulting in the same type and guarantee for privacy. However, this approach does not support (ϵ, δ)-differential privacy directly. Fuzz$^{\epsilon\delta}$ shows how to extend Fuzz to recover (ϵ, δ)-differential privacy by using graded comonadic liftings and path construction. In particular, the type assigned to the family of Gaussian differential privacy mechanisms parametrized by ε and δ is $\mathtt{gauss}_{(\epsilon,\delta)} : \lceil\mathbb{R}\rceil \multimap \bigcirc_{(\epsilon,\delta)}\mathbb{R}$, where $\lceil\mathbb{R}\rceil$ is $\mathbb{R}$ but with the metric rounded up to the nearest integer. In Duet, to support (ϵ, δ)-differential privacy, the Gaussian mechanism requires its own typing rule, shown below. Furthermore, a use of the mechanism looks like $\mathtt{gauss}^s_{\epsilon,\delta}\ e$, where the argument e is a term in the sensitivity language with sensitivity bounded by s. Using privacy closures as described above, we can write $\lambda^{\mathrm{p}}(x : \mathbb{R}).\ \mathtt{gauss}\ 1\ \epsilon\ \delta\ x$, however, note that we have lost the ability to be parametric in s—it must be fixed to 1. This assumption that gauss will be called only with a 1-sensitive argument is enforced in the function application rule in Duet, also shown below:

$$\textsc{Duet: Gauss}\quad \frac{\rceil\Gamma\lceil^{s} \vdash e : \mathbb{R}}{\rceil\Gamma\lceil^{\epsilon,\delta} \vdash \mathtt{gauss}^s_{\epsilon,\delta}\ e : \mathbb{R}} \qquad \textsc{Duet: Privacy-Fun-E}\quad \frac{\Gamma_1 \vdash e_1 : \tau_1@(\epsilon,\delta) \multimap^* \tau_2 \qquad \rceil\Gamma_2\lceil^{1} \vdash e_2 : \tau_1}{\rceil\Gamma_1\lceil^{\infty} + \rceil\Gamma_2\lceil^{\epsilon,\delta} \vdash e_1\ e_2 : \tau_2}\ .$$

In the rule for gauss (left) it allows an argument of any sensitivity s, however, the privacy function application rule (right) restricts that arguments must have sensitivity equal to 1. Restricting gauss to only 1-distance arguments can be overly restrictive (e.g., $\mathtt{gauss}\ 2\ \epsilon\ \delta\ (x + x)$), and relaxing the restriction on function application to an arbitrary $s \neq 1$ in Duet would be unsound.

In Jazz, we extend function introduction to include an explicit bound on the sensitivity of the parameter and enforce this restriction in the application rule. Function introduction syntax introduces the bound and allows us to eta-expand the Gaussian mechanism with relational distance d as a parameter, as shown below. The bound d for the lambda argument is then enforced in function application as the upper bound of argument relational distance, instead of being fixed to 1 as in Duet. Now the use of a variable—like x in the body of eta-expanded gauss below—is not always considered 1-sensitive. To communicate non-zero sensitivities to variables in the type system, an environment of relational distances on lambda arguments must be threaded through the type system, which we notate Δ. After extending this Δ to remember that x has relational distance d in Jazz lambda abstraction, gauss $s\ \epsilon\ \delta\ x$ will see x as d distant inside the lambda body. To do this, we allow lambda-abstracting gauss (including the distance parameter d via singleton types) and extend the structure of typing for sensitivity and privacy terms respectively as follows:

$$\lambda^{\mathrm{s}}(\boxed{d : \mathbb{R}[\hat{d}]}).\ \lambda^{\mathrm{p}}(x : \mathbb{R}\boxed{\cdot\hat{d}}).\ \mathtt{gauss}\ \boxed{d}\ \epsilon\ \delta\ x \qquad \Gamma\ ;\ \boxed{\Delta}\ \vdash\ e : \tau\ ;\ \Sigma \qquad \Gamma\ ;\ \boxed{\Delta}\ \vdash\ \mathtt{e} : \tau\ ;\ \Sigma.$$

6.2 Sensitivity Binding in Privacy Contexts

Jazz improves on prior systems by supporting let-binding intermediate sensitivity computations within the privacy language while also supporting (ϵ, δ)-differential privacy. Fuzz and DFuzz encode let-binding through function application, which scales the sensitivity of the right-hand side of the let with the sensitivity of the let-variable in the body. So, $\mathtt{let}\ y = 2 * x\ \mathtt{in}\ 3 * y$ is 6-sensitive in x, because the right-hand side is 2-sensitive, and this is scaled by 3, the sensitivity of y in the body. However, monadic return and bind in Fuzz can also be used to encode let-binding, e.g.,

$x \leftarrow \texttt{return}\ e_1\ ;\ e_2$ instead of $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$. Unfortunately, this encoding of `let` using `return` and monadic bind does not preserve typeability in Fuzz; instead, it destroys the sensitivity/privacy analysis of the right-hand side, bumping its privacy cost unnecessarily to ∞. For this reason, let statements are encoded exclusively through function application in Fuzz and not through monadic `return`/bind.

In Fuzz, let-binding a sensitivity computation (the pure fragment) inside a privacy computation (the monadic fragment)—via encoding through function application—is supported seamlessly without the addition of extra rules. This flexibility can be extended to advanced privacy variants as shown by de Amorim et al. [29]. In Duet, however, the privacy/monadic fragment of Fuzz is pulled out into its own language with explicit typing rules; the primary reason to do this is to place restrictions on function application to support advanced privacy variants, as described in the previous subsection. This leaves the need for either an explicit typing rule for let-binding inside the privacy language or an escape hatch so privacy analysis is not destroyed for let-binding in privacy contexts à la Fuzz. Duet solves this issue by introducing a *boxed* type that delays the payment of a sensitivity term at the point it is "boxed" and pays for it later when it is "unboxed." This avoids the issue but is unfriendly to program with: Every let-binding requires an explicit box, and every use of a let-bound variable requires an explicit unbox. So, instead of writing the program below on the left, Duet programmers are forced to write the program on the right.

```
let y = expensive x in                  let y = box (expensive x) in
loop 100 initial (λ^P (i : τ). body y i)  loop 100 initial (λ^P (i : τ). body (unbox y) i)
```

In this program, it is essential to let-bind the expensive result, since inlining it would unnecessarily duplicate the computation, and many real programs in differential privacy require support for this pattern [46].

In Jazz, we recover the expressiveness that box types provide while eliminating the need for the programmer to explicitly introduce and eliminate them. In this way, our design can also be seen as a powerful box-inference capability, although we do not demonstrate explicit embeddings between a core language with box types. To recover the expressiveness of boxes without requiring the programmer to write them down, we add new information to typing judgments that has the effect of automatically boxing let-bound variables in privacy contexts and unboxing them at their use. The added information extends typing judgments with a new component Φ that tracks the sensitivities of all let-bound variables w.r.t. the sensitivities of all lambda-bound variables. All sensitivity contexts that mention both let-bound and lambda-bound variables are then reduced using Φ as needed to contexts that only mention lambda-bound variables. Φ can be seen as a matrix, and the reduction of contexts to only lambda-bound variables is then just matrix multiplication—a beautiful coincidence for a linear type system. The final form of type judgments for the sensitivity and privacy type systems are then:

$$\Gamma\ ;\ \Delta\ ;\ \Phi \vdash e : \tau\ ;\ \Sigma \qquad\qquad \Gamma\ ;\ \Delta\ ;\ \Phi \vdash e : \tau\ ;\ \Sigma.$$

Although the prototype implementation adopts the typing rules with Φ, and because the manipulation of Φ is more tedious than insightful, we omit it in the following technical presentation.

7 JAZZ'S DIFFERENTIAL PRIVACY TYPE SYSTEM, FORMALLY

In this section, we present a core subset of Jazz, dubbed λ_J. λ_J is an extension of Sax with support for reasoning about differential privacy. Similarly to Sax, we prove the type safety and type soundness property of λ_J. We discuss how to bridge the gap between λ_J and Jazz in Section 8. Note that our formalism is fixed to (ϵ, δ)-differential privacy, but our design can be instantiated to other forms of advanced differential privacy disciplines, as illustrated in Section 8.

$$
\begin{array}{lll}
e \in \text{sexpr} & ::= \ldots \mid \lambda^{\text{s}}(x : \tau \cdot d).\ e \mid \lambda^{\text{p}}(x : \tau \cdot d).\ \mathtt{e} & \text{sensitivity expressions} \\
\mathtt{e} \in \text{pexpr} & ::= \mathtt{return}\ e \mid x : \tau \leftarrow \mathtt{e}\ ;\ \mathtt{e} \mid \mathtt{e}\ \mathtt{e} & \text{return, bind, applications} \\
& \mid \mathtt{if}\ e\ \mathtt{then}\ \mathtt{e}\ \mathtt{else}\ \mathtt{e} \mid \mathtt{case}\ e\ \mathtt{of}\ \{x \Rightarrow \mathtt{e}\}\ \{y \Rightarrow \mathtt{e}\} & \text{conditionals, case} \\
& \mid \mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{e} & \text{let} \\
p \in \mathit{priv} & \triangleq (\mathbb{R}^{\infty}, \mathbb{R}^{\infty}) & \text{privacies} \\
\Sigma \in \mathit{penv} & \triangleq \varnothing \mid px \mid \Sigma + \Sigma \mid \Sigma \sqcup \Sigma \mid \Sigma \sqcap \Sigma & \text{privacy environments} \\
\tau \in \text{type} & ::= \ldots \mid (x : \tau \cdot d) \xrightarrow{\Sigma} \tau \mid (x : \tau \cdot d) \xrightarrow{\Sigma}\!\!\!\!\rightarrow\ \tau & \text{types} \\
\Gamma \in \text{tenv} & \triangleq \text{var} \rightharpoonup \text{type} ::= \{x : \tau, \ldots, x : \tau\} & \text{type environments}
\end{array}
$$

Fig. 9. λ_{J}: Syntax.

7.1 Syntax and Type System

λ_{J} is divided in two mutually embedded sublanguages: the *sensitivity* sublanguage—an extension of Sax—used to reason about the sensitivity of computations, and the *privacy* sublanguage, used to reason about differential privacy. Thus, the type system of λ_{J} contains two mutually embedded type systems, one for each of the sublanguages. Expressions of the sensitivity sublanguage remain typeset in green and expressions of the privacy sublanguage are typeset in red.

Syntax. Figure 9 presents the syntax of λ_{J}. Expressions of the language are divided into two mutually embedded expressions: sensitivity expressions e and privacy expressions $\mathtt{e}$. Sensitivity expressions are defined the same way as in Sax, except that functions are split into sensitivity lambdas $\lambda^{\text{s}}(x : \tau \cdot d).\ e$ and privacy lambdas $\lambda^{\text{p}}(x : \tau \cdot d).\ \mathtt{e}$. Note that the only difference between a sensitivity lambda and a privacy lambda is that the body of a privacy lambda is a privacy expression $\mathtt{e}$. Also, both sensitivity lambdas and privacy lambdas are parametrized by a relational distance d that represents an upper bound on distance between inputs pertained to the binary relational property of differential privacy: the maximum argument variation for each of two executions.

A privacy expression $\mathtt{e}$ can be a point distribution $\mathtt{return}\ e$, a sequential composition $x : \tau \leftarrow \mathtt{e}\ ;\ \mathtt{e}$, an application $\mathtt{e}\ \mathtt{e}$, a conditional $\mathtt{if}\ e\ \mathtt{then}\ \mathtt{e}\ \mathtt{else}\ \mathtt{e}$, a case expression $\mathtt{case}\ e\ \mathtt{of}\ \{x \Rightarrow \mathtt{e}\}\ \{y \Rightarrow \mathtt{e}\}$, or a let $\mathtt{let}\ x = e\ \mathtt{in}\ \mathtt{e}$.

A privacy cost $p = (\epsilon, \delta)$ is a pair of two (possibly infinite) real numbers, where the first component corresponds to the epsilon and the second to the delta in (ϵ, δ)-differential privacy. We use notation $p.\epsilon$ and $p.\delta$ to extract ϵ and δ, respectively. A privacy environment Σ is either an empty environment $\varnothing$, a pair px representing that variable x has privacy cost p, the addition $\Sigma + \Sigma$ of two privacy environments, the join $\Sigma \sqcup \Sigma$ of two privacy environments, and the meet $\Sigma \sqcap \Sigma$ of two privacy environments. Similarly to sensitivity environments, we also write privacy environments as first-order polynomials when possible. For instance, $p_1 x + p_2 x$ can be written as $(p_1 + p_2)x$, but $p_1 x + (p_2 x \sqcup p_3 y)$ cannot be rewritten as a polynomial without losing precision. Function types are now divided into sensitivity function types $(x : \tau \cdot d) \xrightarrow{\Sigma} \tau$ and privacy function types $(x : \tau \cdot d) \xrightarrow{\Sigma}\!\!\!\!\rightarrow\ \tau$.

Sensitivity type system. The type system for the sensitivity sublanguage is presented in Figure 10. The judgment $\Gamma\ ;\ \Delta \vdash e : \tau\ ;\ \Sigma$ now includes a novel relational distance environment Δ. The relational distance environment Δ stores how much each variable in Γ can vary in every two executions of a program. Most of the rules are straightforward extensions of the type system of Sax to include relational distance environments. We only present interesting cases.

Some of the rules use the sensitivity environment substitution operator $[\Sigma/x]\tau$. We extend the definition of Sax to support privacy functions, as shown in Figure 11. Substitution on privacy

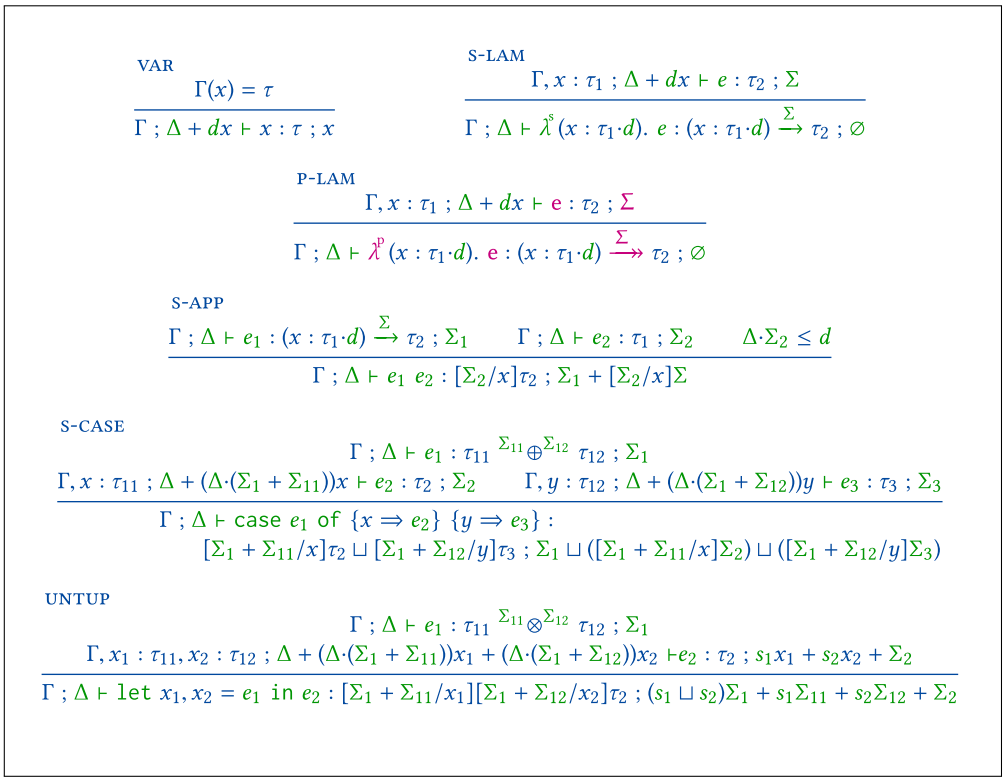

Fig. 10. λ_J: Type system of the sensitivity sublanguage (extract).

function types depends on the definition of sensitivity environment substitution on privacy environments $[\Sigma/x]\Sigma$. $[\Sigma/x]\Sigma$ is defined inductively on the structure of Σ, where the only interesting case is when $\Sigma = px$. Substitution $[\Sigma/x]px$ is defined using the lift operator: $\rceil\Sigma\lceil^{p}$. Intuitively, if we wiggle[6] x on px, then the privacy obtained is at most p (no scaling and zero if x does not change). After substitution, as x depends on all variables on Σ, if we wiggle all variables in Σ at the same time, then the privacy obtained should still be p (scaling p would be an over-approximation). Because of this, $\rceil\Sigma\lceil^{p}$ is defined as the join $px_1 \sqcup \ldots \sqcup px_n$, where $x_i \in dom(\Sigma)$. If all x_i wiggle, then the privacy obtained would be at most p. But as any $\Sigma(x_i)$ can be zero (it means that variable x_i is not used), we multiply each p in px_i, by $\rceil\Sigma(x_i)\lceil^{1}$ to remove those variables from the resulting privacy environment. $\rceil s\lceil^{s'}$ along other lift operators used by Near et al. [46] are defined in Figure 11. For instance, suppose that x depends on $2y + 0z$, then $[(2y + 0z)/x]px$ is computed as $\rceil 2y + 0z\lceil^{p} = py \sqcup 0z = py$.

We now turn to describe the main changes of each type rule with respect to Sax.

- Rule VAR now requires variable x to be present in the relational distance environment Δ. This way, if $\Gamma; \Delta \vdash e : \tau; \Sigma$, we can compute how much the result of evaluating e can change if we wiggle input x by multiplying $\Delta(x)$ by $\Sigma(x)$. For instance, consider program $x + x$ and the following type derivations:

[6]We show how to wiggle variables on privacy environment with the relational distance instantiation operator, later in Section 7.3.

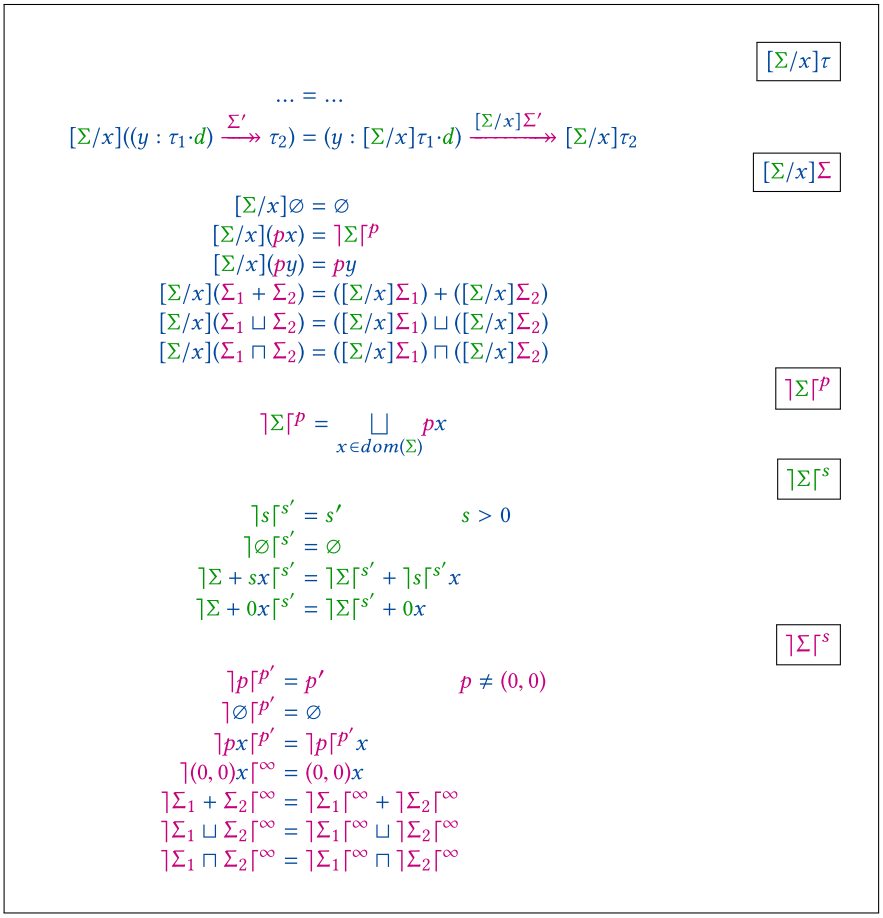

Fig. 11. λ_J: Auxiliary definitions of the static semantics.

$$\mathcal{D} = \dfrac{\dfrac{}{(x:\mathbb{R})(x) = \mathbb{R}}\,\textsc{var}}{x:\mathbb{R}; 3x \vdash x:\mathbb{R}\ ;\ x} \qquad \dfrac{\mathcal{D} \quad \mathcal{D}}{x:\mathbb{R}; 3x \vdash x + x:\mathbb{R}\ ;\ 2x}\,\textsc{plus}.$$

Then, we know that (1) x can change at most by 3, and (2) the expression is 2-sensitive in x, therefore the result can change at most by 6.

