Differential Logical Relations, Part II
Increments and Derivatives

Ugo Dal Lago*, Francesco Gavazzo*

Abstract

We study the deep relation existing between differential logical relations and incremental computing by showing how self-differences in the former precisely correspond to derivatives in the latter. The byproduct of such a relationship is twofold: on the one hand, we 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. On the other hand, we generalize differential logical relations so as to be able to interpret full recursion, something not possible in the original system.

Keywords: differential logical relations, differential semantics, incremental lambda calculus, program difference, program derivatives

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

*Univesrity of Bologna & INRIA Sophia Antipolis

Email addresses: ugodallago@unibo.it (Ugo Dal Lago), francesco.gavazzo2@unibo.it (Francesco Gavazzo)

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* (Mittal, 2016) 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* (Reed and Pierce, 2010; Crubillé and Dal Lago, 2017; Gavazzo, 2018; de Amorim et al., 2017): 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* (Bartholomew-Biggs et al., 2000), which aim to extend the notion of a derivative one finds in mathematical analysis (Spivak, 1971) to arbitrary programs. Examples are given by (Brunel et al., 2020; Abadi and Plotkin, 2020; Shaikhha et al., 2019; Barthe et al., 2020), as well as by references therein.

- Those coming from the *differential λ-calculus* (Ehrhard and Regnier, 2003), whose original motivations rely on quantitative semantics (Girard, 1988) and linear logic (Girard, 1987).
- Those coming from *incremental computing* (Ramalingam and Reps, 1993; Paige and Koenig, 1982), which aim to find ways to incrementally compute outputs as inputs changes.
- Those coming from *differential logical relations* (Dal Lago et al., 2019) via the notion of a *self-difference*, which aim to provide context-sensitive compositional distances between programs.

It is thus natural to ask whether any connections exist 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* (Richardson, 1954), 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 (Cai et al., 2014). On the other hand, we can use results coming from incremental computing to go beyond the current theory of differential logical relations. We witness that by relying on the work by Giarrusso, Régis-Gianas, and Schuster (Giarrusso et al., 2019) on *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 (Dal Lago et al., 2019)).

Structure of the Paper. In Section 2, we introduce 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. In Section 4 we show how to extend the theory developed in Section 3 to a language with sum and list types. Finally, in Section 5 we take advantage of the connection between differential logical relations and incremental computing by extending the former to a calculus with full recursion.

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. To do so, we introduce the vehicle calculus of this work, namely a 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 by Dal Lago et al. to define differential logical relations (Dal Lago et al., 2019). The syntax and static semantics of $\mathsf{ST}_{\mathtt{R}}$ are given in Figure 1, where we assume to have primitives $\underline{r}$ for any real number r and constants $\underline{\varphi}$ for any function[1] $\varphi : \mathbb{R}^n \to \mathbb{R}$.

We use letters $t, s, \dots$ to range over terms, and $v, w, \dots$ to range over values. We follow standard notational conventions (Barendregt, 1985). Accordingly, we work with terms modulo renaming of bound variables and denote by $t[s/x]$ the capture-avoiding substitution of s for x in t. Contexts, i.e. terms with a hole $[-]$, are denoted by the letters $C, D, \dots$ and we write $C[t]$ for the capture-binding substitution of t for the hole $[-]$ in C. We write Λ_σ and $\mathcal{V}_\sigma$ for the collections of terms and values of type σ, respectively, omitting subscripts when types are not relevant. Finally, we use the notation $\Lambda^\bullet_\sigma$ and $\mathcal{V}^\bullet_\sigma$ for the sets of *closed* terms and *closed* values of type σ (again, omitting

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

$$\sigma, \tau ::= \mathtt{R} \;\Big|\; \sigma \times \tau \;\Big|\; \sigma \to \tau$$

$$t, s ::= x \;\Big|\; \underline{r} \;\Big|\; \underline{\varphi} \;\Big|\; \langle t, s \rangle \;\Big|\; \lambda x.t \;\Big|\; \mathbf{out}_1\, t \;\Big|\; \mathbf{out}_2\, t \;\Big|\; ts$$

$$v, w ::= x \;\Big|\; \underline{r} \;\Big|\; \underline{\varphi} \;\Big|\; \langle v, w \rangle \;\Big|\; \lambda x.t$$

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

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

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

Figure 1: Syntax and Statics of $\mathsf{ST}_{\mathtt{R}}$

subscripts when types are not relevant). As customary, we refer to closed terms as *programs*.

We endow $\mathsf{ST}_{\mathtt{R}}$ with a *call-by-value* dynamic semantics defined the reduction rules 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, a standard reducibility proof (Girard et al., 1989) shows 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 reduction 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 reduction steps. As it is customary, we denote by $\to^*$ the reflexive and transitive closure of $\to$.

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*.

First, let us define a suitable notion of program equivalence for $\mathsf{ST}_{\mathtt{R}}$ programs. Due to its simple nature, program equivalence for $\mathsf{ST}_{\mathtt{R}}$ terms can be defined by means of several notions of program equivalence, ranging from

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

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

Figure 2: Dynamics of $\mathsf{ST}_{\mathtt{R}}$

denotationally-based to operationally-based equivalences. Here, we choose *extensional* or *applicative equivalence*.[2]

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* $i \in \{1, 2\}$*):*

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

As usual, we omit type subscripts when not relevant.

It is well-known that extensional equivalence is a congruence, and thus a *compositional* technique for reasoning about program behaviors. In particular, if $t \cong s$, then $C[t] \cong C[s]$ holds for any context C (of the right type). 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.[3] The natural way to overcome this problem is

[2]This is nothing but a simplification of Abramsky's applicative bisimilarity (Abramsky, 1990) that takes advantage of the simple type system of $\mathsf{ST}_{\mathtt{R}}$.

[3]A similar argument can be formulated for any language 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 giving a quantitative flavor to standard semantical notions (such as termination) (Chatzikokolakis et al., 2014; de Amorim et al., 2017; Desharnais et al., 2004; Du et al., 2016; Gebler et al., 2016; Reed and Pierce, 2010; Van Breugel and Worrell, 2005).

