Differential logical relations, part II increments and derivatives

Ugo Dal Lago, Francesco Gavazzo

▶ **To cite this version:**

Ugo Dal Lago, Francesco Gavazzo. Differential logical relations, part II increments and derivatives. Theoretical Computer Science, 2021, 895, pp.34-47. 10.1016/j.tcs.2021.09.027 . hal-03520721

HAL Id: hal-03520721
https://inria.hal.science/hal-03520721

Submitted on 11 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Differential Logical Relations
Part II: Increments and Derivatives

Ugo Dal Lago[1,2] and Francesco Gavazzo[1,2]

[1] University of Bologna, Italy
[2] INRIA Sophia Antipolis, France

Abstract. We study the deep relations existing between differential logical relations and incremental computing, by showing how self-differences in the former precisely correspond to derivatives in the latter. We also show how differential logical relations can be seen as a powerful meta-theoretical tool in the analysis of incremental computations, enabling an easy proof of soundness of differentiation.

1 Introduction

One of the major challenges programming language theory is facing these days is the development of adequate abstractions to deal with the (highly) increasing complexity and heterogeneity of modern software systems. Indeed, since the very birth of the discipline, researchers have been focused on the design of *compositional* techniques for software analysis, whereby one can study the overall behavior of a system by inspecting its constituent parts in isolation. A prime example of the successfulness of such an analysis is given by the theory of *program equivalence*. Notwithstanding its successful history, program equivalence is revealing some weaknesses when applied to nowadays software, where exact reasoning about components is, either because of the very nature of the software involved or because of the cost of such an analysis, oftentimes not feasible. Examples witnessing such weaknesses are given by the fields of *probabilistic computing*, where small perturbations in probabilities break equivalence, *numerical computing*, where program implementing numerical functions can be optimized at the price of introducing an acceptable error in the output, and, more generally, *approximate computing* [24] where accuracy of the result is partially sacrificed to gain efficiency.

The common pattern behind all the aforementioned examples is the shift from an exact analysis of software, whereby equivalent pieces of software are interchangeable within any system, to an *approximate* analysis of software, whereby *non*-equivalent pieces of software are replaced within a system at the price of producing an acceptable error, and thus an approximately correct result. Moving from an exact to an approximate analysis of programs poses several challenges to programming language semantics, the main one arguably concerning compositionality. In fact, once we replace a program P with a non-equivalent one Q in a system $C[-]$, then $C[-]$ may amplify the error introduced by the replacement of P with Q, this way invalidating compositionality of the analysis. This

point becomes evident when studying (higher-order) *program metric* or *program distance* [27, 9, 14, 10]: if the distance between P and Q is upper bounded by a number $\varepsilon > 0$, then it may not be so for $C[P]$ and $C[Q]$.

Motivated by these general observations, researchers are showing an increasing interest in *quantitative* analysis of programs, with a special focus on *differential* properties of programs. Although the expression *differential* has no precise meaning in this context, we may identify it with the collection of properties relating local and global changes in software. Thus, for instance, we can think of the (error produced by the) replacement of P with Q as a local change, and investigate its relationship with the global change we observe between $C[P]$ and $C[Q]$.

The study of such differential properties has led, oftentimes abusing terminology, to the development of several notions of a *program derivative*. Among those, some of the main ones one encounters when looking at the relevant literature are the following.

- Those coming from the field of *automatic differentiation* [4], which aim to extend the notion of a derivative one finds in mathematical analysis [30] to arbitrary programs. Examples are given by [6, 1, 29] (as well as by references therein).
- Those coming from the *differential λ-calculus* [13], whose original motivations rely on quantitative semantics [19] and linear logic [18].
- Those coming from *incremental computing* [26, 25], which aim to find ways to incrementally compute outputs as inputs changes.
- Those coming from *differential logical relations* [21] via the notion of a *self-difference*, which aim to provide context-sensitive compositional distances between programs.

It is thus natural to ask whether there are connections between such notions. Although for some of the aforementioned cases the answer seems to be negative (for instance, the derivatives one finds in incremental computing are generalizations of *finite differences* [28], whereas the ones found in calculi for automatic differentiation are actual derivatives), others have conceptual similarities. This is the case for differential logical relations and incremental computing, as both study the relationship between input and output changes.

In this paper we study such similarities and establish a formal connection between differential logical relations and incremental computing, by showing how self-differences in the former precisely correspond to derivatives in the latter. In fact, as we will see, the derivative of a program P can be seen as a way to quantify how much changes in the input of P influence changes in its output, this way acting as a self-difference for P.

Besides its conceptual implications, the advantages of such a correspondence are twofold. On the one hand, differential logical relations qualify as a lightweight operational technique for incremental computing: we witness that by giving a proof of soundness of differentiation for the incremental λ-calculus of Cai et al. [7]. On the other hand, we can use results coming from incremental computing to go beyond the current theory of differential logical relations. For instance, it is

possible to read the work by Giarrusso, Régis-Gianas, and Schuster [17], where step-indexed logical relations are introduced for proving correctness of *untyped* program derivatives, to define a form of step-indexed differential logical relations, this way giving differential semantics to calculi with full recursion (something not possible in the original formulation of differential logical relations [21]).

Structure of the Paper Section 2 introduces the target calculus of this work, as well as differential logical relations and the incremental λ-calculus. In Section 3, we establish a formal connection between differential logical relations and the incremental λ-calculus by showing that program derivatives (in the sense of incremental computing) are self-distances (in the sense of differential logical relations). Additionally, we give a differential logical relation-based proof of soundness of differentiation, the main result in the theory of incremental computing.