- Rule S-LAM typechecks sensitivity functions. As the body of the lambda has x as a free variable, the relational distance environment Δ is extended with distance d obtained from the type annotation on the argument. For example, consider program $\lambda^s(x:\mathbb{R}\cdot 2).\ x+x$ and its type derivation:

$$\dfrac{\dfrac{}{x:\mathbb{R}; 2x \vdash x + x:\mathbb{R}\ ;\ 2x}\,\textsc{s-lam}}{\varnothing\ ;\varnothing \vdash \lambda^s(x:\mathbb{R}\cdot 2).\ x + x : (x:\mathbb{R}\cdot 2) \xrightarrow{2x} \mathbb{R}\ ;\ \varnothing}.$$

The program is a sensitivity lambda that takes as argument a real with an allowed variation of at most 2 and has a latent contextual effect of $2x$.

- Rule P-LAM is defined analogously to S-LAM, except that its body is a privacy term, therefore it is typechecked using the privacy type system, explained later.

- Rule S-APP deals with sensitivity applications. Note that, from the type of the function, we know that d is an upper bound on the allowed argument variation, therefore, we require that the dot product between the relational distance environment and the sensitivity effect of the argument be less or equal than d. Intuitively, as Δ represents how much the input can change, and Σ_2 represent the sensitivity of variables used in the argument, $\Delta{\cdot}\Sigma_2$ represents how much the argument can change. For example, consider program $\lambda^s(y : \mathbb{R}{\cdot}1).\ (\lambda^s(x : \mathbb{R}{\cdot}2).\ x + x)(y + y)$ and its type derivation:

$$\dfrac{\dfrac{\dfrac{}{y : \mathbb{R}\ ; 1y \vdash (\lambda^s(x : \mathbb{R}{\cdot}2).\ x + x) : (x : \mathbb{R}{\cdot}2) \xrightarrow{2x} \mathbb{R}} \quad \dfrac{}{y : \mathbb{R}\ ; 1y \vdash (2 * y) : \mathbb{R}; 2y} \quad \boxed{(1y{\cdot}2y) \leq 2}}{y : \mathbb{R}\ ; 1y \vdash (\lambda^s(x : \mathbb{R}{\cdot}2).\ x + x)(2 * y) : \mathbb{R}\ ; 4y}\ \text{S-APP}}{\varnothing\ ; \varnothing \vdash \lambda^s(y : \mathbb{R}{\cdot}1).\ (\lambda^s(x : \mathbb{R}{\cdot}2).\ x + x)(2 * y) : (y : \mathbb{R}) \xrightarrow{4y} \mathbb{R}\ ; \varnothing.}\ \text{S-LAM}$$

  The outermost lambda allows a maximum variation of 1 in its argument y. The inner lambda allows a maximum variation of 2 on its argument and its being applied to $2 * y$. As $2 * y$ is 2-sensitive on y and y can wiggle at most by 1, then we know that the argument is going to wiggle at most by 2, which matches the maximum permitted argument variation. If the argument were $3 * y$, then the program would not typecheck, as the argument of the application could wiggle at most by $1 * 3 = 3$.
- Rule S-CASE typechecks subterms e_2 and e_3 by extending the relational distance environment with a sound bound for x and y, respectively. For x (respectively, y), we use the dot product between the relational distance on all variables in scope Δ and the cost of using e_1: the cost Σ_1 of reducing the expression, plus the latent cost of using its subterm Σ_{11} (respectively, Σ_{12}). For example, consider the type derivation of Example 4.4 given, e.g. relational distance environment $x + 2b$:

$$\dfrac{\Gamma; x + 2b \vdash e : \mathbb{R}\ ^{\infty x}\oplus^{x}\ \mathbb{R}; b \qquad \Gamma, x_1 : \mathbb{R}; x + 2b + \boxed{\infty x_1} \vdash 0 : \mathbb{R}; \varnothing \qquad \Gamma, x_2 : \mathbb{R}; x + 2b + \boxed{3x_2} \vdash x_2 : \mathbb{R}; x_2}{\Gamma; x + 2b \vdash \texttt{case}\ e\ \texttt{of}\ \{x_1 \Rightarrow 0\}\{x_2 \Rightarrow x_2\} : \mathbb{R}; b + x}\ \text{CASE}$$

  The relational distance for x_1 on the first branch is computed as the dot product between the maximum distance of all variables in scope, $x + 2b$, and the cost of using e if it were an `inl` expression, i.e., $((x + 2b){\cdot}(\infty x + b)) = \infty$. Analogously, the bound for x_2 on the second branch is computed as $((x + 2b){\cdot}(x + b)) = 3$.
- In Rule UNTUP, as expression e_2 has in scope new variables x_1 and x_2, the relational distance environment Δ is extended accordingly. The relational distance for x_1 is computed as the dot product between the relational distance environment Δ and the cost $\Sigma_1 + \Sigma_{11}$ of accessing the first component (we proceed similarly with x_2). For instance, consider the type derivation of Example 4.2 given some arbitrary relational distance environment $2x + 3y$:

$$\dfrac{\Gamma; 2x + 3y \vdash (2 * x, y) : \mathbb{R}\ ^{2x}\otimes^{y}\ \mathbb{R}\ ; \varnothing \qquad \Gamma, x_1 : \mathbb{R}, x_2 : \mathbb{R}; 2x + 3y + \boxed{4x_1 + 3x_2} \vdash x_1 + 2 * x_2 : \mathbb{R}\ ; x_1 + 2x_2}{\Gamma; 2x + 3y \vdash \texttt{let}\ x_1, x_2 = (2 * x, y)\ \texttt{in}\ x_1 + 2 * x_2 : \mathbb{R}\ ; 2x + 2y}\ \text{UNTUP}\quad .$$

  The relational distance for x_1 is computed as the dot product between the relational distance of all variables in scope $2x + 3y$ and the effect of using the left component of the pair $2x + 0y$, i.e., $(2x + 3y){\cdot}(2x + 0y) = 4$. Similarly, the bound of x_2 is computed as $(2x + 3y){\cdot}(y + 0x) = 3$.

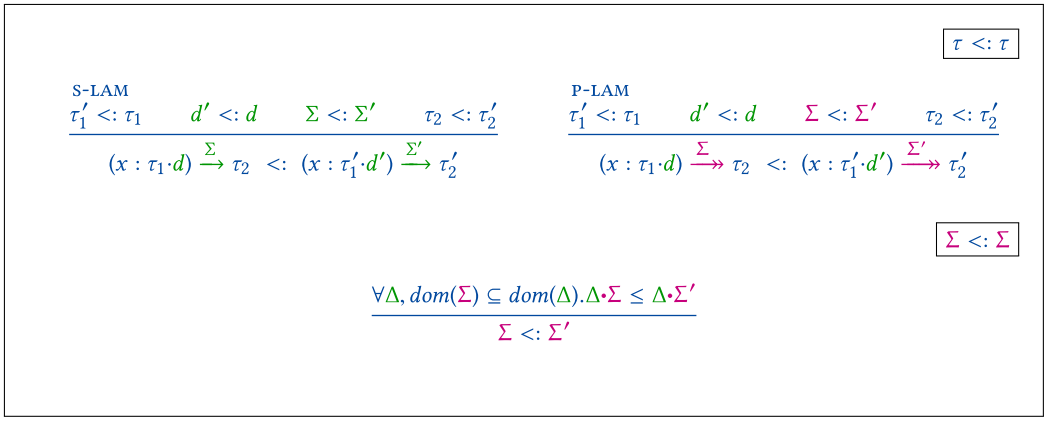

Fig. 12. λ_J: Subtyping.

Subtyping is extended accordingly and presented in Figure 12. Parameterized relational distances on function types are contravariant, and subtyping for privacy function types relies on the definition of subtyping for privacy environment also defined in Figure 12, where $\bullet$ is an operator to close privacy environments defined below:

$$\begin{aligned}
\Delta \bullet \varnothing &= 0 \\
\Delta \bullet (\Sigma_1 + \Sigma_2) &= (\Delta \bullet \Sigma_1) + (\Delta \bullet \Sigma_2) \\
\Delta \bullet (\Sigma_1 \sqcup \Sigma_2) &= (\Delta \bullet \Sigma_1) \sqcup (\Delta \bullet \Sigma_2) \\
\Delta \bullet (\Sigma_1 \sqcap \Sigma_2) &= (\Delta \bullet \Sigma_1) \sqcap (\Delta \bullet \Sigma_2) \\
\Delta \bullet p x &= p \qquad \text{if } \Delta(x) \text{ is defined.}
\end{aligned}$$

Privacy type system. The type system of the privacy part of the language is presented in Figure 13. The judgment $\Gamma \ ; \ \Delta \vdash \mathtt{e} : \tau \ ; \ \Sigma$ says that privacy term $\mathtt{e}$ has type τ and ambient privacy effect Σ under type environment Γ and relational distance environment Δ.

- Rule RETURN uses the type system of the sensitivity language to typecheck its subexpression e. Operationally, `return` constructs a point-distribution, and any sensitive variables in the subexpression e will have their privacy violated, i.e., privacy cost ∞. Notice that ∞ corresponds to the pair $(\epsilon, \delta) = (\infty, \infty)$. The resulting ambient privacy effect is computed by lifting to infinity the ambient effect of the subexpression as well as all free variable in e. The operator that lifts to ∞ free variables is written FP^{∞} and defined in Figure 13. As we pay infinity for every free variable in e, we remove those variables from the reported type τ using the sensitivity environment substitution operator defined in Figure 11. For instance, consider the following type derivation:

$$\frac{\text{RETURN} \quad \Gamma; y + z \vdash \lambda^{s}(x : \mathbb{R} \cdot 1).\ 2x + y : (x : \mathbb{R} \cdot 1) \xrightarrow{2x+y} \mathbb{R}; \varnothing}{\Gamma; y + z \vdash \mathtt{return}\ \lambda^{s}(x : \mathbb{R} \cdot 1).\ 2x + y : (x : \mathbb{R} \cdot 1) \xrightarrow{2x} \mathbb{R}; \infty y\ .}$$

  The resulting type and effect environment is computed by paying in advance for the free variables in scope: The type $[\varnothing/y](2x + y) = 2x$ is computed by erasing the free variables, and the effect environment ∞y is computed by lifting the free variables to infinity.
- Rule BIND typechecks both subexpressions using the type system for the privacy language, as they are privacy expressions. To typecheck $\mathtt{e}_2$, we extend type environment with variable x, therefore the relational distance environment Δ is also extended. We extend Δ with $0x$, as the

$$\textsc{return}\ \frac{\Gamma\ ;\ \Delta \vdash e : \tau\ ;\ \Sigma_1 \qquad \widehat{x} = \mathrm{FV}(e) \cup dom(\Sigma_1)}{\Gamma\ ;\ \Delta \vdash \mathtt{return}\ e : [\varnothing/\widehat{x}]\tau\ ;\ \mathrm{FP}^{\infty}(\widehat{x})}$$

$$\textsc{bind}\ \frac{\Gamma\ ;\ \Delta \vdash \mathrm{e}_1 : \tau_1\ ;\ \Sigma_1 \qquad \Gamma, x : \tau_1\ ;\ \Delta + 0x \vdash \mathrm{e}_2 : \tau_2\ ;\ \Sigma_2}{\Gamma\ ;\ \Delta \vdash x : \tau_1 \leftarrow \mathrm{e}_1\ ;\ \mathrm{e}_2 : [\varnothing/x]\tau_2\ ;\ \Sigma_1 + [\varnothing/x]\Sigma_2}$$

$$\textsc{if}\ \frac{\Gamma\ ;\ \Delta \vdash e_1 : \mathbb{B}\ ;\ \Sigma_1 \qquad \Gamma\ ;\ \Delta \vdash \mathrm{e}_2 : \tau\ ;\ \Sigma_2 \qquad \Gamma\ ;\ \Delta \vdash \mathrm{e}_3 : \tau\ ;\ \Sigma_3}{\Gamma\ ;\ \Delta \vdash \mathtt{if}\ e_1\ \mathtt{then}\ \mathrm{e}_2\ \mathtt{else}\ \mathrm{e}_3 : \tau\ ;\ \rceil\Sigma_1\lceil^{\infty} + (\Sigma_2 \sqcup \Sigma_3)}$$

$$\textsc{p-case}\ \frac{\begin{array}{c}\Gamma\ ;\ \Delta \vdash e_1 : \tau_{11}\ {}^{\Sigma_{11}}\!\oplus^{\Sigma_{12}}\ \tau_{12}\ ;\ \Sigma_1 \\ \Gamma, x : \tau_{11}\ ;\ \Delta + (\Delta{\cdot}(\Sigma_1 + \Sigma_{11}))x \vdash \mathrm{e}_2 : \tau_2\ ;\ \Sigma_2 \qquad \Gamma, y : \tau_{12}\ ;\ \Delta + (\Delta{\cdot}(\Sigma_1 + \Sigma_{12}))y \vdash \mathrm{e}_3 : \tau_3\ ;\ \Sigma_3\end{array}}{\begin{array}{l}\Gamma\ ;\ \Delta \vdash \mathtt{case}\ e_1\ \mathtt{of}\ \{x \Rightarrow \mathrm{e}_2\}\ \{y \Rightarrow \mathrm{e}_3\} : \\ \qquad [\Sigma_{11}/x]\tau_2 \sqcup [\Sigma_{12}/y]\tau_3\ ;\ \rceil\Sigma_1\lceil^{\infty} \sqcup ([\Sigma_{11}/x]\Sigma_2 \sqcup [\Sigma_{12}/y]\Sigma_3)\end{array}}$$

$$\textsc{p-app}\ \frac{\Gamma\ ;\ \Delta \vdash e_1 : (x : \tau_1{\cdot}d) \xrightarrow{\Sigma} \tau_2\ ;\ \Sigma_1 \qquad \Gamma\ ;\ \Delta \vdash e_2 : \tau_1\ ;\ \Sigma_2 \qquad \Delta{\cdot}\Sigma_2 \leq d}{\Gamma\ ;\ \Delta \vdash \mathrm{e}_1\ \mathrm{e}_2 : [\Sigma_2/x]\tau_2\ ;\ \rceil\Sigma_1\lceil^{\infty} + [\Sigma_2/x]\Sigma}$$

$$\mathrm{FP}^{\infty}(\varnothing) = \varnothing$$
$$\mathrm{FP}^{\infty}(\widehat{x}, y) = \mathrm{FP}^{\infty}(\widehat{x}) + \infty y$$

Fig. 13. λ_J: Type system of the privacy sublanguage.

value bound to x is no longer considered sensitive—it has been declassified and can be used without restriction. Finally, as x is out of scope, we remove it from τ_2 and from the resulting ambient privacy effect. For instance, consider the type derivation of program $y : \mathbb{R} \leftarrow \mathrm{laplace}_{\epsilon_1} x\ ;\ z : \mathbb{R} \leftarrow \mathrm{laplace}_{\epsilon_2} x; \mathtt{return}\ y + z$, similar to the example presented in Section 6.2, given an arbitrary relational distance environment x.

$$\textsc{bind}\ \frac{\textsc{bind}\ \dfrac{\Gamma\ ;\ x \vdash \mathrm{laplace}_{\epsilon_1} x : \mathbb{R}\ ;\ (\epsilon_1, 0)x \qquad \Gamma, y : \mathbb{R}\ ;\ x + 0y \vdash \mathrm{laplace}_{\epsilon_2} x : \mathbb{R}\ ;\ (\epsilon_2, 0)x}{\Gamma, y : \mathbb{R}\ ;\ x + 0y + 0z \vdash \mathtt{return}\ y + z : \mathbb{R}\ ;\ (\infty, 0)y + (\infty, 0)z}}{\Gamma, y : \mathbb{R}\ ;\ x + 0y \vdash z : \mathbb{R} \leftarrow \mathrm{laplace}_{\epsilon_2} x; \mathtt{return}\ y + z : \mathbb{R}\ ;\ (\epsilon_2, 0)x + (\infty, 0)y}$$
$$\Gamma\ ;\ x \vdash y : \mathbb{R} \leftarrow \mathrm{laplace}_{\epsilon_1} x\ ;\ z : \mathbb{R} \leftarrow \mathrm{laplace}_{\epsilon_2} x; \mathtt{return}\ y + z : \mathbb{R}\ ;\ (\epsilon_1 + \epsilon_2, 0)x$$

Each laplace call has an effect environment of $(\epsilon_1, 0)x$ and $(\epsilon_2, 0)x$, respectively. The return subexpression lifts to infinite the privacy of variables y and z, but to typecheck the innermost bind expression the privacy on z is dropped: $(\epsilon_2, 0)x + (\infty, 0)y$. Then, to typecheck the outermost bind expression, now the privacy on y is dropped getting a final effect environment of $(\epsilon_1, 0)x + (\epsilon_2, 0)x = (\epsilon_1 + \epsilon_2, 0)x$.

- Rule p-case is similar to rule s-case. Here, we lift the ambient sensitivity effect of the sum expression to infinity, i.e., we pay infinity for all non-zero-sensitive variables used in e_1. For the additional cost of each branch, we compute the join between the cost of each branch by substituting each binder by their appropriated cost: $[\Sigma_{11}/x]\Sigma_2$ for the first branch and $[\Sigma_{12}/y]\Sigma_3$

for the second. Note that we do not use $[\Sigma_1 + \Sigma_{11}/x]\Sigma_2$ and $[\Sigma_1 + \Sigma_{12}/x]\Sigma_3$ as we do in rule S-CASE, because we are already lifting to infinity (or paying for) every cost associated with Σ_1. For example, consider the following type derivation:

$$\frac{\begin{array}{c}\textsc{p-case}\\ \Gamma\,;\, x + 2y \vdash x : \mathbb{R} \,{}^{y}\!\oplus^{\varnothing}\, \mathbb{R}\,;\, x \qquad \Gamma, x_1 : \mathbb{R}\,;\, x + 2y + 3x_1 \vdash \mathtt{e}_2 : \mathbb{R}\,;\, p_x x + p_y y + p_2 x_1 \\ \Gamma, x_2 : \mathbb{R}\,;\, x + 2y + x_2 \vdash \mathtt{e}_3 : \mathbb{R}\,;\, p'_x x + p'_y y + p_3 x_2\end{array}}{\begin{array}{l}\Gamma\,;\, x + 2y \vdash \mathtt{case}\ x\ \mathtt{of}\ \{x_1 \Rightarrow \mathtt{e}_2\}\ \{x_2 \Rightarrow \mathtt{e}_3\} : \\ \qquad \mathbb{R}\,;\, \rceil x \lceil^{\infty} \sqcup (p_x x + p_y y + \rceil y \lceil^{p_2}) \sqcup (p'_x x + p'_y y + \rceil \varnothing \lceil^{p_3})\end{array}}\ .$$

Note that the variation bound of x_1 is computed as $(x + 2y)\cdot(x + y) = 3$ and that of x_2 as $(x + 2y)\cdot(x) = 1$. As $\rceil x \lceil^{\infty} = \infty x$, $\rceil y \lceil^{p_2} = p_2 y$, and $\rceil \varnothing \lceil^{p_3} = \varnothing$, then the resulting effect environment is $\infty x \sqcup (p_x x + (p_y + p_2)y) \sqcup (p'_x x + p'_y y) = \infty x \sqcup ((p_y + p_2) \sqcup p'_y)y$.