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 (Lawvere, 1973). Accordingly, we obtain the following maps:

$$\begin{aligned}
\delta^{\mathcal{V}}_{\mathtt{R}}(\underline{r}, \underline{r}') &\triangleq r' - r \\
\delta^{\mathcal{V}}_{\sigma_1 \times \sigma_2}(\langle v_1, v_2 \rangle, \langle w_1, w_2 \rangle) &\triangleq \max_{i \in \{1,2\}} \delta^{\mathcal{V}}_{\sigma_i}(v_i, w_i) \\
\delta^{\Lambda}_{\sigma}(t, t') &\triangleq \delta^{\mathcal{V}}_{\sigma}(\mathsf{nf}(t), \mathsf{nf}(t')) \\
\delta^{\mathcal{V}}_{\sigma \to \tau}(v, v') &\triangleq \sup_{w \in \mathcal{V}^\bullet_\sigma} \delta^{\Lambda}_{\tau}(vw, v'w)
\end{aligned}$$

which can be easily proved to be generalized metrics[4] (Lawvere, 1973). To make δ a useful notion for reasoning about the (quantitative) behavior of programs, we need it to support the (quantitative refinement of the) compositionality principle ensured by $\cong$. As compositionality of $\cong$ takes 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 do not 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 trivializes, meaning that it collapses to a congruence *relation*. Roughly, given two terms t, s which are $\varepsilon > 0$ apart, for any real number $c \geq 1$ it is always possible to find a context C such that $C[t]$ and $C[s]$ are $c \cdot \varepsilon$ apart. For it is sufficient to take a term t using its input x *enough times*: once the terms $C[t]$ and $C[s]$ 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 map φ in the language of $\mathsf{ST}_{\mathtt{R}}$ 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 (Crubillé and Dal Lago, 2017; Reed and Pierce, 2010; Gavazzo,

[4]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* (Steen and J. Arthur Seebach, 1995).

2018) aiming to restore compositionality. All the 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 amplify distances). The notions of program distance thus obtained are indeed compositional, but still have two major drawbacks. First, they are tailored to linear calculi and are not very informative when applied to non-linear calculi via, e.g., standard translations (Girard, 1987). 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 (Dal Lago et al., 2019) have been introduced 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 (Westbrook and Chaudhuri, 2013) in the setting of approximate program transformations (Misailovic et al., 2011)) 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 (Westbrook and Chaudhuri, 2013) 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. *The map Δ associates to each type σ a type $\Delta\sigma$ which we refer to as the type of σ-differences. We define Δ recursively 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[5] between t and t' (and similarly for values).

Definition 3 (Asymmetric DLRs)**.** *A* differential logical relation *is a type-indexed family of (pairs 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:*

- $\mathsf{D}^{\mathcal{V}}_{\mathtt{R}}(\underline{dr}, \underline{r}, \underline{r}')$ *if and only if* $r' - r = dr$.
- $\mathsf{D}^{\mathcal{V}}_{\sigma_1 \times \sigma_2}(\langle dv_1, dv_2\rangle, \langle v_1, v_2\rangle, \langle v'_1, v'_2\rangle)$ *if and only if* $\mathsf{D}^{\mathcal{V}}_{\sigma_i}(dv_i, v_i, v'_i)$*, for any* $i \in \{1, 2\}$.
- $\mathsf{D}^{\mathcal{V}}_{\sigma\to\tau}(dv, v, v')$ *if and only if for all* dw, w, w'*, if* $\mathsf{D}^{\mathcal{V}}_{\sigma}(dw, w, w')$ *then* $\mathsf{D}^{\Lambda}_{\tau}(dv\, w\, dw, vw, v'w')$.
- $\mathsf{D}^{\Lambda}_{\sigma}(dt, t, t')$ *if and only if* $\mathsf{D}^{\mathcal{V}}_{\sigma}(dv, v, v')$*, where* $dt \Downarrow dv, t \Downarrow v, t' \Downarrow v'$.

[5]Following conventions in, e.g., (Cai et al., 2014), we use the notation $dt, ds, \ldots$ (resp. $dv, dw, \ldots$) for terms (resp. values) of type $\Delta\sigma$. The reader, however, should keep in mind that $\Delta\sigma$ is *de facto* an ordinary type, and thus $dt, \ldots$ are just ordinary terms (and values). Thus, for instance, $\Delta(\mathtt{R} \to \mathtt{R})$ is nothing but $\mathtt{R} \to \mathtt{R} \to \mathtt{R}$ and a closed value $dv : \Delta(\mathtt{R} \to \mathtt{R})$ is just an ordinary λ-abstraction of type $\mathtt{R} \to \mathtt{R} \to \mathtt{R}$.

Remark 4. Contrary to the original formulation of DLRs (Dal Lago et al., 2019), 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 5 (Dal Lago et al. (2019))**.** 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 $\mathsf{D}^{\nu}_{\mathtt{R}\to\mathtt{R}}(dt, t, t')$ requires to prove that for all $\underline{dr}, \underline{r}, \underline{r}'$ such that $\mathsf{D}^{\nu}_{\mathtt{R}}(\underline{dr}, \underline{r}, \underline{r}')$ (meaning that $r + dr = r'$), we have $\mathsf{D}^{\nu}_{\mathtt{R}}(\underline{r + dr - \sin r}, \underline{\sin r}, \underline{r}')$, 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* (Dal Lago et al., 2019).

Lemma 6 (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 6 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.

2.3. The Incremental λ-calculus and Program Derivatives

Albeit enabling compositional reasoning on program differences, Lemma 6 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 6 are precisely the *program derivatives* used in the *incremental λ-calculus*.

The incremental λ-calculus is a formalism introduced by Cai et al. (Cai et al., 2014) as a foundational calculus for incremental computation (Ramalingam and Reps, 1993; Paige and Koenig, 1982). 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 also 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[6] 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. (Cai et al., 2014) studied incrementalization of a simply-typed λ-calculus similar to $\mathsf{ST}_{\mathtt{R}}$. 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 $dt \in \Lambda_{\Delta\sigma}$. To account for incrementalization, the so-called *derivative*[7] $Dt \in \Lambda_{\Delta\sigma}$ of an expression $t \in \Lambda_\sigma$ is also introduced and it is shown 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 (Giarrusso, 2018).

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. 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 (Richardson, 1954).