2 Preliminaries: Differential Logical Relations and the Incremental λ-calculus

In this section we shortly review the main ideas behind *differential logical relations* (DLRs) and the incremental λ-calculus. In order to do so, we introduce the vehicle calculus of (the first part) of this work, namely a call-by-value simply typed λ-calculus with a primitive type for real numbers, which we denote by $\mathsf{ST}_{\mathtt{R}}$. The calculus is standard and it is essentially the same calculus used in [21]. We summarize the syntax and static semantics of $\mathsf{ST}_{\mathtt{R}}$ in Figure 1, where we assume to have constants $\underline{\varphi}$ for any function[3] $\varphi : \mathbb{R}^n \to \mathbb{R}$ and primitives $\underline{r}$ for any real number r.

$$\sigma, \tau ::= \mathtt{R} \mid \sigma \times \tau \mid \sigma \to \tau \qquad \begin{array}{l} t, s ::= x \mid \underline{r} \mid \underline{\varphi} \mid \langle t, s\rangle \mid \lambda x.t \mid \mathbf{out}_1\ t \mid \mathbf{out}_2\ t \mid ts \\ v, w ::= x \mid \underline{r} \mid \underline{\varphi} \mid \langle v, w\rangle \mid \lambda x.t \end{array}$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{}{\Gamma \vdash \underline{r} : \mathtt{R}} \qquad \frac{}{\Gamma \vdash \underline{\varphi} : \underbrace{\mathtt{R} \to \cdots \to \mathtt{R}}_{n} \to \mathtt{R}}$$

$$\frac{\Gamma \vdash t_1 : \sigma_1 \quad \Gamma \vdash t_2 : \sigma_2}{\Gamma \vdash \langle t_1, t_2\rangle : \sigma_1 \times \sigma_2} \qquad \frac{\Gamma \vdash t : \sigma_1 \times \sigma_2}{\Gamma \vdash \mathbf{out}_1\ t : \sigma_1} \qquad \frac{\Gamma \vdash t : \sigma_1 \times \sigma_2}{\Gamma \vdash \mathbf{out}_2\ t : \sigma_2}$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma \vdash t : \sigma \to \tau \quad \Gamma \vdash s : \sigma}{\Gamma \vdash ts : \tau}$$

Fig. 1. Syntax and Statics of $\mathsf{ST}_{\mathtt{R}}$

We use letters $t, s, \ldots$ to range over terms, and $v, w, \ldots$ to range over values. Additionally, we follow standard syntactic conventions as in, e.g., [3]. In partic-

[3] When dealing with standard arithmetic operator, such as $+$, we will overload the notation and write $+$ in place of $\underline{+}$.

ular, we work with terms modulo renaming of bound variables, and denote by $t[v/x]$ the capture-avoiding substitution of v for x in t. Finally, we introduce the following notation and refer to terms in $\Lambda^\bullet$ as *closed* terms or *programs*. Similarly, we refer to values in $\mathcal{V}^\bullet$ as *closed* values, and we use notations such as Λ_σ and $\mathcal{V}^\bullet_\sigma$ with their natural meanings.

The dynamics of $\mathsf{ST}_\mathtt{R}$ is given by a standard call-by-value operational semantics, defined in Figure 2, where for a function $\varphi : \mathbb{R}^n \to \mathbb{R}$ and a number $r \in \mathbb{R}$ we write $\varphi_r : \mathbb{R}^{n-1} \to \mathbb{R}$ for the mapping $(r_1, \ldots, r_{n-1}) \mapsto \varphi(r, r_1, \ldots, r_{n-1})$. Notice, in particular, that $\underline{\varphi}\,\underline{r}_1 \cdots \underline{r}_n$ eventually reduces to $\underline{\varphi(r_1, \ldots, r_n)}$.

Since $\mathsf{ST}_\mathtt{R}$ is simply-typed, standard reducibility [20] suffices to show that $\mathsf{ST}_\mathtt{R}$ is strongly normalizing. In particular, any program t evaluates to a (unique) closed value v—that we indicate as $\mathsf{nf}(t)$—in a finite number of $\to$-steps (notation $t \Downarrow v$). We write $t \Downarrow_n v$, for $n \in \mathbb{N}$, to state that t evaluates to v in n number to $\to$-steps and $\to^*$ for the reflexive and transitive closure of $\to$.

$$\frac{}{(\lambda x.t)v \to t[v/x]} \qquad \frac{}{\underline{\varphi}\,\underline{r} \to \underline{\varphi_r}} \qquad \frac{}{\mathbf{out}_i\ \langle v_1, v_2\rangle \to v_i} \qquad \frac{t \to t'}{ts \to t's} \qquad \frac{s \to s'}{vs \to vs'}$$

$$\frac{t \to t'}{\langle t, s\rangle \to \langle t', s\rangle} \qquad \frac{s \to s'}{\langle v, s\rangle \to \langle v, s'\rangle} \qquad \frac{t \to t'}{\mathbf{out}_i\ t \to \mathbf{out}_i\ t'}$$

Fig. 2. Dynamics of $\mathsf{ST}_\mathtt{R}$

2.1 Program Equivalence and Program Distance: an Overview

Despite its simplicity, $\mathsf{ST}_\mathtt{R}$ allows us to justify the shift from *program equivalence*—that is the family of notions concerning equality between programs—to *program distance*, in general, and to differential logical relations, in particular.

First, let us define a suitable notion of program equivalence for $\mathsf{ST}_\mathtt{R}$ programs. Due to its simple nature, $\mathsf{ST}_\mathtt{R}$ allows us to choose among a large family of notions of program equivalence, ranging from denotationally-based equivalences to operationally-based ones. Here we choose *extensional* or *applicative equivalence*[4].

