Effectful Program Distancing

UGO DAL LAGO, University of Bologna, Italy and INRIA, France
FRANCESCO GAVAZZO, University of Bologna, Italy and INRIA, France

Semantics is traditionally concerned with program equivalence, in which all pairs of programs which are *not* equivalent are treated the same, and simply dubbed as incomparable. In recent years, various forms of program *metrics* have been introduced such that the distance between non-equivalent programs is measured as an element of an appropriate quantale. By letting the underlying quantale *vary* as the type of the compared programs become more complex, the recently introduced framework of differential logical relations allows for a new contextual form of reasoning. In this paper, we show that all this can be generalised to *effectful* higher-order programs, in which not only the *values*, but also the *effects* computations produce can be appropriately distanced in a principled way. We show that the resulting framework is flexible, allowing various forms of effects to be handled, and that it provides compact and informative judgments about program differences.

CCS Concepts: • **Theory of computation** → **Program semantics**; • **Software and its engineering** → **General programming languages**.

Additional Key Words and Phrases: Logical Relations, Program Distances, Lambda Calculus, Monads, Metrics

ACM Reference Format:
Ugo Dal Lago and Francesco Gavazzo. 2022. Effectful Program Distancing. *Proc. ACM Program. Lang.* 6, POPL, Article 19 (January 2022), 30 pages. https://doi.org/10.1145/3498680

1 INTRODUCTION

Program semantics and program equivalence go hand in hand: every way of attributing meaning to programs induces a notion of equivalence, and every form of program equivalence is implicitly a way to attribute semantics to programs. Up to a certain extent, one can well say that program semantics is *the science* of program equivalence.

What defines the meaning of a program **P**, however, is not just the set of programs which behave *exactly the same* as **P**. To properly understand the nature of **P**, one also needs to know how **P** relates to programs which are *not* equivalent to it. Without this information, one could not justify program transformations which are just approximately correct, or judge to which extent a program satisfies a specification which turns out to be too stringent to be met by any concrete program. This is manifested by fields like approximate computing [Mittal 2016] where correctness is traded for efficiency, and small errors are introduced into programs for the purpose of improving their performances [Carbin et al. 2012; Rinard 2011; Sidiroglou-Douskos et al. 2011].

A dual but conceptually related problem is the one of comparing the behaviour of a single program faced with non-equivalent inputs. The classic theory of program equivalence (think about substitutivity, the testbed of any notion of program equivalence) establishes that any program should result in equivalent outputs whenever fed with *equivalent* inputs. Often, however, one is interested in the behaviour of the program on *non*-equivalent (although "similar") inputs, and in

Authors' addresses: Ugo Dal Lago, University of Bologna, Italy and INRIA, Inria Sophia Méditerranée, Sophia Antipolis, France, ugo.dallago@unibo.it; Francesco Gavazzo, University of Bologna, Italy and INRIA, Inria Sophia Méditerranée, Sophia Antipolis, France, francesco.gavazzo2@unibo.it.

2475-1421/2022/1-ART19
https://doi.org/10.1145/3498680

how differences on the input affect the output. A prime example of this interest is given by the fruitful interaction between programming language theory and differential privacy [Barthe et al. 2018; Chaudhuri et al. 2010, 2011; Reed and Pierce 2010], where notions like *program sensitivity*, *continuity*, and *robustness* describing how changes in inputs lead to *bounded* changes in the output play a central role.

But what exactly renders an approximate program transformation acceptable? When could we say that a given program is robust enough to changes in its inputs? Satisfactory answers to this questions are invariably quantitative: one needs to measure *how far* we are from the desired target, possibly checking that the distance, measured as a *number*, is not beyond a certain, acceptable threshold. This is the path program semantics have been taking recently, in which various notions of program metrics have been defined for programs and processes [Chatzikokolakis et al. 2014; de Amorim et al. 2017; Desharnais et al. 2004; Du et al. 2016; Gebler et al. 2016; Van Breugel and Worrell 2005]. When applied to higher-order languages, however, program metrics come with some difficulties, the main one being that compositional reasoning about program metrics requires to have specific quantitative information on *how much* programs are used [Crubillé and Dal Lago 2015; Gebler et al. 2016; Reed and Pierce 2010]. This often results in notions of metric defined on complex structures [Crubillé and Dal Lago 2017] or in notions of metric tailored to specific (usually linear) type systems tracking the desired information and reflecting the *symmetric monoidal closed* structure of categories of metric spaces [de Amorim et al. 2017, 2019; Gavazzo 2018; Reed and Pierce 2010].

Additionally, as observed by Westbrook and Chaudhuri [2013], program metrics are often inadequate as ways to quantitatively compare programs, precisely because a number (or, more generally, an element of a fixed quantale [Flagg and Kopperman 1997; Gavazzo 2018, 2019; Hofmann et al. 2014; Lawvere 1973]) measures the distance between any pair of programs (and any pair of program inputs) independently of their interactive nature. A purely numerical distance is indeed perfectly adequate to compare inert programs, which do not interact with their environment. But as soon as the behaviour of the compared programs becomes *context-dependent*, taking just a number often leads to an unacceptable loss of information. For instance, a well-known [Abelson and Sussman 1996] procedure to *approximately* compute the sine of an angle is to rely on the approximation $x \approx \sin x$, for x sufficiently close to 0, and the trigonometric identity $\sin x = 3 \sin \frac{x}{3} - 4 \sin^3 \frac{x}{3}$ to reduce the size of the argument of sin. Can the standard theory of program metrics justify this program approximation? If we let the distance between two functions be *a number* computed by testing the functions against *any* possible input, then we cannot even justify the approximation $x \approx \sin x$, as the latter holds only if the environment passes to the functions inputs *close to* 0.

A generalization of logical relations called *differential logical relations* has recently been introduced [Dal Lago and Gavazzo 2020, 2021a; Dal Lago et al. 2019] in which the distance between two programs has a type which depends on the type of the compared objects, thus overcoming the limitations metrics suffer from. Differential logical relations associate to each type of the underlying language a space of (non-necessarily numerical) semantic distances reflecting the interactive complexity of programs of that type. Thus, for instance, a distance between functions is itself a function describing how distances between inputs turn into distances between outputs. Moving from fixed, numerical distances to distances reflecting the interactive complexity of programs allows one to account for the role played by the environment when analysing programs, this way obtaining a compositional, context-dependent notion of program distance.

Dal Lago et al. [2019] introduced differential logical relations for a *pure* simply typed language and left open the question of whether differential logical relations could be extended to richer languages. Moving from this question, this paper's contributions are twofold:

- First of all, the notion of *generalized distance space* is introduced as a minimalistic generalization on the notion of a metric space allowing for the computation of differences between elements of the underlying carrier. Generalized distance spaces form a category constituting a semantic foundation for effectful program distancing, through monadic extensions. This is in Section 3.
- On the one hand, the metatheory of differential logical relations is substantially developed by improving the existing theory of *pure* differential logical relations through generalized distance spaces to monadic languages, this way providing the first example of a truly compositional differential analysis of higher-order effectful programs. This is described in Section 4.
- On the other hand, some programs examples coming from diverse application areas like cost analysis, approximate computing, and program sensitivity, are analysed through effectful differential logical relations, obtaining this way some precise relational information. In all these scenarios, crucial to the analysis is the role played by the environment, which makes ordinary, metric-like reasoning ineffective. This is in Section 5.

2 ON PROGRAM DISTANCING AND (DIFFERENTIAL) LOGICAL RELATIONS: AN INTRODUCTION

Modular Relational Reasoning. Software development is a complex task, which needs to be supported at various levels, and in particular by the underlying programming language(s). Any reasonable high-level programming language is built around linguistic features supporting *modular* development, such as procedures, functions, classes, components, services, etc. This influences the way programmers think about the programs they are writing, the way the program is organized, and ultimately what a *module* stands for in a given context. In such a scenario, developing a software system can be done by many people, independently, being guided by a formal or informal *specification* of how each module is supposed to *behave*. There is more: each module **A** can be replaced by another one, call it **B**, provided the former has the *same* behaviour as **B**, or at least that it refines it, see Figure 1. Please observe that if **A** (and thus **B**) are highly interactive modules, then even defining what it means for them to be equivalent can be nontrivial: think about concurrent or higher-order programs. In the case of higher-order programming, one of the most successful ways of supporting such modular reasoning is the one provided by logical relations [Plotkin 1973], which can be spelled out as follows. Each type σ of the language is interpreted as a (syntax-based) relational structure $(Prog_\sigma, \sim_\sigma)$ over the set $Prog_\sigma$ of programs of type σ given by a (logical) equivalence $\sim_\sigma$ between such programs. The equivalence $\sim_\sigma$ is called the *relational interpretation* of σ. One defines logical equivalence for ground types (such as the types of integers or booleans) and then inductively extends it to complex types. The deep essence of this recipe is that the relational interpretation of a complex type $F(\sigma_1, \ldots, \sigma_n)$ is given by a so-called relational extension $\hat{F}$ of the (set-based) construction F to relations: accordingly, one defines the relational interpretation $\sim_{F(\sigma_1,\ldots,\sigma_n)}$ of $F(\sigma_1, \ldots, \sigma_n)$ as $\hat{F}(\sim_{\sigma_1}, \ldots, \sim_{\sigma_n})$. Depending on the structure of F (functor, monad, comonad, etc), the relational extension $\hat{F}$ is required to satisfy some structural properties reflecting those of F. Relying on such structural properties, one proves the so-called Fundamental Lemma of logical relations, which states that logical equivalence is reflexive, with the corollary that programs are invariant under logical equivalence.[1] Additionally, from the Fundamental Lemma it also follows that logical equivalence is a congruence, so that it supports compositional reasoning about program behaviour.

Dealing with Non-Equivalent Programs. But what if **A** is not equivalent to **B**, nor a refinement of it? Well, replacing **A** with **B** can in this case have dramatic consequences on the behaviour of

[1] Semantically, that means that programs behave as a morphisms between relational structures.

the full system. Indeed, even a little discrepancy in behaviour could somehow be exploited by the underlying context **C**, resulting in two systems behaving in completely different ways. But this is not always the case: two modules **A** and **B** which behave in uncorrelated ways *only in certain situations* could result in two overall systems which are highly similar *if* those situations *do not happen*, something which of course heavily depends on the way **C** (and not just **A** and **B**) behaves. In other words, equivalence-breaking program transformations make the whole task of developing correct software much harder, and fundamentally different from the aforementioned classic scenario: everything is now *quantitative*, since errors and discrepancies need to be quantified, and *contextual*, because the transformation can only be justified by looking at the environment, or at least at how it interacts with the module and with the outside world.

Actually, reasoning about non-equivalent programs is not at all something exotic, and it is becoming the norm rather than the exception in many fields. In approximate computing [Mittal 2016], for example, one trades correctness for efficiency, replacing a module with an approximately correct, but more efficient one. Of course, this makes sense only if the environment is benign, i.e., if it queries the approximate program at a point in which it is *just slightly* different from the original one, and/or if it is not too sensitive to changes. In differential privacy, as another example, one is interested in a somehow dual situation in which the program is a single, while the difference is in the input, and one is interested in keeping the sensitivity of the former under control. Even there, it can well be that around certain inputs the sensitivity is greater than around others. Could we account for this fundamentally richer scenario in relational semantics?

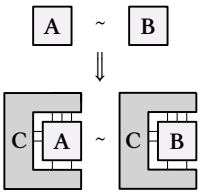

Fig. 1. Program equivalence as a congruence.

From Metrics to Differentials. The exact same schema of relational semantics can indeed be extended to richer notions of relations, such as quantitative ternary relations $\sim_\sigma \subseteq [0,\infty] \times Prog_\sigma \times Prog_\sigma$ akin to program metrics.[2] The relational interpretation of types as quantitative ternary relations (often referred to as *metric logical relations*) follows the same schema used for binary relations, and several relational extensions have been extensively studied in that setting, including functorial and monadic ones [Baldan et al. 2018; Clementino et al. 2004; de Amorim et al. 2017, 2019; Gavazzo 2018; Hofmann 2007]. The drawback of such metric-like relational semantics is that the underlying category of ternary relational structures (or, equivalently, the category of metric spaces) is not cartesian closed, with the strong consequence that it is just not possible to give a nontrivial metric semantics to even simple programming languages. Researchers overcame the problem by defining suitable, linear-like type systems supporting a well-behaved metric semantics in the style of metric logical relations [Reed and Pierce 2010].

This solution is only partially satisfactory, due to the severe limitations of the type systems supporting a (metric-like) relational interpretation. More importantly, even on those languages whose type system support such interpretations, the latter lack the capability of accounting for the context's behavior. To keep our argument concrete, let us consider some approximate program transformations borrowed from the work by Zhang et al. [2014], which consists in replacing a hardware unit **ISR** computing the reciprocal function $x \mapsto \frac{1}{\sqrt{x}}$ by another one, call it **LA**, computing the linear function $x \mapsto 2.08 - 1.1911x$ (Figure 2). For obvious reasons, this produces a small enough error only if x has certain values. More specifically, it is claimed [Zhang et al. 2014] that if $x \in [0.5, 1]$, then the relative error ε is below 11.11%. The way the approximation is constructed is motivated by hardware requirements, themselves driven by energy consumption and performance

[2]Formally, there is a one-to-one correspondence between metric-like functions $\delta : Prog_\sigma \times Prog_\sigma \to [0,\infty]$ and ternary relations $R \subseteq [0,\infty] \times Prog_\sigma \times Prog_\sigma$ that are monotone and meet-preserving in the first argument [Hofmann et al. 2014].

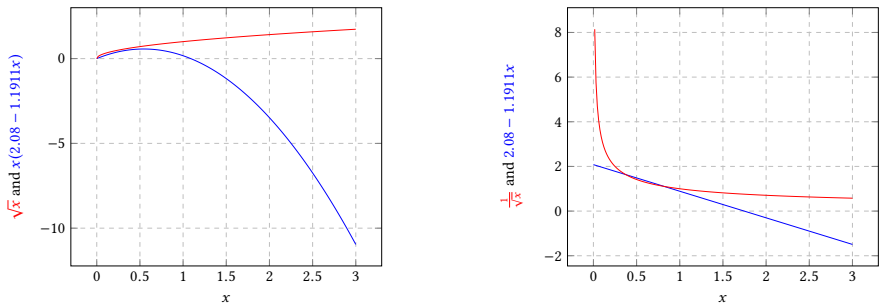

Fig. 2. Two Approximate Program Transformation

requirements. What is crucial for us here, rather, is what happens if such a program transformation is applied inside a complex software project which possibly includes higher-order functions and effects, thus being highly interactive. How could such a program transformation be *justified*? The two programs are clearly not equivalent, so classic program semantic methodologies are simply useless here. How about program metrics? The most common way of comparing two programs **A** and **B**, e.g. of functional type **Real** $\to$ **Real**, is to stipulate that:

$$\mathbf{A} \sim_a \mathbf{B} \quad \Leftrightarrow \quad \forall x.a \geq |\mathbf{A}(x) - \mathbf{B}(x)|.$$

In the case of **ISR** and **LA** as above, we have that **ISR** $\sim_a$ **LA** only when $a = \infty$, i.e. **ISR** and **LA** are at maximal distance.

A paradigm shift seems necessary here: we have to be able to keep track of the discrepancies between the behaviour of **ISR** and **LA** *when called on the different inputs*. How can we do that? Clearly, the distance between programs of type σ cannot be just an element of the interval $[0, \infty]$, but could in general be more complex, depending on the structure of σ. In other words, $\sim_\sigma$ becomes a subset of $Diff_\sigma \times Prog_\sigma \times Prog_\sigma$, where $Diff_\sigma$ varies depending on σ. In particular, any element $\mathrm{d}f$ of $Diff_{\mathbf{Real}\to\mathbf{Real}}$ is a function taking in input a real number x and an error $\mathrm{d}x \in [0, \infty]$ bounding the distance $|x - y|$ between two inputs, and returning an error $\mathrm{d}f(x, \mathrm{d}x)$ bounding the distance $|\mathbf{A}(x) - \mathbf{B}(y)|$ between $\mathbf{A}(x)$ and $\mathbf{B}(y)$, where **A** and **B** are the two programs being compared. For instance, we will see that a distance between the two programs **ISR** and **LA** introduced above is given by $\mathrm{d}f(x, \mathrm{d}x) \triangleq \sup_{|x-y|\leq \mathrm{d}x} |\frac{1}{\sqrt{x}} - 2.08 + 1.1911y|$. Notice that $\mathrm{d}f$ being a function, it can be used to reason about **ISR** and **LA** on *all* inputs, this way providing a modular, *truly input-dependent* notion of distance. Notice also that the same kind of reasoning allows us to find a suitable distance $\mathrm{d}g$ between (programs computing) $\sqrt{x}$ and $x(2.08 - 1.1911x)$ (Figure 2), namely $\mathrm{d}g(x, \mathrm{d}x) \triangleq \sup_{|x-y|\leq \mathrm{d}x} |\sqrt{x} - 2.08y + 1.1911y^2|$. What we have just described is exactly the main idea behind differential logical relations [Dal Lago et al. 2019].

But now that we have a distance $\mathrm{d}f$ between **ISR** and **LA**, what can we say about the approximate program transformation replacing **ISR** with **LA** *inside a complex piece of software* **C**? One of the main features of differential logical relations is that programs have (in general) *non-null self-distances*. That is, any program **P** comes with *at least one* self-distance $\mathrm{d}\mathbf{P}$ producing a generalised *sensitivity* or *differential analysis* for **P**: in fact, any such $\mathrm{d}\mathbf{P}$ tells us how **P** reacts to different inputs. If we now think about the context/environment **C** as a higher-order function taking in input a real-valued function, then we can compute the impact of replacing **ISR** with **LA** inside **C** *compositionally* as $\mathrm{d}\mathbf{C}(\mathbf{ISR}, \mathrm{d}f)$. Notice how such an analysis consists of two, decoupled analyses. On the one hand, we have the difference $\mathrm{d}f$ between **ISR** and **LA** which has been obtained by examining the behaviours

of **ISR** and **LA** independently of the environment **C**. On the other hand, we have the sensitivity analysis of **C** which, again, is independent of the specific input **ISR** (or **LA**) we pass to **C**. In other words, modularity is preserved.