Definition 7. *Given functions $\varphi : \mathbb{R}^n \to \mathbb{R}$ and $d\varphi : (\mathbb{R} \times \mathbb{R})^n \to \mathbb{R}$, define*

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

[7]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* (Richardson, 1954) of φ, i.e. the map $\Delta\varphi : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ defined by $\Delta\varphi(x, dx) \triangleq \varphi(x + dx) - \varphi(x)$.

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. For instance, the finite difference of the sine function sin is the function $\Delta \sin$ defined by $\Delta \sin(x, y) = \sin(x + y) - \sin(x)$. Notice that $\varphi \oplus \Delta\varphi = \varphi$.

Definition 8. *Define the* partial *operator* $\oplus : \Lambda \to \Lambda \to \Lambda$ *as follows:*

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

where the symbol $\oplus$ *in* $\underline{\varphi \oplus d\varphi}$ *is the one in Definition 7.*

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

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

Observe that we can indeed think of Dt as the generalization of finite differences to arbitrary programs. For instance, we have $D(\lambda x.\underline{\varphi}\ x) = \lambda x.\lambda dx.\underline{\Delta\varphi}\ x\ dx$. In a similar fashion, we can compute the derivative of the higher-order function $D(\lambda f.\lambda x.\underline{\varphi}\ (f\ x))$ as $\lambda f.\lambda df.\lambda x.\lambda dx.\underline{\Delta\varphi}\ (f\ x)\ (df\ x\ dx)$. 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. For instance, we have:

$$(\lambda x.\underline{\varphi}\ x) \oplus (\lambda x.\lambda dx.\underline{\Delta\varphi}\ x\ dx) = \lambda x.(\underline{\varphi}\ x) \oplus (\underline{\Delta\varphi}\ x\ dx) = \lambda x.(\underline{\varphi \oplus \Delta\varphi})\ (x \oplus dx)$$

which is nothing by $\lambda x.\underline{\varphi}\ x$.

3. Bridging the Gap

In this section, we relate DLRs with the incremental λ-calculus we have introduced in the previous section. We do so by acting on two orthogonal axes. On the one hand, we show that program derivatives are precisely the self-distances of Lemma 6. That is, for any program t, Dt is a self-distance for t. This result allows us to strengthen the fundamental lemma of DLRs (Lemma 6), 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* (Cai et al., 2014). 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 15).

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 (Dal Lago et al., 2019). 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 $\rho(x) \in \mathcal{V}_\sigma^\bullet$. In particular, we use the notation $d\rho$ to denote substitutions in $\mathsf{S}(d\Gamma)$.

Definition 10. *We extend the notion of a DLR to substitutions over an environment* Γ *as follows:* $\mathsf{D}_\Gamma(d\rho, \rho, \rho') \iff \forall (x : \sigma) \in \Gamma.\ \mathsf{D}_\sigma^\mathcal{V}(d\rho(dx), \rho(x), \rho'(x))$, *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 6, let us observe that DLRs are closed under reduction, in the following sense.

Lemma 11. *The following holds for all closed terms:*

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

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

Lemma 12 (Fundamental Lemma, Version 2)**.** *For any program $t \in \Lambda^{\bullet}_{\sigma}$ we have $\mathsf{D}_{\sigma}(Dt, t, t)$.*

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

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

The proof of the latter is a routine induction on t and v. We show a couple of cases as illustrative examples.

- Consider the case of real-valued maps $\varphi : \mathbb{R}^n \to \mathbb{R}$. We have to show that for all values $\underline{r}_i, \underline{r}'_i, d\underline{r}_i$ such that $\mathsf{D}^{\mathcal{V}}_{\mathtt{R}}(d\underline{r}_i.\underline{r}_i, \underline{r}'_i)$ we have

  $$\mathsf{D}^{\Lambda}_{\mathtt{R}}(\underline{\Delta\varphi}\, \langle \underline{r}_1, d\underline{r}_1 \rangle \cdots \langle \underline{r}_n, d\underline{r}_n \rangle, \underline{\varphi}\, \underline{r}_1 \cdots \underline{r}_n, \underline{\varphi}\, \underline{r}'_1 \cdots \underline{r}'_n)$$

  i.e.

  $$\varphi(r'_1, \dots, r'_n) - \varphi(r_1, \dots, r_n) = \Delta\varphi((r_1, dr_1), \dots, (r_n, dr_n))$$

  We are done since $\mathsf{D}^{\mathcal{V}}_{\mathtt{R}}(d\underline{r}_i.\underline{r}_i, \underline{r}'_i)$ implies $r' = r_i + dr_i$, and

  $$\Delta\varphi((r_1, dr_1), \dots, (r_n, dr_n)) = \varphi(r_1 + dr_1, \dots, r_n + dr_n) - \varphi(r_1, \dots, r_n).$$

- Consider the case of abstractions $\lambda x.t$. Assume $\mathsf{D}^{\mathcal{V}}_{\Gamma}(d\rho, \rho, \rho')$. We have to show

  $$\mathsf{D}^{\mathcal{V}}_{\sigma \to \tau}(\lambda x.\lambda dx.Dt[\rho, d\rho], \lambda x.t[\rho], \lambda x.t[\rho'])$$

  i.e.

  $$\mathsf{D}^{\Lambda}_{\tau}((\lambda x.\lambda dx.Dt[\rho, d\rho])\, v\, dv, (\lambda x.t[\rho])v, (\lambda x.t[\rho'])v')$$

  for all dv, v, v' such that $\mathsf{D}^{\mathcal{V}}_{\sigma}(dv, v, v')$. By Lemma 11, it is sufficient to show

  $$\mathsf{D}^{\Lambda}_{\tau}(Dt[\nu, d\nu], t[\nu], t[\nu'])$$

  where $\nu \triangleq \rho[x \mapsto v]$, $\nu' \triangleq \rho'[x \mapsto v']$, $d\nu \triangleq d\rho[x \mapsto dv]$, which indeed holds by induction hypothesis.

□

Notice how Lemma 12 improves the compositionality principle of DLRs. Given a term $x : \sigma \vdash t : \tau$ and two values $\vdash v, v' : \sigma$ such that $\mathsf{D}^{\mathcal{V}}_{\sigma}(dv, v, v')$ 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 13 (Soundness of Differentiation (Cai et al., 2014; Giarrusso et al., 2019))**.** *For all $t \in \Lambda^{\bullet}_{\sigma\to\tau}$ and values v, v', dv such that $\mathsf{D}^{\mathcal{V}}_{\sigma}(dv, v, v')$, we have $tv' \cong (tv) \oplus (Dt\, v\, dv)$.*

Our proof of Theorem 13 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 14. *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. 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 12, 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 12. □

We can finally prove soundness of differentiation.