Definition 1. *Define the type-indexed family of relations* $(\cong^\Lambda_\sigma \subseteq \Lambda^\bullet_\sigma \times \Lambda^\bullet_\sigma, \cong^\mathcal{V}_\sigma \subseteq \mathcal{V}^\bullet_\sigma \times \mathcal{V}^\bullet_\sigma)_\sigma$ *as follows (where* $i \in \{1, 2\}$*):*

$$\begin{aligned}
t \cong^\Lambda_\sigma t' &\iff \mathsf{nf}(t) \cong^\mathcal{V}_\sigma \mathsf{nf}(t') \\
\langle v_1, v_2\rangle \cong^\mathcal{V}_{\sigma_1 \times \sigma_2} \langle w_1, w_2\rangle &\iff \forall i.\ v_i \cong^\mathcal{V}_{\sigma_i} w_i \\
\underline{r} \cong^\mathcal{V}_\mathtt{R} \underline{r}' &\iff r = r' \\
v \cong^\mathcal{V}_{\sigma\to\tau} v' &\iff \forall w \in \mathcal{V}^\bullet_\sigma.\ vw \cong^\Lambda_\tau v'w
\end{aligned}$$

[4] This is nothing but a simplification of Abramsky's applicative bisimilarity [2] relying on strong normalization of $\mathsf{ST}_\mathtt{R}$ and its simple type system.

It is well-known that extensional equivalence is a congruence relation, this way enabling *compositional* reasoning about program behaviors: $s \cong_\sigma s'$ entails $t[s/x] \cong_\tau t[s'/x]$ for any term $x : \sigma \vdash t : \tau$. Unfortunately, one soon realizes that due to the presence of (constants for) real-numbers, program equivalence is a too coarse notion for reasoning about $\mathsf{ST}_{\mathtt{R}}$ programs. For it is desirable to regard two programs of type $\mathtt{R}$ whose outputs are ε apart to be themselves ε apart, rather than just state that the two are not equivalent.[5]

The natural way to overcome this problem is to refine $\cong_\sigma$ into a map $\delta_\sigma : \Lambda^\bullet_\sigma \times \Lambda^\bullet_\sigma \to \mathbb{R}$ following Lawvere's correspondence between ordered sets and (generalized) metric spaces [22]. Accordingly, we obtain the following maps:

$$\delta^{\mathcal{V}}_{\mathtt{R}}(\underline{r}, \underline{r}') \triangleq r' - r \qquad \delta^{\mathcal{V}}_{\sigma_1 \times \sigma_2}(\langle v_1, v_2\rangle, \langle w_1, w_2\rangle) \triangleq \max_{i \in \{1,2\}} \delta^{\mathcal{V}}_{\sigma_i}(v_i, w_i)$$

$$\delta^{\Lambda}_\sigma(t, t') \triangleq \delta^{\mathcal{V}}_\sigma(\mathsf{nf}(t), \mathsf{nf}(t')) \qquad \delta^{\mathcal{V}}_{\sigma \to \tau}(v, v') \triangleq \sup_{w \in \mathcal{V}^\bullet_\sigma} \delta^{\Lambda}_\tau(vw, v'w)$$

which can be easily proved to be generalized metrics[6] [22]. Of course, in order for δ to serve its purpose, we also need it to support the (quantitative refinement of the) compositionality principle ensured by $\cong$. As compositionality of $\cong$ took the form of a congruence property, it is easy to realize to that compositionality of δ takes the form of *non-expansiveness*: for all terms $s, s' \in \Lambda^\bullet_\sigma$ and $x : \sigma \vdash t : \tau$ we must have $\delta_\sigma(s, s') \geq \delta_\tau(t[s/x], t[s'/x])$. That is, *contexts cannot amplify distances*.

Unfortunately, we immediately see that δ fails to be non-expansive, and thus compositional. Even worse, any reasonable non-expansive metric-like map is bound to trivialize, meaning that it collapses to a congruence *relation*. Roughly, given two terms s, s' which are $\varepsilon \neq 0$ apart, for any positive real number c it is always possible to find an open term $x : \sigma \vdash t : \tau$ such that $t[s]$ and $t[s']$ are c apart. For it is sufficient to take a term t using its input x enough times: once the terms $t[s]$ and $t[s']$ are evaluated, any time t uses x the distance between s and s' is detected and added to the one previously measured. Remarkably, this holds even if any φ is non-expansive (i.e. 1-Lipschitz).

2.2 Differential Logical Relations

The failure of non-expansiveness of quantitative refinements of notions of program equivalence has led researchers to propose several notions of program distance [9, 27, 14] in recent years, all aimed to restore compositionality. All the

[5] A similar argument can be formulated for any language/calculus exhibiting, either in its syntax or in its semantics, some quantitative behavior. Typical examples of such behaviors are given by types for quantitative objects, such as numeric types, but also by probabilistic primitives, the latter making relevant semantic notions, such as termination, quantitative [8, 10–12, 15, 27, 5].

[6] Additionally, by replacing $r' - r$ and $\sup_{r_1,\ldots,r_n} \psi(r_1, \ldots, r_n) - \varphi(r_1, \ldots, r_n)$ with $|r' - r|$ and $\sup_{r_1,\ldots,r_n} |\psi(r_1, \ldots, r_n) - \varphi(r_1, \ldots, r_n)|$, respectively, δ_σ becomes a *pseudometric* [31].

notions proposed share a common feature: they all impose calculi *linearity* constraints, this way providing static information on the number of times a program can use its input (and thus on how much the program can amplifies distances). The notions of program distance thus obtained are indeed compositional, but still have two major drawbacks. First, they are tailored for linear calculi and are not very informative when applied to non-linear calculi (relying, e.g., on standard translations [18]). Second, and most important, they do not account for the role of the environment in determining distances.

Let us expand on this last point by means of an example. Consider the (linear) programs $t \triangleq \lambda x.x$ and $s \triangleq \lambda x.\underline{\sin}\ x$ for the identity and sine function, respectively. It is easy to see that measuring the distance between t and s as we did when defining δ, we are forced to conclude such a distance to be ∞. In fact, for $r \to \infty$ we have $|r - \sin(r)| \to \infty$. This is rather unsatisfactory, as such distance does not take into account which input the environment will actually pass to t and s. For instance, if the environment feeds t and s with an input v close to zero, then the distance between tv and sv should be close to 0 too, and thus we would like to conclude that in all such cases the distance between t and s is itself close to zero.