- Rule P-APP uses the sensitivity type system to typecheck both subterms. The first subterm has to be typed as a privacy function. Just as S-APP, it checks that the sensitivity cost of the argument is bounded by d by computing the dot operation $\Delta\cdot\Sigma_2$ between relational distance environment Δ and sensitivity environment Σ_2. The resulting ambient privacy effect is computed as the lift to infinite of the ambient sensitivity effect of e_1, plus the latent contextual effect of the privacy function, where we substitute Σ_2 by x. Similarly to rule S-APP, rule P-APP also enforces that the relational distance of the argument is bounded by d, i.e., $\Delta\cdot\Sigma_2 \leq d$. For instance, consider the following type derivation:

$$\frac{\begin{array}{c}\textsc{p-app}\\ \Gamma\,;\, y \vdash \mathtt{if}\ y\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : (x : \mathbb{R}\cdot 4) \xrightarrow{p_1 y \sqcup p_2 x} \mathbb{R}\,;\, y \qquad \Gamma\,;\, y \vdash 2 * y : \mathbb{R}\,;\, 2y \qquad 2 \leq 4\end{array}}{\Gamma\,;\, y \vdash (\mathtt{if}\ y\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3\})\ (2 * y) : \mathbb{R}\,;\, \rceil y \lceil^{\infty} + (p_1 y \sqcup \rceil 2y \lceil^{p_2})}\ .$$

The resulting effect environment is computed as $\infty y + [2y/x](p_1 y \sqcup p_2 x) = \infty y + p_1 y \sqcup \rceil 2y \lceil^{p_2} = \infty y + (p_1 \sqcup p_2)y$, which is equivalent to ∞y. If y does not wiggle, then the ambient privacy effect will be zero. If the relational distance environment for y were 3, then this program would be ill-typed, since $3y\cdot 2y \not\leq 4$.

7.2 λ_J: Type Safety

Type safety is defined in the same line of Section 5.2. To establish type safety of the privacy language, we define a non-deterministic sampling big-step semantics of privacy expressions; see Figure 14. We naturally extend the type safety logical relations of Figure 8 to support for both sensitivity and privacy lambdas and privacy expressions as shown in Figure 15. The fundamental property of the type safety logical relation is defined similarly to Proposition 5.2, but now accounting for relational distance environments and expressions:

Proposition 7.1 (Fundamental Property of the Type Safety Logical Relation).

(a) Let $\Gamma; \Delta \vdash e : \tau; \Sigma$, *and* $\gamma \in \mathcal{G}[\![\Gamma]\!]$. *Then* $\gamma \vdash e \in \mathcal{E}[\![\tau/\Gamma]\!]$.
(b) Let $\Gamma; \Delta \vdash \mathtt{e} : \tau; \Sigma$, *and* $\gamma \in \mathcal{G}[\![\Gamma]\!]$. *Then* $\gamma \vdash \mathtt{e} \in \mathcal{E}[\![\tau/\Gamma]\!]$.

Finally, type safety for closed terms is just a corollary of the fundamental property above:

Corollary 7.1 (Type Safety and Normalization of λ_J).

(a) Let $\vdash e : \tau; \varnothing$, *then* $\vdash e \Downarrow v$, *and* $\vdash v : \tau'; \varnothing$, *where* $\tau' <: \tau$.
(b) Let $\vdash \mathtt{e} : \tau; \varnothing$, *then* $\vdash \mathtt{e} \Downarrow v$, *and* $\vdash v : \tau'; \varnothing$, *where* $\tau' <: \tau$.

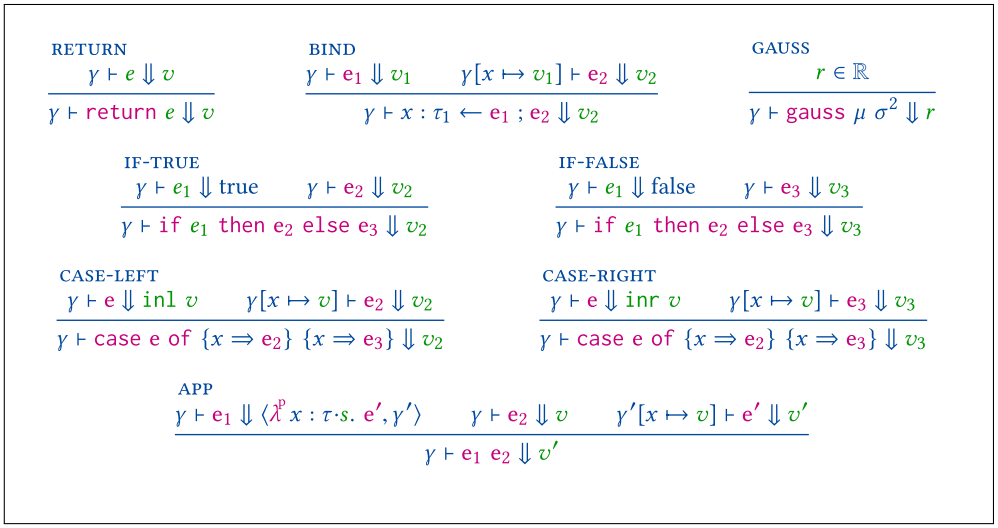

Fig. 14. Non-deterministic sampling semantics for privacy expressions.

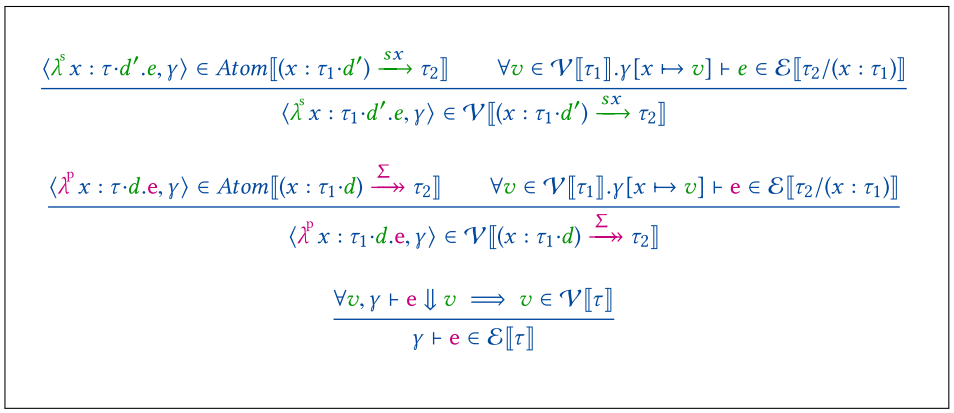

Fig. 15. λ_{J}: Type safety logical relation (selected rules).

7.3 Soundness of λ_{J}: Metric Preservation

This section establishes the *soundness* of λ_{J}, named *metric preservation*. Metric preservation for λ_{J} extends the notion of metric preservation of Sax. In addition to reasoning about sensitivity terms, given a privacy term with free variables, we can reason about the achieved privacy level when closing the privacy term under different (but related) environments.

Contrary to Sax, we establish soundness for λ_{J} using a *step-indexed* logical relation [3]. Although λ_{J} is a strongly normalizing language, step indexing is still required to prove the bind case of the fundamental property of the logical relation.

Probabilistic Semantics. A first step to define the soundness property of λ_{J} is to endow privacy expressions with a probabilistic semantics. An important observation here is that even though to match their traditional (theoretical) presentation, we have introduced the Laplacian and Gaussian mechanisms as sampling from the (uncountable) set of real numbers; for the formal account of the

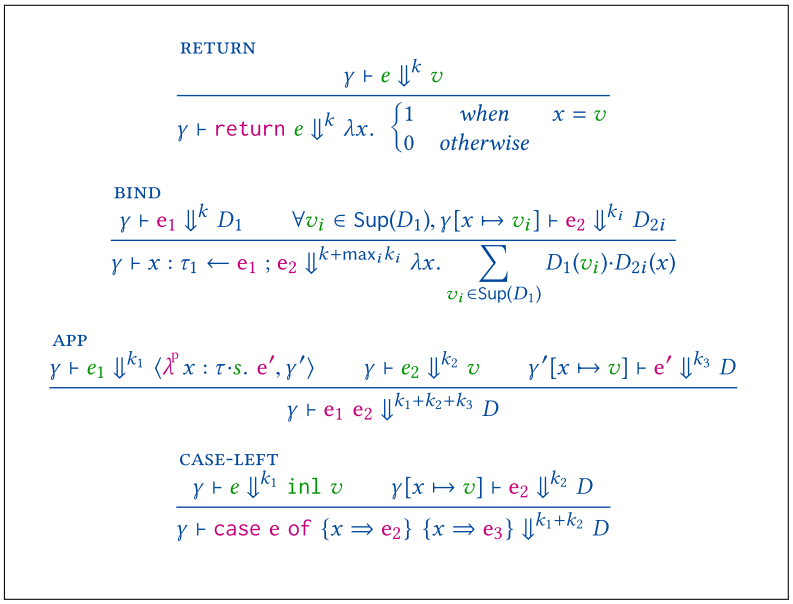

Fig. 16. Probabilistic semantics of privacy expressions (selected rules).

language, we consider *discrete* versions thereof over the set of integers [22]. This discretization is not only a natural but also a necessary requirement for any *implementation* of the language (on "finite" computers), since it is well-known that the naïve use of finite-precision approximations may result in fatal privacy breaches [43]. (However, for the sake of uniformity, in the rest of the presentation, we refer to these mechanisms—at the type level—as operating over the set of real numbers.) Therefore, privacy expressions in λ_{J} sample values only from discrete distributions and can be interpreted as discrete distributions over values.

The probabilistic semantics of a privacy expression e is formally defined in Figure 16. Judgment $\gamma \vdash \mathrm{e} \Downarrow^k D$ denotes that privacy expression e reduces to distribution D within k steps; the probability that the privacy configuration $\gamma \vdash \mathrm{e}$ reduces to value v is then computed as $D(v)$. We encode discrete distributions D as ***probability mass functions*** **(PMF)**, i.e., a discrete distribution over A is modeled as an element of the set $\mathcal{D}(A) = \{f : A \to [0,1] \mid \sum_{a \in A} f(a) = 1\}$.

Let us briefly explain the set of rules in Figure 16. For simplicity, we omit the underlying step indices. The probabilistic semantics of $\texttt{return}\ e$ assigns probability 1 to the (necessarily unique) value to which expression e reduces. The probabilistic semantics of a bind $y : \tau_1 \leftarrow \mathrm{e}_1\ ;\ \mathrm{e}_2$ operates as follows: To compute the probability that it assigns to x, it ranges over the set of values in the support of D_1 denoted as $\mathsf{Sup}(D_1)$, i.e., the set of values v such that $D_1(v) > 0$, and for each $v_i \in \mathsf{Sup}(D_1)$ it sums the product between the probability that e_1 reduces to v_i with the probability that e_2 reduces to x in an extended environment where y is bound to v_i. The discrete Gauss distribution with mean μ and scale σ^2 assigns probability proportional to $e^{-(x-\mu)^2/2\sigma^2}$ to each integer x. The probabilistic semantics of a privacy application is defined as the probabilistic semantics of the body of the resulting privacy closure in an extended environment where the closure formal argument is bound to the value of the real argument. Finally, the probabilistic semantics of a $\texttt{case}$ term is simply the probabilistic semantics of the corresponding branch in an extended environment with the corresponding association for the branch binder variable.

We consider a step-indexed semantics to establish the language metatheory. In particular, the metric preservation theorem is proved by induction on the step index of the logical relation.

$$
\begin{aligned}
(r_1, r_2) \in \mathcal{V}^k_d[\![\mathbb{R}]\!] &\overset{\Delta}{\iff} |r_1 - r_2| \le d \\
(v_1, v_2) \in \mathcal{V}^k_d[\![\mathtt{unit}]\!] &\overset{\Delta}{\iff} v_1 = \mathtt{tt} \wedge v_2 = \mathtt{tt} \\
(\mathtt{inl}\ v_1, \mathtt{inl}\ v_2) \in \mathcal{V}^k_d[\![\sigma_1\ {}^{d_1}\!\oplus^{d_2}\ \sigma_2]\!] &\overset{\Delta}{\iff} (v_1, v_2) \in \mathcal{V}^k_{d+d_1}[\![\sigma_1]\!] \\
(\mathtt{inr}\ v_1, \mathtt{inr}\ v_2) \in \mathcal{V}^k_d[\![\sigma_1\ {}^{d_1}\!\oplus^{d_2}\ \sigma_2]\!] &\overset{\Delta}{\iff} (v_1, v_2) \in \mathcal{V}^k_{d+d_2}[\![\sigma_2]\!] \\
((v_{11}, v_{12}), (v_{21}, v_{22})) \in \mathcal{V}^k_d[\![\sigma_1\ {}^{d_1}\!\&^{d_2}\ \sigma_2]\!] &\overset{\Delta}{\iff} \exists d_1', d_2', d = d_1' \sqcup d_2' \wedge \\
&\quad (v_{11}, v_{21}) \in \mathcal{V}^k_{d_1+d_1'}[\![\sigma_1]\!] \wedge (v_{12}, v_{22}) \in \mathcal{V}^k_{d_2+d_2'}[\![\sigma_2]\!] \\
(\langle v_{11}, v_{12}\rangle, \langle v_{21}, v_{22}\rangle) \in \mathcal{V}^k_d[\![\sigma_1\ {}^{d_1}\!\otimes^{d_2}\ \sigma_2]\!] &\overset{\Delta}{\iff} \exists d_1', d_2', d = d_1' + d_2' \wedge \\
&\quad (v_{11}, v_{21}) \in \mathcal{V}^k_{d_1+d_1'}[\![\sigma_1]\!] \wedge (v_{12}, v_{22}) \in \mathcal{V}^k_{d_2+d_2'}[\![\sigma_2]\!] \\
(v_1, v_2) \in \mathcal{V}^k_d[\![(x : \sigma_1 \cdot d') \xrightarrow{\Delta\cdot\Sigma} \sigma_2]\!] &\overset{\Delta}{\iff} \forall v_1', v_2', \gamma_1, \gamma_2, j < k, d'' \le d', (v_1', v_2') \in \mathcal{V}^j_{d''}[\![\sigma_1]\!] \implies \\
&\quad (\gamma_1 \vdash v_1\ v_1', \gamma_2 \vdash v_2\ v_2') \in \mathcal{E}^j_{d+(\Delta+d''x)\cdot\Sigma}[\![d''x(\sigma_2)]\!] \\
(v_1, v_2) \in \mathcal{V}^k_d[\![(x : \sigma_1 \cdot d') \xrightarrow{\Delta\cdot\Sigma} \sigma_2]\!] &\overset{\Delta}{\iff} \forall v_1', v_2', \gamma_1, \gamma_2, j < k, d'' < d', (v_1', v_2') \in \mathcal{V}^j_{d''}[\![\sigma_1]\!] \implies \\
&\quad (\gamma_1 \vdash v_1\ v_2', \gamma_2 \vdash v_1\ v_2') \in \mathcal{E}^j_{\lceil d \rceil^\infty + (\Delta + d''x)\cdot\Sigma}[\![d''x(\sigma_2)]\!] \\
(\gamma_1 \vdash e_1, \gamma_2 \vdash e_2) \in \mathcal{E}^k_d[\![\sigma]\!] &\overset{\Delta}{\iff} \forall j < k, \forall v_1, (d < \infty \wedge \gamma_1 \vdash e_1 \Downarrow^j v_1) \implies \\
&\quad \exists v_2, (\gamma_2 \vdash e_2 \Downarrow^* v_2 \wedge (v_1, v_2) \in \mathcal{V}^{k-j}_d[\![\sigma]\!]) \\
(\gamma_1 \vdash \mathrm{e}_1, \gamma_2 \vdash \mathrm{e}_2) \in \mathcal{E}^k_p[\![\sigma]\!] &\overset{\Delta}{\iff} \forall j < k, \forall D_1, (p.\epsilon + p.\delta < \infty \wedge \gamma_1 \vdash \mathrm{e}_1 \Downarrow^j D_1) \implies \exists D_2, \\
&\quad (\gamma_2 \vdash \mathrm{e}_2 \Downarrow^* D_2 \wedge \forall S \subseteq val(*), D_1(S) \le e^{p.\epsilon} D_2(S) + p.\delta) \\
(\gamma_1, \gamma_2) \in \mathcal{G}^k_\Delta[\![\Gamma]\!] &\overset{\Delta}{\iff} dom(\gamma_1) = dom(\gamma_2) = dom(\Gamma) \wedge \\
&\quad \forall x \in dom(\Gamma).\ (\gamma_1(x), \gamma_2(x)) \in \mathcal{V}^k_{\Delta(x)}[\![\Delta(\Gamma(x))]\!]
\end{aligned}
$$

Fig. 17. λ_J: logical relations for metric preservation.

However, step indices might not interact very well with the bind rule: When reducing $y : \tau_1 \leftarrow \mathrm{e}_1\ ;\ \mathrm{e}_2$, the reduction of e_2 requires a possibly different number of steps (k_i) for each value (v_i) to which e_1 reduces. This set of steps could in principle be unbounded, making $\max_i k_i$ undefined and thus rendering the semantics partial. However, this is not an issue for our technical development, because all formal results are concerned with programs that reach the distribution of final values within a *finite* number of steps, only.

For convenience throughout this section, we also introduce $D(S)$ to denote the probability of observing S in D, computed as $\sum_{v \in S} D(v)$. Also, we define $\Pr[\varnothing \vdash \mathrm{e} \Downarrow^k v]$ as $D(v)$ if $\varnothing \vdash \mathrm{e} \Downarrow^{k'} D$ for some $k' \le k$ (undefined otherwise).

Logical relation. The logical relations for sensitivity computations, privacy computations, values, and environments are mutually recursive and presented in Figure 17.

Note that each logical relation is also indexed by a relational distance type that now accounts for sensitivity and privacy lambdas:

$$
\begin{aligned}
\sigma &= \ldots \mid (x : \sigma \cdot d) \xrightarrow{\Sigma + d} \sigma \mid (x : \sigma \cdot s) \xrightarrow{\Sigma} \sigma \\
\Sigma &= \ldots \mid p.
\end{aligned}
$$

Note that similarly to sensitivity environments, privacy environments Σ are also extended to include partially instantiated data, for instance, $px + p'$. Notation $(v_1, v_2) \in \mathcal{V}^k_d[\![\sigma]\!]$ indicates that value v_1 is related to v_2 at type σ and distance d for k steps.