Proof of Theorem 13. Assume $\mathsf{D}^{\mathcal{V}}_{\sigma}(dv, v, v')$. From Lemma 12, we obtain $\mathsf{D}_{\sigma\to\tau}(Dt, t, t)$, and thus $\mathsf{D}_{\tau}(Dt\, s\, ds, ts, ts')$, by Lemma 11. We conclude that $tv' \cong (tv) \oplus (Dt\, v\, dv)$ from Proposition 14. □

Remark 15. To the best of the authors' knowledge, all proofs of Theorem 13 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 (Giarrusso et al., 2019; Giarrusso, 2018). 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. Language Extensions

$\mathsf{ST}_{\mathtt{R}}$ is a minimal language which serves as a vehicle to study program differences. However, not so many interesting programs can be written in $\mathsf{ST}_{\mathtt{R}}$, and thus one may wonder whether our techniques scale to richer languages. In Section 5, we will show that differential logical relations can be extended to languages with full recursion. Here, we show by means of examples that our framework is robust with respect to language extensions given in the form of new data types. More specifically, we show how to extend differential logical relations to sum and list types. First, we extend the syntax of $\mathsf{ST}_{\mathtt{R}}$ with the new types and term constructs. This is done in Figure 3.
We also extend the dynamic semantics of $\mathsf{ST}_{\mathtt{R}}$ accordingly (Figure 4).

To extend the results of the previous section to $\mathsf{ST}_{\mathtt{R}}$ with sum and list types, what we have to do is to first define suitable difference spaces for such types, and then to extend DLRs accordingly. A standard notion of a difference space for sum types (Giarrusso, 2018) is given by $\Delta(\sigma + \tau) \triangleq \Delta\sigma + \Delta\tau + \sigma + \tau$. The rationale behind this definition is that whenever we compare elements of type $\sigma + \tau$ we either compare elements coming from the same type (viz. σ or τ) or we compare elements coming from different types (e.g. σ for the first element and τ for the second one). In the first case, a difference is a just a difference between the original terms (thus either an element of $\Delta\sigma$ or an element of $\Delta\tau$). In the second case, instead, we should compare terms coming from different types, something not possible in our framework: we thus simply take as a difference the second term, this way witnessing that we have a 'jump' from one type to another (so that a difference tells that we have such a jump and the target term of the jump). More precisely, let v, v' be values of type $\sigma + \tau$ and dv be a difference between them. Then, we have four possible cases:

$$\sigma,\tau ::= \dots \;\Big|\; \sigma+\tau \;\Big|\; [\sigma]$$

$$v,w ::= \dots \;\Big|\; \mathbf{in}_1\ v \;\Big|\; \mathbf{in}_2\ v \;\Big|\; \mathbf{nil} \;\Big|\; v :: v$$

$$t,s ::= \dots \;\Big|\; \mathbf{in}_1\ t \;\Big|\; \mathbf{in}_2\ t \;\Big|\; \mathbf{case}\ t\ \mathbf{of}\ (x_1 \Rightarrow t \mid x_2 \Rightarrow t) \;\Big|\; t :: t$$
$$\Big|\; \mathbf{case}\ t\ \mathbf{of}\ (\,\mathbf{nil} \Rightarrow t \mid x_{hd} :: x_{tl} \Rightarrow t)$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \mathbf{in}_1\ t : \sigma_1+\sigma_2} \qquad \frac{\Gamma \vdash t : \sigma_2}{\Gamma \vdash \mathbf{in}_2\ t : \sigma_1+\sigma_2} \qquad \frac{\Gamma \vdash t_{hd} : \sigma \quad \Gamma \vdash t_{tl} : [\sigma]}{\Gamma \vdash t_{hd} :: t_{tl} : [\sigma]}$$

$$\frac{\Gamma \vdash t : \sigma_1+\sigma_2 \quad \Gamma, x_1 : \sigma_1 \vdash t_1 : \tau \quad \Gamma, x_2 : \sigma_2 \vdash s_2 : \tau}{\Gamma \vdash \mathbf{case}\ t\ \mathbf{of}\ (x_1 \Rightarrow s_1 \mid x_2 \Rightarrow s_2) : \tau}$$

$$\frac{}{\Gamma \vdash \mathbf{nil}\ : [\sigma]} \qquad \frac{\Gamma \vdash t : [\sigma] \quad \Gamma \vdash s_{nil} : \tau \quad \Gamma, x_{hd} : \sigma, x_{tl} : [\sigma] \vdash s_{cons} : \tau}{\Gamma \vdash \mathbf{case}\ t\ \mathbf{of}\ (\,\mathbf{nil} \Rightarrow s_{nil} \mid x_{hd} :: x_{tl} \Rightarrow s_{cons}) : \tau}$$

Figure 3: Syntax of $\mathsf{ST}_{\mathtt{R}}$ with sums and lists.

$$\frac{t \to t' \quad i \in \{1,2\}}{\mathbf{in}_i\ t \to \mathbf{in}_i\ t'} \qquad \frac{i \in \{1,2\}}{\mathbf{case}\ (\mathbf{in}_i\ v)\ \mathbf{of}\ (x_1 \Rightarrow s_1 \mid x_2 \Rightarrow s_2) \to s_i[v/x_i]}$$

$$\frac{t \to t'}{\mathbf{case}\ t\ \mathbf{of}\ (x_1 \Rightarrow s_1 \mid x_2 \Rightarrow s_2) \to \mathbf{case}\ t'\ \mathbf{of}\ (x_1 \Rightarrow s_1 \mid x_2 \Rightarrow s_2)}$$

$$\frac{t_{hd} \to t'_{hd}}{t_{hd} :: t_{tl} \to t'_{hd} :: t_{tl}} \qquad \frac{t_{tl} \to t'_{tl}}{v_{hd} :: t_{tl} \to v_{hd} :: t'_{tl}}$$

$$\frac{}{\mathbf{case}\ \ \mathbf{nil}\ \ \mathbf{of}\ (\,\mathbf{nil} \Rightarrow s_{nil} \mid x_{hd} :: x_{tl} \Rightarrow s_{cons}) \to s_{nil}}$$

$$\frac{}{\mathbf{case}\ v_{hd} :: v_{tl}\ \mathbf{of}\ (\,\mathbf{nil} \Rightarrow s_{nil} \mid x_{hd} :: x_{tl} \Rightarrow s_{cons}) \to s_{cons}[v_{hd}/x_{hd}, v_{tl}/x_{tl}]}$$

$$\frac{t \to t'}{\mathbf{case}\ t\ \mathbf{of}\ (\,\mathbf{nil} \Rightarrow s_{nil} \mid x_{hd} :: x_{tl} \Rightarrow s_{cons}) \to \mathbf{case}\ t'\ \mathbf{of}\ (\,\mathbf{nil} \Rightarrow s_{nil} \mid x_{hd} :: x_{tl} \Rightarrow s_{cons})}$$