Summing up, ordinary notions of program distance are not sensitive to the context in which programs are used. This ultimately relies on the fact that measuring the distance between two programs (regarded as functions) as just *one single number* there is no way to give information on how such programs interact with the environments in which they are used. Differential logical relations have been introduced in [21] as a way to define a *context-sensitive* notion of program distance on non-linear calculi. The main novelty of differential logical relations (which was previously theorized by Westbrook and Chaudhuri [32] in the setting of approximate program transformations [23]) is to consider richer notions of distance (also called *differences*) between programs, whereby the difference between two programs is, in general, not a *number*, but a *function* describing how differences between inputs turn into differences between outputs.

Differential logical relations take the form of (type-indexed) ternary relations D_σ relating pairs of programs together with differences between them. When dealing with programs of type $\sigma \to \tau$, differences take the form of functions mapping input programs of type σ and differences for such programs to differences for programs of type τ. This is why here we consider a computationally-oriented notion of difference whereby differences between programs are defined as programs themselves (cf. [32]) rather than as semantical objects.

We formalize these ideas by assigning to each type σ a type $\Delta\sigma$ whose inhabitants are terms acting as differences between terms of type σ.

Definition 2. *For any type σ, we define the type $\Delta\sigma$ of σ-differences as follows:*

$$\Delta\mathtt{R} \triangleq \mathtt{R}; \qquad \Delta(\sigma \times \tau) \triangleq \Delta\sigma \times \Delta\tau; \qquad \Delta(\sigma \to \tau) \triangleq \sigma \to \Delta\sigma \to \Delta\tau.$$

Notice, in particular, that a difference between two programs of type $\sigma \to \tau$ is a program taking an input of type σ and a σ-difference, and returning a τ-difference.

Obviously, given two programs t, t' of type σ, not all programs of type $\Delta\sigma$ can act as (meaningful) differences between t and t'. Differential logical relations (DLRs for short) are ternary relations specifically designed to isolate meaningful differences between programs. More precisely, a DLR is a type-indexed family of ternary relations $\mathsf{D} \triangleq (\mathsf{D}^{\Lambda}_{\sigma}, \mathsf{D}^{\mathcal{V}}_{\sigma})_{\sigma}$, where $\mathsf{D}^{\Lambda}_{\sigma} \subseteq \Lambda^{\bullet}_{\Delta\sigma} \times \Lambda^{\bullet}_{\sigma} \times \Lambda^{\bullet}_{\sigma}$ and $\mathsf{D}^{\mathcal{V}}_{\sigma} \subseteq \mathcal{V}^{\bullet}_{\Delta\sigma} \times \mathcal{V}^{\bullet}_{\sigma} \times \mathcal{V}^{\bullet}_{\sigma}$, such that $\mathsf{D}^{\Lambda}_{\sigma}(dt, t, t')$ holds if and only if dt is a difference[7] between t and t' (and similarly for values).

Definition 3 (Asymmetric DLRs). *A* differential logical relation *is a type-indexed family of ternary relations* $(\mathsf{D}^{\Lambda}_{\sigma} \subseteq \Lambda^{\bullet}_{\Delta\sigma} \times \Lambda^{\bullet}_{\sigma} \times \Lambda^{\bullet}_{\sigma}, \mathsf{D}^{\mathcal{V}}_{\sigma} \subseteq \mathcal{V}^{\bullet}_{\Delta\sigma} \times \mathcal{V}^{\bullet}_{\sigma} \times \mathcal{V}^{\bullet}_{\sigma})_{\sigma}$ *such that:*

$$
\begin{aligned}
\mathsf{D}^{\mathcal{V}}_{\mathtt{R}}(\underline{dr}, \underline{r}, \underline{r}') &\iff r' - r = dr \\
\mathsf{D}^{\mathcal{V}}_{\sigma_1 \times \sigma_2}(dv, v, v') &\iff \forall i \in \{1, 2\}.\ \mathsf{D}^{\mathcal{V}}_{\sigma_i}(\mathbf{out}_i\ dv, \mathbf{out}_i\ v, \mathbf{out}_i\ v') \\
\mathsf{D}^{\mathcal{V}}_{\sigma \to \tau}(dv, v, v') &\iff \forall dw, w, w'.\ \mathsf{D}^{\mathcal{V}}_{\sigma}(dw, w, w') \implies \mathsf{D}^{\Lambda}_{\tau}(dv\, w\, dw, vw, v'w') \\
\mathsf{D}^{\Lambda}_{\sigma}(dt, t, t') &\iff \mathsf{D}^{\mathcal{V}}_{\sigma}(dv, v, v') \text{ where } dt \Downarrow dv, t \Downarrow v, t' \Downarrow v'.
\end{aligned}
$$

Remark 1. Contrary to the original formulation of DLRs [21], here we work with *asymmetric* DLRs: if dt is a difference between t and t', then dt may not be a difference between t' and t. For instance, $\underline{3}$ is a difference between $\underline{2}$ and $\underline{5}$, as by adding 3 to 2 we reach 5. Yet, according to such a reading, it is not true that $\underline{3}$ is a difference between $\underline{5}$ and $\underline{2}$ (the desired difference being, in fact, $\underline{-3}$).