Going Effectful. So far we have considered differences on *pure* higher-order programs, and this is where the state of the art is. But what happens if the compared programs are *effectful*? Suppose, for example, that the compared programs are randomized [Barthe et al. 2018; Misailovic et al. 2011] or that their performances rather than their value, need to be compared, like in relational cost analysis [Çiçek et al. 2017; Qu et al. 2019; Radicek et al. 2018]. How could we proceed? Would it be possible to generalise the theory of differential logical relations to the new setting? This is precisely the question this paper gives an answer to, and as such deserves to be discussed, even if just informally and by way of an example.

Suppose we consider a combinator for the numerical integration of functions, which in a higher-order language could take the form of a program **integral** of type

$$(\mathbf{Real} \to \mathbf{Real}) \to \mathbf{Real} \to \mathbf{Real} \to \mathbf{Real}.$$

The program computes an approximation of the integral of the input function f (the first parameter) between a and b (the second and third parameters) based on a simple quadrature formula. However implemented, this method gives rise to a certain approximation error, which obviously depends on the input values and which as such can be captured through differential logical relations as we have introduced them in the previous paragraph. But what happens if the underlying quadrature algorithm is *randomized*, and chooses the intervals to consider on the basis of some carefully chosen probabilistic distributions? How could the impact of this randomization be assessed? A naive answer to this question may be given in the general setting of monadic effects by saying that the distance between monadic objects must be given by a monadic distance. This idea certainly has a *raison d'etre*: to every possible input and error in the input, such an algorithm would give rise to *a distribution of errors* in the output, which is, for instance, in a probabilistic coupling relation with the output distributions [Villani 2008]. But is this generally true of all monads? As we will see from the next section, the problem is more complex than it appears. As an example, output or cost monads does not give rise to distances the same way and it seems that a unique pattern for effectful distances is missing. We thus take a different, more structural perspective on the problem and define an axiomatics characterising those distances and distance spaces that qualify as distances between monadic objects: we call objects satisfying our axiomatics *differential extensions of monads*. This is what we are going to talk about in the next session. By the way, we will extensively talk about the program **integral** in Section 5.

3 GENERALISED DISTANCE SPACES AND THEIR MONADIC EXTENSIONS

In this section, we define the semantic foundations for our theory of effectul distances. Such a foundation actively uses the notion of a monad, which we recall for the sake of completeness. A monad [MacLane 1971] (tacitly assumed to be on the category of sets and functions) is a triple $\mathcal{T} = (T, \eta, \gg\!=)$, where T associates to each set A a set $T(A)$ of T-computations, η (called *unit* of T) is a set-indexed family of functions $\eta_A : A \to T(A)$, and for any function $f : A \to T(B)$ we have a function $\gg\!= f : T(A) \to T(B)$ (the operator $\gg\!=$ is called *bind*). Moreover, these data have to satisfy some suitable equational laws. As it is customary, we often write $\alpha \gg\!= f$ in place of $\gg\!= f(\alpha)$.

In this paper we will use the following monads as running examples [Moggi 1989]: (i) the Maybe monad $(\mathcal{M}, \eta, \gg\!=)$, where $\mathcal{M}(A) \triangleq \{just\ a \mid a \in A\} \cup \{\bot\}$, $\eta(a) \triangleq just\ a$, and $t \gg\!= f \triangleq f(a)$ if $t = just\ a$ and $\bot$ otherwise; (ii) the Writer monad $(\mathcal{W}, \eta, \gg\!=)$, where $(W, \cdot, \varepsilon)$ is a monoid, $\mathcal{W}(X) \triangleq W \times X$, $\eta(x) \triangleq (\varepsilon, x)$, and $(w, x) \gg\!= f \triangleq (w \cdot u, y)$, with $(u, y) = f(x)$; (iii) the powerset

monad $(\mathcal{P}, \eta, \gg=)$, where $\eta(x) \triangleq \{x\}$, and $\alpha \gg= f \triangleq \bigcup_{a \in \alpha} f(a)$; $(i v)$ the discrete distribution monad $(\mathcal{D}, \eta, \gg=)$, where $\mathcal{D}(X)$ is the collection of distributions $\mu : X \to [0,1]$ (so that $\sum_x \mu(x) = 1$) with countable support (recall that the support $\mathrm{supp}(\mu)$ of $\mu : X \to [0,1]$ is the set $\{x \mid \mu(x) > 0\}$, so that the expression $\sum_x \mu(x)$ is defined), $\eta(x) \triangleq \delta_x$ (Dirac distribution on x), and $\mu \gg= f \triangleq \sum_i p_i \cdot f(x_i)$, where $\sum_i p_i \cdot \delta_{x_i}$ is a representation of μ as convex sums of Dirac distributions.

3.1 The Category of Generalised Distance Spaces

The basic semantic element we will use to define the differential semantics of monadic higher-order programs is the notion of a *generalised distance space*, GDS for short. A GDS generalises the notion of a generalised metric domain [Dal Lago et al. 2019] and resembles semantic structures used to study incremental properties of programs, such as change actions [Alvarez-Picallo and Ong 2019] and change structures [Cai et al. 2014; Giarrusso 2018]. GDSs generalise quantitative relational (i.e. metric) structures, exactly as the latter generalise ordinary relational structures. As in quantitative relational structures, in GDSs distances between objects are described by means of ternary relations. Such relations, however, relate pairs of elements $x, y \in A$ not with numerical distances, but with generalised distances (just distances) $\mathrm{d}x \in (\!|A|\!)$.

Definition 3.1. A *generalised distance space* is a triple $\mathcal{A} = (A, (\!|A|\!), R)$, where $R \subseteq A \times (\!|A|\!) \times A$ is a ternary relation. We call $(\!|A|\!)$ the distance space of A and refer to relations R as differential relations.

Given a GDS $\mathcal{A} = (A, (\!|A|\!), R)$, the distance space $(\!|A|\!)$ contains distances for elements in A. Such a set depends on A and reflects the complexity of objects in A: for instance, if A is a set of functions Y^X, then $(\!|A|\!)$ is itself a set of functions, namely $(\!|Y|\!)^{X \times (\!|X|\!)}$. Finally, a GDS comes with a ternary relation $R \subseteq A \times (\!|A|\!) \times A$ with the intended meaning that whenever $R(x, \mathrm{d}x, y)$ holds, then $\mathrm{d}x$ is a distance from x to y.[3] Other possible informal interpretations of $R(x, \mathrm{d}x, y)$ can be obtained borrowing the vocabulary of incremental computing [Ramalingam and Reps 1993], approximate computing [Mittal 2016], or generalised differentiation [Alvarez-Picallo and Ong 2019]. Accordingly, we think about $\mathrm{d}x$ as a change from x to (an over-approximation of) y or as an error or perturbation of x. Let us now introduce morphisms between GDSs.

Definition 3.2. A morphism between GDSs $\mathcal{A} = (A, (\!|A|\!), R)$ and $\mathcal{B} = (B, (\!|B|\!), S)$ is a pair of maps $(f, \mathrm{d}f)$ where $f : A \to B$, $\mathrm{d}f : A \times (\!|A|\!) \to (\!|B|\!)$ such that:

$$R(x, \mathrm{d}x, y) \implies S(f(x), \mathrm{d}f(x, \mathrm{d}x), f(y)).$$

A morphism between GDSs is thus given by an ordinary function $f : A \to B$ together with a map $\mathrm{d}f$ that given an input $x \in A$ and a distance $\mathrm{d}x$ between x and y computes a distance between $f(x)$ and $f(y)$. Functions $\mathrm{d}f$ are sometimes called derivatives[4] of f since by interpreting $\mathrm{d}x$ as an error (or perturbation) in the input, we see that $\mathrm{d}f(x, \mathrm{d}x)$ gives the error (or perturbation) in the output. Notice that in general one can define more than one $\mathrm{d}f$ for a given map f. Given morphisms

$$\mathcal{A} \xrightarrow{(f, \mathrm{d}f)} \mathcal{B} \xrightarrow{(g, \mathrm{d}g)} C$$

we define the composition of $(f, \mathrm{d}f)$ and $(g, \mathrm{d}g)$ as the morphism $(g \circ f, \mathrm{d}g \odot \mathrm{d}f)$, where $g \circ f : A \to C$ is ordinary function composition and $\mathrm{d}g \odot \mathrm{d}f : A \times (\!|A|\!) \to (\!|C|\!)$ is defined relying on the so-called chain rule:

$$(\mathrm{d}g \odot \mathrm{d}f)(x, \mathrm{d}x) \triangleq \mathrm{d}g(f(x), \mathrm{d}f(x, \mathrm{d}x)).$$

[3] The relation R is in general not symmetric, so that a distance from x to y may not be a distance from y to x.

[4] A better name would be a difference since, as we will see in Example 1, maps $\mathrm{d}f$ generalise the notion of a *finite difference* of f [Richardson 1954].

Composition is associative and has as unit element $(1, \mathrm{d}1) : \mathcal{A} \to \mathcal{A}$, with $\mathrm{d}1(x, \mathrm{d}x) \triangleq \mathrm{d}x$, meaning that we have a category **GDS** whose objects are GDSs and whose arrows are GDS morphisms.

Our differential semantics will associate to each type a GDS on expressions of that type whose differential relation is given by means of a differential logical relation. To handle product and arrow types (as well as other types, such as sum types, which we have not included in our language), we rely on the cartesian closed structure of **GDS**.

Definition 3.3. Given GDSs $(A, |\!|A|\!|, R)$ and $(B, |\!|B|\!|, S)$, we define their product and exponential as the GDSs $(A \times B, |\!|A|\!| \times |\!|B|\!|, R \times S)$ and $(B^A, |\!|B|\!|^{A \times |\!|A|\!|}, [R \to S])$, respectively, where:

$$(R \times S)((x, z), (\mathrm{d}x, \mathrm{d}z), (y, w)) \overset{\triangle}{\iff} R(x, \mathrm{d}x, y) \ \&\ S(z, \mathrm{d}z, w)$$

$$[R \to S]\ (f, \mathrm{d}f, g) \overset{\triangle}{\iff} \forall x, y \in A, \mathrm{d}x \in |\!|A|\!|.(R(x, \mathrm{d}x, y) \implies S(f(x), \mathrm{d}f(x, \mathrm{d}x), g(y))).$$

Proposition 3.4. **GDS** *is cartesian closed.*

Example 1. Consider the GDS $\mathcal{R} = (\mathbb{R}, [0, \infty], R)$ where $R(x, \mathrm{d}x, y) \overset{\triangle}{\iff} \mathrm{d}x \geq d(x, y)$, for $d : \mathbb{R} \times \mathbb{R} \to [0, \infty]$. The map d associates a numerical distance[5] between real numbers, typical examples of those distances being the Euclidean distance and the asymmetric Euclidean distance.[6] The exponential of $\mathcal{R}$ with itself is the GDS $(\mathbb{R}^{\mathbb{R}}, [0, \infty]^{\mathbb{R} \times [0, \infty]}, [R \to R])$. In particular, we have

$$[R \to R]\ (f, \mathrm{d}f, g) \iff \forall x, \mathrm{d}x, y.(\mathrm{d}x \geq d(x, y) \implies \mathrm{d}f(x, \mathrm{d}x) \geq d(f(x), g(y))).$$

Let us now recall the approximate program transformation by Zhang et al. [2014] mapping a program computing the function $f(x) \triangleq \frac{1}{\sqrt{x}}$ to one computing $g(x) \triangleq 2.08 + 1.1911x$ that we have discussed in Section 2. Obviously, we can always take the map $\mathrm{d}f(x, \mathrm{d}x) \triangleq \infty$ as a distance between f and g. That, however, is not informative (see Section 4.1). Instead, easy calculations show that for $\mathrm{d}f(x, \mathrm{d}x) \triangleq \sup_{|x-y| \leq \mathrm{d}x} \left| \frac{1}{\sqrt{x}} - 2.08 + 1.1911y \right|$ we indeed have $[R \to R]\ (f, \mathrm{d}f, g)$. Similarly, recalling the approximation of the sine function in terms of the identity function for inputs closed to 0 mentioned in Section 1, we see that for $\mathrm{d}f(x, \mathrm{d}x) \triangleq |\sin(x) - x| + \mathrm{d}x$, we have $[R \to R]\ (\sin, \mathrm{d}f, 1)$. Actually, given any two real-valued functions f, g, the map $\Delta(f, g)(x, \mathrm{d}x) \triangleq \sup_{|x-y| \leq \mathrm{d}x} |f(x) - g(y)|$ gives a distance between f and g (i.e. $[R \to R]\ (f, \Delta(f, g), g)$). In particular, we can instantiate the above formula with $g = f$, hence obtaining that $\Delta(f, f)(x, \mathrm{d}x) \triangleq \sup_{|x-y| \leq \mathrm{d}x} |f(x) - f(y)|$ is a self-distance for f. Please observe the similarity between such a self-distance and the notion of a *finite difference* one meets in (discrete) calculus [Richardson 1954]. ⊠

Proposition 3.4 hints that **GDS** can be used to give a differential semantics to simply typed *pure* calculi. What we are interesting in, is to use GDSs to define a differential semantics for *effectful* languages. Given a monad $\mathcal{T} = (T, \eta, \gg\!=)$ modelling a notion of computational effect, effectful relational semantics are obtained relying on suitable constructions (known as *lax extensions* [Barr 1970; Hoffman 2015] or *relators* [Backhouse and Hoogendijk 1993; Bird and de Moor 1997; Thijs 1996]) extending binary relations $R \subseteq X \times Y$ to relations $\Gamma R \subseteq T(X) \times T(Y)$ in such a way that the unit and bind of $\mathcal{T}$ extend to morphisms between relational structures. For instance, if one considers as relational structures the category **EBRel** whose objects (X, R) are sets together with binary relations on them and morphisms $f : (X, R) \to (Y, S)$ are functions $f : X \to Y$ preserving relations (so that $x\ R\ y$ implies $f(x)\ S\ f(y)$), then a lax extension Γ of a monad $\mathcal{T}$ ensures that η extends to a map between relational structures — so that $\eta : (X, R) \to (TX, \Gamma R)$ — and that whenever $f : (X, R) \to (TY, \Gamma S)$ is a map between relational structures, so is $\gg\!= f : (TX, \Gamma R) \to (TY, \Gamma S)$.

[5] Notice, however, that our treatment is general: we do not require any metric-like axiom on d.

[6] Recall that the asymmetric Euclidean distance between two real numbers x and y is $y \dot{-} x$, where $\dot{-}$ denotes truncated subtraction.

In metric semantics, one does the same [Clementino et al. 2004; Hofmann et al. 2014], this time extending ternary relations $R \subseteq X \times [0,\infty] \times Y$ to relations $\Gamma R \subseteq T(X) \times [0,\infty] \times T(Y)$. In both cases, one thus defines axioms ensuring that $\mathcal{T}$ suitably extends to the relational structures of interest. This idea can be compactly expressed relying on a weak version of the notion of a *lifting of a monad* [Katsumata et al. 2018].[7] For the sake of exposition, we consider the case of **EBRel**. Let $U : \mathbf{EBRel} \to \mathbf{Set}$ be the forgetful functor mapping (A, R) to A and acting as the identity on morphisms. Given a monad $\mathcal{T} = (T, \eta, \gg\!=)$ on **Set**, a lifting of $\mathcal{T}$ to **EBRel** (along U) is given by a monad $\hat{\mathcal{T}} = (\hat{T}, \hat{\eta}, \hat{\gg\!=})$ on **EBRel** such that $U \circ \hat{\mathcal{T}} = \mathcal{T} \circ U$, $\eta_{U(\mathcal{A})} = U(\hat{\eta}_{\mathcal{A}})$, and $U(\hat{\gg\!=} f) = \gg\!= U(f)$ $(= \gg\!= f)$. According to the previous discussion, however, what we are interesting in is not having a monad $\hat{\mathcal{T}}$ on **EBRel**: all we want (and need, for our purposes) are the extensions of the unit and bind of $\mathcal{T}$ to arrows in **EBRel**. A lax extension Γ precisely gives (mapping (A, R) to $(T(A), \Gamma R)$) a lifting of $\mathcal{T}$ in which we drop the requirement that $\hat{\mathcal{T}}$ must itself be a monad. We call the resulting notion of extensions as *weak* liftings of $\mathcal{T}$. The advantage of relying on relators/lax extensions to define weak liftings of monads is that the latter are defined *relationally* by giving a way to extend relations to monadic elements. We apply the same methodology in our differential setting.

Informally, given a monad $\mathcal{T} = (T, \eta, \gg\!=)$ on **Set**, an extension of T to **GDS** consists of a collection of data that allow us to map a GDS $(A, (\!|A|\!), R)$ to a GDS $(T(A), (\!|T(A)|\!), \Gamma R)$ in such a way that η and $\gg\!=$ extend to morphisms between GDSs. Compared to the relational and metric case, in our setting giving a relator-like construction Γ is not enough. In fact, since distances between elements in A and elements in $T(A)$ will be, in general, different kinds of elements, we need to specify not only how to extend ternary relations to elements in $T(A)$, but also *what* distances between such elements *are*. Moreover, since morphisms in **GDS** are given by pairs of maps $(f, \mathrm{d}f)$, if we want to extend η and $\gg\!=$, we also need to explicitly give the differential counterparts of these data. Putting these observations together, we obtain the definition of a differential extension of a monad.