Figure 4: Dynamics of $\mathsf{ST}_{\mathtt{R}}$ with sum and list types

- Both v and v' come from the type σ (i.e. $v = \mathbf{in}_1\ (w)$ and $v' = \mathbf{in}_1\ (w')$). In that case, dv will be an element of $\Delta\sigma$ (injected into $\Delta(\sigma+\tau)$) acting as a difference between w and w'.

- Both v and v' come from the type τ. As before, in this case dv will be an element of $d\tau$ acting as a difference between w and w'.

- $v = \mathbf{in}_1\ (w)$ comes from the type σ and $v' = \mathbf{in}_2\ (w')$ from τ, so that we have a 'jump' from σ to τ. In that case, dv is just w' (injected into $\Delta(\sigma+\tau)$, and witnesses that we have 'jumped' from σ to τ.

- $v = \mathbf{in}_1\ (w)$ comes from the type σ and $v' = \mathbf{in}_2\ (w')$ from τ, so that we have a 'jump' from σ to τ. As before, in this case dv is just w' (injected into $\Delta(\sigma+\tau)$, and witnesses that we have 'jumped' from τ to σ.

For list types, things are simpler since we may take as difference space just lists of differences. Formally: $\Delta[\sigma] \triangleq [\Delta\sigma]$. Before extending differential logical relations, it is useful to introduce some syntactic sugar.

Remark 16. Due to the form of $\Delta(\sigma+\tau)$, it is convenient to introduce the following syntactic sugar for nested (with depth two) injection and case analysis on sum types: that allows us to inject and do pattern matching over 4-ary sums.

$$\frac{\Gamma \vdash t : \sigma_i}{\Gamma \vdash \mathbf{in}_i\ t : \sigma_1 + \sigma_2 + \sigma_3 + \sigma_4}\ i \in \{1,2,3,4\}$$

$$\frac{\Gamma \vdash t : \sigma_1 + \sigma_2 + \sigma_3 + \sigma_4 \quad \Gamma, x_i : \sigma_i \vdash s_i : \tau}{\Gamma \vdash \mathbf{case}\ t\ \mathbf{of}\ (x_1 \Rightarrow s_1 \mid x_2 \Rightarrow s_2 \mid x_3 \Rightarrow s_3 \mid x_4 \Rightarrow s_4) : \tau}\ i \in \{1,\dots,4\}$$

We are now ready to extend DLRs.

Definition 17. *We extend Definition 3 with the following clauses.*

- $\mathsf{D}^{\nu}_{\sigma_1+\sigma_2}(dv, v, v')$ *if and only if the following holds:*

$$\begin{aligned}
(v = \mathbf{in}_1\ w) \wedge (v' = \mathbf{in}_1\ (w')) &\implies (dv = \mathbf{in}_1\ dw) \wedge \mathsf{D}^{\nu}_{\sigma_1}(dw, w, w');\\
(v = \mathbf{in}_2\ w) \wedge (v' = \mathbf{in}_2\ (w')) &\implies (dv = \mathbf{in}_2\ dw) \wedge \mathsf{D}^{\nu}_{\sigma_2}(dw, w, w');\\
(v = \mathbf{in}_1\ w) \wedge (v' = \mathbf{in}_2\ (w')) &\implies (dv = \mathbf{in}_4\ w');\\
(v = \mathbf{in}_2\ w) \wedge (v' = \mathbf{in}_1\ (w')) &\implies (dv = \mathbf{in}_3\ w').
\end{aligned}$$

- $\mathsf{D}^{\nu}_{[\sigma]}(dv, v, v')$ *if and only either* $v = v' = dv =$ **nil** *or* $v = v_{hd} :: v_{tl}$, $v' = v'_{hd} :: v'_{tl}$, $dv = dv_{hd} :: dv_{tl}$, *and* $\mathsf{D}^{\nu}_{\sigma}(dv_{hd}, v_{hd}, v'_{hd})$ *and* $\mathsf{D}^{\nu}_{[\sigma]}(dv_{tl}, v_{tl}, v'_{tl})$.

To extend Lemma 12 to Definition 17, we need to extend the notion of a derivative to our new type constructors. To do so, we introduce some syntactic sugar which will improve readability. Given lists t, s, we write $\mathbf{case}\ t, s\ \mathbf{of}\ ((\,\mathbf{nil}\,,\,\mathbf{nil}\,) \Rightarrow r_{nil} \;\Big|\; (x_{hd} :: x_{tl}, y_{hd} :: y_{tl} \Rightarrow r_{cons})$ for the nested case analysis on t and s returning a dummy value for those cases not listed in the pattern matching (such a syntax will be used only for terms related by differential logical relations, this way ensuring the incomplete case analysis of our syntax to be indeed exhaustive). We use a similar syntax for nested case analysis on sum types.