Example 1 ([21]). Let $t \triangleq \lambda x.\underline{\sin}\ x$ and $t' \triangleq \lambda x.x$. Then $dt \triangleq \lambda x.\lambda dx.x + dx - \underline{\sin}\ x$ is a difference between t and t'. For, proving $\mathsf{D}^{\mathcal{V}}_{\mathtt{R} \to \mathtt{R}}(dt, t, t')$ requires to prove that for all $\underline{dr}, \underline{r}, \underline{r}'$ such that $\mathsf{D}^{\mathcal{V}}_{\mathtt{R}}(\underline{dr}, \underline{r}, \underline{r}')$ (meaning that $r + dr = r'$), we have $\mathsf{D}^{\mathcal{V}}_{\mathtt{R}}(r + dr - \sin r, \underline{\sin}\, r, \underline{r}')$, i.e. $r + dr - \sin r + \sin r = r'$, which is indeed the case. Observe how $dt\ \underline{x}\ \underline{\varepsilon}$ evaluates to a real number which is indeed small when the two arguments are themselves close to 0.

As already remarked, DLRs have been introduced with the goal of developing a compositional theory of program distance. This goal is achieved by the so-called *Fundamental Lemma* [21].

Lemma 1 (Fundamental Lemma, Version 1). *For any program $t \in \Lambda^{\bullet}_{\sigma}$ there exists a self-difference dt for it. That is, $\mathsf{D}_{\sigma}(dt, t, t)$.*

Lemma 1 enables compositional reasoning on program differences. Informally, by regarding a context $x : \sigma \vdash t : \tau$ as a term $\lambda x.t$ we are guaranteed a self-difference dt for t to exist, so that given two programs s, s' of type σ with difference ds between them, one can compute the difference between $t[s/x]$ and $t[s'/x]$ starting from s, ds, and dt alone.

[7] Following conventions in, e.g., [7], we use the notation $dt, ds, \ldots$ for term differences—i.e. terms of type $\Delta\sigma$.

2.3 The Incremental λ-calculus and Program Derivatives

Albeit enabling compositional reasoning on program differences, Lemma 1 has the major drawback of guaranteeing the existence of self-distances *without* giving any explicit information on how to construct them. As we will see in the next section, the self-distances of Lemma 1 are precisely the *program derivatives* used in the *incremental λ-calculus*.

The incremental λ-calculus is a formalism introduced by Cai et al. [7] as a foundational calculus for incremental computation [26, 25]. Roughly, suppose we are given a program f regarded as a function, and an input a (think, for instance, of a as a database). Let us now suppose to compute $f(a)$ and then to modify the input a by a change da, this way obtaining a new input $a \oplus da$ (for instance, we may add a new entry to the database a). Incremental computing seeks for ways to obtain the result of $f(a \oplus da)$ without computing f on the new input $a \oplus da$ from scratch. In fact, sometimes it is indeed possible to obtain such a result in terms of $f(a)$ and $f'(a, da)$, for a suitable function[8] f'. For instance, let $f(x) \triangleq x^2$ and suppose we have computed $f(a)$, for some a. Let us now change a to $a+da$. When asked to compute $f(a + da)$ we can take advantage of having already computed $f(a) = a^2$ by observing that $f(a + da) = a^2 + 2ada + da^2 = f(a) + f'(a, da)$, where $f'(x, dx) \triangleq 2xdx + dx^2$.

In order to provide a formal foundation for higher-order incremental computation, Cai et al. [7] studied incrementalization of a simply-typed λ-calculus similar to the one introduced in the previous section. More precisely, for any type σ a type of σ-changes coinciding with $\Delta\sigma$ is introduced, as well as an operator $\oplus$ (called *change update*) building an expression $t \oplus dt \in \Lambda_\sigma$ from an expression $t \in \Lambda_\sigma$ and a change $dt \in \Lambda_{\Delta\sigma}$. In order to account for incrementalization, they also introduced the so-called *derivative*[9] $Dt \in \Lambda_{\Delta\sigma}$ of an expression $t \in \Lambda_\sigma$, and showed that for all terms $t \in \Lambda^\bullet_{\sigma\to\tau}$, $s \in \Lambda^\bullet_\sigma$, and σ-change ds, one has: $t\,(s \oplus ds) \equiv (ts) \oplus (Dt\,s\,ds)$, where $\equiv$ stands for denotational equality. All the aforementioned results are proved by means of denotational semantics, although some operationally-based proofs employing techniques resembling DLRs are given in Giarrusso's PhD thesis [16].

In the next section, we will show how we can easily prove such results using DLRs, and, dually, how by identifying differences with changes we can improve on the current theory of program differences. In order to do so, however, we first need to formally introduce the update operator $\oplus$ and the notion of a program derivative. We begin recalling a couple of basic definitions from finite difference calculus.

---

[8] Obviously, to be practically useful, the map f' should be such that computing $f'(a, da)$ is cheaper than computing $f(a \oplus da)$.

[9] The terminology is misleading. For instance, we should not think of the derivative $D\underline{\varphi}$ of a term $\underline{\varphi} : \mathtt{R} \to \mathtt{R}$ as the syntactic counterpart of the derivative of $\varphi : \mathbb{R} \to \mathbb{R}$. Rather, $D\underline{\varphi}$ represents the *finite difference* [28] of φ, i.e. the map $\Delta\varphi : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ defined by $\Delta\varphi(x, dx) \triangleq \varphi(x + dx) - \varphi(x)$.

Definition 4. *Given functions* $\varphi : \mathbb{R}^n \to \mathbb{R}$ *and* $d\varphi : (\mathbb{R} \times \mathbb{R})^n \to \mathbb{R}$*, define the maps* $\varphi \oplus d\varphi : \mathbb{R}^n \to \mathbb{R}$ *and* $\Delta\varphi : (\mathbb{R} \times \mathbb{R})^n \to \mathbb{R}$ *by:*

$$(\varphi \oplus d\varphi)(x_1, \ldots, x_n) \triangleq \varphi(x_1, \ldots, x_n) + d\varphi((x_1, 0), \ldots, (x_n, 0));$$
$$\Delta\varphi((x_1, dx_1), \ldots, (x_n, dx_n)) \triangleq \varphi(x_1 + dx_1, \ldots, x_n + dx_n) - \varphi(x_1, \ldots, x_n).$$

The map $\Delta\varphi$ is known as the *finite difference* of φ. Notice that $\varphi \oplus \Delta\varphi = \varphi$.