Definition 3.5 (Differential Extension). Given a monad $\mathcal{T} = (T, \eta, \gg\!=)$ on **Set**, a differential extension of $\mathcal{T}$ is given by (i) a map associating to any set A and distance space $(\!|A|\!)$, a distance space $(\!|T(A)|\!)$ for $T(A)$; (ii) a family Γ of mappings from $R \subseteq A \times (\!|A|\!) \times A$, to $\Gamma R \subseteq T(A) \times (\!|TA|\!) \times T(A)$; (iii) family of maps $\mathrm{d}\eta : A \times (\!|A|\!) \to (\!|T(A)|\!)$ and extensions $\overset{\star}{\gg\!=}\mathrm{d}f : T(A) \times (\!|T(A)|\!) \to (\!|T(B)|\!)$ of maps $\mathrm{d}f : A \times (\!|A|\!) \to (\!|T(B)|\!)$ such that the following laws hold for all $R \subseteq A \times (\!|A|\!) \times A$ and $S \subseteq B \times (\!|B|\!) \times B$:[8]

$$R(x, \mathrm{d}x, y) \implies \Gamma R(\eta(x), \mathrm{d}\eta(x, \mathrm{d}x), \eta(y))$$

$$\frac{\forall x, \mathrm{d}x, y.\ R(x, \mathrm{d}x, y) \implies \Gamma S(f(x), \mathrm{d}f(x, \mathrm{d}x), f(y))}{\forall \alpha, \mathrm{d}\alpha, \beta.\ \Gamma R(\alpha, \mathrm{d}\alpha, \beta) \implies \Gamma S(\alpha \gg\!= f, (\alpha, \mathrm{d}\alpha) \overset{\star}{\gg\!=} \mathrm{d}f, \beta \gg\!= f)}$$

A differential extension of $\mathcal{T}$ thus associates to each GDS $(A, (\!|A|\!), R)$ a GDS $(T(A), (\!|T(A)|\!), \Gamma R)$ together with maps $\mathrm{d}\eta$ and $\overset{\star}{\gg\!=}\mathrm{d}f$ extending the unit and bind of $\mathcal{T}$ to morphisms in **GDS**. That is:

$$\frac{}{(A, (\!|A|\!), R) \xrightarrow{(\eta, \mathrm{d}\eta)} (T(A), (\!|T(A)|\!), \Gamma R)} \qquad \frac{(A, (\!|A|\!), R) \xrightarrow{(f, \mathrm{d}f)} (T(B), (\!|T(B)|\!), \Gamma S)}{(T(A), (\!|T(A)|\!), \Gamma R) \xrightarrow{(\gg\!= f, \overset{\star}{\gg\!=} \mathrm{d}f)} (T(B), (\!|T(B)|\!), \Gamma S)}$$

Indeed, a differential extension of $\mathcal{T}$ gives a relational way to define a weak extension of $\mathcal{T}$ to **GDS**. In fact, let $U : \mathbf{GDS} \to \mathbf{Set}$ be the forgetful functor mapping a GDS $(A, (\!|A|\!), R)$ to A and an arrow $(f, \mathrm{d}f)$ to f. It is easy to see that the mapping from $(A, (\!|A|\!), R)$ to $(T(A), (\!|TA|\!), \Gamma R)$ together with the

[7] Other accounts of the notion of a relator/lax extension, can be given in terms of monads on double categories [Hofmann et al. 2014]. See Remark 1.

[8] Following standard notational conventions, oftentimes we write $(\alpha, \mathrm{d}\alpha) \overset{\star}{\gg\!=} \mathrm{d}f$ in place of $\overset{\star}{\gg\!=}\mathrm{d}f(\alpha, \mathrm{d}\alpha)$.

morphisms $(\eta, \mathrm{d}\eta)$ and the extensions mapping $(f, \mathrm{d}f)$ to $(\gg\!=\! f, \overset{\star}{\gg\!=}\! f)$ indeed give a weak extension of $\mathcal{T}$ to **GDS**. In particular, we have $\eta_A = U(\eta, \mathrm{d}\eta)$, and $U(\gg\!=\! f, \overset{\star}{\gg\!=}\! \mathrm{d}f) = \gg\!=\! U(f, \mathrm{d}f) = \gg\!=\! f$.

Remark 1. As already pointed out, another way to understand relators/lax extensions is in terms of monads on double categories of relations. In the classical relational setting,[9] one considers the double category having sets as objects, (binary) relations as horizontal morhisms, functions as vertical morhisms, and set-theoretic inclusions between relations as 2-morphisms. The notion of a lax extension/relator of a monad $\mathcal{T}$ is nothing but one possible definition of the notion of a monad on the double category of relations. Following this observation, one may then try to define generalisations of relators to our differential setting in terms of double categories. Such an approach, however, cannot readily transfer to the setting of GDSs, as endowing the latter with a double category structure is highly nontrivial (and outside the scope of this work). Nonetheless, we may, for the sake of the intuition, pretend to have a double category having sets as objects, differential relations as horizontal arrows, and functions as vertical arrows. 2-morphisms are then essentially given by maps $\mathrm{d}f$. This definition does not give a double category, as we lack a notion of composition between differential relations. Still, one can regard the notion of a differential extension as giving a definition of a monad on this imaginary double category of GDSs. Since, the latter does not meet all the requirements defining a double category (and thus it cannot be regarded as such), our notion of a differential extension lacks those conditions defining a monad on double categories corresponding to the aforementioned missing requirements. As a consequence, one way of viewing our notion of a differential extension is as imagining GDSs to give a double category as defined above; giving a definition of a monad on such a double category; removing from such a definition all conditions corresponding to the double category structure that GDSs actually lack.

Example 2. Let us consider the Maybe monad $(\mathcal{M}, \eta, \gg\!=)$. Given $(A, (\!|A|\!), R)$, we define $(\!|\mathcal{M}(A)|\!) \triangleq \mathcal{M}(\!|A|\!)$ and

$$\Gamma R(t, \mathrm{d}t, s) \overset{\triangle}{\iff} \begin{cases} t = \mathit{just}\ x\ \&\ \mathrm{d}t = \mathit{just}\ \mathrm{d}x\ \&\ s = \mathit{just}\ y\ \&\ R_A(x, \mathrm{d}x, y) & \text{or} \\ t = \bot\ \&\ \mathrm{d}t = \bot & \text{or} \\ s = \bot\ \&\ \mathrm{d}t = \bot. \end{cases}$$

A distance between two defined elements of the form $\mathit{just}\ x, \mathit{just}\ y$ is thus a distance between x and y. Otherwise (meaning that at least one element is undefined), the only distance is $\bot$. Notice that this structure gives us information only about defined elements and tells nothing about undefined expressions. An easy way to overcome the problem is to define $(\!|\mathcal{M}(A)|\!) \triangleq (\!|A|\!) + (A \times \{\bot\}) + (\{\bot\} \times A) + \{\bot\}$ and to change the definition of Γ accordingly, so that now a distance gives information on which element is undefined (for instance, we have $\Gamma R(\mathit{just}\ x, (x, \bot), \bot)$, $\Gamma R(\bot, (\bot, y), \mathit{just}\ y)$, and $\Gamma R(\bot, \bot, \bot)$). Although we could use the latter, more informative structure, the former one is enough for our goals. We define the extension of the unit and bind of $\mathcal{M}$ as: $\mathrm{d}\eta(x, \mathrm{d}x) \triangleq \mathit{just}\ \mathrm{d}x$ and $\overset{\star}{\gg\!=}\! \mathrm{d}f(t, \mathrm{d}t) \triangleq \mathrm{d}f(x, \mathrm{d}x)$ if $t = \mathit{just}\ x$ and $\mathrm{d}t = \mathit{just}\ \mathrm{d}x$; and $\bot$ otherwise Standard calculations show that we indeed have a differential extension of $\mathcal{M}$ ⊠

Example 3. Let us consider the writer monad $(\mathcal{W}, \eta, \gg\!=)$ for a monoid $(W, \cdot, \varepsilon)$. Given $(A, (\!|A|\!), R)$, we define:

$$(\!|\mathcal{W}(A)|\!) \triangleq [0, \infty] \times (\!|A|\!)$$

$$\Gamma R((w, x), (n, \mathrm{d}x), (u, y)) \overset{\triangle}{\iff} n \geq d(w, u)\ \&\ R(x, \mathrm{d}x, y),$$

[9] A similar observation holds in the setting of quantitative relations, where one simply replaces binary relations with relations of the form $R \subseteq A \times [0, \infty] \times B$ [Hofmann et al. 2014].

where $d : W \times W \to [0, \infty]$ is a (numerical) distance on W making $\cdot$ non-expansive.[10] For instance, if $W = \Sigma^*$, then we could define d as the longest common prefix distance. A distance between two computations with output (w, x), (u, y) thus consists of a distance between x and y together with a numerical value bounding the distance between the output strings. Let us now move to unit and bind:

$$\mathrm{d}\eta(\mathrm{d}x) \triangleq (0, \mathrm{d}x)$$

$$\overset{\star}{\gg\!=}\mathrm{d}f((w, x), (n, \mathrm{d}x)) \triangleq (n + m, \mathrm{d}y) \quad \text{where } (m, \mathrm{d}y) = \mathrm{d}f(x, \mathrm{d}x).$$

Easy calculations show that $\mathrm{d}\eta$ and $\overset{\star}{\gg\!=}\mathrm{d}f$ both satisfy the required properties. ⊠

Example 4. By taking $(W, \cdot, \varepsilon) = (\mathbb{N}, +, 0)$ in the previous example, we obtain the so-called *cost* or *tick* monad [Çiçek et al. 2017; Dal Lago and Gavazzo 2019a; Sands 1998] used in (relational) cost analysis. Defining e as the Euclidean norm between natural numbers, we see that a distance between two computations with cost gives us a distance between the returned values of the computations together with an (over-)approximation of the numerical distance between the computational costs of producing such values. Moreover, if we take e as the asymmetric Euclidean norm, we obtain information on which of the two computations is more efficient. We will extensively apply differential logical relations to perform context-dependent cost analysis in Section 5. ⊠

3.2 Weak Pullback Preserving Monads and Coupling-Based Differential Extensions

In this section, we define a general differential extension based on (a suitable generalisaton of) the notion of a coupling on weak pullback preserving monads. Coupling have been introduced in probability theory and they play a central role in optimal transport and optimisation theory (via the notion of a transportation plan and the celebrated Kantorovich-Rubinstein theorem [Villani 2008]), differential privacy (via the notion of expected sensitivity [Barthe et al. 2018, 2016] and the Strassen Theorem [Strassen 1965]), and program semantics (via the notion of probabilistic bisimilarity and bisimulation metrics [Crubillé and Dal Lago 2015; Dal Lago and Gavazzo 2019b; Dal Lago et al. 2014; Deng 2015; Desharnais et al. 2004; Larsen and Skou 1989]). Generalisations of couplings to a more general class of monads have been extensively used to define both relational [Kurz and Velebil 2016] (starting with the seminal work by Barr [1970] in categorical topology) and metric [Clementino et al. 2004; Hofmann 2007; Hofmann et al. 2014] extensions of monads. Here, we extend such extensions to a differential setting.

Through this section, we restrict our analysis to monads $\mathcal{T}$ preserving weak pullbacks, and thus preserving injections. As a consequence, if $\iota : A \hookrightarrow X$ is the subset inclusion map, then $T(\iota) : T(A) \hookrightarrow T(X)$ is an injection, which can be regarded as monadic inclusion. Intuitively, given an element $\alpha \in T(X)$, we think about the smallest set A with $\iota : A \hookrightarrow X$ such that $\alpha \in T(A)$ as the support of α, and denote such a set as $\mathrm{supp}(\alpha)$. Of course, in general the support of an element α may not exist and therefore we restrict our analysis to monads coming with a notion of *countable support* [Hofmann 2007; Manes 2002]. Accordingly, we say that a monad is countable if for any set X and any element $\alpha \in T(X)$, there exists the smallest countable subset $\mathrm{supp}(\alpha)$ of X (so that $\iota : \mathrm{supp}(\alpha) \hookrightarrow X$) such that $\alpha \in T(\mathrm{supp}(\alpha))$ (i.e. there exists $\beta \in T(\mathrm{supp}(\alpha))$ such that $\alpha = T(\iota)(\beta)$). All monads seen so far are countable (for instance, the distribution monad is countable by definition and has the usual notion of a support of a distribution), with the exception of the powerset monad. We overcome the problem by (tacitly) working with countable restriction of the powerset monad (this is in line with the way we model nondeterminism, whereby a nondeterministic programs can nondeterministically return countable values only).

[10] So that $d(w \cdot u, v \cdot z) \leq d(w, v) + d(u, z)$.

To justify our design choice, we first review coupling-based metric extensions. Let us begin with the probabilistic case.

Definition 3.6. A *coupling* between two distributions μ, ν over sets X, Y, respectively, is a distribution ω over $X \times Y$ such that $\sum_y \omega(x,y) = \mu(x)$ and $\sum_x \omega(x,y) = \nu(y)$. We denote the collection of couplings of μ and ν by $\Omega(\mu,\nu)$.

One of the main applications of probabilistic couplings is the definition of relational and metric extensions of the distribution monad. In particular, given a relation $R \subseteq X \times Y$ and a distance $d : X \times Y \to [0,\infty]$, we define the extensions $\hat{R} \subseteq \mathcal{D}(X) \times \mathcal{D}(Y)$ and $\hat{d} : \mathcal{D}(X) \times \mathcal{D}(Y) \to [0,\infty]$ by:

$$\mu \,\hat{R}\, \nu \overset{\triangle}{\iff} \exists \omega \in \Omega(\mu,\nu).(\forall x, y.\ \omega(x,y) > 0 \implies x\, R\, y)$$
$$\hat{d}(\mu,\nu) \triangleq \inf_{\omega \in \Omega(\mu,\nu)} \sum_{x,y} \omega(x,y) \cdot d(x,y).$$

The reader may have recognised that $\hat{R}$ is nothing but the probabilistic lifting used by Larsen and Skou to define notions of probabilistic bisimulations [Larsen and Skou 1989] whereas $\hat{d}$ is the well-known Wasserstein-Kantorovich distance [Villani 2008].

To understand the rationale behind the definition of the aforementioned probabilistic extension (as well as of our coupling-based differential extension), let us explain where the definition of $\hat{d}$ comes from. First, we notice that a coupling $\omega \in \Omega(\mu,\nu)$ formalises the informal notion of a transportation plan (Figure 3). Thinking about the distribution μ as assigning weights to points in X, a transportation plan from μ to ν is a mapping specifying how (and how much) weight one has to move from X to Y to transform μ into ν. Formally, a transportation plan from μ to ν is a X-indexed family of maps $\pi_x : Y \to [0,1]$ specifying for each $y \in Y$ how much weight of $\mu(x)$ has to be moved from x to y. Accordingly, π_x cannot move out of x more weight than the available one and actually, since we want to move the whole weight mass μ, the plan π has to transport the whole μ: we thus require $\sum_y \pi_x(y) = \mu(x)$, for all $x \in X$. Applying π to μ, we obtain a new distribution (a new mass) $\vec{\pi}(\mu) \in \mathcal{D}(Y)$ defined by $\vec{\pi}(\mu)(y) \triangleq \sum_{x \in X} \pi_x(y)$. Obviously, since π has to transform μ into ν, we require $\vec{\pi}(\mu) = \nu$. At this point, it is straightforward to see that any transportation plan π from μ to ν induces a coupling $\omega_\pi \in \Omega(\mu,\nu)$ defined by $\omega_\pi(x,y) \triangleq \pi_x(y)$, and that any coupling $\omega \in \Omega(\mu,\nu)$ induces a transportation plan π^ω defined by $\pi^\omega_x(y) \triangleq \omega(x,y)$, so that the notions of a transportation plan and of a coupling are equivalent. Finally, we interpret the distance function $d : X \times Y \to [0,\infty]$ as a cost function, so that we may read $d(x,y)$ as the cost of moving (something) from point x to point y. As a consequence, given a transportation plan π from μ to ν, it is natural to define the total cost of transforming μ into ν as the expected cost of transforming μ into ν following plan π, i.e. $\sum_{x,y} \pi_x(y) \cdot d(x,y)$. The distance $\hat{d}$ is then obtained by taking the plan that minimises such an expected cost.

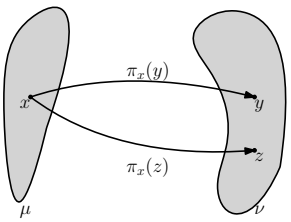

Fig. 3. Transportation plan from μ to ν

Now that we have understood where the definition of $\hat{d}$ comes from, let us generalise it to arbitrary monads. In fact, a coupling $\omega \in \Omega(\mu,\nu)$ is nothing but a distribution in $\mathcal{D}(X \times Y)$ such that $\mathcal{D}(\pi_1)(\omega) = \mu$ and $\mathcal{D}(\pi_2)(\omega) = \nu$, where $\pi_i : X_1 \times X_2 \to X_i$ is the i-th projection function. As a consequence, given a monad T and elements $\alpha \in T(X)$, $\beta \in T(Y)$, a coupling of α and β is just an element $\gamma \in T(X \times Y)$ such that $T(\pi_1)(\gamma) = \alpha$ and $T(\pi_2)(\gamma) = \beta$. Thinking about a function $d : X \times Y \to [0,\infty]$ as a (generalised) 'distance', then any coupling γ gives a 'monadic distance'

$T(d)(\gamma) \in T[0,\infty]$. To obtain a numerical distance, we need a way to extract a numerical value out of $T[0,\infty]$, i.e. a map $\xi : T[0,\infty] \to [0,\infty]$. For the distribution monad $\mathcal{D}$, the map ξ is nothing but the expectation map $\mathbb{E} : \mathcal{D}[0,\infty] \to [0,\infty]$ defined by $\mathbb{E}(\mu) \triangleq \sum_x x \cdot \mu(x)$. Therefore, given such a map $\xi : T[0,\infty] \to [0,\infty]$ (usually called structure map [Hofmann 2007] or evaluation map [Baldan et al. 2018]), any coupling γ induces a numerical distance (measured via d) between α and β, namely $\xi(T(d)(\gamma))$. Among all these possible distances, we then take the minimal one.

Definition 3.7. Given a map $\xi : T[0,\infty] \to [0,\infty]$ and distance $d : X \times Y \to [0,\infty]$, define $\hat{T}_\xi(d) : TX \times TY \to [0,\infty]$ by $\hat{T}_\xi(d)(\alpha,\beta) \triangleq \inf_{\gamma \in \Omega(\alpha,\beta)} \xi(T(d)(\gamma))$, where $\Omega(\alpha,\beta)$ is the set of *couplings* of α, β.

Let us now progressively move to a differential setting. First, we rephrase Definition 3.7 in terms of ternary relations. Given a ternary relation $R \subseteq X \times [0,\infty] \times Y$, we define the relation $\tilde{T}_\xi(R) \subseteq T(X) \times [0,\infty] \times T(Y)$ thus:

$$\tilde{T}_\xi(R)(\alpha,a,\beta) \overset{\triangle}{\iff} \exists \gamma \in T(X \times [0,\infty] \times Y). \begin{cases} T(\pi_1)(\gamma) = \alpha \\ T(\pi_3)(\gamma) = \beta \\ \forall x,b,y.\ (x,b,y) \in \mathrm{supp}(\gamma) \implies R(x,b,y) \\ a \geq \xi(T(\pi_2)(\gamma)) \end{cases}$$

Indeed, we have $\inf\{a \mid \tilde{T}_\xi(R)(\alpha,a,\beta)\} = \hat{T}_\xi(d)(\alpha,\beta)$. Notice that $\tilde{T}_\xi(R)$ is obtained by constructing a *ternary coupling*, i.e. an element $\gamma \in T(X \times [0,\infty] \times Y)$ whose support contains elements $(x,y) \in X \times Y$ together with distances between them. That is, if (x,a,y) belongs to the support of γ, then a must bound the distance between x and y, i.e. $R(x,a,y)$ holds. Moreover, once such a ternary coupling γ is given, we can extract a monadic distance from it, namely $T(\pi_2)(\gamma) \in T[0,\infty]$. Finally, the information given by $T(\pi_2)(\gamma)$ is processed by the map ξ, this way obtaining a numerical distance.