Definition 18. *We extend Definition 9 with the following clauses:*

$$D(\mathbf{in}_i\ t) \triangleq \mathbf{in}_i\ Dt;$$

$$D(t_{hd} :: t_{tl}) \triangleq Dt_{hd} :: Dt_{tl};$$

$$D(\mathbf{case}\ t\ \mathbf{of}\ (x_1 \Rightarrow s_1 \mid x_2 \Rightarrow s_2)) \triangleq \left(\begin{array}{l} \mathbf{case}\ t, Dt\ \mathbf{of} \\ \quad x_1, dx_1 \Rightarrow Ds_1 \\ \quad\quad \mid\ x_2, dx_2 \Rightarrow Ds_2 \\ \quad\quad \mid\ x_1, x_4 \Rightarrow s_2[x_4/x_2] \\ \quad\quad \mid\ x_2, x_3 \Rightarrow s_1[x_3/x_1] \end{array}\right);$$

$$D\left(\begin{array}{l} \mathbf{case}\ t\ \mathbf{of} \\ \quad \mathbf{nil} \Rightarrow s_{nil} \\ \quad\quad \mid\ x_{hd} :: x_{tl} \Rightarrow s_{cons} \end{array}\right) \triangleq \left(\begin{array}{l} \mathbf{case}\ t, Dt\ \mathbf{of} \\ \quad (\,\mathbf{nil}\,,\,\mathbf{nil}\,) \Rightarrow Ds_{nil} \\ \quad\quad \mid\ (x_{hd} :: x_{tl}, dx_{hd} :: dx_{tl}) \Rightarrow Ds_{cons} \end{array}\right).$$

We now have all the ingredients to extend the Fundamental Lemma (Lemma 12) to sum and list types.

Lemma 19 (Fundamental Lemma, Version 3)**.** *For any program* $t \in \Lambda^{\bullet}_{\sigma}$ *we have* $\mathsf{D}_{\sigma}(Dt, t, t)$.

The proof of Lemma 19 is a straightforward extension of the one of Lemma 12. Notice how the incomplete case analysis of the derivative of the pattern matching constructs for sums and lists perfectly matches the one in Definition 17.

We have thus achieved the desired extension of DLRs (and their meta-theory) to list and sum types. Moreover, the exact same methodology we have used here can be used to extend DLRs to richer languages. However, so far we have focused on theoretical aspects of DLRs giving little attention on how they can be used in practice. Can we apply derivatives and differential logical relations to perform differential analysis of interesting higher-order programs? Consider, for instance, the higher-order function $\mathbf{map} : (\sigma \to \tau) \to [\sigma] \to [\tau]$ obeying the laws

$$\begin{aligned}\mathbf{map}\ (\lambda x.t)\ \mathbf{nil} &\to \mathbf{nil}\\ \mathbf{map}\ (\lambda x.t)\ (v_{hd} :: v_{tl}) &\to t[v_{hd}/x] :: (\mathbf{map}\ (\lambda x.t)\ v_{tl}).\end{aligned}$$

Can we define the derivative $D\mathbf{map}$, this way giving (by Lemma 19) a self-difference for **map**? We answer this question affirmatively in the next section, where we extend differential logical relations to full recursion.

5. Differential Logical Relations in Presence of Recursion via Step-indexing

The connection between DLRs and the incremental λ-calculus of previous sections can also be used as the starting point towards an extension of DLRs beyond the simply-typed setting of previous sections, this way generalizing Lemma 12 to Turing complete calculi. In this section, we show how to achieve such a goal for an extension of $\mathsf{ST}_{\mathtt{R}}$, which we call $\mathsf{PCF}_{\mathtt{R}}$, with a fixed point combinator. To handle full recursion, we refine DLRs using the so-called *step-indexing* technique (Appel and McAllester, 2001; Ahmed, 2006). A similar technique has been recently proposed for an untyped incremental λ-calculus (Giarrusso et al., 2019).

The syntax and static semantics of $\mathsf{PCF}_{\mathtt{R}}$ are obtained by extending the one of $\mathsf{ST}_{\mathtt{R}}$ with a fixed point operator $\mathbf{fix}(f, x).t$.

$$v, w ::= \dots \ \big|\ \mathbf{fix}(f, x).t$$

We extend the notational conventions used for $\mathsf{ST}_{\mathtt{R}}$ to $\mathsf{PCF}_{\mathtt{R}}$ *mutatis mutandis*. The static and dynamic semantics of $\mathsf{PCF}_{\mathtt{R}}$ are given by extending the one of $\mathsf{ST}_{\mathtt{R}}$ with the rules in Figure 5. Notice also that we can ignore the λ-abstraction constructor and encode terms of the form $\lambda x.t$ as $\mathbf{fix}(f, x).t$ where f does not appear (free) in t. We extend the notations $t \Downarrow v$ and $t \Downarrow_n v$ to $\mathsf{PCF}_{\mathtt{R}}$ in the obvious way. All this is standard, and does not pose any significant problem.

To extend DLRs to $\mathsf{PCF}_{\mathtt{R}}$, we need to handle possibly infinitary behaviors. We do so relying on step-indexing (Appel and McAllester, 2001).

$$\frac{\Gamma, f : \sigma \to \tau, x : \sigma \vdash t : \tau}{\Gamma \vdash \mathbf{fix}(f,x).t : \sigma \to \tau} \qquad \frac{}{(\mathbf{fix}(f,x).t)v \to t[v/x, \mathbf{fix}(f,x).t/f]}$$

Figure 5: Statics and Dynamics of $\mathsf{PCF}_{\mathtt{R}}$

Definition 20 (Step-indexed Asymmetric DLRs)**.** *A* step-indexed *DLR consists of a family of type-indexed (pairs of) relations* $\mathsf{D}^{\mathcal{V}}_{\sigma} \subseteq \mathbb{N} \times \mathcal{V}^{\bullet}_{\Delta\sigma} \times \mathcal{V}^{\bullet}_{\sigma} \times \mathcal{V}^{\bullet}_{\sigma};$, $\mathsf{D}^{\Lambda}_{\sigma} \subseteq \mathbb{N} \times \Lambda^{\bullet}_{\Delta\sigma} \times \Lambda^{\bullet}_{\sigma} \times \Lambda^{\bullet}_{\sigma};$ *such that:*

- $\mathsf{D}^{\mathcal{V}}_{\mathtt{R}}(n, \underline{dr}, \underline{r}, \underline{r'})$ *if and only if* $r + dr = r'$*.*
- $\mathsf{D}^{\mathcal{V}}_{\sigma_1\times\sigma_2}(n, \langle dv_1, dv_2\rangle, \langle v_1, v_2\rangle, \langle v'_1, v'_2\rangle)$ *if and only if* $\mathsf{D}^{\mathcal{V}}_{\sigma_i}(n, dv_i, v_i, v'_i)$*, for any* $i \in \{1,2\}$*.*
- $\mathsf{D}^{\mathcal{V}}_{\sigma\to\tau}(n, ds, \mathbf{fix}(f,x).t, \mathbf{fix}(f,x).t')$ *if and only if for all* k, v, v', dv*, we have:*

  $$\mathsf{D}^{\mathcal{V}}_{\sigma}(k, dv, v, v') \wedge k < n \implies \mathsf{D}^{\Lambda}_{\tau}(k, ds\, v\, dv, (\mathbf{fix}(f,x).t)v, (\mathbf{fix}(f,x).t')v').$$