The sensitivity parts of the logical relations are defined analogously to Figure 8 with the addition of a step index k. We only present relevant changes:

- Two sensitivity closures are also related if, given related inputs, they produce related computations. Specifically, first the environments have to be related for any step $j < k$. Second, inputs v_1' and v_2' have to be related at distance d'' not greater than d' and for j steps. Finally, the bodies of the functions in extended environments have to be related computations for j steps.
- Similarly to sensitivity closures, two privacy closures are related if they produce related computations when applied to related inputs. The computations are related at privacy $\lceil d \rceil^{\infty} + (\Delta + d''x)\cdot\Sigma$. Note that we lift d to infinite, because we cannot record relational distances as a privacy result. In addition to that, we also pay the latent contextual effect of the function instantiated to $d''x$, i.e., $(\Delta + d''x)\cdot\Sigma$.

Two sensitivity configurations are related computations at type σ and distance d for k steps, noted $(\gamma_1 \vdash e_1, \gamma_2 \vdash e_2) \in \mathcal{E}_d^k[\![\sigma]\!]$, when for any $j < k$, if the first configuration reduces in j steps to a value, then the second configuration also reduces to a value in any number of steps, and these values are related for the remaining $k - j$ steps at type σ and distance d. We write $\gamma \vdash e \Downarrow^k v$ to say that the configuration $\gamma \vdash e$ reduces to value v in k steps.

We now turn to the definition of related privacy computations. Notation $(\gamma_1 \vdash \mathrm{e}_1, \gamma_2 \vdash \mathrm{e}_2) \in \mathcal{E}_p^k[\![\sigma]\!]$ indicates that two privacy configurations are related computations at type σ and privacy $p = (\epsilon, \delta)$ for k steps. Two privacy configurations are related when, for any $j < k$, if the first privacy configuration reduces in j steps to a distribution, then the second configuration also reduces to a distribution in any number of steps, and the probability of observing S in the first distribution is no greater than e^{ϵ} times the probability of observing S in the second distribution, plus δ.

Metric Preservation. Armed with these logical relations, we can establish the notion of type soundness for λ_{J} and prove the fundamental property.

Theorem 7.2 (Metric Preservation).

(1) $\Gamma, \Delta \vdash e : \tau \,;\, \Sigma \Rightarrow \forall k \geq 0, \forall \Delta' \sqsubseteq \Delta, \forall (\gamma_1, \gamma_2) \in \mathcal{G}_{\Delta'}^k[\![\Gamma]\!].(\gamma_1 \vdash e, \gamma_2 \vdash e) \in \mathcal{E}_{\Delta'\cdot\Sigma}^k[\![\Delta'(\tau)]\!]$,
(2) $\Gamma, \Delta \vdash \mathrm{e} : \tau \,;\, \Sigma \Rightarrow \forall k \geq 0, \forall \Delta' \sqsubseteq \Delta, \forall (\gamma_1, \gamma_2) \in \mathcal{G}_{\Delta'}^k[\![\Gamma]\!].(\gamma_1 \vdash \mathrm{e}, \gamma_2 \vdash \mathrm{e}) \in \mathcal{E}_{\Delta'\cdot\Sigma}^k[\![\Delta'(\tau)]\!]$,

where $\Delta' \sqsubseteq \Delta \iff dom(\Delta') = dom(\Delta) \wedge \forall x \in dom(\Delta'), \Delta'(x) \leq \Delta(x)$. The theorem says that if a sensitivity term (respectively, privacy term) is well-typed, then for any number of steps k, valid relational distance environment Δ' (not greater than Δ) and value environments (γ_1, γ_2), the two configurations are related computations at distance type $\Delta'(\tau)$ (closing all free sensitivity or privacy variables) and at relational distance $\Delta'\cdot\Sigma$ (respectively, $\Delta'\cdot\Sigma$). Note that as $dom(\Sigma) \subseteq dom(\Delta') = dom(\Delta)$ and $dom(\Sigma) \subseteq dom(\Delta') = dom(\Delta)$, then $\Delta'\cdot\Sigma \in \mathbb{R}^{\infty}$ and $\Delta'\cdot\Sigma \in priv$.

To prove the fundamental property, we rely on the following three lemmas that connect types, sensitivity, and privacy environments from the type system, with distances and privacy costs from the logical relations:

Lemma 7.3. *If* $\Delta\cdot\Sigma = d$ *and* $x \notin dom(\Sigma) \cup dom(\Delta)$, *then* $\Delta\cdot([\Sigma/x]\Sigma') = (\Delta + dx)\cdot\Sigma'$.

Lemma 7.4. *If* $\Delta\cdot\Sigma = d$ *and* $x \notin dom(\Sigma) \cup dom(\Delta)$, *then* $\Delta\cdot([\Sigma/x]\Sigma') = (\Delta + dx)\cdot\Sigma'$.

Lemma 7.5. *Let* $\Delta\cdot\Sigma = d$ *and* $x \notin dom(\Sigma) \cup dom(\Delta)$, *then* $\Delta([\Sigma/x]\tau) = dx(\Delta(\tau))$.

We can also derive from the fundamental property some corollaries about closed terms.

Corollary 7.6 (FP for Closed Sensitivity Terms). *If* $\varnothing; \varnothing \vdash e : \tau \,;\, \varnothing$, *then* $\forall k \geq 0, (\varnothing \vdash e, \varnothing \vdash e) \in \mathcal{E}_0^k[\![\tau]\!]$

Corollary 7.7 (FP for Closed Privacy Terms). *If* $\varnothing; \varnothing \vdash \mathrm{e} : \tau \,;\, \varnothing$, *then* $\forall k \geq 0, (\varnothing \vdash \mathrm{e}, \varnothing \vdash \mathrm{e}) \in \mathcal{E}_{(0,0)}^k[\![\tau]\!]$

In addition to sensitivity type soundness at base types (Prop 5.3), from the fundamental property, we can now establish privacy type soundness at base types:

Theorem 7.8 (Privacy Type Soundness at Base Types). *If* $\varnothing \vdash e : (x : \mathbb{R}\cdot 1) \xrightarrow{(\epsilon,\delta)x} \mathbb{R} \; ; \varnothing$, $|r_1 - r_2| \leq 1, \forall r, Pr[\varnothing \vdash \mathrm{e}\ r_1 \Downarrow^{\infty} r] \leq e^{\epsilon} Pr[\varnothing \vdash \mathrm{e}\ r_2 \Downarrow^{\infty} r] + \delta$

Finally, we observe that our technical development relies on the specific variant of differential privacy considered in restricted places: the bind case of the soundness theorem (Theorem 7.2); the definition of subtyping for privacy costs, specially the base case (Figure 23); operations over privacies p, such as dot product, addition, meet, join, lifting (Figures 25 and 26); monotonicity of meet and join of privacies w.r.t. subtyping (Reference [54, Lemmas C.7 and C.8]); monotonicity of dot product w.r.t. privacy ordering (Reference [54, Lemma C.13]); distributivity of dot product w.r.t. substitution (Lemma 7.4); weakening of related private computations (Reference [54, Lemma C.10]). The remaining definitions and lemmas are independent and could be reused as such to adapt this work to deal with other variants of differential privacy.

8 FROM λ_J TO JAZZ

The full prototype implementation of Jazz includes several extensions to the core language λ_J and address the non-determinism of multiplicative and additive products by using type annotations.

Type Polymorphism. Jazz implements System F (universal quantification over well-kinded types) and parametric polymorphism over all compound types, including vector/matrix schemas, allowing all data objects and functions in the language to be fully generic. This feature requires the use of type-level quantifiers and application.

Value Dependency. Jazz supports type-level dependency on values through singleton types—an approach we borrow directly from DFuzz [34]. This allows differentially private algorithms to be verified with respect to privacy parameters that are not fixed and instead are function arguments.

More specifically, singleton types [33] are a technique for supporting limited forms of value dependency that builds on standard (System-F-style) polymorphic type system features and an enriched kind system. In a type system with native support for dependent types, a dependent function with a real-valued argument is written $(x : \mathbb{R}) \to \tau$ where the return type τ can use x to refer symbolically to the eventual runtime value of x. In a singleton type encoding of dependent types, the same function is written $x : \mathbb{R}[\hat{x}] \to \tau$, where the return type τ can use $\hat{x}$ to refer symbolically to the eventual runtime value of x. In essence, there is still a syntactic split between term-level variables (x) and type-level variables ($\hat{x}$), and the type declaration $(x : \mathbb{R}[\hat{x}])$ links them, so $\hat{x}$ is the type-level *proxy* for the term-level variable x.

Let-binding Sensitivity Terms in Privacy Contexts. As described in Section 6.2, we implement latent sensitivity via local bindings in the privacy language. We implement this feature using an environment Φ to delay the "payments" of a value's sensitivity, which fulfills the same role as the boxed type introduced by Near et al. [46]. Unlike boxed types, this feature requires no additional annotations—sensitivity is inferred automatically. The complete type systems that includes Φ are presented in Appendix A, Figures 19 and 20.

Context Polymorphism. When implementing flexible primitives in Jazz, it becomes convenient to abstract over latent contextual effects. We label this form of abstraction as context polymorphism. This form of polymorphism in our language is what enables us to give a single generalized type to the primitives gauss and seqloop (a looping combinator that uses sequential composition, further discussed in Section 9.2). Because Jazz implements quantification over latent contextual effects, it is possible to afford privacy in type signatures to closed-over variables involved in the

differentially private computation. This feature requires the use of type-level quantifiers, application, substitution, and annotations for context schemas. A context schema is an angle bracket enclosed list of variables. Angle bracket context schemas in Jazz denote the set of variables that we care about preserving privacy for and are used in the introduction forms for sums, pairs, and functions, as well as in type-level application. For example, gauss <x> ($x + y * y$) is 1-sensitive in x, and bumps y to infinity privacy. This demonstrates the use of context polymorphism to indicate which variables we care about preserving privacy for. The use of seqloop in Section 9 provides another example of context polymorphism in action. Note that context-polymorphic functions in Jazz are required to be primitives.

Variants of Differential Privacy. In addition to (ϵ, δ)-differential privacy, Jazz supports zero-concentrated differential privacy [21] and Rényi differential privacy [44] and has built-in constructs for mixing the variants. Each variant has different privacy parameters and rules for composition, but all of them follow the same basic pattern as (ϵ, δ)-differential privacy. For example, we can give the Gaussian mechanism the following types for **Rényi differential privacy** (**RDP**; privacy parameters α and ϵ) and **zero-concentrated differential privacy** (**zCDP**; privacy parameter ρ):

$$\overset{\text{RDP}}{\text{gauss}} : \forall\ (\hat{d}{:}\mathbb{R})\ (\hat{\alpha}{:}\mathbb{R})\ (\hat{\epsilon}{:}\mathbb{R}).\ (d{:}\mathbb{R}[\hat{d}]) \to (\alpha{:}\mathbb{R}[\hat{\alpha}]) \to (\epsilon{:}\mathbb{R}[\hat{\epsilon}]) \to (x{:}\mathbb{R}{\cdot}\hat{d}) \xrightarrow{\infty(d+\alpha+\epsilon)+(\hat{\alpha},\hat{\epsilon})x} \mathbb{R}$$
$$\overset{\text{zCDP}}{\text{gauss}} : \forall\ (\hat{d}{:}\mathbb{R})\ (\hat{\rho}{:}\mathbb{R}).\ (d{:}\mathbb{R}[\hat{d}]) \to (\rho{:}\mathbb{R}[\hat{\rho}]) \to (x{:}\mathbb{R}{\cdot}\hat{d}) \xrightarrow{\infty(d+\rho)+\hat{\rho}x} \mathbb{R}.$$

Since RDP and zCDP guarantees can be converted to (ϵ, δ) guarantees, Jazz provides constructs for converting between variants. For example, the following code uses the Gaussian mechanism twice, each time satisfying $(20, 0.25)$-RDP. By sequential composition, the total cost is $(20, 0.5)$-RDP. The program then converts this guarantee to (ϵ, δ)-differential privacy, using $\delta = 10^{-5}$.

$$\lambda^{\mathrm{p}}(x{\cdot}1).\ \texttt{RENYI}[\delta = 10^{-5}]\{\ r_1 \leftarrow \text{gauss } 1\ 20\ 0.25\ x;\ r_2 \leftarrow \text{gauss } 2\ 20\ 0.25\ (x+x);\ \texttt{return}\ (r_1 + r_2)\ \} \ :\ (x : \mathbb{R}{\cdot}1) \xrightarrow{(1.08,\,10^{-5})x} \mathbb{R}$$

Jazz automatically finds the privacy cost of this function, in (ϵ, δ)-differential privacy, by performing the appropriate conversion. The ability to mix privacy variants in Jazz makes it easy to frame the privacy guarantee of any program in terms of (ϵ, δ) privacy cost, allowing privacy costs to be directly compared. In addition, it enables embedding iterative RDP and zCDP algorithms inside of (ϵ, δ) programs, allowing these programs to take advantage of the improved composition properties RDP and zCDP provide. This approach—leveraging recent variants for composition, but reporting privacy costs in terms of ϵ and δ—is extremely common in recent work on differentially private machine learning [2]. We make extensive use of variant-mixing in our case studies, described next.

9 IMPLEMENTATION & CASE STUDIES

Jazz enables programmers to implement and verify largely the same set of *applications* as Duet, but Jazz empowers the programmer to construct these applications in *simpler ways*, e.g., via composition of reusable library functions. This is possible, because Jazz gives types to many privacy functions and looping combinators that required custom typing rules in Duet. Our case studies demonstrate that instead of encoding these applications as a single monolithic function, Jazz enables their implementation through composition of multiple helper functions and their verification without the use of custom typing rules.

In particular, we highlight two important features of Jazz that enable refactoring programs to reuse library functions:

- Jazz gives types to privacy primitives and looping combinators that are not typeable in Duet, enabling privacy functions to be parameterized by these components (see Section 9.2).
- Jazz's privacy functions allow sensitivity arguments with arbitrary sensitivity bounds, enabling code reuse in more places than Duet's sensitivity-1 privacy functions (see Section 9.3).

In addition, we select realistic algorithms previously verified using other systems to demonstrate that Jazz maintains the capabilities of previous work.

A summary of our case study programs appears in Table 3. We present two new representative case study algorithms we have implemented and verified using Jazz: the MWEM algorithm [38] for a workload of linear queries, and a recently proposed algorithm for differentially private deep learning with adaptive clipping [53]. In both case studies, privacy mechanisms (e.g., laplace and exponential) and looping constructs (e.g., `aloop`—advanced composition) can be expressed with regular functions, provided in a library of primitives. We mark these two case studies with a * in Table 3 and describe them in detail later in this section. The other case study programs are available in our source code repository.

9.1 Implementation

We have implemented a prototype of the Jazz typechecker in Haskell and used it to verify the case studies from Table 3. The prototype implementation is available on GitHub.[7] Table 3 lists the time needed to typecheck each of the case studies; our typechecker takes just a few milliseconds for each one.

Type inference & annotations. Our prototype implements type inference for both Sax and Jazz. Type annotations for sensitivity and privacy are required for *inputs* at top-level functions and lambda-expressions, but no additional annotations are required for function outputs or elsewhere in the program. The case studies described later in this section have been typeset for readability, but are otherwise identical to the input for our prototype; in particular, the actual examples typechecked by our prototype have the same annotations as the examples in this section, except for the input sensitivity annotations on inputs to top-level functions. The types given for primitives in this section are also drawn directly from our implementation.

Constraint solving. As described earlier, and detailed in Section 10, prior work has made extensive use of SMT solvers for the equations over real expressions that arise in type inference for sensitivity. Because SMT solvers are incomplete for non-linear operations (like the logarithms and square roots used in advanced composition), our implementation does not follow the same path.

Instead, we implement a custom solver for inequalities over symbolic real expressions, based on the solver from Duet [46]. Our custom solver is based on a simple decidable (but incomplete) theory; it supports logarithms, square roots, and polynomial formulas over real numbers. The solver is transparent to the programmer and produces readable output expressions for the privacy costs in our case studies.

9.2 MWEM

The MWEM algorithm [38] generates differentially private synthetic data approximating the target sensitive data by iteratively optimizing the accuracy of a set of workload queries on the synthetic data. In each iteration, the algorithm uses the exponential mechanism to pick a query from the workload for which the synthetic data produces an *inaccurate* result, uses the Laplace mechanism to run that query on the real data, and uses the result to update the synthetic data via the *multiplicative weights update rule*. The Jazz program shown below implements the MWEM algorithm.

[7] https://github.com/uvm-plaid/contextual-duet/.

Its inputs are a sensitive dataset X over a domain D, a workload Q of linear queries, the number of iterations to be performed k, the privacy parameter ϵ, initial synthetic data Y_0 (n times the uniform distribution over D), and the dimensions of the input data set (matrix) m rows by n columns. The algorithm performs k iterations, invoking laplace and exponential in each iteration. The privacy parameter for each invocation is $\epsilon/2k$, yielding a total privacy cost of ϵ. We omit the sensitivity annotations on the function's inputs for readability.

```
MWEM ≜ λˢ n. λˢ k. λˢ ϵ. λˢ X. λˢ Q. λˢ loop. λᵖ Y₀.
  loop k Y₀ <X> (λᵖ A_{i-1}.
    q_iIndex ← exponential (ϵ / (2 ∗ k)) Q X (λˢ X′. λˢ q′. |q′ A_{i-1} − q′ X′|);
    let q_i = matrixIndex Q (index ℕ[0]) q_iIndex in
    m_i ← laplace 1 (ϵ / (2 ∗ k)) (q_i X);
    return mmap-row X A_{i-1} (λˢ x.
      mScale x (exp (q_i x ∗ (m_i − q_i A_{i-1} / (2 ∗ n))))))
```

The Jazz typechecker produces the following type for this implementation, indicating that the algorithm satisfies ϵ-differential privacy. Note the homogenous matrix type notation used here is $\mathbb{M}[m,n]\ \tau$, where m denotes the number of rows, n the number of columns, and τ the type of each entry. $<X>$ is the context schema argument for type-application of loop and indicates the program variable we want to preserve privacy for in this expression.

$$\text{MWEM} : \forall\ (\hat{m}:\mathbb{N})\ (\hat{n}:\mathbb{N})\ (\hat{k}:\mathbb{N})\ (\hat{d}:\mathbb{R}^+)\ (\hat{\epsilon}:\mathbb{R}^+).\ \overbrace{(n:\mathbb{N}[\hat{n}])}^{\text{columns}} \to \overbrace{(k:\mathbb{N}[\hat{k}])}^{\text{iterations}} \to \overbrace{(d:\mathbb{R}^+[\hat{d}])}^{\text{relational distance}}$$

$$\to \overbrace{(\epsilon:\mathbb{R}^+[\hat{\epsilon}])}^{\text{desired privacy}} \to \overbrace{(X:(\mathbb{M}[\hat{m},\hat{n}]\ \mathbb{D})\cdot\hat{d})}^{\text{sensitive data}} \to \overbrace{(Q:\text{List}\ ((xs:\mathbb{M}[1,\hat{n}]) \xrightarrow{xs} \mathbb{N}))}^{\text{linear queries}} \to \overbrace{(Y_0:\mathbb{M}[\hat{m},\hat{n}]\ \mathbb{D})}^{\text{initial synthetic db}}$$

$$\to \overbrace{(\tau_{\text{loop}})}^{\text{looping combinator}}\ \overbrace{\infty(n+k+\epsilon+Q+Y_0)+(\hat{\epsilon},0)X}^{\text{privacy effect}}\ \overbrace{\mathbb{M}[\hat{m},\hat{n}]\ \mathbb{R}}^{\text{synthetic db}}$$

$$\text{where } \tau_{\text{loop}} = \forall\ (\hat{k}:\mathbb{N})\ (\hat{d}:\mathbb{R}^+)\ (\hat{\epsilon}:\mathbb{R}^+)\ (\hat{\delta}:\mathbb{R}^+)\ (\hat{\epsilon}':\mathbb{R}^+)\ (\hat{\delta}':\mathbb{R}^+)\ (\Gamma:\texttt{cxt}).$$

$$(k:\mathbb{N}[\hat{k}]) \to (X:\tau\cdot\hat{d}) \to (f:(x:\tau\cdot\hat{d} \xrightarrow{(\hat{\epsilon},\hat{\delta})\Gamma} \tau)) \xrightarrow{\infty(k+d+f)+(\hat{\epsilon}',\hat{\delta}')\Gamma} \tau$$

On an average of 10 runs, it takes the Jazz typechecker 5.2 ms to produce this type for the MWEM algorithm.