In a differential setting, distances need not be numeric values and thus we do not need the map ξ anymore. Moreover, the set $[0,\infty]$ is replaced by $(\!|A|\!)$. This suggests that a distance between two monadic elements $\alpha \in T(A)$, $\beta \in T(A)$ is a monadic distance $\mathrm{d}\alpha \in T(\!|A|\!)$ such that $\alpha, \mathrm{d}\alpha, \beta$ stay in a ternary coupling relation. Elements $\mathrm{d}\alpha \in T(\!|A|\!)$, however, do not have enough information to act as distances. In fact, given $\mathrm{d}\alpha \in T(\!|A|\!)$ and $\alpha \in T(A)$ and thinking about $\mathrm{d}\alpha$ as a change to α, we would be able to 'apply' such a change to α. But to do so, we need $\mathrm{d}\alpha$ to specify which changes $\mathrm{d}x$ in its support applies to which elements x in the support of α. We overcome this deficiency by taking $\mathrm{d}\alpha \in T(A \times (\!|A|\!))$ and requiring $T(\pi_1)(\mathrm{d}\alpha) = \alpha$. Notice that this way we have $T(\pi_2)(\mathrm{d}\alpha) \in T(\!|A|\!)$. An intuition behind this choice comes from the transportation plan reading of probabilistic couplings. In the probabilistic case, distances are ultimately defined by transportation plans, the latter specifying how much weight one has to move from a point another. In our differential setting, we can think about distances $\mathrm{d}x$ as 'edges' between points, so that we not only transport weight from x to y, but we do so along 'edges' $\mathrm{d}x$. As a consequence, we have to specify how much weight can be moved from x along edges $\mathrm{d}x$. A distribution $\mathrm{d}\mu \in \mathcal{D}(A \times (\!|A|\!))$ does exactly that: $\mathrm{d}\mu(x,\mathrm{d}x)$ tells us how much weight we can move from x along $\mathrm{d}x$. Again, since we want to move all $\mu(x)$ from x, we need to have $\sum_{\mathrm{d}x} \mathrm{d}\mu(x,\mathrm{d}x) = \mu(x)$, i.e. $\mathcal{D}(\pi_1)(\mathrm{d}\mu) = \mu$ The differential relation which we are going to define then tells us when a plan $\mathrm{d}\mu \in \mathcal{D}(A \times (\!|A|\!))$ can indeed be used to turn μ into a distribution ν. The formalisation of these ideas give the following differential extension of T.

Definition 3.8. The coupling-based differential extension of a monad $\mathcal{T} = (T, \eta, \gg=)$ is defined thus, where $(A, (\!|A|\!), R)$ is a GDS.

$$(\!|T(A)|\!) \triangleq T(A \times (\!|A|\!))$$

$$\Gamma R(\alpha, \mathrm{d}\alpha, \beta) \overset{\triangle}{\Longleftrightarrow} \exists \gamma \in T(A \times (\!|A|\!) \times A).\ \begin{cases} T(\pi_1) = \alpha \\ T(\langle \pi_1, \pi_2 \rangle)(\gamma) = \mathrm{d}\alpha \\ T(\pi_3)(\gamma) = \beta \\ \forall (x, \mathrm{d}x, y) \in \mathrm{supp}(\gamma).\ R_A(x, \mathrm{d}x, y) \end{cases}$$

$$\mathrm{d}\eta(x, \mathrm{d}x) \triangleq \eta(x, \mathrm{d}x)$$

$$\overset{\star}{\gg=}\mathrm{d}f(\alpha, \mathrm{d}\alpha) \triangleq \mathrm{d}\alpha \gg= \mathrm{d}f$$

Notice that from $T(\pi_1) = \alpha$ and $T(\langle \pi_1, \pi_2 \rangle)(\gamma) = \mathrm{d}\alpha$ it follows $T(\pi_1)(\mathrm{d}\alpha) = \alpha$, meaning that the relevant information about α is contained in $\mathrm{d}\alpha$. This is the reason why $\overset{\star}{\gg=}\mathrm{d}f(\alpha, \mathrm{d}\alpha)$ *de facto* relies on $\mathrm{d}\alpha$ only. Before proving that Definition 3.8 indeed gives a differential extension of T, let us see some examples.

Example 5. Instantiating Definition 3.8 to the distribution monad $\mathcal{D}$, we obtain the following differential structure:

$$(\!|\mathcal{D}(A)|\!) \triangleq \mathcal{D}(A \times (\!|A|\!))$$

$$\Gamma R(\mu, \mathrm{d}\mu, \nu) \overset{\triangle}{\Longleftrightarrow} \exists \omega \in \mathcal{D}(A \times (\!|A|\!) \times A).\ \begin{cases} \sum_{\mathrm{d}x, y} \omega(x, \mathrm{d}x, y) = \mu(x) \\ \sum_{y} \omega(x, \mathrm{d}x, y) = \mathrm{d}\mu(x, \mathrm{d}x) \\ \sum_{x, \mathrm{d}x} \omega(x, \mathrm{d}x, y) = \nu(y) \\ \omega(x, \mathrm{d}x, y) > 0 \implies R_A(x, \mathrm{d}x, y) \end{cases}$$

Notice that if we have $\Gamma R(\mu, \mathrm{d}\mu, \nu)$, then there is a way to transport weights given by μ to weights given by ν following $\mathrm{d}\mu$. ⊠

Example 6. Instantiating Definition 3.8 to the powerset monad $\mathcal{P}$, we obtain the following differential structure:

$$(\!|\mathcal{P}(A)|\!) \triangleq \mathcal{P}(A \times (\!|A|\!))$$

$$\Gamma R(\alpha, \mathrm{d}\alpha, \beta) \overset{\triangle}{\Longleftrightarrow} \begin{cases} \forall x \in \alpha.\ \exists \mathrm{d}x, y.\ (x, \mathrm{d}x) \in \mathrm{d}\alpha\ \&\ y \in \beta\ \&\ R(x, \mathrm{d}x, y) \\ \forall (x, \mathrm{d}x) \in \mathrm{d}\alpha.\ \exists y \in \beta.\ R\ (x, \mathrm{d}x, y) \\ \forall y \in \beta.\ \exists (x, \mathrm{d}x) \in \mathrm{d}\alpha.\ R\ (x, \mathrm{d}x, y). \end{cases}$$

⊠

Finally, notice that our coupling-based extension is indeed a differential extension.

Proposition 3.9. *If $\mathcal{T}$ is countable and preserves weak pullbacks, then the coupling-based extension from Definition 3.8 is a differential extension of $\mathcal{T}$.*

3.3 Sensitivity Analysis: Local, General, and Expected

In this section, we show how GDSs can be used to reason about function sensitivity [Chaudhuri et al. 2010, 2011; Reed and Pierce 2010] and how they give a natural way to extend such a notion to a higher-order setting. Recall that $f : \mathbb{R} \to \mathbb{R}$ has sensitivity $s : [0, \infty] \to [0, \infty]$, if for any pair of inputs x, y, it holds that $s(d(x, y)) \geq d(f(x), f(y))$, where d is a distance on $\mathbb{R}$. Let us consider the GDS $(\mathbb{R}, [0, \infty], R)$ with $R(x, \mathrm{d}x, y) \overset{\triangle}{\Longleftrightarrow} \mathrm{d}x \geq d(x, y)$. As usual, we have $R(x, d(x, y), y)$. We

claim that if $[R \to R]\ (f, \mathrm{d}f, f)$ for $\mathrm{d}f$ such that $\mathrm{d}f(x, \mathrm{d}x) = s(\mathrm{d}x)$, then f has sensitivity s. In fact, from $R(x, \mathrm{d}x, y)$ it follows, by very definition of $[R \to R]$, that $R(f(x), \mathrm{d}f(x, d(x, y), f(y))$, i.e. $\mathrm{d}f(x, d(x, y)) = s(d(x, y)) \geq d(f(x), f(y))$.

Local Sensitivity. The notion of sensitivity is by its very nature input independent. This may be unfortunate in situations where we deal with functions having high sensitivity only on specific intervals, and where we (post)compose such functions with maps whose outputs do not belong to such intervals. A natural way to fix this deficiency is by generalising sensitivity to what we may call *local sensitivity* [Nissim et al. 2007], i.e. an input dependent notion of sensitivity. We say that f has *local sensitivity* $k : \mathbb{R} \to [0, \infty]$ if for all input x, y, we have $k(x) \cdot d(x, y) \geq d(f(x), f(y))$. We can use GDSs to characterise local sensitivities simply noticing that f has local sensitivity k if there exists $\mathrm{d}f$ such that $\mathrm{d}f(x, \mathrm{d}x) = k(x) \cdot \mathrm{d}x$. But why should one work with local (instead of global) sensitivity? One of the advantages of working with local sensitivity is that the latter leads to more accurate sensitivity analyses. Let us consider two real functions g, f with self-distances $\mathrm{d}g$, $\mathrm{d}f$ such that $\mathrm{d}f(x, \mathrm{d}x) = k(x) \cdot \mathrm{d}x$ and $\mathrm{d}g(x, \mathrm{d}x) = h(x) \cdot \mathrm{d}x$, so that f and g have local sensitivity k and h, respectively. What can we say about the local sensitivity of $g \circ f$? By very definition of composition on **GDS**, we have that $\mathrm{d}g \odot \mathrm{d}f$ is self-distance for $g \circ f$. In particular, we have $(\mathrm{d}g \odot \mathrm{d}f)(x, \mathrm{d}x) = \mathrm{d}g(f(x), \mathrm{d}f(x, \mathrm{d}x)) = h(f(x)) \cdot k(x) \cdot \mathrm{d}x$, which gives the local sensitivity of $g \circ f$. Notice that if f maps an input x to an output $f(x)$ for which g has low sensitivity, then $g \circ f$ will have low sensitivity on x (provided that f has low sensitivity on x too).

General Sensitivity. Let us now come back to the sensitivity of a function $f : \mathbb{R} \to \mathbb{R}$. We have seen that f has sensitivity $s : [0, \infty] \to [0, \infty]$ if f has a self-distance $\mathrm{d}f$ such that $\mathrm{d}f(x, \mathrm{d}x) = s(\mathrm{d}x)$. We can express this condition in a purely algebraic fashion recalling that the derivative of the identity function $1 : A \to A$ is $\mathrm{d}1$ defined by $\mathrm{d}1(x, \mathrm{d}x) \triangleq \mathrm{d}x$. We can then say that f has sensitivity s if $s \circ \mathrm{d}1$ is a self-distance of f. This definition has essentially no reference to real numbers, so that we can extend the notion of sensitivity to arbitrary sets by saying that a function $f : A \to A$ has sensitivity $s : (\!|A|\!) \to (\!|A|\!)$ if $s \circ \mathrm{d}1$ is a self-distance of f.

Expected Sensitivity. General sensitivity obviously applies to effectful distances, in those case sensitivities being functions $s : (\!|T(A)|\!) \to (\!|T(A)|\!)$. That, however, is a rather liberal notion of sensitivity, and one would like to say that the sensitivity of an effectful function/program $f : A \to T(A)$ is a function $s : (\!|A|\!) \to (\!|A|\!)$, rather than a function of $(\!|T(A)|\!)$. We show how this can be done in the case of coupling-based extensions, this way extending the so-called expected sensitivity [Barthe et al. 2018] to a differential setting. The notion of expected sensitivity is inherently probabilistic: we thus restrict our analysis to the distribution monad (although in principle we could work a larger class of monads [Hofmann 2007]). Let us recall that a function $f : \mathbb{R} \to \mathcal{D}(\mathbb{R})$ is expected s-sensitive, with $s : [0, \infty] \to [0, \infty]$, if for every two inputs x, y, $s(d(x, y)) \geq \hat{d}(f(x), f(y))$, where d is a distance on $\mathbb{R}$ and $\hat{d}$ is its Wasserstein-Kantorovich lifting. The notion of expected sensitivity crucially relies on the Wasserstein-Kantorovich lifting of a distance, which itself build on two main components: probabilistic couplings and the the expectation function $\mathbb{E} : \mathcal{D}[0, \infty] \to [0, \infty]$. To generalise the notion of expected sensitivity to a differential setting we thus need to generalise the expectation function to an arbitrary distance space $(\!|A|\!)$. Although this is in general not possible, it is so for those distance spaces giving semantics to our calculus.

Definition 3.10. Define the map $\xi_{(\!|\mathbb{R}|\!)} : \mathcal{D}[0,\infty] \to [0,\infty]$ as $\mathbb{E}$. Assuming $\xi_{(\!|A|\!)}$ and $\xi_{(\!|B|\!)}$ to be defined, we define:

$$\xi_{(\!|B^A|\!)} : \mathcal{D}((\!|B|\!)^{A\times(\!|A|\!)}) \to (\!|B|\!)^{A\times(\!|A|\!)} \qquad \xi_{(\!|A\times B|\!)} : \mathcal{D}((\!|A|\!)\times(\!|B|\!)) \to (\!|A|\!)\times(\!|B|\!)$$

$$\xi_{(\!|B^A|\!)}(\mu)(x,\mathrm{d}x) \triangleq \xi_{(\!|B|\!)}\left(\sum_{\mathrm{d}f}\mu(\mathrm{d}f)\cdot\delta_{\mathrm{d}f(x,\mathrm{d}x)}\right) \quad \xi_{(\!|A\times B|\!)}(\mu) \triangleq \langle \xi_{(\!|A|\!)}(\mathcal{D}(\pi_1)(\mu)), \xi_{(\!|B|\!)}(\mathcal{D}(\pi_2)(\mu))\rangle.$$

For instance, we have $\xi_{(\!|\mathbb{R}\to\mathbb{R}|\!)}(\mu)(x,\mathrm{d}x) = \xi_{\mathbb{R}}(\sum_{\mathrm{d}f}\mu(\mathrm{d}f)\cdot\delta_{\mathrm{d}f(x,\mathrm{d}x)}) = \sum_{\mathrm{d}f}\mu(\mathrm{d}f)\cdot\mathrm{d}f(x,\mathrm{d}x)$, so that $\xi_{(\!|\mathbb{R}\to\mathbb{R}|\!)}(\mu)(x,\mathrm{d}x)$ gives the expected distance/error of the distribution of functions μ on input x with perturbation $\mathrm{d}x$. As a sanity check, we prove the following 'adequacy' lemma stating that the expected distance between two distributions approximates the Wasserstein-Kantorovich lifting distance between the distributions.

Lemma 3.11 (Adequacy). *Given $\mu,\nu\in\mathcal{D}(\mathbb{R})$, and a distance d on $\mathbb{R}$, recall that we have a GDS $(\mathbb{R},[0,\infty],R)$ with $R(x,\mathrm{d}x,y) \overset{\triangle}{\Longleftrightarrow} \mathrm{d}x \geq d(x,y)$. Then,*

$$\Gamma R(\mu,\mathrm{d}\mu,\nu) \implies \xi_{(\!|\mathbb{R}|\!)}(\mathcal{D}(\pi_2)(\mathrm{d}\mu)) \geq \hat{d}(\mu,\nu).$$

We can now give an abstract notion of expected sensitivity.

Definition 3.12. Given a GDS $(A,(\!|A|\!),R)$ such that $\xi_{(\!|A|\!)} : T(\!|A|\!) \to (\!|A|\!)$ is defined, we say that a function $f : A \to \mathcal{D}(A)$ has expected sensitivity $s : (\!|A|\!) \to (\!|A|\!)$ if there exists $\mathrm{d}f$ such that $[R\to\Gamma R]\ (f,\mathrm{d}f,f)$ and $\xi_{(\!|A|\!)}\circ\mathcal{D}(\pi_2)\circ\mathrm{d}f = s\circ\mathrm{d}1$.

Let us take $A=\mathbb{R}$ and assume $f:\mathbb{R}\to\mathcal{D}(\mathbb{R})$ to have sensitivity $s:[0,\infty]\to[0,\infty]$, so that there exists $\mathrm{d}f$ such that $[R\to\Gamma R]\ (f,\mathrm{d}f,f)$ and $\xi_{(\!|\mathbb{R}|\!)}\circ\mathcal{D}(\pi_2)\circ\mathrm{d}f = s\circ\mathrm{d}1$. Given a distance d on $\mathbb{R}$, we have $R(x,d(x,y),y)$, and thus $\Gamma R(f(x),\mathrm{d}f(x,d(x,y)),f(y))$. Since f is expected s-sensitive, by Lemma 3.11 we:

$$s(d(x,y)) = (\xi_{(\!|\mathbb{R}|\!)}\circ\mathcal{D}(\pi_2))(\mathrm{d}f(x,\mathrm{d}x)) = \mathbb{E}[\mathcal{D}(\pi_2)(\mathrm{d}f(x,\mathrm{d}x)] \geq \hat{d}(f(x),f(y))$$

which is nothing but the original notion of expected sensitivity of real valued functions.

4 EFFECTFUL DIFFERENTIAL LOGICAL RELATIONS

In this section, we apply GDSs to define contextual distances for a higher-order calculus with computational effects. More specifically, we associate to any type σ a syntax based GDS whose distance relation is defined by an extension of *differential logical relations* (DLRs) [Dal Lago et al. 2019] to an effectful and monadic setting. Such an extension builds upon the notion of a differential extension of monad introduced in the previous section: remarkably, the defining axioms of such a notion are what is needed to extend the so-called fundamental lemma (the main result of this section) to computational effects. In Section 5, we will then show applications of the theoretical results proved in this section to fields such as relational cost analysis and approximate computing.