- $\mathsf{D}^{\Lambda}_{\sigma}(n, dt, t, t')$ *if and only if* $\forall k < n.\ t \Downarrow_k v \wedge t' \Downarrow v'$ *implies* $dt \Downarrow dv \wedge \mathsf{D}^{\mathcal{V}}_{\sigma}(n-k, dv, v, v')$*.*

Definition 20 is not conceptually far from the original definition of DLRs. Its main novelty is the introduction of a new parameter (the natural number n in its defining clauses) which behaves as a standard step-index in ordinary logical relations. Notice that we can also straightforwardly refine Definition 17 with the step-index parameter. For instance, we have $\mathsf{D}^{\mathcal{V}}_{\sigma_1+\sigma_2}(n, dv, v, v')$ if and only if the following holds:

$$\begin{aligned}
v = \mathbf{in}_1\ w \wedge v' = \mathbf{in}_1\ (w') &\implies dv = \mathbf{in}_1\ dw \wedge \mathsf{D}^{\mathcal{V}}_{\sigma_1}(n, dw, w, w');\\
v = \mathbf{in}_2\ w \wedge v' = \mathbf{in}_2\ (w') &\implies dv = \mathbf{in}_2\ dw \wedge \mathsf{D}^{\mathcal{V}}_{\sigma_2}(n, dw, w, w');\\
v = \mathbf{in}_1\ w \wedge v' = \mathbf{in}_2\ (w') &\implies dv = \mathbf{in}_4\ w';\\
v = \mathbf{in}_2\ w \wedge v' = \mathbf{in}_1\ (w') &\implies dv = \mathbf{in}_3\ w'.
\end{aligned}$$

To prove the $\mathsf{PCF}_{\mathtt{R}}$ counterpart of Lemma 12, we need to extend the notion of a derivative to $\mathsf{PCF}_{\mathtt{R}}$ terms. In order to improve the understanding and readability of our results, it is convenient to introduce some syntactic sugar in the form of a term $\mathbf{fix}(df, dx, f, x).dt$ whose dynamics is the following

$$(\mathbf{fix}(df, dx, f, x).dt)\, v\, dv \to^* dt[\mathbf{fix}(df, dx, f, x).dt/df, \mathbf{fix}(f,x).t/f, dv/dx, v/x]$$

and which can be easily constructed via nested **fix**. We are now ready to define $\mathsf{PCF}_{\mathtt{R}}$ derivatives.

Definition 21. *The derivative Dt of a $\mathsf{PCF}_{\mathtt{R}}$ term t is defined by extending Definition 9 (resp. Definition 18) with the following clause:*

$$D(\mathbf{fix}(f,x).t) \triangleq \mathbf{fix}(df,dx,f,x).Dt.$$

The rationale behind the definition of $D(\mathbf{fix}(f,x).t)$ is as follows (Arntzenius, 2017; Alvarez-Picallo et al., 2019). Given a function $f : X \to X$, let μf be its (least) fixed point, so that $\mu f = f(\mu f)$. As a consequence, by the very definition of the derivative of an application, we have: $D(\mu f) = D(f(\mu f)) = (Df(\mu f))(D(\mu f))$. Therefore, $D(\mu f)$ is a fixed point of the map $x \mapsto (Df(\mu f))(x)$, and thus we can stipulate $D(\mu f) \triangleq \mu((Df)(\mu f))$. This way, obtain the desired result:

$$D(\mu f) = \mu((Df)(\mu f)) = ((Df)(\mu f))(\mu((Df)(\mu f))) = ((Df)(\mu f))(D(\mu f)).$$

The last piece we need in order to prove the extension of Lemma 12 to $\mathsf{PCF}_{\mathtt{R}}$ is the extension of step-indexed DLRs to substitution maps:

Definition 22. *We extend the notion of a step-indexed DLR to substitutions over an environment Γ as follows: $\mathsf{D}_\Gamma(n, d\rho), \rho, \rho') \iff \forall (x : \sigma) \in \Gamma.\ \mathsf{D}^{\mathcal{V}}_\sigma(n, d\rho(dx), \rho(x), \rho'(x))$, where $\rho, \rho' \in \mathsf{S}(\Gamma)$ and $d\rho \in \mathsf{S}(\Delta\Gamma)$.*

We are finally ready to extend Lemma 12 to $\mathsf{PCF}_{\mathtt{R}}$.

Lemma 23 (Fundamental Lemma, Version 4)**.** *For any term $\Gamma \vdash t : \sigma$ and value $\Gamma \vdash v : \sigma$:*

$$\mathsf{D}_\Gamma(n, d\rho, \rho, \rho') \implies \mathsf{D}^{\mathcal{V}}_\sigma(n, Dv[\rho, d\rho], v[\rho], v[\rho']) \wedge \mathsf{D}^{\Lambda}_\sigma(n, Dt[\rho, d\rho], t[\rho], t[\rho']),$$

for all $d\rho, \rho, \rho'$ and for any $n \geq 0$.

Proof sketch. The proof is by induction on n, t, and v, mimicking the one of Lemma 12 and using the following standard results:

$$\begin{aligned}
\forall d\rho, \rho, \rho'.\ \forall n. \mathsf{D}_\Gamma(n, d\rho, \rho, \rho') &\implies \forall k \leq n.\ \mathsf{D}_\Gamma(k, d\rho, \rho, \rho');\\
\forall dt, t, t' \in \Lambda^\bullet_\sigma.\ \forall n. \mathsf{D}^{\Lambda}_\sigma(n, dt, t, t') &\implies \forall k \leq n.\ \mathsf{D}^{\Lambda}_\sigma(k, dt, t, t');\\
\forall dv, v, v' \in \mathcal{V}^\bullet_\sigma.\ \forall n. \mathsf{D}^{\mathcal{V}}_\sigma(n, dv, v, v') &\implies \forall k \leq n.\ \mathsf{D}^{\mathcal{V}}_\sigma(k, dv, v, v').
\end{aligned}$$