Definition 5. *Define the* partial *operator* $\oplus : \Lambda \to \Lambda \to \Lambda$ *as follows:*

$$x \oplus dx \triangleq x; \qquad (\mathbf{out}_i\ t) \oplus (\mathbf{out}_i\ dt) \triangleq \mathbf{out}_i\ (t \oplus dt);$$
$$\underline{r} \oplus d\underline{r} \triangleq \underline{r + dr}; \qquad (\lambda x.t) \oplus (\lambda x.\lambda dx.dt) \triangleq \lambda x.(t \oplus dt);$$
$$\underline{\varphi} \oplus \underline{d\varphi} \triangleq \underline{\varphi \oplus d\varphi}; \qquad (ts) \oplus (dt\, s\, ds) \triangleq (t \oplus dt)\,(s \oplus ds).$$
$$\langle t, s\rangle \oplus \langle dt, ds\rangle \triangleq \langle t \oplus dt, s \oplus ds\rangle;$$

The definition of $t \oplus dt$ may appear weird at first, but it will become clear once the notion of a derivative is introduced. Intuitively, given a term t and a change dt, we can see $t \oplus dt$ as the term obtained by changing t according to dt. Clearly, this is possible only if dt has the 'right' structure (for instance, it would be meaningless to do something like changing a function according to a number). We immediately notice that if v is a value and $v \oplus dv$ is defined, then dv and $v \oplus dv$ are values too. Moreover, the following typing rule is admissible, whenever $t \oplus dt$ is defined:

$$\frac{\Gamma \vdash t : \sigma \quad d\Gamma, \Gamma \vdash dt : \Delta\sigma}{\Gamma \vdash t \oplus dt : \sigma}$$

where for $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$, we have $d\Gamma \triangleq dx_1 : \Delta\sigma_1, \ldots, dx_n : \Delta\sigma_n$.

Next, we define the notion of a derivative of a term.

Definition 6. *The derivative* Dt *of a term* t *is thus defined:*

$$Dx \triangleq dx; \qquad D\underline{r} \triangleq \underline{0}; \qquad D(ts) \triangleq Dt\, s\, Ds.$$
$$D\underline{\varphi} \triangleq \underline{\Delta\varphi}; \qquad D\langle t, s\rangle \triangleq \langle Dt, Ds\rangle;$$
$$D(\mathbf{out}_i\ t) = \mathbf{out}_i\ Dt; \qquad D(\lambda x.t) \triangleq \lambda x.\lambda dx.Dt;$$

Observe that we can indeed think of Dt as the generalization of finite differences to arbitrary programs. Moreover, we easily see that if $\Gamma \vdash t : \sigma$, then $\Gamma, d\Gamma \vdash Dt : \Delta\sigma$ and that $t \oplus Dt$ is defined and equal to t itself.

3 Bridging the Gap

In this section we relate DLRs with the incremental λ-calculus we introduced in the previous section. We do so acting on two orthogonal axes. On the one hand, we show that derivatives are precisely the self-distances of Lemma 1. That is, for

any program t, Dt is a self-distance for t. This result allows us to strength the fundamental lemma of DLRs (Lemma 1), as now self-differences can be effectively computed. On the other hand, we prove by means of DLRs a major result on the incremental λ-calculus, namely *soundness of differentiation* [7]. To the best of the authors' knowledge, all proofs of such a result rely on either denotational semantics or logical relations tailored for such purpose (see Remark 2).

Let us begin proving that derivatives are actually self-differences. In order to achieve such a result, we have to first extend the notion of a DLR to open terms [21]. Given an environment Γ, we denote by $\mathsf{S}(\Gamma)$ the collection of Γ-substitutions, i.e. the collection of maps ρ mapping variables $(x : \sigma) \in \Gamma$ to closed values $\rho(x) \in \mathcal{V}^\bullet_\sigma$. In particular, we use the notation $d\rho$ to denote substitutions in $\mathsf{S}(d\Gamma)$.

Definition 7. *We extend the notion of a DLR to substitutions over an environment Γ as follows:* $\mathsf{D}_\Gamma(d\rho, \rho, \rho') \iff \forall (x : \sigma) \in \Gamma.\ \mathsf{D}^\nu_\sigma(d\rho(dx), \rho(x), \rho'(x))$, *where $\rho, \rho' \in \mathsf{S}(\Gamma)$ and $d\rho \in \mathsf{S}(d\Gamma)$.*

As it is customary, we write $t[\rho]$ for the application of the substitution ρ to the term t, and $\rho[x \mapsto v]$ for the substitution mapping x to v and behaving as ρ otherwise. Before proving our refinement of Lemma 1, let us observe that DLRs are closed under reduction, in the following sense.

Lemma 2. *The following holds for all closed terms:*

$$\mathsf{D}^\Lambda_\sigma(dt, t, t') \wedge t \to^* s \wedge t' \to^* s' \implies \mathsf{D}^\Lambda_\sigma(dt, s, s');$$
$$\mathsf{D}^\Lambda_\sigma(dt, s, s') \wedge s \to^* t \wedge s' \to^* t' \implies \mathsf{D}^\Lambda_\sigma(dt, t, t').$$

We are now ready to prove our new version of the Fundamental Lemma.