The target calculus we use is a computational simply-typed call-by-value λ-calculus extended with a type **Real** for real numbers. We call such a calculus Λ^T. To calculus is given in a fine-grain style [Levy et al. 2003], so that we have a syntactic distinction between values and computations. The latter are defined by the following grammar, where $r\in\mathbb{R}$ and $F:\mathbb{R}^n\to\mathbb{R}$.

$$v,w ::= x \mid \underline{r} \mid \langle\rangle \mid \langle v,w\rangle \mid \lambda x.e; \qquad e,f ::= \mathbf{val}\ v \mid \underline{F}(v,\ldots,v) \mid v.i \mid vw \mid \mathbf{let}\ x = e\ \mathbf{in}\ f$$

The static semantics of Λ^T is defined as usual relying on judgments of the form $\Theta\vdash v:\sigma$ and $\Theta\vdash e:\sigma$. Since all these notions are standard, we omit them. Moreover, we tacitly employ standard syntactic and notational conventions and refer the reader to any textbook on the subject [Harper

$$\underline{r} \overset{\mathrm{d}r}{\longmapsto}_{\mathbf{Real}} \underline{s} \overset{\triangle}{\Longleftrightarrow} \mathrm{d}r \geq d(r,s)$$

$$\langle\rangle \overset{*}{\mapsto}_{\mathbf{Unit}} \langle\rangle$$

$$\langle v_1, v_2\rangle \overset{(\mathrm{d}v_1, \mathrm{d}v_2)}{\longmapsto}_{\sigma_1\times\sigma_2} \langle w_1, w_2\rangle \overset{\triangle}{\Longleftrightarrow} \forall i \in \{1,2\}.\ v_i \overset{\mathrm{d}v_i}{\longmapsto}_{\sigma_i} w_i$$

$$\lambda x.e \overset{\mathrm{d}e}{\longmapsto}_{\sigma\to\tau} \lambda x.f \overset{\triangle}{\Longleftrightarrow} \forall v, \mathrm{d}v, w.\ v \overset{\mathrm{d}v}{\longmapsto}_{\sigma} w \implies e[v/x] \overset{\mathrm{d}e(v,\mathrm{d}v)}{\longmapsto}_{\tau} f[w/x]$$

$$e \overset{\mathrm{d}e}{\longmapsto}_{\sigma} f \overset{\triangle}{\Longleftrightarrow} [\![e]\!]^{\mathcal{E}} \overset{\mathrm{d}e}{\Longmapsto}_{\sigma} [\![f]\!]^{\mathcal{E}}.$$

Fig. 4. Effectful Differential Logical Relations

2016]. We write $\mathcal{V}_{\Theta\vdash\sigma}$ and $\Lambda_{\Theta\vdash\sigma}$ for the collections of values and computations with type σ in environment Θ, respectively. If Θ is empty, we simply write $\mathcal{V}_\sigma$ and Λ_σ.

The dynamic semantics of Λ^T is defined with respect to a monad $(T, \eta, \gg\!=)$ modelling the computational effects of interest. The semantics is given by a monadic evaluation function [Dal Lago et al. 2017a] $[\![-]\!]$ mapping closed computations of type σ to monadic values in $T(\mathcal{V}_\sigma)$:

$$[\![\mathbf{val}\ v]\!] \triangleq \eta(v) \qquad [\![\underline{F}(\underline{r}_1, \ldots, \underline{r}_n)]\!] \triangleq \eta(\underline{F(r_1, \ldots, r_n)}) \qquad [\![\langle v_1, v_2\rangle.i]\!] \triangleq \eta(v_i)$$

$$[\![(\lambda x.e)v]\!] \triangleq [\![e[v/x]]\!] \qquad [\![\mathbf{let}\ x = e\ \mathbf{in}\ f]\!] \triangleq [\![e]\!] \gg\!= [\![f[-/x]]\!]$$

Notice that because Λ^T is simply-typed, $[\![-]\!]$ is well-defined. Λ^T actually lacks real effect-producing operations. We may overcome this problem by endowing Λ^T with effect-triggering (algebraic) operations in the style of Plotkin and Power [2001]. Our results being mostly type-driven, they are rather independent of the concrete details of the language used. For this reason, we work with a minimal calculus, although we remark that our results are robust to language extensions.

We now define GDSs for values and computations following the structure of types. We assume a differential extension of the monad $(T, \eta, \gg\!=)$ used for effects as given, and we employ the same notation used in the previous section for it. We begin by defining a GDS for closed values and closed computations for each type.

Definition 4.1. The distance spaces $(\!|\mathcal{V}_\sigma|\!)$ and $(\!|\Lambda_\sigma|\!)$ are recursively defined as follows:

$$(\!|\mathcal{V}_{\mathbf{Real}}|\!) \triangleq [0,\infty] \qquad (\!|\mathcal{V}_{\mathbf{Unit}}|\!) \triangleq 1 \qquad (\!|\mathcal{V}_{\sigma\to\tau}|\!) \triangleq (\!|\Lambda_\tau|\!)^{\mathcal{V}_\sigma\times(\!|\mathcal{V}_\sigma|\!)}$$

$$(\!|\mathcal{V}_{\sigma\times\tau}|\!) \triangleq (\!|\mathcal{V}_\sigma|\!) \times (\!|\mathcal{V}_\tau|\!) \qquad (\!|\Lambda_\sigma|\!) \triangleq (\!|T(\mathcal{V}_\sigma)|\!).$$

Notice that distances between computations are given by the differential extension of T. Next, we endow $(\mathcal{V}_\sigma, (\!|\mathcal{V}_\sigma|\!))$ and $(\Lambda_\sigma, (\!|\Lambda_\sigma|\!))$ with a differential logical relation which will turn them into GDSs. Given the importance of such a relation, we introduce a special notation for it. We denote by $\longmapsto^{\mathcal{V}}_\sigma \subseteq \mathcal{V}_\sigma \times (\!|\mathcal{V}_\sigma|\!) \times \mathcal{V}_\sigma$ and $\longmapsto^{\Lambda}_\sigma \subseteq \Lambda_\sigma \times (\!|\Lambda_\sigma|\!) \times \Lambda_\sigma$ the differential logical relations on values and computations of type σ, respectively. We write $e \overset{\mathrm{d}e}{\longmapsto}_\sigma f$ if $(e, \mathrm{d}e, f) \in \longmapsto^{\Lambda}_\sigma$ (and similarly for values). Notice that we omit value/computations superscripts. Finally, we write $\Longmapsto_\sigma$ for $\Gamma(\longmapsto^{\mathcal{V}}_\sigma)$ and write $\alpha \overset{\mathrm{d}\alpha}{\Longmapsto}_\sigma \beta$ in place of $(\alpha, \mathrm{d}\alpha, \beta) \in \Gamma(\longmapsto^{\mathcal{V}}_\sigma)$.

Definition 4.2. Let $d : \mathbb{R}\times\mathbb{R} \to [0,\infty]$ be a distance function. The type-indexed families of ternary relations $\longmapsto^{\mathcal{V}}_\sigma \subseteq \mathcal{V}_\sigma \times (\!|\mathcal{V}_\sigma|\!) \times \mathcal{V}_\sigma$ and $\longmapsto^{\Lambda}_\sigma \subseteq \Lambda_\sigma \times (\!|\Lambda_\sigma|\!) \times \Lambda_\sigma$, called *differential logical relations* (DLRs), are defined recursively in Figure 4.

We thus have syntactic GDSs $(\mathcal{V}_\sigma, (\!|\mathcal{V}_\sigma|\!), \longmapsto_\sigma^{\mathcal{V}})$ and $(\Lambda_\sigma, (\!|\Lambda_\sigma|\!), \longmapsto_\sigma^{\Lambda})$ which provide a differential model for Λ^T. Notice that the clauses for product and arrow types in Figure 4 correspond to the definition of cartesian products and exponentials in **GDS**.

Remark 2. Example 1 can be now straightforwardly translated in the language of DLRs. That highlights an important feature of GDSs, namely that programs may have *non-null self-distances*. For instance, a self-distance for a program $\lambda x.e$ is a function $\mathrm{d}e$ describing *how* $\lambda x.e$ reacts to *different inputs*. For instance, a self distance for $I \triangleq \lambda x.x$ is given by the map $\mathrm{d}I(v, \mathrm{d}v) \triangleq \mathrm{d}v$ (cf. with the identity morphism in **GDS**).

Fundamental Lemma and Compositionality. Before giving further examples of DLRs, we state the so-called fundamental lemma of DLRs, which will be the subject of Section 4.1.

Lemma 4.3 (Fundamental Lemma). *For any program e (resp. closed value) of type σ, there exists a distance $\mathrm{d}e \in (\!|\Lambda_\sigma|\!)$ (resp. $\mathrm{d}v \in (\!|\mathcal{V}_\sigma|\!)$) such that $e \overset{\mathrm{d}e}{\longmapsto}_\sigma e$ (resp. $v \overset{\mathrm{d}v}{\longmapsto}_\sigma v$).*

Why is Lemma 4.3 so important to deserve to be called Fundamental Lemma? First, let us observe that indeed Lemma 4.3 plays the same role played by the so-called fundamental lemmas in relational and metric scenarios. The latter state that expressions behave as morphisms in the underlying relational and metric models. Since morphisms in **GDS** are pairs of functions and their derivatives, to interpret a term e (and similarly for values) as a morphism in **GDS**, we need to ensure such a term to have a derivative/self-distance $\mathrm{d}e$. This is precisely what Lemma 4.3 does.[11] The reader may be puzzled by the fact that Lemma 4.3 gives just the existence of one (possibly not informative) self-distance of a term. This is indeed an important observation, that we will discuss in Section 4.1.

Besides its semantical interpretation, Lemma 4.3 ensures *compositionality* of differential reasoning about programs, whereby we can reason compositionally about the impact of replacing non-equivalent programs e, f for one another in a context C. Actually, a formal treatment of compositionality of DLRs requires to first extend DLRs to open expressions and then to generalise Lemma 4.3 to such an extension (Section 4.1). Nonetheless, for the sake of the argument, we can regard a context $x : \sigma \vdash C : \tau$ as the term $\lambda x.C : \sigma \to \tau$. Accordingly, we write $C[v]$ in place of $C[v/x]$ and $C \overset{\mathrm{d}C}{\longmapsto} C$ in place of $\lambda x.C \overset{\mathrm{d}C}{\longmapsto}_{\sigma\to\tau} \lambda x.C$. Let us now clarify what we mean by compositionality with a simple example. Let us consider the calculus with no effects and say we have the context $y : \mathbf{Real} \to \mathbf{Real} \vdash C : \mathbf{Real}$, with $C \triangleq \mathbf{let}\ x = \underline{y r}\ \mathbf{in}\ (\mathbf{let}\ z = (yx) + (yx)\ \mathbf{in}\ (\mathbf{val}\ z))$, and two functions $v, w : \mathbf{Real} \to \mathbf{Real}$ with distance $\mathrm{d}v$; what can we say about the distance between $C[v]$ and $C[w]$? That is, how $C[w]$ differs from $C[v]$, given that the "local" distance between v and w is $\mathrm{d}v$? Obviously, we may try to find a distance $\mathrm{d}C[v]$ between $C[v]$ and $C[w]$ from scratch. That, however, would be rather unsatisfactory due to the lack of compositionality. Indeed, the distance $\mathrm{d}C[v]$ is seen as a monolithic block, whereas we would like to describe it as a function of C, $\mathrm{d}v$, and possibly v and w. In fact, what seems relevant to compute distances between $C[v]$ and $C[w]$ is how the environment C uses its input (and thus how it modifies distances) and the concrete distance between the inputs. What we are looking for is thus a *differential analysis* of C, i.e. a function $\mathrm{d}C$ describing how C reacts to different inputs. Notice that $\mathrm{d}C$ depends on C only, and it is parametric with respect to the concrete inputs given: once we have such a differential analysis $\mathrm{d}C$, we can freely apply it to all desired inputs.

How should $\mathrm{d}C$ be defined? Let us try to understand its structure. First, $\mathrm{d}C$ should be parametric with respect to the inputs one passes to C and their distances. Actually, the very definition of DLRs shows that what is relevant for distances is just the first input and its distance with the second

[11] Actually, what we need is a generalisation of Lemma 4.3 to open expressions: such a generalisation is the content of Section 4.1.

one. Therefore, $\mathrm{d}C$ should belong to $\mathcal{V}_{\mathbf{Real}\to\mathbf{Real}} \times (\!|\mathcal{V}_{\mathbf{Real}\to\mathbf{Real}}|\!) \to (\!|\Lambda_{\mathbf{Real}}|\!)$. But the latter is exactly $(\!|\mathcal{V}_{(\mathbf{Real}\to\mathbf{Real})\to\mathbf{Real}}|\!)$. There is a natural subset of $(\!|\mathcal{V}_{(\mathbf{Real}\to\mathbf{Real})\to\mathbf{Real}}|\!)$ whose elements depend on C only, namely the set of *self-distances* for C, i.e. the collection of distances $\mathrm{d}C$ such that $C \overset{\mathrm{d}C}{\longmapsto} C$. Lemma 4.3 ensures that there always exists a self-distance for C, and thus it is always possible to perform a differential analysis of C.

Now that we have a differential analysis $\mathrm{d}C$ of C, we can indeed say something about the distance between $C[v]$ and $C[w]$. In fact, the very definition of DLRs ensures that $C[v] \overset{\mathrm{d}C(v,\mathrm{d}v)}{\longmapsto}_{\mathbf{Real}} C[w]$, meaning that $\mathrm{d}C(v, \mathrm{d}v)$ gives indeed a desired distance. Notice that such a distance has been obtained *compositionally* by means of two, decoupled analysis. On the one hand, we have the study of distances between v and w in *isolation*. On the other hand, we have the differential analysis (whose existence is guaranteed by Lemma 4.3) $\mathrm{d}C$ of C which is independent of the specific inputs v and w, and can thus be done once and for all. Concluding our example, we define $\mathrm{d}C$ by $\mathrm{d}C(v, \mathrm{d}v) \triangleq 2 \cdot \mathrm{d}v(v\underline{r}, \mathrm{d}v(\underline{r}, 0))$.

4.1 The Fundamental Lemma of Effectful Differential Logical Relations

In the next section, we will see several applications and examples of DLRs. Before doing so, however, we extend DLRs to open expressions and prove Lemma 4.3. The reader interested in examples of DLRs can jump to Section 5 and come back to this section on a second reading. To extend DLRs to open expressions, we first need to define GDSs for such expressions. We begin defining a GDS for typing environments.[12]

Definition 4.4. To any typing environment Θ we associate the GDS $(\mathsf{Subst}(\Theta), (\!|\Theta|\!), \longmapsto_\Theta)$ where $\mathsf{Subst}(\Theta) \triangleq \prod_{(x:\sigma)\in\Theta} \mathcal{V}_\sigma$, $(\!|\Theta|\!) \triangleq \prod_{(x:\sigma)\in\Theta} (\!|\mathcal{V}_\sigma|\!)$, and

$$[\boldsymbol{v}/\boldsymbol{x}] \overset{[\mathbf{d}\boldsymbol{v}/\boldsymbol{x}]}{\longmapsto}_\Theta [\boldsymbol{w}/\boldsymbol{x}] \overset{\triangle}{\iff} \forall (x_i : \sigma_i) \in \Theta.\ v_i \overset{\mathrm{d}v_i}{\longmapsto}_{\sigma_i} w_i$$

Next, we extend Definition 4.1 and Definition 4.2 to open terms and values.

Definition 4.5.

1. The distance spaces $(\!|\mathcal{V}_{\Theta\vdash\sigma}|\!)$ and $(\!|\Lambda_{\Theta\vdash\sigma}|\!)$ are defined thus:
$$(\!|\mathcal{V}_{\Theta\vdash\sigma}|\!) \triangleq \mathsf{Subst}(\Theta) \times (\!|\Theta|\!) \to (\!|\mathcal{V}_\sigma|\!) \qquad (\!|\Lambda_{\Theta\vdash\sigma}|\!) \triangleq \mathsf{Subst}(\Theta) \times (\!|\Theta|\!) \to (\!|\Lambda_\sigma|\!)$$
2. The relations $\longmapsto^{\mathcal{V}}_{\Theta\vdash\sigma} \subseteq \mathcal{V}_{\Theta\vdash\sigma} \times (\!|\mathcal{V}_{\Theta\vdash\sigma}|\!) \times \mathcal{V}_{\Theta\vdash\sigma}$ and $\longmapsto^{\Lambda}_{\Theta\vdash\sigma} \subseteq \Lambda_{\Theta\vdash\sigma} \times (\!|\Lambda_{\Theta\vdash\sigma}|\!) \times \Lambda_{\Theta\vdash\sigma}$ are defined thus:
$$v \overset{\mathrm{d}v}{\longmapsto}_{\Theta\vdash\sigma} w \overset{\triangle}{\iff} [\boldsymbol{v}/\boldsymbol{x}] \overset{[\mathbf{d}\boldsymbol{v}/\boldsymbol{x}]}{\longmapsto}_\Theta [\boldsymbol{w}/\boldsymbol{x}] \implies v[\boldsymbol{v}/\boldsymbol{x}] \overset{\mathrm{d}v([\boldsymbol{v}/\boldsymbol{x}],[\mathbf{d}\boldsymbol{v}/\boldsymbol{x}])}{\longmapsto}_\sigma w[\boldsymbol{w}/\boldsymbol{x}]$$
$$e \overset{\mathrm{d}e}{\longmapsto}_{\Theta\vdash\sigma} f \overset{\triangle}{\iff} [\boldsymbol{v}/\boldsymbol{x}] \overset{[\mathbf{d}\boldsymbol{v}/\boldsymbol{x}]}{\longmapsto}_\Theta [\boldsymbol{w}/\boldsymbol{x}] \implies e[\boldsymbol{v}/\boldsymbol{x}] \overset{\mathrm{d}e([\boldsymbol{v}/\boldsymbol{x}],[\mathbf{d}\boldsymbol{v}/\boldsymbol{x}])}{\longmapsto}_\sigma f[\boldsymbol{w}/\boldsymbol{x}]$$

We are now ready to prove the extension of Lemma 4.3 to open terms. It is important to stress that our proof crucially relies on the definition of a differential extension of a monad.

Proposition 4.6. *For all open terms and values $\Theta \vdash e : \sigma$ and $\Theta \vdash v : \sigma$, there exist distances $\mathrm{d}e \in (\!|\Lambda_{\Theta\vdash\sigma}|\!)$ and $\mathrm{d}v \in (\!|\mathcal{V}_{\Theta\vdash\sigma}|\!)$ such that $e \overset{\mathrm{d}e}{\longmapsto}_{\Theta\vdash\sigma} e$ and $v \overset{\mathrm{d}v}{\longmapsto}_{\Theta\vdash\sigma} v$.*

Proof Sketch. We proceed by simultaneous induction on the derivation of $\Theta \vdash e : \sigma$ and $\Theta \vdash v : \sigma$. The most challenging case is the one of sequencing which is handled relying on the differential extension of monad T. □

[12] In this section, we extensively employ vector-like notations. In particular, we write $\boldsymbol{s}$ for lists $s_1, \ldots, s_n$ of symbols (so that we agree that s_i denotes the i-th element of $\boldsymbol{s}$). In particular, we write $[\boldsymbol{v}/\boldsymbol{x}]$ for the substitution simultaneously replacing variables $x_1, \ldots, x_n$ with values $v_1, \ldots, v_n$.