Beyond Duet. This example demonstrates Jazz's ability to define algorithms in terms of library functions and to parameterize algorithms by the choice of component pieces. In this case, we define MWEM in terms of a generic looping privacy combinator loop; the caller of MWEM can specify looping combinators based on sequential composition, advanced composition, or even a custom combinator. Jazz similarly allows functions like MWEM to be parameterized by the choice of basic mechanism (e.g., laplace vs. gauss) with the appropriate privacy function type. In both cases, the relevant functions can be pulled from libraries or defined by the programmer.

This kind of modularity is impossible in Duet. Functions cannot be parameterized by basic privacy mechanisms or looping combinators, because it is not possible to write their types in Duet.

Primitives used. This case study demonstrates the composition of a complex iterative algorithm from basic privacy mechanisms encoded as Jazz primitives (e.g., laplace and exponential) and privacy combinators (e.g., seqloop, which implements looping with sequential composition for

Table 3. List of Case Studies Included with the Jazz Implementation

Technique	Ref.	Privacy Concept	Typecheck Time
Machine Learning Algorithms			
Noisy Gradient Descent	[18]	Composition	4.1 ms
Gradient Descent w/ Output Perturbation	[57]	Parallel comp. (sens.)	4.2 ms
Noisy Frank-Wolfe	[52]	Exponential mechanism	5.9 ms
Variations on Gradient Descent			
Minibatching	[18]	Privacy amplification	5.5 ms
Parallel-composition minibatching	—	Parallel composition	5.9 ms
Gradient clipping	[2]	Sensitivity bounds	4.5 ms
Adaptive gradient clipping* (Section 9.3)	[53]	Advanced variants	5.6 ms
Preprocessing & Deployment			
Hyperparameter tuning	[23]	Exponential mechanism	6.9 ms
Adaptive clipping	—	Sparse Vector Technique	7.7 ms
Z-Score normalization	[1]	Composition	6.9 ms
Algorithms for Linear Queries			
Multiplicative Weights (MWEM)* (Section 9.2)	[38]	Exponential mechanism	5.2 ms

Case studies marked with a * are described in detail in this section.

privacy). These primitives with types shown above would require explicit typing rules in the core Duet language. In Jazz, they can be given regular types, as shown below:

$$\text{exponential} : \forall\ (\hat{m} : \mathbb{N})\ (\hat{n} : \mathbb{N})\ (\hat{d} : \mathbb{R}^+)\ (\hat{\epsilon} : \mathbb{R}^+)\ (\Gamma : \texttt{cxt})\ (\tau : \bigstar).\ \overbrace{(d : \mathbb{R}^+[\hat{d}])}^{\text{relational distance}} \to \overbrace{(\epsilon : \mathbb{R}^+[\hat{\epsilon}])}^{\text{desired privacy}}$$
$$\to \overbrace{(Q : \text{List}\ \ ((xs : \mathbb{M}[1, \hat{n}]) \xrightarrow{xs} \mathbb{N}))}^{\text{linear queries}} \to \overbrace{(X : (\text{List}\ \tau)\cdot\hat{d})}^{\text{sensitive data}} \to \overbrace{(\phi : (x : \tau) \xrightarrow{\hat{d}(x+\Gamma)} \mathbb{R})}^{\text{scoring function}}$$
$$\xrightarrow{\overbrace{\infty(d + \epsilon + Q + \phi) + (\hat{\epsilon}, 0)(X + \Gamma)}^{\text{privacy effect}}} \tau$$

$$\text{seqloop} : \forall\ (\hat{k} : \mathbb{N})\ (\hat{d} : \mathbb{R}^+)\ (\hat{\epsilon} : \mathbb{R}^+)\ (\hat{\delta} : \mathbb{R}^+)\ (\Gamma : \texttt{cxt}).\ (k : \mathbb{N}[\hat{k}]) \to (d : \mathbb{R}^+[\hat{d}]) \to (\epsilon : \mathbb{R}^+[\hat{\epsilon}])$$
$$\to (\delta : \mathbb{R}^+[\hat{\delta}]) \to (X : (\text{List}\ \mathbb{R})\cdot\hat{d}) \to (f : (x : \text{List}\ \mathbb{R}\cdot\hat{d} \xrightarrow{(\hat{\epsilon}, \hat{\delta})\Gamma} \text{List}\ \mathbb{R}))$$
$$\xrightarrow{\infty(k+d+\epsilon+\delta+f)+(\hat{k}\hat{\epsilon}, \hat{k}\hat{\delta})\Gamma} \text{List}\ \mathbb{R}$$

To typecheck MWEM, the privacy closure rule in Jazz creates a function type for the λ^{p} that has a privacy effect for the body of $\frac{\epsilon}{k}$ because of the two uses of mechanisms that give $\frac{\epsilon}{2k}$ differential privacy. If we pass seqloop as the looping combinator loop, then this privacy effect is multiplied by the loop iteration number k as a result of the type of seqloop : $\mathbb{N}[k] \to \tau \to (\tau \xrightarrow{\rceil\Sigma\lceil^{\epsilon}} \tau) \xrightarrow{\rceil\Sigma\lceil^{k\epsilon}} \tau$ and the type rule for privacy function application. Finally, the new let rule for the Jazz privacy fragment that tracks latent contextual sensitivities is used in the let-binding for q_i to precompute an intermediate value that is used multiple times without the need for explicit boxing.

9.3 Differentially Private Deep Learning with Adaptive Clipping

The current state-of-the-art in differentially private machine learning is *noisy gradient descent* [2]: At each iteration of training, compute the gradient, *clip* the gradient to have bounded *L2* norm, and add noise in proportion to the clipping parameter. The clipping parameter is typically treated

as a hyperparameter set by the analyst before training.

```
gauss : (s : sens) → (x : ℝ·s) -∞,(ε,δ)x->> ℝ
zeros(n) ≜ mcreate_{L∞} 1 n (λ^s i. λ^s j. 0.0)

DPMean : ∀ (ε̂ : ℝ) (δ̂ : ℝ). (ε : ℝ[ε̂]) → (δ : ℝ[δ̂])
    → (s : sens) → (x : (𝕄[m̂, n̂] 𝔻)·s)
    -∞,(ε,δ)x->> (𝕄[1, n̂] ℝ) ≜
  mgauss ε δ (mean x)

clipUpdate(C^t, d, γ, gs, ε, δ) ≜
  β ← DPMean ε δ (𝕀_{‖d‖≤C^t} gs);
  return C^t − (β − γ)
```

```
DPAL(X, y, k, γ, ε, δ, δ′) ≜
  let C^0 = MAXNUM in
  let θ_0 = zeros (cols X) in
  aloop δ′ k θ_0 <X, y> (λ^p (θ^t, C^t).
      let gs = (gradients θ^t X y) in
      g_p ← DPMean ε δ (mclip C^t gs);
      let d = (θ^t − g_p) in
      C^t′ ← clipUpdate C^t d γ gs ε δ;
      return (d, C^t′))
```

Recent work by Thakkar et al. [53] proposed an algorithm for *adaptively* determining the clipping parameter during training by adaptively improving the clipping parameter based on a differentially private estimate of the percentage of gradients clipped in each iteration.

In each iteration, the implementation computes the gradients for a batch of examples gs, clips each gradient using the current parameter C^t, and uses the Gaussian mechanism to compute a differentially private average gradient g_p. Then, the algorithm updates the clipping parameter for the next iteration $C^{t\prime}$ using clipUpdate, which computes a noisy count β of the number of gradients in gs that are clipped under the clipping parameter C^t and uses the count to update the parameter. The inputs to the algorithm are the training data X, the training labels y, the number of iterations k, and the target percentage of gradients remaining un-clipped γ. ϵ, δ, and δ' are the privacy cost parameters. We omit the sensitivity annotations on top-level function inputs for readability. The Jazz typechecker typechecks DPAL in 5.6 ms (averaged over 10 runs).

Beyond Duet. In this case study, we implement a library function for the differentially private average (DPMean) and use it in two places. This refactoring is not possible in Duet, because Duet's privacy functions require all sensitive arguments to have a sensitivity of 1. In this algorithm, one of the uses of DPMean in fact has *data-dependent* sensitivity (the sensitivity of mclip C^t gs is C^t).

Practical implementations of algorithms like this one often rely on libraries of differentially private functions like DPMean (e.g., the Opacus library for differentially private deep learning [58], or the OpenDP library for differentially private analytics [35]). By lifting the limitations of Duet's privacy functions, Jazz makes it possible to implement and use such libraries.

Primitives used. This algorithm demonstrates the use of privacy combinators (e.g., aloop) that can be specified as primitives in Jazz but require special typing rules in Duet:

$$\texttt{aloop} : \forall\ \epsilon : \mathbb{R}^+, \delta : \mathbb{R}^+, \delta' : \mathbb{R}^+, s : \mathbb{R}^+, \Gamma : \texttt{cxt}.\ \overbrace{(\delta' : \mathbb{R}^+[\delta'])}^{\delta\text{ relaxation}} \to \overbrace{(k : \mathbb{N})}^{\text{iterations}} \to \overbrace{(init : \tau)}^{\text{initial value}}$$
$$\to \overbrace{(\mathcal{M} : (x : \tau \xrightarrow{(\epsilon,\delta)\Gamma} \tau))}^{\text{loop body}} \xrightarrow{\overbrace{(\epsilon', k\delta + \delta')\Gamma}^{\text{total privacy cost of the loop}}} \tau$$
$$\textit{where } \epsilon' \triangleq 2\epsilon\left(\sqrt{2k(\log(1/\delta'))}\right).$$

This type for aloop encodes the advanced composition theorem (introduced in Section 2). Our advanced composition combinator runs an (ϵ, δ)-differentially private function ($\mathcal{M}$) representing the body of the loop k times, for a total privacy cost of $\left(2\epsilon\left(\sqrt{2k(\log(1/\delta'))}\right), k\delta + \delta'\right)$ (a significant improvement over the sequential composition cost of $(k\epsilon, k\delta)$).

Our version of the adaptive clipping gradient descent algorithm uses advanced composition and (ϵ, δ)-differential privacy to demonstrate the encoding of `aloop` as a regular function in Jazz. Our source code repository contains an alternative implementation that uses zero-concentrated differential privacy for improved composition and converts the privacy guarantee to (ϵ, δ)-differential privacy at the end of the algorithm.

10 RELATED WORK

Verification techniques based on type systems. There are two threads of prior work in type-system-based verification of differential privacy for high-level programs: those based on linear types and those based on relational refinement types. Reed and Pierce [49] proposed Fuzz, the first type system for differential privacy based on linear typing; its fundamental components are a linear type system with an indexed "scaling" modality $!_s$ for tracking the sensitivity of programs and a monadic connective $\bigcirc$ to model randomized computations. An s-sensitive function is encoded in Fuzz as a linear function with scaled domain $!_s A \multimap B$ and often notated $A \multimap_s B$. An ϵ-differential privacy mechanism is represented as an ϵ-sensitive function with monadic return type as in $A \multimap_\epsilon \bigcirc B$. DFuzz [34] extends Fuzz with dependent types to encode sensitivity and privacy bounds that depend on the values of function arguments. This allows, e.g., reasoning about the privacy of iterative algorithms whose privacy cost depend on the number of iterations. Fuzz and DFuzz can be characterized by strong support for higher-order programming and potential for automation via type inference. They support pure differential privacy but approximate, and any other recent variants of differential privacy fall out of their scope due to nonlinear scaling. Several recently proposed approaches allow a Fuzz-like analysis for (ϵ, δ)-differentially private programs: Azevedo de Amorim et al. [29] leverage a *path construction* and a Fuzz-like type system. Fuzzi [60] integrates a Fuzz-like type system with an expressive program logic. Fuzzi directly connects (automated) type-based proofs of composition for sensitivity and privacy properties with (manual) apRHL proofs for basic constructs such as sequential composition and the Laplace mechanism. In our approach, however, properties of basic mechanisms must be axiomatized. The Fuzzi system targets imperative programs and does not provide support for higher-order programming with privacy functions. Finally, Duet [46] proposes a two-language design with linear types for tracking sensitivity and privacy; crucial restrictions in the typing rules of the privacy language allow encoding advanced differential privacy variants such as approximate, Rényi, zero-concentrated, and truncated-concentrated differential privacy. In Duet privacy functions are n-ary and written $(\tau_1 @ p_1, \ldots, \tau_n @ p_n) \multimap^* \tau$ for privacy quantities p_i such as (ϵ_i, δ_i) in the case of approximate differential privacy.

Unlike previous works based on linear type systems, HOARe^2 [14] uses relational refinement types to encode arbitrary relational properties of programs, including differential privacy. In HOARe^2, an s-sensitive function type is written $\Pi s'.\ \{x :: \tau_1 \mid \mathfrak{D}_{\tau_1}(x_\triangleleft, x_\triangleright) \leq s'\} \to \{y :: \tau_2 \mid \mathcal{D}_{\tau_2}(y_\triangleleft, y_\triangleright) \leq s s'\}$ where $\mathfrak{D}_\tau$ is a type-indexed distance metric, and $x_\triangleleft$ and $x_\triangleright$ are explicit symbolic representations of the "left" and "right" executions of the program in support of encoding relational properties. To account for probabilistic private computations, HOARe^2 uses an indexed monad $\mathfrak{M}_{\epsilon,\delta}[\tau]$: The type of an (ϵ, δ)-differentially private function is written $\{x :: \tau_1 \mid \mathfrak{D}_{\tau_1}(x_\triangleleft, x_\triangleright) \leq 1\} \to \mathfrak{M}_{\epsilon,\delta}[\{y :: \tau_2 \mid y_\triangleleft = y_\triangleright\}]$. A limitation of this encoding is that a function of two arguments that provides different privacy bounds for each argument (as described in Section 3.3) will report a summed, global privacy bound, because the tracking of privacy occurs in a single global index to the privacy monad $\mathfrak{M}_{\epsilon,\delta}$. Because privacy is proved as a relational property, rather than as a sensitivity/Lipschitz continuity property, HOARe^2 is also capable of placing relational distance bounds on arguments to functions. Regarding the implementation, both HOARe^2 and Jazz use dependent types to capture sizes. Regarding typechecking, even though some

Table 4. How Each System —*(D)*Fuzz, HOARe^2, Duet, and Jazz (This Article)—(1) Encodes Function Sensitivity in Types, (2) Structures Typing Judgments for Function Sensitivity, (3) Encodes Differential Privacy in Types, and (4) Structures Typing Judgments for Differential Privacy

System	Sens. Function	Sens. Typing	Priv. Function	Priv. Typing
*(D)*Fuzz	$\tau_1 \multimap_s \tau_2$	$\Gamma \vdash e : \tau$	$\tau_1 \multimap_\epsilon \bigcirc\tau_2$	$\Gamma \vdash e : \bigcirc\tau$
HOARe^2	$\Pi s'.\{x :: \tau_1 \mid \mathfrak{D}_{\tau_1}(x_\triangleleft, x_\triangleright) \leq s'\} \to \{y :: \tau_2 \mid \mathfrak{D}_{\tau_2}(y_\triangleleft, y_\triangleright) \leq ss'\}$	$\mathfrak{G} \vdash e :: \tau$	$\{x :: \tau_1 \mid \mathfrak{D}_{\tau_1}(x_\triangleleft, x_\triangleright) \leq 1\} \to \mathfrak{M}_{\epsilon,\delta}[\{y :: \tau_2 \mid y_\triangleleft = y_\triangleright\}]$	$\mathfrak{G} \vdash e :: \mathfrak{M}_{\epsilon,\delta}[\tau]$
Duet	$\tau \multimap_s \tau$	$\Gamma \vdash e : \tau$	$(\tau@p, \ldots, \tau@p) \multimap^* \tau$	$\Gamma \vdash e : \tau$
Jazz	$(x : \tau) \xrightarrow{\Sigma} \tau$	$\Gamma \vdash e : \tau \,;\, \Sigma$	$(x : \tau) \xrightarrow{\Sigma} \tau$	$\Gamma \vdash e : \tau \,;\, \Sigma$

automation has been achieved [24], the automation relies heavily on what is achievable with SMT solvers and has limited application to programs that make generous use of compositional or higher-order programming techniques or metric-distance relationships between values at non-base types. SMT solvers are not complete for non-linear operations, which are common in complex differential privacy mechanisms. Consider, for example, the expression for privacy cost under advanced composition: $\epsilon' = k\epsilon(e^\epsilon - 1) + \epsilon\sqrt{2k\ln(1/\delta')}$; SMT solvers are not capable of proving universally quantified qualities between equations like these, which limits their ability to automate reasoning about privacy cost.

It is important to note that all of these type systems support some form of recursion. Adding any form of recursion in the formalism of Jazz would make the technical development even more complicated. We just focused on a small core that could illustrate the main novelties of the latent or contextual approach.

To conclude the overview about type-system-based verification techniques, we refer the reader to Table 4, comparing different aspects of the reviewed type systems.

Techniques based on couplings and program logics. Approximate couplings [12] are a probabilistic abstraction that witnesses differential privacy properties of programs and have been successfully exploited for verification purposes. The relational Hoare logic apRHL [16] and its successors apRHL$^+$ [15] and span-apRHL [50] internalize the compositional construction of such couplings and capture from pure and approximate differential privacy to more recent variants such as Rényi, zero-concentrated, and truncated-concentrated differential privacy. While compared to other methods, these program logics are rather expressive going beyond the composition of (a set of predefined) basic mechanisms, derivations in the logics involve complex quantitative reasoning, not always amenable to automation. Even though there has been a successful report on partial automation [11], e.g., the synthesis of (quantitative relational) loop invariants for iterative algorithms remains challenging. To synthesize couplings, Albarghouthi and Hsu [5] use an alternative approach based on constraint solving, which is highly amenable to automation; the approach is, however, confined to ϵ-DP. Finally, to verify programs that achieve (ϵ, δ)-DP composing basic mechanisms, Barthe et al. [13] use a customized program product construction and traditional (non-relational and non-probabilistic) Hoare logic augmented with mechanism-specific rules. In recent work, Barthe et al. [9] show that checking differential privacy for imperative programs is undecidable in general but present a reduction to a decidable fragment of first-order logic for a restricted class of programs. Barthe et al. [10] also show that checking accuracy bounds for differentially private programs is also undecidable (in general). A common limitation of all these approaches is that they are restricted to first-order imperative programs.

Techniques based on randomness alignment. LightDP [59] and ShadowDP [55] take a third approach to verifying differential privacy based on *randomness alignments*. A randomness alignment

is an injective function relating the randomness from one execution of a differentially private mechanism to a second execution of the same mechanism (i.e., $\mathcal{M}(x)$ outputs the same result with noise H as $\mathcal{M}(x')$ outputs with noise $f(H)$, where f is the randomness alignment). Both LightDP and ShadowDP are capable of verifying complex low-level mechanisms like the sparse vector technique in just a few seconds. However, both tools target a first-order imperative programming language and have limited support for higher-order programming.