Lemma 3 (Fundamental Lemma, Version 2). *For any program $t \in \Lambda^\bullet_\sigma$ we have $\mathsf{D}_\sigma(Dt, t, t)$.*

Proof (sketch). The thesis follows from the stronger statement: for any term $\Gamma \vdash t : \sigma$ and value $\Gamma \vdash v : \sigma$ we have:

$$\forall d\rho, \rho, \rho'.\ \mathsf{D}_\Gamma(d\rho, \rho, \rho') \implies \mathsf{D}^\nu_\sigma(Dv[\rho, d\rho], v[\rho], v[\rho']) \wedge \mathsf{D}^\Lambda_\sigma(Dt[\rho, d\rho], t[\rho], t[\rho']).$$

The proof of the latter is a routine induction on t and v. □

Notice how Lemma 3 improves the compositionality principle of DLRs. Given a term $x : \sigma \vdash t : \tau$ and two values $\vdash v, v' : \sigma$ such that $\mathsf{D}^\nu_\sigma(dv, v, v')$ the impact of replacing v with v' in t can be computed as $Dt[v/x, dv/dx]$. Next, we show how the incremental λ-calculus can benefit from DLRs by showing how the latter support an easy proof of *soundness of differentiation*.

Theorem 1 (Soundness of Differentiation [7, 17]). *For all $t \in \Lambda^\bullet_{\sigma\to\tau}$ and values v, v', dv such that $\mathsf{D}^\nu_\sigma(dv, v, v')$, we have: $tv' \cong (tv) \oplus (Dt\, v\, dv)$.*

Our proof of Theorem 1 is based on the following result which states that changes indeed behave as such. Recall that $\cong$ extends to open terms by stipulating that for $\Gamma \vdash t, t' : \sigma$ we have $t \cong^{\Lambda}_{\sigma} t'$ iff $t[\rho] \cong^{\Lambda}_{\sigma} t'[\rho]$, for any substitution $\rho \in \mathsf{S}(\Gamma)$ (and similarly for values).

Proposition 1. *The following holds for all (possibly open) terms t, t', dt and values v, v', dv such that $t \oplus dt$ and $v \oplus dv$ is defined.*

$$\mathsf{D}^{\Lambda}_{\sigma}(dt, t, t') \implies t' \cong t \oplus dt; \qquad \mathsf{D}^{\mathcal{V}}_{\sigma}(dv, v, v') \implies v' \cong v \oplus dv.$$

Proof (sketch). The proof is by induction on σ, the relevant case being the one of values. We show how to handle the case for arrow types. Assume $\Gamma \vdash t, t' : \sigma \to \tau$. We have to show that for any $\rho \in \mathsf{S}(\Gamma)$, $t'[\rho] \cong (t \oplus dt)[\rho]$. First, observe that we have the following general result (by induction on t), where $(D\rho)(dx) \triangleq D\rho(x)$: $(t \oplus dt)[\rho] = t[\rho] \oplus dt[\rho, D\rho]$. By Lemma 3, we have $\mathsf{D}_{\Gamma}(D\rho, \rho, \rho)$, hence $\mathsf{D}^{\mathcal{V}}_{\sigma\to\tau}(dt[\rho, D\rho], t[\rho], t'[\rho])$. In particular, we must have $t[\rho] = \lambda x.s$, $t'[\rho] = \lambda x.s'$, and $dt[\rho, D\rho] = \lambda x.\lambda dx.ds$, for some s, s', and ds. Since $(\lambda x.s) \oplus (\lambda x.\lambda dx.ds) = \lambda x.(s \oplus ds)$ (notice that this term is indeed defined, as it is obtained from $t \oplus dt$, which is defined by hypothesis, replacing variables y, dy with closed values v, Dv), in order to prove the thesis we have to show $s'[v/x] \cong (s \oplus ds)[v/x]$ for any closed value v of type σ. Since $(s \oplus ds)[v/x] = s[v/x] \oplus ds[v/x, Dv/dx]$ we obtain the thesis from $\mathsf{D}^{\mathcal{V}}_{\sigma\to\tau}(dt[\rho, D\rho], t[\rho], t'[\rho])$ and $\mathsf{D}^{\mathcal{V}}_{\sigma}(Dv, v, v)$, the latter being a consequence of Lemma 3. □

We can finally prove soundness of differentiation.

Proof (Theorem 1). Assume $\mathsf{D}^{\mathcal{V}}_{\sigma}(dv, v, v')$. By Lemma 3 we have $\mathsf{D}_{\sigma\to\tau}(Dt, t, t)$, and thus $\mathsf{D}_{\tau}(Dt\, s\, ds, ts, ts')$, by Lemma 2. We conclude that $tv' \cong (tv) \oplus (Dt\, v\, dv)$ from Proposition 1. □

Remark 2. To the best of the authors' knowledge, all proofs of Theorem 1 in the literature are based on either denotational semantics or on logical relations resembling DLRs, but specifically extended with a clause requiring $t \oplus dt = t'$ for all related terms dt, t, t', at any type [17, 16]. Notice the use of syntactic equality: the reason behind such a choice is that the logical relation obtained is meant to relate only programs with their derivative (in which case we indeed have $t \oplus Dt = t$), rather than as a tool to reason about program differences.

4 Related Work

Differential logical relations have been introduced by the authors and Yoshimizu [21], building over intuitions by Westbrook and Chaudhuri [32] and are currently under investigation. Differently from the ones considered in this work, the first formulation of differential logical relations [21] is symmetric and considers semantical difference spaces, so that differences between programs are semantical objects (such as numbers and functions), rather than programs themselves. Whereas we have found that working with asymmetric DLRs makes proofs

clearer (besides, asymmetry is in line with Lawvere's analysis of the notion of a distance [22]), working with syntactic difference spaces does not really affect our results. In fact, we could consider semantic-based difference spaces and show that the denotation of a derivative of a program is a self-difference for the program.