Notice that Lemma 4.3 immediately follows from Proposition 4.6, from which it also follows that for any term $x_1 : \sigma_1, \ldots, x_1 : \sigma_1 \vdash e : \sigma$ we have (at least!) one arrow

$$\prod_i (\mathcal{V}_{\sigma_i}, (\!|\mathcal{V}_{\sigma_i}|\!), \longmapsto^{\mathcal{V}}_{\sigma_i}) \xrightarrow{(e, \mathrm{d}e)} (\Lambda_\sigma, (\!|\Lambda_\sigma|\!), \longmapsto_\sigma)$$

which, essentially, is a morphism to $(T(\mathcal{V}_\sigma), (\!|T(\mathcal{V}_\sigma)|\!), \Longmapsto_\sigma)$. Notice also that the we have the following compositionality principle, which we have informally discussed in previous section and that we can finally write explicitly as

$$C \overset{\mathrm{d}C}{\longmapsto}_{x:\sigma\vdash\tau} C \;\&\; e \overset{\mathrm{d}e}{\longmapsto}_\sigma f \implies C[e] \overset{\mathrm{d}C(e,\mathrm{d}e)}{\longmapsto}_\tau C[f].$$

Trivial and Optimal Self-distances. Before moving to applications of DLRs, we comment on Proposition 4.6. In fact, the value of this result is not in the statement of the proposition, but in its proof. The proof of Proposition 4.6 shows us how starting with a distance at the ground type **Real**, one hereditary extends it at all types. Obviously, the more informative the ground distance chosen, the more informative its inductive extension. For instance, by changing the GDS associated to $\mathcal{V}_{\mathbf{Real}}$ stipulating that, e.g., $\underline{r} \overset{\mathrm{d}r}{\longmapsto}_{\mathbf{Real}} \underline{s} \overset{\triangle}{\iff} \mathrm{d}r = s - r$, then distances are *a fortiori* informative. Please notice that Proposition 4.6 keeps holding *mutatis mutandi* even with the aforementioned modification of the GDS of $\mathcal{V}_{\mathbf{Real}}$.

The defining axioms of a differential extension play a key role here: they tell us how to extend a distance on 'ground objects' to a distance on 'monadic objects'. Again, the degree of information given by the resulting distance depends on the concrete differential extension chosen. But even if we start with the trivial, infinity distance on **Real**, to extend it at all types we still need to have a differential extension of the underlying monad.

Altogether, these observations raises some questions, the most crucial one being: do we have optimal, most informative distances? First, notice that such a question may not be semantically meaningful, as not all distance spaces may come with a notion of order between distances. For instance, in the case of coupling-based differential extensions, extending an order on $(\!|A|\!)$ to $T(A \times (\!|A|\!))$ is nontrivial, and often relies on suitable relational extensions of T (which are themselves defined using coupling-based methods). On the other hand, however, for some monads — such as the output, cost, and maybe monad — there are natural extensions of orders on $(\!|A|\!)$ to $(\!|TA|\!)$. In these cases, one can compare self-distances and ask whether there exists an optimal one. The answer to this question is in the affirmative for the aforementioned monads.

Proposition 4.7 (Strong Fundamental Lemma). *Let us consider Λ^T restricted to one of the following monads: identity, maybe, output, and cost. For all open terms and values $\Theta \vdash e : \sigma$ and $\Theta \vdash v : \sigma$, there exist optimal distances $\partial e \in (\!|\Lambda_{\Theta\vdash\sigma}|\!)$ and $\partial v \in (\!|\mathcal{V}_{\Theta\vdash\sigma}|\!)$ such that $e \overset{\partial e}{\longmapsto}_{\Theta\vdash\sigma} e$ and $v \overset{\partial v}{\longmapsto}_{\Theta\vdash\sigma} v$.*

Proof sketch. In the case of the identity monad, we observe that $[0, \infty]$ is a complete lattice, and that such a structure can be extended to both products and function spaces. As a consequence, given two terms e, f, we can consider the meet of distances between them: $\partial(e, f) \triangleq \bigwedge_{e \overset{\mathrm{d}e}{\longmapsto}_\sigma f} \mathrm{d}e$. We then follow the same strategy of the proof of Proposition 4.6 and prove that $e \overset{\partial e}{\longmapsto}_\sigma f$ holds. The same strategy can be applied to the the cost and output monads simply observing that $(\!|\mathcal{O}(\mathcal{V}_\sigma)|\!) = [0, \infty] \times (\!|\mathcal{V}_\sigma|\!)$ is a complete lattice. This is not anymore the case when one moves to the maybe monad $\mathcal{M}$. Nonetheless, $(\!|\mathcal{M}(\mathcal{V}_\sigma)|\!)$ is a dcppo [Davey and Priestley 1990] and the set of self-distances of a term e is directed (and similarly for values), meaning that it has a greatest lower bound ∂e, which is itself a self-distance of e. □

Proposition 4.7 is a useful result, which we will extensively use in the next section. An interesting question is whether one can find some general conditions on monads and their differential extensions that guarantee Proposition 4.7 to hold. Finding such conditions seems nontrivial, and we will leave that for future work.

5 DIFFERENTIAL LOGICAL RELATIONS AT WORK

In previous sections, we have seen some examples and applications of differential semantics and DLRs. For instance, in Example 1 we have seen how GDSs (and thus DLRs) can be used to formally justify the approximate program transformations seen in the introduction of this work; and in Section 3.3 we have seen how differential semantics allows for generalisations of the notion of expected sensitivity. In this section, we give further examples and applications of DLRs and differential semantics. Our applications come from the field of approximate computing, probabilistic programming, and (relational) cost analysis.

5.1 Cost Analysis I: Adaptive Sorting

To reason about program efficiency, we consider the cost monad, so that the evaluation semantics of a program e consists of a pair (n, v), where v is the actual value computed by e and n is the cost of the computation. To actually compute the cost of a computation, we consider the generic effect operation **tick** : **Unit**. Once evaluated, **tick** returns the pair $(1, \langle\rangle)$ made of the unit value $\langle\rangle$ and a unit of cost. By a suitable decoration of programs, we can use **tick** to keep track of the number of computation steps performed. For instance, if we replace any application occurrence vw with **tick**; vw, where we write $e; f$ for the trivial sequencing, then we see that if a program e evaluates to (n, v), then it performs n computation steps to reduce to v. Notice that with the exact the same strategy, we can use **tick** to count other operations (for instance, we may count how many probabilistic choices a program fires during its execution). For readability, we sometimes write $cost(e)$ for the number n such that $[\![e]\!] = (n, v)$, and usually do not explicit write occurrences of **tick** in programs. Nonetheless, we assume applications to be suitably decorated with ticking operations.

We work with an extension of Λ_Σ with the type **Nat** of natural numbers and types $\sigma^{\boldsymbol{m}}$ for lists of fixed length. We associate to **Nat** the GDS $(\mathcal{V}_{\mathbf{Nat}}, \mathbb{N}, \overset{\mathcal{V}}{\longmapsto}_{\mathbf{Nat}})$, where $\underline{n} \overset{\mathrm{d}n}{\longmapsto}_{\mathbf{Nat}} \underline{m} \overset{\triangle}{\iff} \mathrm{d}n = m - n$. The type $\sigma^{\boldsymbol{m}}$ is the type of lists of type σ of fixed length m. Morally, $\sigma^{\boldsymbol{m}}$ is nothing but the arrow type $\boldsymbol{m} \to \sigma$ (with $\boldsymbol{m}$ being the type corresponding to the set $\{1, \ldots, m\}$), except for the fact that an inhabitant xs of $\sigma^{\boldsymbol{m}}$ associates to each $1 \leq i \leq m$ a *value* $xs[i]$ of type σ. We define $(\!|\mathcal{V}_{\sigma^{\boldsymbol{m}}}|\!) \triangleq (\!|\mathcal{V}_\sigma|\!)^{\{1,\ldots,m\}\times\{-m,\ldots,m\}}$. A distance $\mathrm{d}xs$ between two lists xs and ys of type $\sigma^{\boldsymbol{m}}$ takes in input an index $i \leq m$ and an index-distance $\mathrm{d}i$, and returns a distance $\mathrm{d}xs(i, \mathrm{d}i)$ between $xs[i]$ and $ys[i + \mathrm{d}i]$ [Dal Lago and Gavazzo 2021a]:

$$xs \overset{\mathrm{d}xs}{\longmapsto}_{\sigma^{\boldsymbol{m}}} ys \iff \forall i \in \{1, \ldots, m\}, \mathrm{d}i \in \{-m, \ldots, m\}.\ xs[i] \overset{\mathrm{d}xs(i,\mathrm{d}i)}{\longmapsto}_{\sigma} ys[i + \mathrm{d}i].$$

We show how to use DLRs to reason about the *adaptive cost* of sorting algorithms [Estivill-Castro and Wood 1992]. In particular, we show how to use DLRs to prove that for suitable families of lists, insertion sort is an efficient sorting algorithm. Let us consider the following implementation of insertion sort.

$$\begin{aligned}&\mathbf{isort} : \mathit{List}(\mathbf{Nat}) \to \mathit{List}(\mathbf{Nat})\\ &\mathbf{isort} = \lambda\ell.\mathbf{case}\ \ell\ \mathbf{of}\ (\mathsf{nil} \Rightarrow \mathsf{nil} \mid x :: xs \Rightarrow \mathbf{insert}(x, \mathbf{isort}(xs)))\end{aligned}$$

$$\begin{aligned}
&\mathbf{insert} : \mathbf{Nat} \times \mathit{List}(\mathbf{Nat}) \Rightarrow \mathit{List}(\mathbf{Nat})\\
&\mathbf{insert} = \lambda(x, \ell).\mathbf{case}\ \ell\ \mathbf{of}\\
&\qquad\qquad \mathsf{nil} \Rightarrow [x]\\
&\qquad\qquad \mid y :: ys \Rightarrow \mathbf{if}\ x \leq y\ \mathbf{then}\ x :: y :: ys\ \mathbf{else}\ y :: \mathbf{insert}(x, ys)
\end{aligned}$$

Insertion sort is an *adaptive algorithm*, meaning that its running time depends on its input: the more the input list is ordered, the better performances insertion sort has on that list. Let us expand on that. Suppose to have a list $xs = [x_1, \ldots, x_n]$. Then, $\mathbf{isort}(xs)$ proceeds by traversing xs backward, so that when examining an element x_i, the sub-list $[x_{i+1}, \ldots, x_n]$ is already sorted — say obtaining a list $ys = [y_1, \ldots, y_{n-i}]$ — and the function **insert** is called on x_i and ys. The latter function inserts x_i in ys by permuting x_i with $y_1, y_2, \ldots$ until the new list is sorted. As a consequence, the cost of $\mathbf{isort}(xs)$ is given by a linear function in the number of permutations each call to **insert** makes plus $|xs|$ (the latter capturing the cost of the remaining operations). The aforementioned permutations are known as *inversions* [Sedgewick and Flajolet 2013] and are used to measure the degree of (dis)order of a list. More formally, given a list xs, an inversion of xs is any pair of entries of xs that are out of order. The inversion number I of xs is $|\{(i, j) \mid i < j \wedge xs[i] > xs[j]\}|$.

For our analysis, it is convenient to count the inversions with respect to a given index i. We thus define $I_i = |\{j \mid i < j \wedge xs[i] > xs[j]\}|$, so that $I = \sum_i I_i$. Coming back to our analysis of **isort**, when we call **insert** on x_i and the (sorted) list ys, the number of operations we make is essentially the number of inversions of $[x_{i+1}, \ldots, x_n]$ with respect to i, that is the number of elements in $[x_{i+1}, \ldots, x_n]$ that are strictly bigger than x_i. But that number is I_i, so that we can express the cost of $\mathbf{isort}(xs)$ as $F(I + n) = F(\sum_{i=1}^{n} I_i + n)$, for a suitable linear function F.

But what about DLRs? The key observation is that we can use self distances $\mathrm{d}xs$ of xs to obtain information on the trend of xs, and thus on its inversion number. Recall that for any self-distance $\mathrm{d}xs$ of a list xs we have $\mathrm{d}xs(i, \mathrm{d}i) = xs[i + \mathrm{d}i] - xs[i]$, for all $i, \mathrm{d}i$. As a consequence, $xs[i] \geq xs[i + \mathrm{d}i]$ if and only if $\mathrm{d}xs(i, \mathrm{d}i) = 0$, and thus xs is sorted if and only if $\forall i < |xs|.\ \mathrm{d}xs(i, 1) = 0$. If we additionally require $\mathrm{d}xs(i + \mathrm{d}i, -\mathrm{d}i) > 0$, then we also obtain that $xs[i] > xs[i + j]$, so that we recover I_i as $|\{\mathrm{d}i \leq n - i \mid \mathrm{d}xs(i, \mathrm{d}i) = 0 \wedge \mathrm{d}xs(i + \mathrm{d}i, -\mathrm{d}i) > 0\}|$. We can thus characterise the cost of $\mathbf{isort}(xs)$ as

$$F(\sum_{i=1}^{n} |\{\mathrm{d}i \leq n - i \mid \mathrm{d}xs(i, \mathrm{d}i) = 0 \wedge \mathrm{d}xs(i + \mathrm{d}i, -\mathrm{d}i) > 0\}|)$$

for a suitable linear function F. Notice that the cost of $\mathbf{isort}(xs)$ indeed depends on xs, and actually on $\mathrm{d}xs$, the latter describing how much xs is ordered. For instance, in the worst case scenario, we have $I_i = n - 1$ (meaning that all elements in (sub)list should be permuted) which gives $\mathbf{isort}(\ell) \in O(n^2)$.

Using DLRs we can make the cost analysis of **isort** *truly input-dependent* by using the self-distance $\mathrm{d}xs$ to formalize specific classes of lists. For instance, we may say that a list is *almost ordered* if $I_i = \log_2(n - i)$. In this case, we see that if xs is almost ordered, then $\mathbf{isort}(xs) \in O(n \log_2 n)$. Similarly, if xs is already sorted, then there is no inversion and thus $I_i = 0$. As a consequence, we see that $\mathbf{isort}(\ell) \in O(n)$, meaning that if xs is sorted, then $\mathbf{isort}(xs)$ runs in linear time.

Finally, we can incorporate these results in a differential analysis between **isort** and other sorting algorithms. For instance, let **msort** be an implementation of mergesort and let xs be a list of length n as above. We know that mergesort is not adaptive, so that the cost of $\mathbf{msort}(xs)$ will depend on n only. In particular, we know that $\mathit{cost}(\mathbf{msort}(xs)) = G(n \log_2 n)$, for some suitable linear function G. We also know that **msort** and **isort** are extensionally equivalent. As we are interested in passing to **msort** and **isort** the *same* list as input (so to compare their performances), we can simplify our analysis by looking at candidate distances between them that behave as expected only on

extensionally equivalent lists. Define:

$$\mathbf{disort}(xs, \mathrm{d}xs) = \left(G(n \log_2 n) - F(n + \sum_{i=1}^{n} I_i), (i, \mathrm{d}i) \mapsto xs^{\mathrm{s}}[i + \mathrm{d}i] - xs^{\mathrm{s}}[i] \right)$$

where xs^{s} is the sorted version of xs (also, recall that I_i can be given in terms of $\mathrm{d}xs$). Notice that **disort** is not a distance between **isort** and **msort**, as **disort** does not behave well when we pass to **isort** and **msort** different lists. However, it does so if we pass them the same list ℓ. Does that mean that to complete our analysis we need both ordinary and differential logical relations, the former to reason about list equivalence and the latter to make cost analysis? No, we can use DLRs to express extensional equality between programs and thus formulate our relational cost analysis entirely in terms of DLRs. In fact, it is sufficient to observe that two values $\underline{n}, \underline{m}$ of type **Nat** are equal if and only if both $\underline{n} \overset{0}{\mapsto}_{\mathbf{Nat}} \underline{m}$ and $\underline{m} \overset{0}{\mapsto}_{\mathbf{Nat}} \underline{n}$ hold. Similarly, given $xs \overset{\mathrm{d}xs}{\longmapsto}_{\mathbf{Nat}^{\mathbf{m}}} ys$, we use $\mathrm{d}xs(i, 0)$ to compare $xs[i]$ and $ys[i]$, so that we have xs is extensionally equivalent to ys if and only if there exist distances $\mathrm{d}xs, \mathrm{d}ys$ such that:

$$(xs \overset{\mathrm{d}xs}{\longmapsto}_{\mathbf{Nat}^{\mathbf{m}}} ys) \wedge (ys \overset{\mathrm{d}ys}{\longmapsto}_{\mathbf{Nat}^{\mathbf{m}}} xs) \wedge (\forall i \leq m.\ \mathrm{d}xs(i, 0) = 0 \wedge \mathrm{d}ys(i, 0) = 0).$$

Putting all the pieces together, we obtain a cost analysis **disort** for **isort** and **msort**. In particular, for any list xs the first component of $\mathbf{disort}(xs, \mathrm{d}xs)$ gives the distance between the cost of $\mathbf{isort}(xs)$ and $\mathbf{msort}(xs)$, where we remark that I_i is defined in terms of $\mathrm{d}xs$. Finally, given a context C, we can then compute the distance cost between $C[\mathbf{isort}]$ and $C[\mathbf{msort}]$ using a self-distance $\mathrm{d}C(\mathbf{isort}, \mathbf{disort})$. This analysis goes beyond what standard operational techniques — such as improvement theory [Sands 1998] — the latter being *context-independent*. For instance, improvement theory tells only when a program performs better than another one on any possible input, and thus it gives no useful information on the relationship between **msort** and **isort**. On the contrary, DLRs provide a genuine context-dependent analysis. For instance, if C passes to its argument an already sorted list, then $\mathrm{d}C(\mathbf{isort}, \mathbf{disort})$ tells us that $C[\mathbf{isort}]$ is asymptotically better than $C[\mathbf{msort}]$.