Techniques based on testing. Since differential privacy mechanisms are randomized, traditional methods of software testing do not apply. Two recent works by Bichsel et al. [19] and Ding et al. [30] address this challenge by automatically generating neighboring inputs for the mechanism being tested and sampling from their outputs many times to approximate their output distributions. For privacy mechanisms with major bugs, these tools are able to show that the approximated distributions do not satisfy the claimed differential privacy guarantee. Wilson et al. [56] have implemented a testing tool based on this approach in their open-source library. DPCheck [61] combines static analysis with instrumented concrete execution of the target program to detect bugs in even more complex algorithms.

Type systems and contextual information. The technical device of contextual latent effects used in Sax and Jazz is related to prior approaches to expose information about captured variables in function types. Leroy [40] and Hannan et al. [37] use function types augmented with the set of captured variables, the former for tracking dangerous type variables for polymorphic generalization and the latter for lifetime analysis. Scherer and Hoffmann [51] introduce *open closure types* to track additional information about closed-over variables in first-class functions. An open closure type augments the traditional arrow type with a lexical environment of closed over variables, further decorated by a mapping characterizing the use of each variable. For instance, they formalize a system where the mapping marks each variable with a Boolean indicating whether the evaluation of the body of the function depends on the variable or not. An open closure type also includes the name of the function argument similarly decorated. For the considered system, they prove a non-interference property, which states that closing an open term with two valuations that coincide on used variables yield the same result, up to used variables. Function types in Sax and Jazz can be seen as specific cases of open closure types, in which the information attached to captured variables (and arguments) is not a Boolean value, but sensitivity and privacy information. Consequently, the type soundness result we establish is more general than noninterference; for instance, in Sax, for two valuations that are at a given distance apart, the distance between the results is bounded by the sensitivities of each variable. The technical development of contextual linear types shares many concerns with that of open closure types, notably regarding the proper handling of the typing environment—in which order matters—and of scoping. However, our soundness results rely on logical relations, while they adopt a more restricted technique, sufficient for the purpose of the simple type system tracking Boolean information.

As observed by Scherer and Hoffmann, using open closure types allows delaying the accounting of information flow into closures from abstraction time to application time. Likewise, contextual sensitivity in Sax delays sensitivity accounting to application time. We extend this principle to positive type constructors such as products and sums, generally deferring accounting to elimination forms; we believe this would likewise apply to the simpler information-flow control setting they study.

Recently, Bao et al. [8] use a similar context-annotation technique to track reachability information in types. For instance, they augment the type of reference cell with the variables in scope that alias the cell. This information is tracked on function types as well to track the reachability information from closures. In all these approaches, type information depends on variable names; this is

quite different from dependent types, however, where types depend on arbitrary terms. The telescope nature of the typing environment is similar, but many of the deep challenges of dependent types do not manifest in this restricted setting.

Hoare Type Theory [45] supports specifying effectful, heap-manipulating computations by introducing Hoare-style pre/postconditions in types. Computation types include contexts of variables and heap locations to track footprints of logical assertions. In a similar vein as Leroy [40], the context of variables is not decorated with any information, so it is unclear whether one could handle contextual sensitivity and privacy in this approach.

Finally, it would be interesting to study if generic approaches such as *coeffects* [48] and *graded modal types* [47] can express delayed sensitivity and privacy tracking as developed here. These generic approaches use resource algebras (such as the semiring of natural numbers) for capturing modalities in types. To the best of our knowledge, these are not contextual: For instance, the arrow type is annotated with a scalar, label, and so on, not with a decorated environment as in open closure types and this work. Extending these generic approaches to support contextual information on all type constructors could bring the benefits of lazy accounting to a wide range of type-based quantitative program reasoning.

11 CONCLUSION

We have presented Jazz, a language and type system for differentially private programming with strong support for both higher-order programming and advanced variants of differential privacy. The key insight of our approach is latent contextual tracking of both privacy and sensitivity, which enables sum, product, and function types to describe their privacy effects—even for closure variables—making it possible to delay the payment of the effects until actual elimination, sometimes yielding advantages on the precision of the analysis and the annotation burden.

We have formalized a core subset of Jazz and proved its soundness using a step-indexed logical relation, following a novel strategy. Case studies demonstrate the ability to encode basic privacy mechanisms and privacy combinators as primitives in Jazz and to compose them to develop more complicated iterative differentially private algorithms.

Jazz extends the expressive power of systems like Fuzz [49] to advanced variants of differential privacy. Like Fuzz, it remains incapable of proving the correctness of basic privacy mechanisms like the Laplace mechanism. One interesting avenue for future work lies in combining Jazz with an expressive program logic like apRHL [16, 17] in the style of Fuzzi [60]. Such a combined system would provide a complete programming framework supporting higher-order programming and end-to-end privacy proofs. Another line of future work is to incorporate some form of recursion to Jazz such as recursive types.

APPENDICES

Throughout the appendix, we use symbol s instead of d and Σ instead of Δ, as they are interchangeable.

A λ_J: STATIC SEMANTICS

In this section, we present some definitions of the static semantics of λ_J not presented in the main document. Figure 18 presents the syntax of λ_J. Figures 19 and 20 present the complete sensitivity type system of λ_J. Figure 21 presents the complete type system of λ_J. They include the usage of Φ and the type rules for the derived expressions: Boolean, conditional, and let expressions. Figure 22 presents the sensitivity instantiation or dot product operator and the sensitivity type instantiation operator. Figure 23 presents the subtyping rules for λ_J. Figure 24 presents the

sensitivity environment substitution environment. Figure 25 presents the different lift operators. Finally, Figure 26 presents the join and meet operator between types.

$$
\begin{array}{lll}
r \in \mathbb{R} & & \\
b \in \mathbb{B} & & \\
x \in \mathrm{var} & & \\
e \in \mathrm{sexpr} & ::= r \mid e + e \mid e * e \mid e \leq e & \text{real numbers} \\
& \mid x \mid \lambda^{s}(x : \tau \cdot s).\ e \mid \lambda^{p}(x : \tau \cdot s).\ \mathtt{e} \mid e\ e & \text{vars, functions, apps} \\
& \mid \mathtt{tt} & \text{unit} \\
& \mid \mathtt{inl}^{\tau_2}\ e \mid \mathtt{inr}^{\tau_1}\ e \mid \mathtt{case}\ e\ \mathtt{of}\ \{x \Rightarrow e\}\ \{y \Rightarrow e\} & \text{sums} \\
& \mid (e, e) \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e & \text{add. products} \\
& \mid \langle e, e \rangle \mid \mathtt{let}\ x, x = e\ \mathtt{in}\ e & \text{mult. products} \\
& \mid e :: \tau & \text{ascription} \\
& \mid b \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e & \text{(derived) booleans} \\
& \mid \mathtt{let}\ x = e\ \mathtt{in}\ e & \text{(derived) let} \\
\mathrm{e} \in \mathrm{pexpr} & ::= \mathtt{return}\ e \mid x : \tau \ \leftarrow\ \mathrm{e}\ ;\ \mathrm{e} \mid \mathrm{e}\ \mathrm{e} & \text{return, bind, applications} \\
& \mid \mathtt{if}\ e\ \mathtt{then}\ \mathrm{e}\ \mathtt{else}\ \mathrm{e} \mid \mathtt{case}\ e\ \mathtt{of}\ \{x \Rightarrow \mathrm{e}\}\ \{y \Rightarrow \mathrm{e}\} & \text{conditionals, case} \\
& \mid \mathtt{let}\ x = e\ \mathtt{in}\ \mathrm{e} & \text{let} \\
s \in \mathit{sens} & \triangleq \mathbb{R}^{\infty} & \text{sensitivities} \\
\Sigma \in \mathit{senv} & \triangleq \mathrm{var} \rightharpoonup \mathit{sens} ::= sx + \ldots + sx & \text{sensitivity environments} \\
p \in \mathit{priv} & \triangleq (\mathbb{R}^{\infty}, \mathbb{R}^{\infty}) & \text{privacy costs} \\
\Sigma \in \mathit{penv} & \triangleq \varnothing \mid px \mid \Sigma + \Sigma \mid \Sigma \sqcup \Sigma \mid \Sigma \sqcap \Sigma & \text{privacy environments} \\
\tau \in \mathrm{type} & ::= \mathbb{R} \mid \mathbb{B} \mid \mathtt{unit} \mid (x : \tau \cdot s) \xrightarrow{\Sigma} \tau \mid (x : \tau \cdot s) \xrightarrow{\Sigma\!\!\!\!\rightarrow} \tau & \\
& \mid \tau\ {}^{\Sigma}\!\oplus^{\Sigma}\ \tau \mid \tau\ {}^{\Sigma}\!\&^{\Sigma}\ \tau \mid \tau\ {}^{\Sigma}\!\otimes^{\Sigma}\ \tau & \text{types} \\
\Gamma \in \mathrm{tenv} & \triangleq \mathrm{var} \rightharpoonup \mathrm{type} ::= \{x : \tau, \ldots, x : \tau\} & \text{type environments}
\end{array}
$$

Fig. 18. λ_{J}: Syntax.

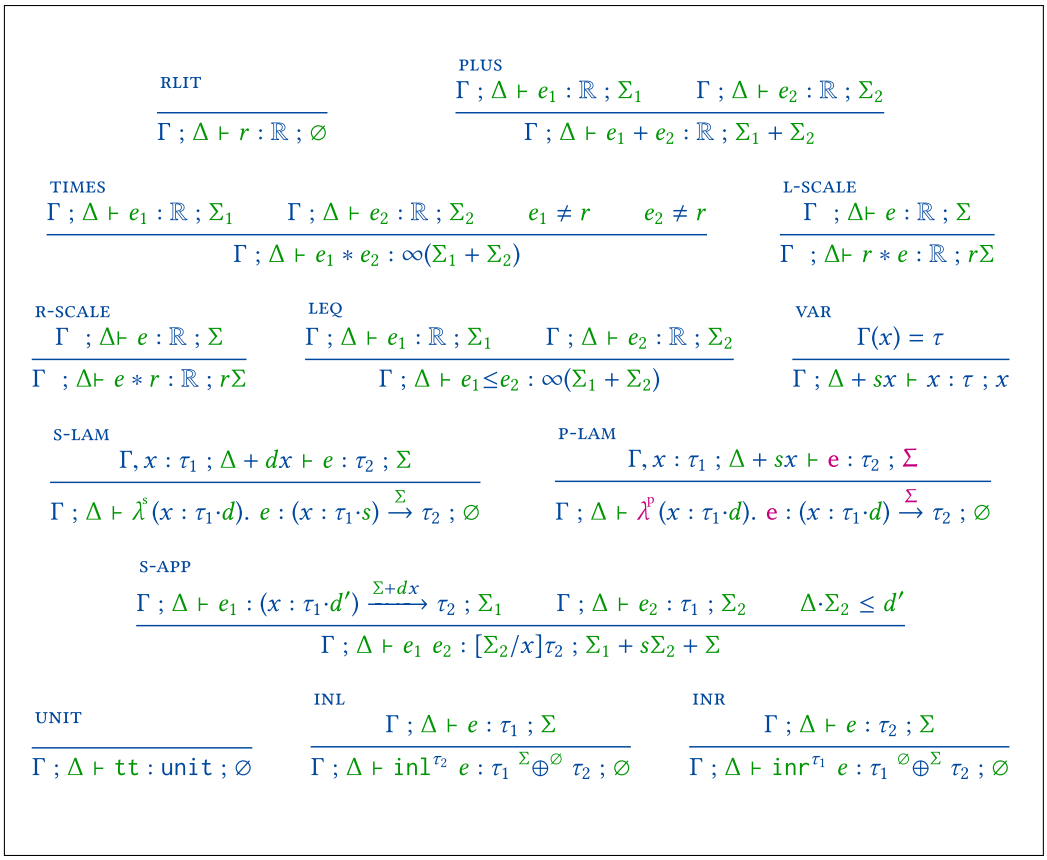

Fig. 19. λ_J: Complete sensitivity type system (part 1).

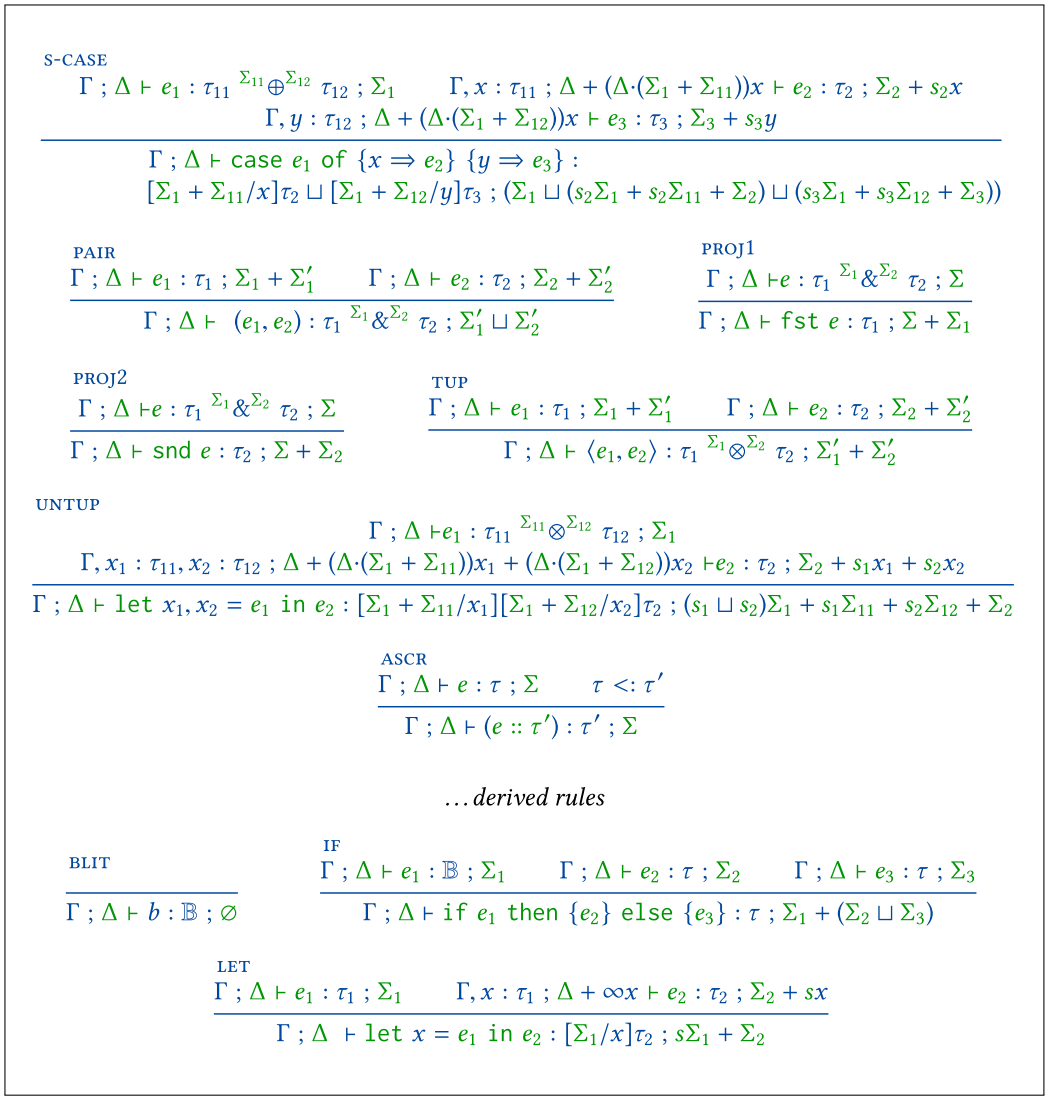

Fig. 20. λ_J: Complete sensitivity type system (part 2).

$$\textsc{return}\quad \frac{\Gamma\ ;\ \Delta \vdash e : \tau\ ;\ \Sigma_1 \qquad \widehat{x} = \mathrm{FV}(e) \cup dom(\Sigma_1)}{\Gamma\ ;\ \Delta \vdash \mathtt{return}\ e : [\varnothing/\widehat{x}]\tau\ ;\ \mathrm{FP}^\infty(\widehat{x})}$$

$$\textsc{bind}\quad \frac{\Gamma\ ;\ \Delta \vdash \mathrm{e}_1 : \tau_1\ ;\ \Sigma_1 \qquad \Gamma, x : \tau_1\ ;\ \Delta + 0x \vdash \mathrm{e}_2 : \tau_2\ ;\ \Sigma_2}{\Gamma\ ;\ \Delta \vdash x : \tau_1 \leftarrow \mathrm{e}_1\ ;\ \mathrm{e}_2 : [\varnothing/x]\tau_2\ ;\ \Sigma_1 + [\varnothing/x]\Sigma_2}$$

$$\textsc{if}\quad \frac{\Gamma\ ;\ \Delta \vdash e_1 : \mathbb{B}\ ;\ \Sigma_1 \qquad \Gamma\ ;\ \Delta \vdash \mathrm{e}_2 : \tau\ ;\ \Sigma_2 \qquad \Gamma\ ;\ \Delta \vdash \mathrm{e}_3 : \tau\ ;\ \Sigma_3}{\Gamma\ ;\ \Delta \vdash \mathtt{if}\ e_1\ \mathtt{then}\ \mathrm{e}_2\ \mathtt{else}\ \mathrm{e}_3 : \tau\ ;\ \rceil\Sigma_1\lceil^\infty + (\Sigma_2 \sqcup \Sigma_3)}$$

$$\textsc{p-case}\quad \frac{\begin{array}{c}\Gamma\ ;\ \Delta \vdash e_1 : \tau_{11}\ {}^{\Sigma_{11}}\!\oplus^{\Sigma_{12}}\ \tau_{12}\ ;\ \Sigma_1 \\ \Gamma, x : \tau_{11}\ ;\ \Delta + (\Delta\cdot(\Sigma_1 + \Sigma_{11}))x \vdash \mathrm{e}_2 : \tau_2\ ;\ \Sigma_2 \qquad \Gamma, y : \tau_{12}\ ;\ \Delta + (\Delta\cdot(\Sigma_1 + \Sigma_{12}))y \vdash \mathrm{e}_3 : \tau_3\ ;\ \Sigma_3\end{array}}{\begin{array}{l}\Gamma\ ;\ \Delta \vdash \mathtt{case}\ e_1 \mathtt{of}\ \{x \Rightarrow \mathrm{e}_2\}\ \{y \Rightarrow \mathrm{e}_3\} : \\ \qquad [\Sigma_{11}/x]\tau_2 \sqcup [\Sigma_{12}/y]\tau_3\ ;\ \rceil\Sigma_1\lceil^\infty \sqcup ([\Sigma_{11}/x]\Sigma_2 \sqcup [\Sigma_{12}/y]\Sigma_3)\end{array}}$$

$$\textsc{p-app}\quad \frac{\Gamma\ ;\ \Delta \vdash e_1 : (x : \tau_1\cdot s) \xrightarrow{\Sigma} \tau_2\ ;\ \Sigma_1 \qquad \Gamma\ ;\ \Delta \vdash e_2 : \tau_1\ ;\ \Sigma_2 \qquad \Delta\cdot\Sigma_2 \leq s}{\Gamma\ ;\ \Delta \vdash \mathrm{e}_1\ \mathrm{e}_2 : [\Sigma_2/x]\tau_2\ ;\ \rceil\Sigma_1\lceil^\infty + [\Sigma_2/x]\Sigma}$$

$$\begin{aligned}
\mathrm{FP}^\infty(\mathbb{R}) &= \varnothing \\
\mathrm{FP}^\infty(\mathtt{unit}) &= \varnothing \\
\mathrm{FP}^\infty((x : \tau_1\cdot s) \xrightarrow{\Sigma} \tau_2) &= [\varnothing/x]\rceil\Sigma\lceil^\infty + \mathrm{FP}^\infty(\tau_1) + \mathrm{FP}^\infty([\varnothing/x]\tau_2) \\
\mathrm{FP}^\infty((x : \tau_1\cdot s) \xrightarrow{\Sigma} \tau_2) &= [\varnothing/x]\rceil\Sigma\lceil^\infty + \mathrm{FP}^\infty(\tau_1) + \mathrm{FP}^\infty([\varnothing/x]\tau_2) \\
\mathrm{FP}^\infty(\tau_1\ {}^{\Sigma_1}\!\oplus^{\Sigma_2}\ \tau_2) &= \rceil\Sigma_1\lceil^\infty + \rceil\Sigma_2\lceil^\infty + \mathrm{FP}^\infty(\tau_1) + \mathrm{FP}^\infty(\tau_2) \\
\mathrm{FP}^\infty(\tau_1\ {}^{\Sigma_1}\!\&^{\Sigma_2}\ \tau_2) &= \rceil\Sigma_1\lceil^\infty + \rceil\Sigma_2\lceil^\infty + \mathrm{FP}^\infty(\tau_1) + \mathrm{FP}^\infty(\tau_2) \\
\mathrm{FP}^\infty(\tau_1\ {}^{\Sigma_1}\!\otimes^{\Sigma_2}\ \tau_2) &= \rceil\Sigma_1\lceil^\infty + \rceil\Sigma_2\lceil^\infty + \mathrm{FP}^\infty(\tau_1) + \mathrm{FP}^\infty(\tau_2)
\end{aligned}$$

Fig. 21. λ_J: Complete privacy type system.