The incremental λ-calculus has been introduced by Cai et al. [7] as a simply-typed calculus, and by Giarrusso et al. [17] as an untyped calculus. The former work introduces the notions of a program derivative and change update, and gives a denotational proof of soundness of differentiation. Operationally-based proofs of the same result are given in Giarrusso PhD's thesis [17, 16] by means of logical relations (see Remark 2). Remarkably, both Giarrusso's thesis [16] and the work by Giarrusso et al. [17] use ternary logical relations nearly identical to differential logical relations to relate programs with changes between them. Moreover, the logical relations introduced in the aforementioned papers have been mechanized in CoQ. The authors believe it is important to stress how essentially the same technique has independently emerged in different fields (and with different purposes) to prove two different kinds of differential properties of programs.

5 Conclusion

In this work we have established a formal connection between differential logical relations and the incremental λ-calculus of Cai et al. [7], whereby the self-differences of the former are identified with the program derivatives of the latter. Albeit the results proved here are not technically involved, by establishing a formal connection between two different fields they improve the current understanding of differential properties of programs, such an understanding being still in its infancy. The fact that essentially the same technique has been independently developed in different fields, one looking at software optimization and the other studying semantical notions of distance between programs, witnesses that, at least in the authors' opinion, the technique deserves to be further investigated.

In addition to its conceptual relevance, the connection established in the present work also allows us to obtain technical improvements both on the theory of incremental λ-calculus and on the one of differential logical relations. Concerning the former, we have showed how differential logical relations constitute a lightweight operational technique for incremental computing, and we have witnessed that by giving a new, relatively easy proof of soundness of differentiation. Concerning the latter, we have strengthened the fundamental lemma of DLRs [21] by showing how program derivatives constitute self-differences, this way reaching an higher level of compositionality. A further consequence of such a connection is the extension of DLRs to calculi with full recursion by means of the step-indexed logical relations of Giarrusso, Régis-Gianas, and Schuster [17].

Acknowledgment The authors are supported by the ERC Consolidator Grant DLV-818616 DIAPASoN as well as by the ANR project 16CE250011 REPAS.

References

1. Martín Abadi and Gordon D. Plotkin. A simple differentiable programming language. *PACMPL*, 4(POPL):38:1–38:28, 2020.
2. Samson Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison Wesley, 1990.
3. Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.
4. Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1):171 – 190, 2000. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.
5. Frank Van Breugel and James Worrell. A behavioural pseudometric for probabilistic transition systems. *Theoretical Computer Science*, 331(1):115–142, 2005.
6. Aloïs Brunel, Damiano Mazza, and Michele Pagani. Backpropagation in the simply typed lambda-calculus with linear negation. *PACMPL*, 4(POPL):64:1–64:27, 2020.
7. Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In *Proc. of PLDI 2014*, pages 145–155, 2014.
8. Konstantinos Chatzikokolakis, Daniel Gebler, Catuscia Palamidessi, and Lili Xu. Generalized bisimulation metrics. In *Proc. of CONCUR 2014*, pages 32–46, 2014.
9. Raphaëlle Crubillé and Ugo Dal Lago. Metric reasoning about λ-terms: The general case. In *Proc. of ESOP 2017*, pages 341–367, 2017.
10. Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-yaKatsumata, and Ikram Cherigui. A semantic account of metric preservation. In *Proc. of POPL 2017*, pages 545–556, 2017.
11. Josee Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Metrics for labelled markov processes. *Theoretical Computer Science*, 318(3):323–354, 2004.
12. Wenjie Du, Yuxin Deng, and Daniel Gebler. Behavioural pseudometrics for nondeterministic probabilistic systems. In *Proc. of SETTA 2016*, pages 67–84, 2016.
13. Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3):1–41, 2003.
14. Francesco Gavazzo. Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances. In *Proc. of LICS 2018*, pages 452–461, 2018.
15. Daniel Gebler, Kim G. Larsen, and Simone Tini. Compositional bisimulation metric reasoning with probabilistic process calculi. *Logical Methods in Computer Science*, 12(4), 2016.
16. Paolo G. Giarrusso. *Optimizing and incrementalizing higher-order collection queries by AST transformation*. PhD thesis, University of Tbingen, 2018.
17. Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. Incremental lambda-calculus in cache-transfer style - static memoization by program transformation. In *Proc. of ESOP 2019*, pages 553–580, 2019.
18. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
19. Jean-Yves Girard. Normal functors, power series and λ-calculus. *Annals of Pure and Applied Logic*, 37(2):129–177, 1988.
20. Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
21. Ugo Dal Lago, Francesco Gavazzo, and Akira Yoshimizu. Differential logical relations, part I: the simply-typed case. In *Proc. of ICALP 2019*, pages 111:1–111:14, 2019.

22. F. William Lawvere. Metric spaces, generalized logic, and closed categories. *Rendiconti del Seminario Matematico e Fisico di Milano*, 43:135–166, 1973.
23. Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistically accurate program transformations. In *In Proc. of SAS 2011*, pages 316–333, 2011.
24. Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys*, 48(4):62:1–62:33, 2016.
25. Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982.
26. Ganesan Ramalingam and Thomas W. Reps. A categorized bibliography on incremental computation. In *Proc. of POPL 1993*, pages 502–510, 1993.
27. Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *Proc. of ICFP 2010*, pages 157–168, 2010.
28. Clarence Hudson Richardson. *An Introduction to the Calculus of Finite Differences*. New York, 1954.
29. Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. Efficient differentiable programming in a functional array-processing language. *PACMPL*, 3(ICFP):97:1–97:30, 2019.
30. Michael Spivak. *Calculus On Manifolds: A Modern Approach To Classical Theorems Of Advanced Calculus*. Avalon Publishing, 1971.
31. Lynn Arthur Steen and Jr. J. Arthur Seebach. *Counterexamples in Topology*. Dover books on mathematics. Dover Publications, 1995.
32. Edwin M. Westbrook and Swarat Chaudhuri. A semantics for approximate program transformations. *CoRR*, abs/1304.5531, 2013. URL: `http://arxiv.org/abs/1304.5531`.