5.2 Numeric Integration

In this section, we show how EDLR can help us in the differential analysis of approximate integration as studied by Zhu et al. [2012]. Given a function $f : \mathbb{R} \to \mathbb{R}$, we want to numerically integrate f over a fixed (non-negative interval) $[a, b]$. A standard way to do so is by dividing the interval $[a, b]$ into n equal-sized subintervals of length $\frac{b-a}{n}$ and then to calculate f on those intervals. The integral can be thus computed by the program:

$$\mathbf{integral}\ f\ a\ b \triangleq \mathbf{let}\ x_i = a + i \cdot \frac{b-a}{2n}\ \mathbf{in}\ \frac{1}{n} \sum_{i=1}^{n} (b-a) \cdot f(x_i).$$

Zhu et al. [2012] propose two general approximate transformations for the purpose of improving the overall efficiency. The first technique — called *substitution* — replaces the function f with a function g approximating f. For instance, if f is the reciprocal square root function $x \mapsto \frac{1}{\sqrt{x}}$ we may replace it with $x \mapsto 2.08 - 1.1911x$. We have already seen that a formal account for this technique can be given in terms of pure DSs. The second technique studied — called *sampling* — approximates $\mathbf{integral}\ f\ a\ b$ with a *probabilistic program* that instead of computing f on each x_i, it does so on only $s < n$ randomly chosen $x_{i_1}, \ldots, x_{i_s}$. Due to its probabilistic nature, a formal account of sampling cannot be given in terms of pure GDSs (and thus pure DLRs). We show that effectful (notably probabilistic) GDSs enable a formal analysis of sampling. Using that result, we also show how to give a highly compositional analysis of approximate transformations using *both* substitution and sampling.

Substitution. Let us assume parameters a, b to be fixed, so and let $C : (\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$ be the function computed by $\lambda x.\mathbf{integral}\ x\ a\ b$. For simplicity, we work with the GDS $(\mathbb{R}, [0,\infty], R)$, where $R(x, \mathrm{d}x, y) \overset{\triangle}{\iff} \mathrm{d}x = y - x$ (but our argument scales *mutatis mutandis* to, e.g., the GDS defined by taking $R(x, \mathrm{d}x, y) \overset{\triangle}{\iff} \mathrm{d}x = |y - x|$). We first give a differential analysis $\mathrm{d}C : (\mathbb{R} \to \mathbb{R}) \times (\!|\mathbb{R} \to \mathbb{R}|\!) \to (\!|\mathbb{R}|\!)$ of C. Define:

$$\mathrm{d}C(f, \mathrm{d}f) \triangleq \frac{1}{n}\sum_{i=1}^{n}(b-a)\cdot(f(x_i) + \mathrm{d}f(x_i, 0)) - \frac{1}{n}\sum_{i=1}^{n}(b-a)\cdot f(x_i) = \frac{1}{n}\sum_{i=1}^{n}(b-a)\cdot \mathrm{d}f(x_i, 0).$$

Notice that we indeed have $[[R \to R] \to R]\ \ (C, \mathrm{d}C, C)$. In particular, given functions f, g with distance $\mathrm{d}f$, a distance between $C(f)$ and $C(g)$ is given by $\mathrm{d}C(f, \mathrm{d}f)$.

Sampling. Let $D : (\mathbb{R} \to \mathbb{R}) \to \mathcal{D}(\mathbb{R})$ be the function implemented by the following probabilistic program

$$\begin{array}{ll} \mathbf{let} & x_i = a + i \cdot \frac{b-a}{2n} \\ & [j_1, \ldots, j_s] = [\mathbf{sample}(1, \ldots, n), \ldots, \mathbf{sample}(1, \ldots, n)] \\ \mathbf{in} & \frac{1}{n}\sum_{k\in[j_1,\ldots,j_s]}(b-a)\cdot f(x_k) \end{array}$$

We write $\boldsymbol{n}^{\boldsymbol{s}}$ for the set of lists $[j_1, \ldots, j_n]$ of length s and with elements in $\{1, \ldots, n\}$. Notice that we have $D(f) = \sum_{[j_1,\ldots,j_s]\in\mathbf{n}^{\mathbf{s}}} \frac{1}{n^s} \cdot \delta_{\frac{1}{n}\sum_{k\in[j_1,\ldots,j_s]}(b-a)\cdot f(x_k)}$. To compare C and D, we regard C as trivially probabilistic. A distance between C and D is a map

$$\mathrm{d}C : (\mathbb{R} \to \mathbb{R}) \times (\mathbb{R} \times [0,\infty] \to [0,\infty]) \to \mathcal{D}(\mathbb{R} \times [0,\infty])$$

Define:

$$\mathrm{d}C(f, \mathrm{d}f) \triangleq \sum_{\{j_1,\ldots,j_s\}\in\boldsymbol{n}^{\boldsymbol{s}}} \frac{1}{n^s} \cdot \delta_{\left(\frac{1}{n}\sum_{i=1}^{n}(b-a)\cdot f(x_i),\ \frac{b-a}{n}\left(\sum_{k\in\{j_1,\ldots,j_s\}}\mathrm{d}f(x_k,0) - \sum_{i\in\boldsymbol{n}\setminus\{j_1,\ldots,j_s\}} f(x_i)\right)\right)}$$

The distribution $\mathrm{d}C(f, \mathrm{d}f)$ corresponds to the plan moving weight $\frac{1}{n^s}$ from $\frac{1}{n}\sum_{i=1}^{n}(b-a)\cdot f(x_i)$ along the edge $\frac{b-a}{n}\left(\sum_{k\in\{j_1,\ldots,j_s\}}\mathrm{d}f(x_k,0) - \sum_{i\in\boldsymbol{n}\setminus\{j_1,\ldots,j_s\}} f(x_i)\right)$. If we are given a function g such that $\mathrm{d}f$ is a distance between f and g, then we can apply the plan $\mathrm{d}C(f, \mathrm{d}f)$ to $C[f]$ to recover $D[g]$. In fact, if $[R \to R]\ (f, \mathrm{d}f, g)$, then

$$\frac{b-a}{n}\cdot\left(\sum_{k\in\{j_1,\ldots,j_s\}}\mathrm{d}f(x_k,0) - \sum_{i\in\boldsymbol{n}\setminus\{j_1,\ldots,j_s\}} f(x_i)\right) = \frac{1}{n}\sum_{k\in\{j_1,\ldots,j_s\}}(b-a)\cdot g(x_k) - \frac{1}{n}\sum_{i=1}^{n}(b-a)\cdot f(x_i).$$

We can then build the ternary coupling completing the plan $\mathrm{d}C(f, \mathrm{d}f)$ by assigning probability $\frac{1}{n^s}$ to triples

$$\left(\frac{1}{n}\sum_{i=1}^{n}(b-a)f(x_i),\ \frac{b-a}{n}\left(\sum_{k\in\{j_1,\ldots,j_s\}}\mathrm{d}f(x_k,0) - \sum_{i\in\boldsymbol{n}\setminus\{j_1,\ldots,j_s\}} f(x_i)\right),\ \frac{1}{n}\sum_{k\in\{j_1,\ldots,j_s\}}(b-a)g(x_k)\right)$$

We can now make a compositional analysis of performing both substitution and sampling: given a substitution-based approximation of f by g with error $\mathrm{d}f$ and a sampling-based approximation of C by D with error $\mathrm{d}C$, what is the total error obtained by approximating $C[f]$ by $D[g]$? By compositionality/chain rule, such an error is given by $\mathrm{d}C(f, \mathrm{d}f)$. Notice that the error of performing a mixed approximation based on both substitution and sampling is computed compositionally in terms of the differential analyses of substitution and sampling, which are now completely decoupled.

5.3 Cost Analysis II: The Sieve of Eratosthenes

Our last example is taken from the work by Çiçek et al. on relational cost analysis [Çiçek et al. 2017] and highlights how DLRs can be used to mix *extensional* and *intensional* analysis of programs. We study the higher-order program **erat** implementing the sieve of Eratosthenes:

$$\mathbf{erat} : (\mathbf{Nat} \times \mathit{List}(\mathbf{Nat}) \to \mathit{List}(\mathbf{Nat})) \times \mathit{List}(\mathbf{Nat}) \to \mathit{List}(\mathbf{Nat})$$
$$\mathbf{erat} = \lambda(d, ys).\mathbf{case}\ xs\ \mathbf{of}\ (\ \mathsf{nil} \Rightarrow \mathsf{nil} \mid x :: xs \Rightarrow x :: (\mathbf{erat}(d, (d(x, xs)))))$$

The program **erat** takes in input a drop function $d : \mathbf{Nat} \times \mathit{List}(\mathbf{Nat}) \to \mathit{List}(\mathbf{Nat})$ and a list xs of the form $[2, 3, \ldots, n]$ containing all natural numbers that are less than or equal to a given number n, and returns the list of all prime numbers in the list. It does so by recursively applying the drop function d, which takes as input a natural number m and a list h, and returns the list obtained from h by removing all multiples of m.

Our goal is to make a cost analysis of **erat** in terms of the relative cost of the drop function. That is, suppose to have two *extensionally/functionally* equivalent implementations $\mathbf{drop}_1, \mathbf{drop}_2$ of the drop function, which are *intensionally* different, meaning that they have different costs (for instance, $\mathbf{drop}_2$ may be more efficient than $\mathbf{drop}_1$). What can we say about the performances of $\mathbf{erat}(\mathbf{drop}_1, xs)$ and $\mathbf{erat}(\mathbf{drop}_2, xs)$ in terms of the relative cost of $\mathbf{drop}_1$ and $\mathbf{drop}_2$? Notice that answering this question requires us to analyse both about extensional (i.e. equivalence) and intensional (i.e. cost) properties of programs. As we did when working with sorting algorithms, we actually work with lists of fixed length. As before, we reason about the cost of programs using the ticking monad so that we can assume the two aforementioned implementations $\mathbf{drop}_1, \mathbf{drop}_2 : \mathbf{Nat} \times \mathit{List}(\mathbf{Nat}) \to \mathit{List}(\mathbf{Nat})$ of the drop function to be suitably decorated with occurrences of the ticking operation. We have already remarked in Section 5.1 that we can use DLRs to express extensional equality between programs.

Let us now analyse the two implementations $\mathbf{drop}_1, \mathbf{drop}_2$ of the drop function. Recall that given lists xs_1, xs_2 with distance $\mathrm{d}xs$ and numbers n_1, n_2 with distance $\mathrm{d}n$, a distance between $\mathbf{drop}_1$ and $\mathbf{drop}_2$ is a map $\mathrm{d}f$ such that $\mathrm{d}f(n_1, xs_1, \mathrm{d}n, \mathrm{d}xs)$ gives a pair $(\mathrm{d}k, \mathrm{d}v)$ consisting of a bound $\mathrm{d}k$ for $\mathit{cost}(\mathbf{drop}_1(n_1, xs_1)) - \mathit{cost}(\mathbf{drop}_2(n_2, xs_2))$ and a distance $\mathrm{d}v$ between the values v_1, v_2 computed by $\mathbf{drop}_1(n_1, xs_1)$ and $\mathbf{drop}_2(n_2, xs_2)$, respectively. Putting this observations together, what we need are distances $\mathbf{drop}_1 \overset{\mathrm{d}f}{\longmapsto}_{\mathbf{Nat}\times\mathbf{Nat}^{\mathcal{P}}\to\mathbf{Nat}^{\mathcal{P}}} \mathbf{drop}_2$ such that for all $xs_1, xs_2, \mathrm{d}xs, n_1, n_2, \mathrm{d}n$ as above, we have that $n_1 = n_2$ and $xs_1 \simeq xs_2$ imply $v_1 \simeq v_2$, for v_1, v_2 as above. Notice that v_1, v_2 are lists of numbers, and thus we can use $\mathrm{d}f(n_1, xs_1, m, \varphi) = (\mathrm{d}k, \mathrm{d}v)$ (most notably, $\mathrm{d}v$) to express such an equivalence. The distance $\mathrm{d}f$ thus plays a double role: on the one hand it is used to witness that $\mathbf{drop}_1$ and $\mathbf{drop}_2$ are *extensionally equivalent*, as they map related inputs to related outputs. On the other hand, the number $\mathrm{d}k$ gives an *intensional* distance between $\mathbf{drop}_1$ and $\mathbf{drop}_2$. For instance, if $\mathrm{d}k > 0$, then we not only know that $\mathbf{drop}_2$ is more efficient than $\mathbf{drop}_1$, but also *how much* $\mathbf{drop}_2$ is more efficient than $\mathbf{drop}_1$, this way going beyond improvement theory.

Next, given an intensional analysis $\mathrm{d}f$ of $\mathbf{drop}_1$ and $\mathbf{drop}_2$, what can we say about the intensional behaviour of **erat**? Let us consider the differential analysis $\mathbf{derat}$ of **erat**, which exists by Lemma 4.3 (and can actually take as optimal by Proposition 4.7). Given a list xs, $\mathbf{derat}(\mathbf{drop}_1, xs, \mathrm{d}f, \mathrm{d}xs)$ gives a cost analysis of $\mathbf{erat}(\mathbf{drop}_1, xs)$ and $\mathbf{erat}(\mathbf{drop}_2, xs)$ in terms of the cost analysis $\mathrm{d}f$ of $\mathbf{drop}_1$ and $\mathbf{drop}_2$. As a consequence, the desired cost analysis is given as: $\pi_1(\mathbf{derat}(\mathbf{drop}_1, xs, \mathrm{d}f, \mathrm{d}xs))$.

Finding an explicit formula for $\mathbf{derat}$, however, may be complicated. Nonetheless, we can easily circumnavigate the problem by reasoning in an approximate, yet informative way. Suppose we decorate **erat** with ticking operations for counting the number of times the drop function is called. Then, given an input list xs and a drop function **drop**, the cost of $\mathbf{erat}(\mathbf{drop}, xs)$ will be bounded by the number of times **erat** calls **drop** multiplied by the cost of **drop** (when applied

to a specific input). We can approximate such a cost as follows, where we write $xs \ominus xs_{1,\dots,i}$ for the list $xs \setminus [xs(1),\dots,xs(i)]$: $cost(\mathbf{erat}(\mathbf{drop}, xs)) \leq \sum_{i=1}^{|xs|} cost(\mathbf{drop}(xs(i), xs \ominus xs_{1,\dots,i-1}))$. Moreover, since the distance $\mathrm{d}f$ gives the relative cost of $\mathbf{drop}_1$ and $\mathbf{drop}_2$ on an input n, xs as $\pi_1(\mathrm{d}f(n, xs, 0, \mathrm{d}xs))$, we obtain an analysis of the relative cost of $\mathbf{erat}(\mathbf{drop}_1, xs)$ and $\mathbf{erat}(\mathbf{drop}_2, xs)$ as

$$\begin{aligned}
&\pi_1(\mathbf{derat}(\mathbf{drop}_1, xs, \mathrm{d}f, \mathrm{d}xs))\\
&\leq \sum_{i=1}^{|xs|} cost(\mathbf{drop}_1(xs(i), xs \ominus xs_{1,\dots,i-1})) - cost(\mathbf{drop}_1(xs(i), xs \ominus xs_{1,\dots,i-1}))\\
&= \sum_{i=1}^{|xs|} \pi_1(\mathrm{d}f(xs(i), xs \ominus xs_{1,\dots,i-1}, 0, \mathrm{d}(xs \ominus xs_{1,\dots,i-1}))).
\end{aligned}$$

6 RELATED WORK

Semantical approaches to program distancing and program approximation for higher-order languages have been proposed in a number of recent papers. Closest to our work are the work by Dal Lago and Gavazzo [2021a]; Dal Lago et al. [2019]; Pistone [2021] on differential logical relations, and the unpublished work by Westbrook and Chaudhuri [2013] that inspired the former. There are several dissimilarities between the present paper and the aforementioned work on differential logical relations, the main one being that the latter can deal with *pure* languages only. As a consequence, previous accounts of DLRs can used to perform neither differential analysis of, e.g., probabilistic computations, nor (relational) cost analysis. Additionally, defining satisfactory effectful extensions of such accounts seems hardly possible. In fact, much in the same way we have built upon suitable extensions of monads to the category of GDSs, one may expect a similar approach to work on the aforementioned notions of DLRs. However, the resulting notions of monadic DLRs turned out to be unable to account for almost all interesting notions of effects. That is due to the strong conditions such accounts imposed on DLRs — the main one being that distance spaces are required to form a quantale — which rule out essentially all the examples analysed in the present work. Our notion of a GDS generalises all previous proposals of (semantical foundations of) DLRs, and can be thus seen as a minimal structure to perform differential analysis of higher-order programs.

All the aforementioned works on differential semantics have their roots in program metrics [Chatzikokolakis et al. 2014; de Amorim et al. 2017; Desharnais et al. 2004; Du et al. 2016; Gebler et al. 2016; Van Breugel and Worrell 2005], and, in particular, in the work by Chaudhuri et al. [Chaudhuri et al. 2010, 2011] on program robustness and by Reed and Pierce [Reed and Pierce 2010] on language-based differential privacy. Both these works study how distances in inputs impact distance in outputs of programs, and identify the notions of program sensitivity and robustness as a central tool for such an analysis. Their approach has been developed on several axes, including type theory [Barthe et al. 2015; D'Antoni et al. 2013; Gaboardi et al. 2013], logic [Barthe et al. 2014, 2013; Zhang et al. 2019], and effectful extensions of languages [Barthe et al. 2018, 2016; de Amorim et al. 2019; Gavazzo 2018; Sato et al. 2019]. We showed in Section 3.3 how program sensitivity can be expressed in terms of differential logical relations, and also how the latter allows us to talk about local and expected notions of program sensitivity.

Approximate program transformations have also been studied in recent works by Rinard et al. [Misailovic et al. 2011; Rinard 2011; Sidiroglou-Douskos et al. 2011]. There, the focus is on probabilistic analyses of loop perforation techniques aiming to produce probabilistic guarantees on the distance between the original and the perforated programs. Other, deterministic, analyses of loop perforations techniques include the already mentioned work by Chaudhuri and Westbrook

[Chaudhuri et al. 2011; Westbrook and Chaudhuri 2013]. We are currently investigating whether the results by Rinard et al. can also be obtained using differential logical relations along the lines of our analysis in Section 5. There are also a number of works on language-based approximate computing [Boston et al. 2015; Carbin et al. 2013; Sampson et al. 2011]. In particular, languages such as EnerJ [Sampson et al. 2011] and Rely [Carbin et al. 2013] use type modifiers to specify whether data are exact or inexact, as well as the probability that programs are reliable. Such languages naturally support static reasoning about approximate programs. In addition to that, probabilistic logics [Carbin et al. 2012, 2013; Sampson et al. 2014] have been designed for reasoning about them.