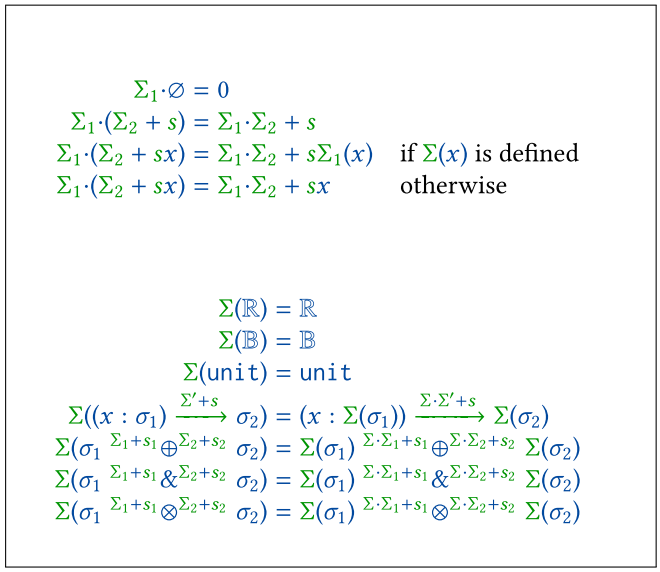

Fig. 22. Sensitivity instantiation/“dot product” and sensitivity type instantiation.

$\boxed{\Sigma <: \Sigma}$

$$\Sigma <: \Sigma' \overset{\Delta}{\iff} \forall x.\ \Sigma(x) \le \Sigma'(x)$$

$\boxed{\Sigma <: \Sigma}$

$$\frac{p.\epsilon \le p'.\epsilon \qquad p.\delta \le p'.\delta}{px <: p'x} \qquad \frac{}{\varnothing <: \Sigma} \qquad \frac{\Sigma_1 <: \Sigma_1' \qquad \Sigma_2 <: \Sigma_2'}{\Sigma_1 + \Sigma_2' <: \Sigma_1' + \Sigma_2'}$$

$$\frac{\Sigma_1 <: \Sigma_1' \vee \Sigma_1 <: \Sigma_2'}{\Sigma_1 <: \Sigma_1' \sqcup \Sigma_2'} \qquad \frac{\Sigma_1 <: \Sigma_1' \wedge \Sigma_1 <: \Sigma_1'}{\Sigma_1 \sqcup \Sigma_2 <: \Sigma_1'} \qquad \frac{\Sigma_1 <: \Sigma_1' \vee \Sigma_2 <: \Sigma_1'}{\Sigma_1 \sqcap \Sigma_2 <: \Sigma_1'}$$

$$\frac{\Sigma_1 <: \Sigma_1' \wedge \Sigma_1 <: \Sigma_2'}{\Sigma_1 <: \Sigma_1' \sqcap \Sigma_2'}$$

$\boxed{\tau <: \tau}$

BASE
$$\frac{\tau \in \{\mathbb{R}, \mathtt{unit}\}}{\tau <: \tau}$$

S-LAM
$$\frac{\tau_1' <: \tau_1 \qquad d' <: d \qquad \Sigma <: \Sigma' \qquad \tau_2 <: \tau_2'}{(x : \tau_1 \cdot d) \xrightarrow{\Sigma} \tau_2 \ <: \ (x : \tau_1' \cdot d') \xrightarrow{\Sigma'} \tau_2'}$$

P-LAM
$$\frac{\tau_1' <: \tau_1 \qquad d' <: d \qquad \Sigma <: \Sigma' \qquad \tau_2 <: \tau_2'}{(x : \tau_1 \cdot d) \xrightarrow{\Sigma}\!\!\!\!\rightarrow \tau_2 \ <: \ (x : \tau_1' \cdot d') \xrightarrow{\Sigma'}\!\!\!\!\rightarrow \tau_2'}$$

SUM
$$\frac{\tau_1 <: \tau_1' \qquad \Sigma_1 <: \Sigma_1' \qquad \tau_2 <: \tau_2' \qquad \Sigma_2 <: \Sigma_2'}{\tau_1 \ {}^{\Sigma_1}\!\oplus^{\Sigma_2} \tau_2 \ <: \ \tau_1' \ {}^{\Sigma_1'}\!\oplus^{\Sigma_2'} \tau_2'}$$

PAIR
$$\frac{\tau_1 <: \tau_1' \qquad \Sigma_1 <: \Sigma_1' \qquad \tau_2 <: \tau_2' \qquad \Sigma_2 <: \Sigma_2'}{\tau_1 \ {}^{\Sigma_1}\!\&^{\Sigma_2} \tau_2 \ <: \ \tau_1' \ {}^{\Sigma_1'}\!\&^{\Sigma_2'} \tau_2'}$$

TUP
$$\frac{\tau_1 <: \tau_1' \qquad \Sigma_1 <: \Sigma_1' \qquad \tau_2 <: \tau_2' \qquad \Sigma_2 <: \Sigma_2'}{\tau_1 \ {}^{\Sigma_1}\!\otimes^{\Sigma_2} \tau_2 \ <: \ \tau_1' \ {}^{\Sigma_1'}\!\otimes^{\Sigma_2'} \tau_2'}$$

Fig. 23. Subtyping.

$$
\begin{aligned}
[\_/x]\_ &: \mathsf{sensv} \times \mathsf{type} \to \mathsf{type} \\
[\Sigma/x]\mathbb{R} &= \mathbb{R} \\
[\Sigma/x]\mathbb{B} &= \mathbb{B} \\
[\Sigma/x]\mathtt{unit} &= \mathtt{unit} \\
[\Sigma/x]((y : \tau_1 \cdot d) \xrightarrow{\Sigma'} \tau_2) &= (y : [\Sigma/x]\tau_1 \cdot d) \xrightarrow{[\Sigma/x]\Sigma'} [\Sigma/x]\tau_2 \\
[\Sigma/x](\tau_1 \mathbin{^{\Sigma_1}\oplus^{\Sigma_2}} \tau_2) &= [\Sigma/x]\tau_1 \mathbin{^{[\Sigma/x]\Sigma_1}\oplus^{[\Sigma/x]\Sigma_2}} [\Sigma/x]\tau_2 \\
[\Sigma/x](\tau_1 \mathbin{^{\Sigma_1}\&^{\Sigma_2}} \tau_2) &= [\Sigma/x]\tau_1 \mathbin{^{[\Sigma/x]\Sigma_1}\&^{[\Sigma/x]\Sigma_2}} [\Sigma/x]\tau_2 \\
[\Sigma/x](\tau_1 \mathbin{^{\Sigma_1}\otimes^{\Sigma_2}} \tau_2) &= [\Sigma/x]\tau_1 \mathbin{^{[\Sigma/x]\Sigma_1}\otimes^{[\Sigma/x]\Sigma_2}} [\Sigma/x]\tau_2
\end{aligned}
$$

$$
\begin{aligned}
[\_/x]\_ &: \mathsf{sensv} \times \mathsf{sensv} \to \mathsf{sensv} \\
[\Sigma/x]\varnothing &= \varnothing \\
[\Sigma/x](\Sigma' + sx) &= [\Sigma/x]\Sigma + s\Sigma' \\
[\Sigma/x](\Sigma' + sy) &= [\Sigma/x]\Sigma + sy
\end{aligned}
$$

$$
\begin{aligned}
[\_/x]\_ &: \mathsf{sensv} \times (\mathsf{sensv} + \mathsf{sens}) \to (\mathsf{sensv} + \mathsf{sens}) \\
[\Sigma/x](\Sigma' + s) &= [\Sigma/x]\Sigma' + s \\
[\Sigma/x]s &= s
\end{aligned}
$$

Fig. 24. Sensitivity environment substitution.

$$
\begin{aligned}
\rceil s \lceil^{s'} &= s' && s > 0 \\
\rceil \varnothing \lceil^{s'} &= \varnothing \\
\rceil \Sigma + sx \lceil^{s'} &= \rceil \Sigma \lceil^{s'} + \rceil s \lceil^{s'} x \\
\rceil \Sigma + 0x \lceil^{s'} &= \rceil \Sigma \lceil^{s'} + 0x \\
\rceil p \lceil^{p'} &= p' && p \neq (0,0) \\
\rceil (0,0) \lceil^{p'} &= (0,0) \\
\rceil \varnothing \lceil^{p'} &= \varnothing \\
\rceil px \lceil^{p'} &= \rceil p \lceil^{p'} x \\
\rceil \Sigma_1 + \Sigma_2 \lceil^{\infty} &= \rceil \Sigma_1 \lceil^{\infty} + \rceil \Sigma_2 \lceil^{\infty} \\
\rceil \Sigma_1 \sqcup \Sigma_2 \lceil^{\infty} &= \rceil \Sigma_1 \lceil^{\infty} \sqcup \rceil \Sigma_2 \lceil^{\infty} \\
\rceil \Sigma_1 \sqcap \Sigma_2 \lceil^{\infty} &= \rceil \Sigma_1 \lceil^{\infty} \sqcap \rceil \Sigma_2 \lceil^{\infty} \\
\rceil \Sigma \lceil^{p'} &= \bigsqcup_{x \in dom(\Sigma)} (\rceil \Sigma(x) \lceil^{1} p) x
\end{aligned}
$$

Fig. 25. Lift operators.

$$
\begin{aligned}
&\_ \sqcup \_ : \mathrm{type} \times \mathrm{type} \to \mathrm{type}\\
&\mathbb{R} \sqcup \mathbb{R} = \mathbb{R}\\
&\mathbb{B} \sqcup \mathbb{B} = \mathbb{B}\\
&\mathtt{unit} \sqcup \mathtt{unit} = \mathtt{unit}\\
&(y : \tau_1 \cdot d) \xrightarrow{\Sigma} \tau_1 \sqcup (y : \tau_1' \cdot d') \xrightarrow{\Sigma'} \tau_2' = (y : (\tau_1 \sqcap \tau_1') \cdot (d \sqcap d')) \xrightarrow{\Sigma \sqcup \Sigma'} (\tau_2 \sqcup \tau_2')\\
&(y : \tau_1 \cdot d) \xrightarrow{\Sigma} \!\!\!\!\to\ \tau_1 \sqcup (y : \tau_1' \cdot d') \xrightarrow{\Sigma'} \!\!\!\!\to\ \tau_2' = (y : (\tau_1 \sqcap \tau_1') \cdot (d \sqcap d')) \xrightarrow{\Sigma \sqcup \Sigma'} \!\!\!\!\to\ (\tau_2 \sqcup \tau_2')\\
&\tau_1 \,{}^{\Sigma_1}\!\oplus^{\Sigma_2}\, \tau_2 \sqcup (\tau_1' \,{}^{\Sigma_1'}\!\oplus^{\Sigma_2'}\, \tau_2') = (\tau_1 \sqcup \tau_1') \,{}^{\Sigma_1 \sqcup \Sigma_1'}\!\oplus^{\Sigma_2 \sqcup \Sigma_2'}\, (\tau_2 \sqcup \tau_2')\\
&\tau_1 \,{}^{\Sigma_1}\!\&^{\Sigma_2}\, \tau_2 \sqcup (\tau_1' \,{}^{\Sigma_1'}\!\&^{\Sigma_2'}\, \tau_2') = (\tau_1 \sqcup \tau_1') \,{}^{\Sigma_1 \sqcup \Sigma_1'}\!\&^{\Sigma_2 \sqcup \Sigma_2'}\, (\tau_2 \sqcup \tau_2')\\
&\tau_1 \,{}^{\Sigma_1}\!\otimes^{\Sigma_2}\, \tau_2 \sqcup (\tau_1' \,{}^{\Sigma_1'}\!\otimes^{\Sigma_2'}\, \tau_2') = (\tau_1 \sqcup \tau_1') \,{}^{\Sigma_1 \sqcup \Sigma_1'}\!\otimes^{\Sigma_2 \sqcup \Sigma_2'}\, (\tau_2 \sqcup \tau_2')
\end{aligned}
$$

$$
\begin{aligned}
&\_ \sqcap \_ : \mathrm{type} \times \mathrm{type} \to \mathrm{type}\\
&\mathbb{R} \sqcap \mathbb{R} = \mathbb{R}\\
&\mathbb{B} \sqcap \mathbb{B} = \mathbb{B}\\
&\mathtt{unit} \sqcap \mathtt{unit} = \mathtt{unit}\\
&(y : \tau_1 \cdot d) \xrightarrow{\Sigma} \tau_1 \sqcap (y : \tau_1' \cdot d') \xrightarrow{\Sigma'} \tau_2' = (y : (\tau_1 \sqcup \tau_1') \cdot (d \sqcup d')) \xrightarrow{\Sigma \sqcap \Sigma'} (\tau_2 \sqcap \tau_2')\\
&(y : \tau_1 \cdot d) \xrightarrow{\Sigma} \!\!\!\!\to\ \tau_1 \sqcap (y : \tau_1' \cdot d') \xrightarrow{\Sigma'} \!\!\!\!\to\ \tau_2' = (y : (\tau_1 \sqcup \tau_1') \cdot (d \sqcup d')) \xrightarrow{\Sigma \sqcap \Sigma'} \!\!\!\!\to\ (\tau_2 \sqcap \tau_2')\\
&\tau_1 \,{}^{\Sigma_1}\!\oplus^{\Sigma_2}\, \tau_2 \sqcap (\tau_1' \,{}^{\Sigma_1'}\!\oplus^{\Sigma_2'}\, \tau_2') = (\tau_1 \sqcap \tau_1') \,{}^{\Sigma_1 \sqcap \Sigma_1'}\!\oplus^{\Sigma_2 \sqcap \Sigma_2'}\, (\tau_2 \sqcap \tau_2')\\
&\tau_1 \,{}^{\Sigma_1}\!\&^{\Sigma_2}\, \tau_2 \sqcap (\tau_1' \,{}^{\Sigma_1'}\!\&^{\Sigma_2'}\, \tau_2') = (\tau_1 \sqcap \tau_1') \,{}^{\Sigma_1 \sqcap \Sigma_1'}\!\&^{\Sigma_2 \sqcap \Sigma_2'}\, (\tau_2 \sqcap \tau_2')\\
&\tau_1 \,{}^{\Sigma_1}\!\otimes^{\Sigma_2}\, \tau_2 \sqcap (\tau_1' \,{}^{\Sigma_1'}\!\otimes^{\Sigma_2'}\, \tau_2') = (\tau_1 \sqcap \tau_1') \,{}^{\Sigma_1 \sqcap \Sigma_1'}\!\otimes^{\Sigma_2 \sqcap \Sigma_2'}\, (\tau_2 \sqcap \tau_2')
\end{aligned}
$$

$$
\begin{aligned}
&\_ \sqcup \_ : \mathrm{sensv} \times \mathit{sensv} \to \mathit{sensv}\\
&\varnothing \sqcup \varnothing = \varnothing\\
&(\Sigma + sx) \sqcup (\Sigma' + s'x) = (\Sigma \sqcup \Sigma') + (s \sqcup s')x \;\; x \notin \mathit{dom}(\Sigma \sqcup \Sigma')\\
&\Sigma \sqcup (\Sigma' + sx) = (\Sigma \sqcup \Sigma') + sx \qquad (x \notin \mathit{dom}(\Sigma))\\
&(\Sigma + sx) \sqcup \Sigma' = (\Sigma \sqcup \Sigma') + sx \qquad (x \notin \mathit{dom}(\Sigma'))\\
&\_ \sqcup \_ : \mathrm{sensv} \times \mathit{sensv} \to \mathit{sensv}\\
&\varnothing \sqcap \varnothing = \varnothing\\
&(\Sigma + sx) \sqcap (\Sigma' + s'x) = (\Sigma \sqcap \Sigma') + (s \sqcap s')x \;\; x \notin \mathit{dom}(\Sigma \sqcap \Sigma')\\
&\Sigma \sqcap (\Sigma' + sx) = (\Sigma \sqcap \Sigma') \qquad (x \notin \mathit{dom}(\Sigma))\\
&(\Sigma + sx) \sqcap \Sigma' = (\Sigma \sqcap \Sigma') \qquad (x \notin \mathit{dom}(\Sigma'))
\end{aligned}
$$

Fig. 26. Join and Meet of types and sensitivity environments.

B λ_J: DYNAMIC SEMANTICS

Figure 27 presents the dynamic semantics of the sensitivity language. Figure 28 presents the dynamic semantics of the privacy language.