Finally, (higher-order) relational logics [Aguirre et al. 2019; Benton 2004] have been used not only to reason about program sensitivity and approximate computing, but also for cost analysis [Çiçek et al. 2017; Qu et al. 2019; Radicek et al. 2018]. In particular, the example studied in Section 5.3 is taken from the work by Çiçek et al. [2017], where it is used to show the limit of the proposed logics, which cannot handle both relational cost analysis and program equivalence (however, the authors remark that such a limitation can be easily overcame by adding suitable rules to the logic). A cost analysis of sorting algorithms close to the one we did in Section 5.1 is also done in the work by Qu et al. [Qu et al. 2019] but for a higher-order language with imperative features (viz. arrays). The exact relationship between differential logical relations and relational logics is currently under investigation. In particular, it is unknown whether differential logical relations can be formalised in the setting of relational logic and, dually, if differential logical relations can be used as a tool to prove soundness of such logics.

7 CONCLUSION

In this paper, effectful differential logical relations have been proved to be a powerful and robust methodology to analyze the difference between non-equivalent but similarly behaved inpure higher-order programs, and the sensitivity of such programs to changes in their input. As such, we see it as a little but decisive step towards turning program semantics into a science of *program differences*, replacing Figure 1 in Section 2 with Figure 5.

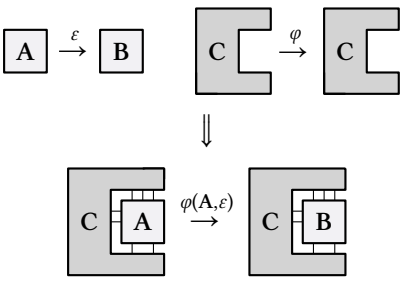

Fig. 5. A new kind of congruence rule.

There are plenty of problems which we have left open. For example, whether differential logical relations can be turned into a *formal system* (perhaps following, e.g., the work by Abadi et al. [1993]), this way supporting (semi)automated reasoning about program distances. We are confident this to be feasible. However, we think that one of the main roles the framework will have in the future is in validating expressive and *contextual* program logics and relational formal systems, suggesting extensions and adaptations of existing ones, e.g., to effects. Another topic the authors plan to investigate is the applicability of DLRs to reason about program relations in a resource-sensitive setting [Dal Lago and Gavazzo 2021b, 2022]. Finally, as *coinductive* techniques play a key role when reasoning about effectful program equivalence [Dal Lago and Gavazzo 2019a; Dal Lago et al. 2017a,b, 2020], the authors plan to study *coinductive* techniques for effectful higher-order program distances.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is supported by the ERC CoG “DIAPASoN” under Grant No. 818616.

REFERENCES

Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. 1993. Formal Parametric Polymorphism. *Theor. Comput. Sci.* 121, 1&2 (1993), 9–58. https://doi.org/10.1016/0304-3975(93)90082-5

Harold Abelson and Gerald J. Sussman. 1996. *Structure and Interpretation of Computer Programs, Second Edition.* MIT Press.

Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 2019. A relational logic for higher-order programs. *J. Funct. Program.* 29 (2019), e16. https://doi.org/10.1017/S0956796819000145

Mario Alvarez-Picallo and C.-H. Luke Ong. 2019. Change Actions: Models of Generalised Differentiation. In *Proc. of FOSSACS 2019*. 45–61. https://doi.org/10.1007/978-3-030-17127-8_3

Roland Carl Backhouse and Paul F. Hoogendijk. 1993. Elements of a Relational Theory of Datatypes. In *Formal Program Development - IFIP TC2/WG 2.1 State-of-the-Art Report*. 7–42.

Paolo Baldan, Filippo Bonchi, Henning Kerstan, and Barbara König. 2018. Coalgebraic Behavioral Metrics. *Log. Methods Comput. Sci.* 14, 3 (2018). https://doi.org/10.23638/LMCS-14(3:20)2018

M. Barr. 1970. Relational algebras. *Lect. Notes Math.* 137 (1970), 39–55.

Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2018. Proving expected sensitivity of probabilistic programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 57:1–57:29. https://doi.org/10.1145/3158145

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. 2014. Proving Differential Privacy in Hoare Logic. In *Proc. of CSF 2014*. 411–424. https://doi.org/10.1109/CSF.2014.36

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 *Proc. of POPL 2015*. 55–68. https://doi.org/10.1145/2676726.2677000

Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving Differential Privacy via Probabilistic Couplings. In *Proc. of LICS 2016*. 749–758. https://doi.org/10.1145/2933575.2934554

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2013. Probabilistic Relational Reasoning for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 35, 3 (2013), 9:1–9:49. https://doi.org/10.1145/2492061

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proc. of POPL 2004*. 14–25. https://doi.org/10.1145/964001.964003

Richard S. Bird and Oege de Moor. 1997. *Algebra of programming.* Prentice Hall.

Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. 2015. Probability type inference for flexible approximate programming. In *Proc. of OOPSLA 2015*. 470–487. https://doi.org/10.1145/2814270.2814301

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. 2014. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In *Proc. of PLDI '14*. 145–155. https://doi.org/10.1145/2594291.2594304

Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. 2012. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Proc. of PLDI 2012*. 169–180. https://doi.org/10.1145/2254064.2254086

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proc. of OOPSLA 2013*. 33–52. https://doi.org/10.1145/2509136.2509546

Konstantinos Chatzikokolakis, Daniel Gebler, Catuscia Palamidessi, and Lili Xu. 2014. Generalized Bisimulation Metrics. In *Proc. of CONCUR 2014*. 32–46. https://doi.org/10.1007/978-3-662-44584-6_4

Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2010. Continuity analysis of programs. In *Proc. of POPL 2010*. 57–70. https://doi.org/10.1145/1706299.1706308

Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara NavidPour. 2011. Proving programs robust. In *Proc. of SIGSOFT/FSE'11*. 102–112. https://doi.org/10.1145/2025113.2025131

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational cost analysis. In *Proc. of POPL 2017*. 316–329. https://doi.org/10.1145/3009837.3009858

Maria Manuel Clementino, Dirk Hofmann, and Walter Tholen. 2004. One Setting for All: Metric, Topology, Uniformity, Approach Structure. *Appl. Categorical Struct.* 12, 2 (2004), 127–154. https://doi.org/10.1023/B:APCS.0000018144.87456.10

Raphaëlle Crubillé and Ugo Dal Lago. 2015. Metric Reasoning about λ-Terms: The Affine Case. In *Proc. of LICS 2015*. 633–644. https://doi.org/10.1109/LICS.2015.64

Raphaëlle Crubillé and Ugo Dal Lago. 2017. Metric Reasoning About λ-Terms: The General Case. In *Proc. of ESOP 2017*. 341–367. https://doi.org/10.1007/978-3-662-54434-1_13

Ugo Dal Lago and Francesco Gavazzo. 2019a. Effectful Normal Form Bisimulation. In *Proc. of ESOP 2019*. 263–292. https://doi.org/10.1007/978-3-030-17184-1_10

Ugo Dal Lago and Francesco Gavazzo. 2019b. On Bisimilarity in Lambda Calculi with Continuous Probabilistic Choice. In *Proc. of MFPS 2019*. 121–141. https://doi.org/10.1016/j.entcs.2019.09.007

Ugo Dal Lago and Francesco Gavazzo. 2020. Differential Logical Relations Part II: Increments and Derivatives. In *Proc. of ICTCS 2020*. 101–114.

Ugo Dal Lago and Francesco Gavazzo. 2021a. Differential logical relations, part II increments and derivatives. *Theoretical Computer Science* (2021). https://doi.org/10.1016/j.tcs.2021.09.027

Ugo Dal Lago and Francesco Gavazzo. 2021b. Resource Transition Systems and Full Abstraction for Linear Higher-Order Effectful Programs. In *Proc. of FSCD 2021 (LIPIcs, Vol. 195)*. 23:1–23:19. https://doi.org/10.4230/LIPIcs.FSCD.2021.23

Ugo Dal Lago and Francesco Gavazzo. 2022. A Relational Theory of Effects and Coeffects. *Proc. ACM Program. Lang.* 6, POPL (2022), 36:1–36:32. https://doi.org/10.1145/3498692

Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy. 2017a. Effectful applicative bisimilarity: Monads, relators, and Howe's method. In *Proc. of LICS 2017*. 1–12. https://doi.org/10.1109/LICS.2017.8005117

Ugo Dal Lago, Francesco Gavazzo, and Ryo Tanaka. 2017b. Effectful Applicative Similarity for Call-by-Name Lambda Calculi. In *Proc. of ICTCS 2017*. 87–98.

Ugo Dal Lago, Francesco Gavazzo, and Ryo Tanaka. 2020. Effectful applicative similarity for call-by-name lambda calculi. *Theor. Comput. Sci.* 813 (2020), 234–247. https://doi.org/10.1016/j.tcs.2019.12.025

Ugo Dal Lago, Francesco Gavazzo, and Akira Yoshimizu. 2019. Differential Logical Relations, Part I: The Simply-Typed Case. In *Proc. of ICALP 2019*. 111:1–111:14. https://doi.org/10.4230/LIPIcs.ICALP.2019.111

Ugo Dal Lago, Davide Sangiorgi, and Michele Alberti. 2014. On coinductive equivalences for higher-order probabilistic functional programs. In *Proc. of POPL 2014*. 297–308. https://doi.org/10.1145/2535838.2535872

Loris D'Antoni, Marco Gaboardi, Emilio Jesús Gallego Arias, Andreas Haeberlen, and Benjamin C. Pierce. 2013. Sensitivity analysis using type-based constraints. In *Proc. of FPCDSL@ICFP 2013*. 43–50. https://doi.org/10.1145/2505351.2505353

B.A. Davey and H.A. Priestley. 1990. *Introduction to lattices and order*. Cambridge University Press.

A.A. de Amorim, M. Gaboardi, J. Hsu, S. Katsumata, and I. Cherigui. 2017. A semantic account of metric preservation. In *Proc. of POPL 2017*. 545–556. https://doi.org/10.1145/3009837.3009890

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2019. Probabilistic Relational Reasoning via Metrics. In *Proc. of LICS 2019*. 1–19. https://doi.org/10.1109/LICS.2019.8785715

Y. Deng. 2015. *Semantics of Probabilistic Processes: An Operational Approach*. Springer Berlin Heidelberg.

Josee Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. 2004. Metrics for labelled Markov processes. *Theor. Comput. Sci.* 318, 3 (2004), 323–354. https://doi.org/10.1016/j.tcs.2003.09.013

W. Du, Y. Deng, and D. Gebler. 2016. Behavioural Pseudometrics for Nondeterministic Probabilistic Systems. In *Proc. of SETTA 2016*. 67–84. https://doi.org/10.1007/978-3-319-47677-3_5

Vladimir Estivill-Castro and Derick Wood. 1992. A Survey of Adaptive Sorting Algorithms. *ACM Comput. Surv.* 24, 4 (1992), 441–476. https://doi.org/10.1145/146370.146381

Bob Flagg and Ralph Kopperman. 1997. Continuity Spaces: Reconciling Domains and Metric Spaces. *Theor. Comput. Sci.* 177, 1 (1997), 111–138. https://doi.org/10.1016/S0304-3975(97)00236-3

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear dependent types for differential privacy. In *Proc. of POPL*. 357–370. https://doi.org/10.1145/2429069.2429113

Francesco Gavazzo. 2018. Quantitative Behavioural Reasoning for Higher-order Effectful Programs: Applicative Distances. In *Proc. of LICS 2018*. 452–461. https://doi.org/10.1145/3209108.3209149

Francesco Gavazzo. 2019. *Coinductive Equivalences and Metrics for Higher-order Languages with Algebraic Effects*. Ph. D. Dissertation. University of Bologna, Italy. http://amsdottorato.unibo.it/9075/

Daniel Gebler, Kim G. Larsen, and Simone Tini. 2016. Compositional bisimulation metric reasoning with Probabilistic Process Calculi. *Log. Methods Comput. Sci.* 12, 4 (2016). https://doi.org/10.2168/LMCS-12(4:12)2016

P.G. Giarrusso. 2018. *Optimizing and incrementalizing higher-order collection queries by AST transformation*. Ph. D. Dissertation. University of Tübingen.

Robert Harper. 2016. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press.

D. Hoffman. 2015. A cottage industry of lax extensions. *Categories and General Algebraic Structures with Applications* 3, 1 (2015), 113–151.

D. Hofmann. 2007. Topological theories and closed objects. *Adv. Math.* 215 (2007), 789–824. https://doi.org/10.1016/j.aim.2007.04.013

D. Hofmann, G.J. Seal, and W. Tholen (Eds.). 2014. *Monoidal Topology. A Categorical Approach to Order, Metric, and Topology*. Number 153 in Encyclopedia of Mathematics and its Applications. Cambridge University Press.

Shin-ya Katsumata, Tetsuya Sato, and Tarmo Uustalu. 2018. Codensity Lifting of Monads and its Dual. *Log. Methods Comput. Sci.* 14, 4 (2018). https://doi.org/10.23638/LMCS-14(4:6)2018

A. Kurz and J. Velebil. 2016. Relation lifting, a survey. *J. Log. Algebr. Meth. Program.* 85, 4 (2016), 475–499. https://doi.org/10.1016/j.jlamp.2015.08.002

Kim Guldstrand Larsen and Arne Skou. 1989. Bisimulation Through Probabilistic Testing. In *Proceedings of POPL 1989*. 344–352. https://doi.org/10.1145/75277.75307

F.W. Lawvere. 1973. Metric spaces, generalized logic, and closed categories. *Rend. Sem. Mat. Fis. Milano* 43 (1973), 135–166.

P.B. Levy, J. Power, and H. Thielecke. 2003. Modelling Environments in Call-by-Value Programming Languages. *Inf. Comput.* 185, 2 (2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

S. MacLane. 1971. *Categories for the Working Mathematician*. Springer-Verlag.

Ernest G. Manes. 2002. Taut Monads and T0-spaces. *Theor. Comput. Sci.* 275, 1-2 (2002), 79–109. https://doi.org/10.1016/S0304-3975(00)00415-1

Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. 2011. Probabilistically Accurate Program Transformations. In *Proc. of SAS*. 316–333. https://doi.org/10.1007/978-3-642-23702-7_24

Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* 48, 4, Article 62 (2016), 33 pages. https://doi.org/10.1145/2893356

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proc. of LICS 1989*. IEEE Computer Society, 14–23. https://doi.org/10.1109/LICS.1989.39155

Kobbi Nissim, Sofya Raskhodnikova, and Adam D. Smith. 2007. Smooth sensitivity and sampling in private data analysis. In *Proc. of STOC 2007*. 75–84. https://doi.org/10.1145/1250790.1250803

Paolo Pistone. 2021. On Generalized Metric Spaces for the Simply Typed Lambda-Calculus. In *Proc. of LICS 2021*. 1–14. https://doi.org/10.1109/LICS52264.2021.9470696

Gordon D. Plotkin. 1973. Lambda-Definability and Logical Relations. (1973). Memorandum SAI-RM-4, University of Edinburgh.

Gordon D. Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Proc. of FOSSACS 2001*. 1–24. https://doi.org/10.1007/3-540-45315-6_1

Weihao Qu, Marco Gaboardi, and Deepak Garg. 2019. Relational cost analysis for functional-imperative programs. *Proc. ACM Program. Lang.* 3, ICFP (2019), 92:1–92:29. https://doi.org/10.1145/3341696

Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2018. Monadic refinements for relational cost analysis. *Proc. ACM Program. Lang.* 2, POPL (2018), 36:1–36:32. https://doi.org/10.1145/3158124

Ganesan Ramalingam and Thomas W. Reps. 1993. A Categorized Bibliography on Incremental Computation. In *Proc. of POPL 1993*. 502–510. https://doi.org/10.1145/158511.158710

J. Reed and B.C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proc. of ICFP 2010*. 157–168. https://doi.org/10.1145/1863543.1863568

C.H. Richardson. 1954. *An Introduction to the Calculus of Finite Differences*. New York.

Martin C. Rinard. 2011. Probabilistic accuracy bounds for perforated programs: a new foundation for program analysis and transformation. In *Proceedings of PEPM 2011*. 79–80. https://doi.org/10.1145/1929501.1929517

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: approximate data types for safe and general low-power computation. In *Proc. of PLDI 2011*. 164–174. https://doi.org/10.1145/1993498.1993518

Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and verifying probabilistic assertions. In *Proc. of PLDI 2014*. 112–122. https://doi.org/10.1145/2594291.2594294

D. Sands. 1998. Improvement Theory and Its Applications. In *Higher Order Operational Techniques in Semantics*, A. D. Gordon and A. M. Pitts (Eds.). Cambridge University Press, 275–306.

Tetsuya Sato, Gilles Barthe, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2019. Approximate Span Liftings: Compositional Semantics for Relaxations of Differential Privacy. In *Proc. of LICS 2019*. 1–14. https://doi.org/10.1109/LICS.2019.8785668

R. Sedgewick and P. Flajolet. 2013. *An Introduction to the Analysis of Algorithms*. Pearson Education.

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proc. of SIGSOFT/FSE'11*. 124–134. https://doi.org/10.1145/2025113.2025133

V. Strassen. 1965. The existence of probability measures with given marginals. *Ann. Math. Statist.* 36, 2 (1965), 423–439.

A.M. Thijs. 1996. *Simulation and fixpoint semantics*. Rijksuniversiteit Groningen.

F. Van Breugel and J. Worrell. 2005. A behavioural pseudometric for probabilistic transition systems. *Theor. Comput. Sci.* 331, 1 (2005), 115–142. https://doi.org/10.1016/j.tcs.2004.09.035

C. Villani. 2008. *Optimal Transport: Old and New*. Springer Berlin Heidelberg.

Edwin M. Westbrook and Swarat Chaudhuri. 2013. A Semantics for Approximate Program Transformations. *CoRR* abs/1304.5531 (2013). http://arxiv.org/abs/1304.5531

Hang Zhang, Mateja Putic, and John Lach. 2014. Low Power GPGPU Computation with Imprecise Hardware. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*. 99:1–99:6. https://doi.org/10.1145/2593069.2593156

Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C. Pierce, and Aaron Roth. 2019. Fuzzi: a three-level logic for differential privacy. *Proc. ACM Program. Lang.* 3, ICFP (2019), 93:1–93:28.

Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin C. Rinard. 2012. Randomized accuracy-aware program transformations for efficient approximate computations. In *Proc. of POPL 2012*, John Field and Michael Hicks (Eds.). ACM, 441–454. https://doi.org/10.1007/978-3-642-35632-2_26