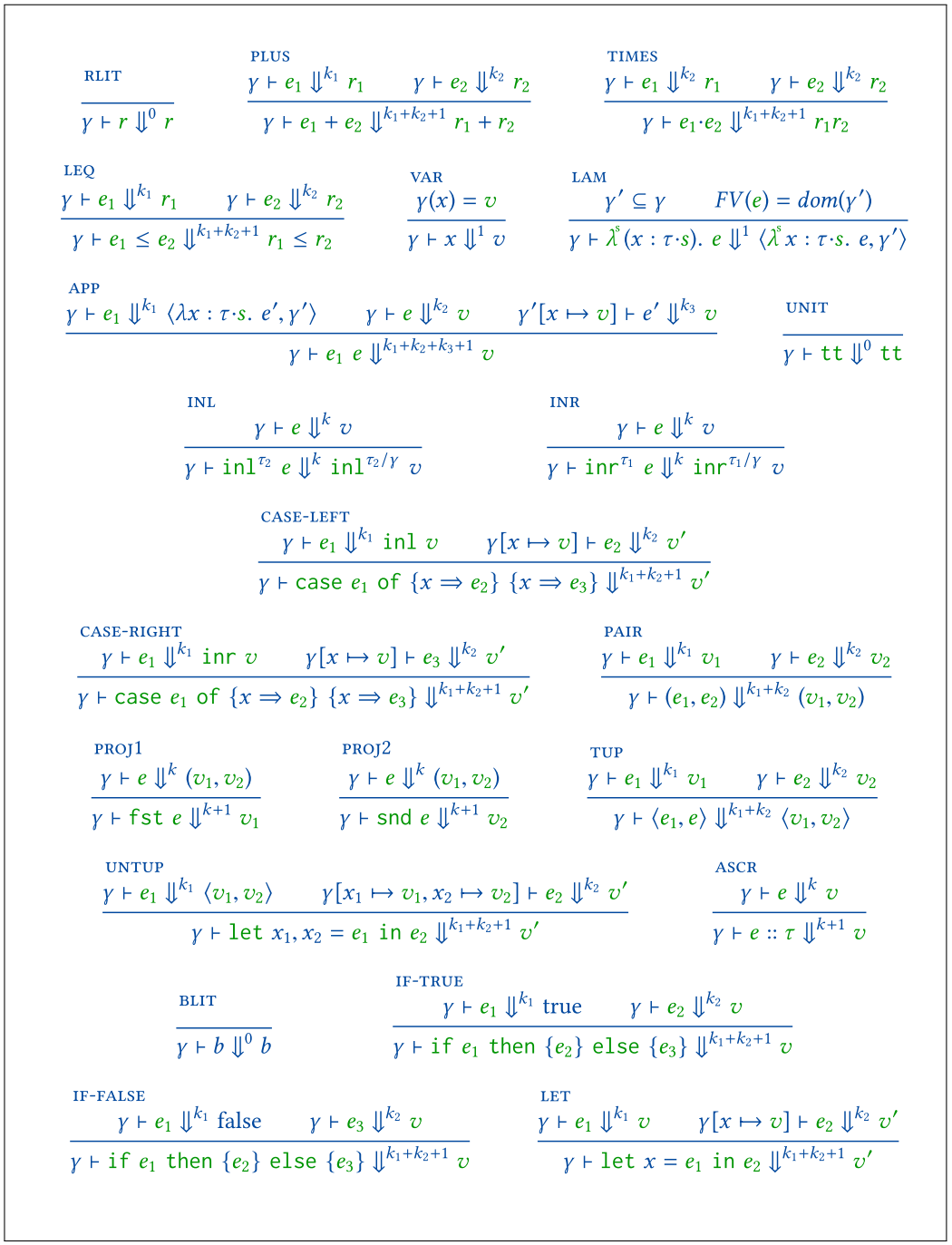

Fig. 27. λ_J: Sensitivity dynamic semantics.

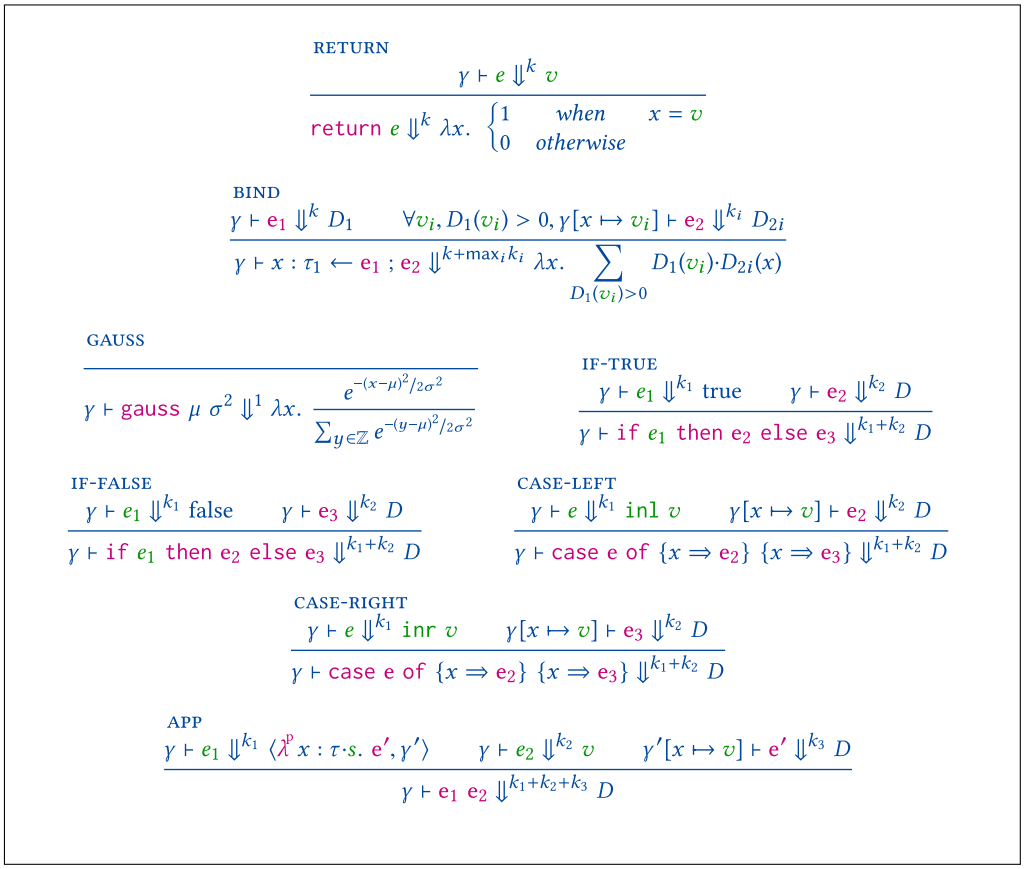

Fig. 28. λ_J: Probabilistic semantics.

C λ_J: SOUNDNESS

In this section, we present auxiliary definitions used in Section 7.3 and the proof of the fundamental property. Figure 29 presents the join and meet operators for sensitivity environments and sensitivities $\Sigma + s$. Figure 30 presents the subtyping relation between sensible types.

$$
\begin{aligned}
\varnothing \sqcup \varnothing &= \varnothing \\
(\Sigma + sx) \sqcup (\Sigma' + s'x) &= (\Sigma \sqcup \Sigma') + (s \sqcup s')x \; x \notin dom(\Sigma \sqcup \Sigma') \\
\Sigma \sqcup (\Sigma' + sx) &= (\Sigma \sqcup \Sigma') + sx \qquad (x \notin dom(\Sigma)) \\
(\Sigma + sx) \sqcup \Sigma' &= (\Sigma \sqcup \Sigma') + sx \qquad (x \notin dom(\Sigma')) \\
(\Sigma + s) \sqcup (\Sigma' + s') &= (\Sigma \sqcup \Sigma') + (s \sqcup s') \qquad s'' \notin \Sigma \sqcup \Sigma' \\
\_ \sqcup \_ &: \text{sensv} \times sensv \to sensv \\
\varnothing \sqcap \varnothing &= \varnothing \\
(\Sigma + sx) \sqcap (\Sigma' + s'x) &= (\Sigma \sqcap \Sigma') + (s \sqcap s')x \; x \notin dom(\Sigma \sqcap \Sigma') \\
\Sigma \sqcap (\Sigma' + sx) &= (\Sigma \sqcap \Sigma') + sx \qquad (x \notin dom(\Sigma)) \\
(\Sigma + sx) \sqcap \Sigma' &= (\Sigma \sqcap \Sigma') + sx \qquad (x \notin dom(\Sigma')) \\
(\Sigma + s) \sqcap (\Sigma' + s') &= (\Sigma \sqcap \Sigma') + (s \sqcap s') \qquad s'' \notin \Sigma \sqcap \Sigma'
\end{aligned}
$$

Fig. 29. Join and Meet of sensitivity environment and sensitivities.

$\boxed{\sigma <: \sigma}$

BASE
$$\frac{\sigma \in \{\mathbb{R}, \texttt{unit}\}}{\sigma <: \sigma}$$

S-FUN
$$\frac{\sigma_1' <: \sigma_1 \qquad s' <: s \qquad \Sigma_1 <: \Sigma_2' \qquad \Sigma_2 <: \Sigma_2' \qquad \sigma_2 <: \sigma_2'}{(x : \sigma_1 \cdot s) \xrightarrow{\Sigma_1 + \Sigma \cdot \Sigma_2} \sigma_2 \;<:\; (x : \sigma_1' \cdot s') \xrightarrow{\Sigma_1' + \Sigma \cdot \Sigma_2'} \sigma_2'}$$

P-FUN
$$\frac{\sigma_1' <: \sigma_1 \qquad s' <: s \qquad \Sigma_1 <: \Sigma_2' \qquad \Sigma_2 <: \Sigma_2' \qquad \sigma_2 <: \sigma_2'}{(x : \sigma_1 \cdot s) \xrightarrow{\Sigma_1 + \Sigma \bullet \Sigma_2} \sigma_2 \;<:\; (x : \sigma_1' \cdot s') \xrightarrow{\Sigma_1' + \Sigma \bullet \Sigma_2'} \sigma_2'}$$

SUM
$$\frac{\sigma_1 <: \sigma_1' \qquad s_1 \leq s_1' \qquad \sigma_2 <: \sigma_2' \qquad s_2 \leq s_2'}{\sigma_1 \; {}^{s_1}\!\oplus^{s_2} \sigma_2 \;<:\; \sigma_1' \; {}^{s_1'}\!\oplus^{s_2'} \sigma_2'}$$

PAIR
$$\frac{\sigma_1 <: \sigma_1' \qquad s_1 \leq s_1' \qquad \sigma_2 <: \sigma_2' \qquad s_2 \leq s_2'}{\sigma_1 \; {}^{s_1}\!\&^{s_2} \sigma_2 \;<:\; \sigma_1' \; {}^{s_1'}\!\&^{s_2'} \sigma_2'}$$

TUP
$$\frac{\sigma_1 <: \sigma_1' \qquad s_1 \leq s_1' \qquad \sigma_2 <: \sigma_2' \qquad s_2 \leq s_2'}{\sigma_1 \; {}^{s_1}\!\otimes^{s_2} \sigma_2 \;<:\; \sigma_1' \; {}^{s_1'}\!\otimes^{s_2'} \sigma_2'}$$

Fig. 30. Subtyping of sensible types.

REFERENCES

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830. https://scikit-hep.org/citing.

[2] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 308–318.

[3] Amal Ahmed. 2004. *Semantics of Types for Mutable State*. Ph.D. Dissertation. Princeton University.

[4] Amal Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27–28, 2006, Proceedings (Lecture Notes in Computer Science)*, Vol. 3924. Springer, 69–83.

[5] Aws Albarghouthi and Justin Hsu. 2018. Synthesizing coupling proofs of differential privacy. *Proc. ACM Program. Lang.* 2, POPL (2018), 58:1–58:30. DOI: https://doi.org/10.1145/3158146

[6] Andrew W. Appel and David McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (Sept. 2001), 657–683.

[7] Borja Balle and Yu-Xiang Wang. 2018. Improving the Gaussian mechanism for differential privacy: Analytical calibration and optimal denoising. In *Proceedings of the International Conference on Machine Learning*. PMLR, 394–403.

[8] Yuyan Bao, Guannan Wei, Oliver Bracevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: Tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–32.

[9] Gilles Barthe, Rohit Chadha, Vishal Jagannath, A. Prasad Sistla, and Mahesh Viswanathan. 2020. Deciding differential privacy for programs with finite inputs and outputs. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. 141–154.

[10] Gilles Barthe, Rohit Chadha, Paul Krogmeier, A. Prasad Sistla, and Mahesh Viswanathan. 2021. Deciding accuracy of differential privacy schemes. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30.

[11] Gilles Barthe, George Danezis, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. 2013. Verified computational differential privacy with applications to smart metering. In *Proceedings of the IEEE 26th Computer Security Foundations Symposium*. IEEE Computer Society, 287–301. DOI: https://doi.org/10.1109/CSF.2013.26

[12] Gilles Barthe, Thomas Espitau, Justin Hsu, Tetsuya Sato, and Pierre-Yves Strub. 2019. Relational ★★\star-liftings for differential privacy. *Logic. Meth. Comput. Sci.* 15, 4 (2019). Retrieved from https://lmcs.episciences.org/5989.

[13] G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, C. Kunz, and P. Strub. 2014. Proving differential privacy in Hoare logic. In *Proceedings of the IEEE 27th Computer Security Foundations Symposium*. 411–424. DOI: https://doi.org/10.1109/CSF.2014.36

[14] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*. Association for Computing Machinery, New York, NY, 55–68. DOI: https://doi.org/10.1145/2676726.2677000

[15] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving differential privacy via probabilistic couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'16)*. Association for Computing Machinery, New York, NY, 749–758. DOI: https://doi.org/10.1145/2933575.2934554

[16] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. Association for Computing Machinery, New York, NY, 97–110. DOI: https://doi.org/10.1145/2103656.2103670

[17] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. 2013. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst.* 35, 3, (Nov. 2013). DOI: https://doi.org/10.1145/2492061

[18] Raef Bassily, Adam Smith, and Abhradeep Thakurta. 2014. Private empirical risk minimization: Efficient algorithms and tight error bounds. In *Proceedings of the IEEE 55th Annual Symposium on Foundations of Computer Science (FOCS'14)*. IEEE, 464–473.

[19] Benjamin Bichsel, Timon Gehr, Dana Drachsler-Cohen, Petar Tsankov, and Martin Vechev. 2018. DP-finder: Finding differential privacy violations by sampling and optimization. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 508–524.

[20] Mark Bun, Cynthia Dwork, Guy N. Rothblum, and Thomas Steinke. 2018. Composable and versatile privacy via truncated CDP. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*. 74–86.

[21] Mark Bun and Thomas Steinke. 2016. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In *Proceedings of the Theory of Cryptography Conference*. Springer, 635–658.

[22] Clément L. Canonne, Gautam Kamath, and Thomas Steinke. 2020. The discrete Gaussian for differential privacy. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS'20)*.

[23] Kamalika Chaudhuri and Staal A. Vinterbo. 2013. A stability-based validation procedure for differentially private machine learning. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 2652–2660.

[24] Ezgi Çiçek, Weihao Qu, Gilles Barthe, Marco Gaboardi, and Deepak Garg. 2019. Bidirectional type checking for relational properties. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 533–547. DOI: https://doi.org/10.1145/3314221.3314603

[25] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (Sept. 2010), 1157–1210.

[26] Arthur Azevedo de Amorim, Emilio Jesús Gallego Arias, Marco Gaboardi, and Justin Hsu. 2015. Really natural linear indexed type checking. *CoRR* abs/1503.04522 (2015).

[27] Arthur Azevedo de Amorim, Marco Gaboardi, Emilio Jesús Gallego Arias, and Justin Hsu. 2014. Really natural linear indexed type checking. In *Proceedings of the 26th International Symposium on Implementation and Application of Functional Languages (IFL'14)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2746325.2746335

[28] Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A semantic account of metric preservation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 545–556.

[29] Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2019. Probabilistic relational reasoning via metrics. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*. IEEE, 1–19.

[30] Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. 2018. Detecting violations of differential privacy. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 475–489.

[31] Cynthia Dwork, Moni Naor, Omer Reingold, Guy N. Rothblum, and Salil Vadhan. 2009. On the complexity of differentially private data release: Efficient algorithms and hardness results. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC'09)*. Association for Computing Machinery, New York, NY, 381–390. DOI: https://doi.org/10.1145/1536414.1536467

[32] Cynthia Dwork and Aaron Roth. 2014. The algorithmic foundations of differential privacy. *Found Trends® Theoret. Comput. Sci.* 9, 3–4 (2014), 211–407.

[33] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently typed programming with singletons. In *Proceedings of the Haskell Symposium*. Association for Computing Machinery, New York, NY, 117–130.

[34] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 357–370.

[35] Marco Gaboardi, Michael Hay, and Salil Vadhan. 2020. A programming framework for OpenDP. *6th Workshop on the Theory and Practice of Differential Privacy (TPDP'20)*. https://salil.seas.harvard.edu/publications/programming-framework-opendp.

[36] David K. Gifford and John M. Lucassen. 1986. Integrating functional and imperative programming. In *Proceedings of the ACM Conference on Lisp and Functional Programming*. 28–38.

[37] J. Hannan, P. Hicks, and D. Liben-Nowell. 1997. *A Lifetime Analysis for Higher-order Languages*. Technical Report. Pennsylvania State University.

[38] Moritz Hardt, Katrina Ligett, and Frank McSherry. 2012. A simple and practical algorithm for differentially private data release. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 2339–2347.

[39] Susumu Hayashi. 1991. Singleton, union and intersection types for program extraction. In *Theoretical Aspects of Computer Software, International Conference TACS'91, Sendai, Japan, September 24–27, 1991, Proceedings (Lecture Notes in Computer Science)*, Takayasu Ito and Albert R. Meyer (Eds.), Vol. 526. Springer, 701–730.

[40] Xavier Leroy. 1992. *Polymorphic Typing of an Algorithmic Language*. Technical Report Research Report 1778. INRIA.

[41] Min Lyu, Dong Su, and Ninghui Li. 2017. Understanding the sparse vector technique for differential privacy. *PVLDB* 10, 6 (2017), 637–648. DOI: https://doi.org/10.14778/3055330.3055331

[42] Frank McSherry. 2009. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 19–30. DOI: https://doi.org/10.1145/1559845.1559850

[43] Ilya Mironov. 2012. On significance of the least significant bits for differential privacy. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'12)*. ACM, 650–661.

[44] Ilya Mironov. 2017. Rényi differential privacy. In *Proceedings of the IEEE 30th Computer Security Foundations Symposium (CSF'17)*. IEEE, 263–275.

[45] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18, 5-6 (2008), 865–911.

[46] Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan et al. 2019. Duet: An expressive higher-order language and linear type system for statically enforcing differential privacy. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 1–30.

[47] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP (2019), 110:1–110:30.

[48] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. 2014. Coeffects: A calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ACM, 123–135.
[49] Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: A calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. 157–168.
[50] T. Sato, G. Barthe, M. Gaboardi, J. Hsu, and S. Katsumata. 2019. Approximate span liftings: Compositional semantics for relaxations of differential privacy. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*. 1–14. DOI: https://doi.org/10.1109/LICS.2019.8785668
[51] Gabriel Scherer and Jan Hoffmann. 2013. Tracking data-flow with open closure types. In *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'13) (Lectures Notes in Computer Science)*. Springer, 710–726.
[52] Kunal Talwar, Abhradeep Guha Thakurta, and Li Zhang. 2015. Nearly optimal private lasso. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 3025–3033.
[53] Om Thakkar, Galen Andrew, and H. Brendan McMahan. 2019. Differentially private learning with adaptive clipping. *arXiv preprint arXiv:1905.03871* (2019).
[54] Matías Toro, David Darais, Chike Abuah, Joseph P. Near, Damián Árquez, Federico Olmedo, and Éric Tanter. 2023. *Contextual Linear Types for Differential Privacy — Extended with Proofs*. Technical Report TR/DCC-2023-1. University of Chile.
[55] Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. 2019. Proving differential privacy with shadow execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 655–669.
[56] Royce J. Wilson, Celia Yuxing Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. 2020. Differentially private SQL with bounded user contribution. *Proc. Privac. Enhanc. Technol.* 2020, 2 (2020).
[57] Xi Wu, Fengan Li, Arun Kumar, Kamalika Chaudhuri, Somesh Jha, and Jeffrey Naughton. 2017. Bolt-on differential privacy for scalable stochastic gradient descent-based analytics. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'17)*. ACM, New York, NY, 1307–1322. DOI: https://doi.org/10.1145/3035918.3064047
[58] Ashkan Yousefpour, Igor Shilov, Alexandre Sablayrolles, Davide Testuggine, Karthik Prasad, Mani Malek, John Nguyen, Sayan Ghosh, Akash Bharadwaj, Jessica Zhao, et al. 2021. Opacus: User-friendly differential privacy library in PyTorch. *arXiv preprint arXiv:2109.12298* (2021).
[59] Danfeng Zhang and Daniel Kifer. 2017. LightDP: Towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 888–901.
[60] Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C. Pierce, and Aaron Roth. 2019. Fuzzi: A three-level logic for differential privacy. *Proc. ACM Program. Lang.* 3, ICFP (2019), 1–28.
[61] Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C. Pierce, and Aaron Roth. 2020. Testing differential privacy with dual interpreters. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 1–26.

Received 22 February 2021; revised 21 February 2023; accepted 25 February 2023