□

Let us now come back to the example of the higher-order combinator **map**. First, let us define **map** formally:

$$\mathbf{map} = \mathbf{fix}(m, f).\lambda\ell.\mathbf{case}\ \ell\ \mathbf{of}\ (\,\mathbf{nil} \Rightarrow \mathbf{nil} \mid x_{hd} :: x_{tl} \Rightarrow f\ x_{hd} :: m\ f\ x_{tl})$$

where $m : (\sigma \to \tau) \to [\sigma] \to [\tau]$ and $f : \sigma \to \tau$. We can now compute D**map**, obtaining:

$$\begin{aligned}
&\mathbf{fix}(dm, df, m, f).\lambda\ell.\lambda d\ell.\\
&\qquad\qquad \mathbf{case}\ \ell, d\ell\ \mathbf{of}\\
&\qquad\qquad\qquad (\,\mathbf{nil}\,, \mathbf{nil}\,) \Rightarrow \mathbf{nil}\\
&\qquad\qquad\qquad \Big|\ (x_{hd} :: x_{tl}, dx_{hd} :: dx_{tl}) \Rightarrow df\ x_{hd}\ dx_{hd} :: dm\ f\ df\ x_{tl}\ dx_{tl}.
\end{aligned}$$

Notice that in the same way as **map** obeys the laws **map** $(\lambda x.t)$ **nil** $\to$ **nil** and **map** $(\lambda x.t)$ $(v_{hd} :: v_{tl}) \to t[v_{hd}/x] :: (\mathbf{map}\ (\lambda x.t)\ v_{tl})$, we have $D\mathbf{map}\ (\lambda x.t)\ (\lambda x.\lambda dx.Dt)\, \mathbf{nil}\ \mathbf{nil} \to \mathbf{nil}$ and

$$\begin{aligned}
&D\mathbf{map}\ (\lambda x.t)\ (\lambda x.\lambda dx.Dt)\ (v_{hd} :: v_{tl})\ (Dv_{hd} :: Dv_{tl})\\
&\quad \to Dt[v_{hd}/x, Dv_{hd}/dx] :: D\mathbf{map}\ (\lambda x.t)\ (\lambda x.\lambda dx.Dt)\ v_{tl}\ Dv_{tl}.
\end{aligned}$$

The Fundamental Lemma (Lemma 23) ensures that D**map** is indeed a self-difference of **map**, and thus provides a way to perform a differential (and context-sensitive) analysis of **map**.

6. Related Work

Differential logical relations have been introduced by the authors and Yoshimizu (Dal Lago et al., 2019), building over intuitions by Westbrook and Chaudhuri (Westbrook and Chaudhuri, 2013), and are currently under investigation. Differently from the one considered in this work, the first formulation of differential logical relations (Dal Lago et al., 2019) 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 (Lawvere, 1973)), 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. This, however, would uselessly complicate the exposition of our results, as we should have to go throw the denotational semantics of both $\mathsf{ST}_{\mathtt{R}}$ and $\mathsf{PCF}_{\mathtt{R}}$.

The incremental λ-calculus has been introduced by Cai et al. (Cai et al., 2014) as a simply-typed calculus, and by Giarrusso et al. (Giarrusso et al., 2019) 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 (Giarrusso et al., 2019; Giarrusso, 2018) by means of logical relations (see Remark 15). Remarkably, both Giarrusso's thesis (Giarrusso, 2018) and the work by Giarrusso et al. (Giarrusso et al., 2019) 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.

Finally, semantical investigations of abstract notions of difference have been given in terms of category theory by Alvarez-Picallo and Ong (Alvarez-Picallo and Ong, 2019).

7. Conclusion

In this work, we have established a formal connection between differential logical relations and the incremental λ-calculus, 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 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 the step-indexing.

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

References

References

Abadi, M., Plotkin, G.D., 2020. A simple differentiable programming language. PACMPL 4, 38:1–38:28.

Abramsky, S., 1990. The lazy lambda calculus, in: Turner, D. (Ed.), Research Topics in Functional Programming, Addison Wesley. pp. 65–117.

Ahmed, A.J., 2006. Step-indexed syntactic logical relations for recursive and quantified types, in: Proc. of ESOP 2006, pp. 69–83.

Alvarez-Picallo, M., Eyers-Taylor, A., Jones, M.P., Ong, C.L., 2019. Fixing incremental computation - derivatives of fixpoints, and the recursive semantics of datalog, in: Proc. of ESOP 2019, pp. 525–552.

Alvarez-Picallo, M., Ong, C.L., 2019. Change actions: Models of generalised differentiation, in: Proc. of FOSSACS 2019, pp. 45–61.

de Amorim, A.A., Gaboardi, M., Hsu, J., Shin-yaKatsumata, Cherigui, I., 2017. A semantic account of metric preservation, in: Proc. of POPL 2017, pp. 545–556.

Appel, A.W., McAllester, D.A., 2001. An indexed model of recursive types for foundational proof-carrying code. ACM Transactions on Programming Languages and Systems 23, 657–683.

Arntzenius, M., 2017. Static differentiation of monotone fixpoints. URL: `http://www.rntz.net/files/fixderiv.pdf`.

Barendregt, H.P., 1985. The lambda calculus - its syntax and semantics. volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland.

Barthe, G., Crubillé, R., Dal Lago, U., Gavazzo, F., 2020. On the versatility of open logical relations - continuity, automatic differentiation, and a containment theorem, in: Proc. of ESOP 2020, pp. 56–83.

Bartholomew-Biggs, M., Brown, S., Christianson, B., Dixon, L., 2000. Automatic differentiation of algorithms. Journal of Computational and Applied Mathematics 124, 171 – 190. Numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations.

Brunel, A., Mazza, D., Pagani, M., 2020. Backpropagation in the simply typed lambda-calculus with linear negation. PACMPL 4, 64:1–64:27.

Cai, Y., Giarrusso, P.G., Rendel, T., Ostermann, K., 2014. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation, in: Proc. of PLDI 2014, pp. 145–155.

Chatzikokolakis, K., Gebler, D., Palamidessi, C., Xu, L., 2014. Generalized bisimulation metrics, in: Proc. of CONCUR 2014, pp. 32–46.

Crubillé, R., Dal Lago, U., 2017. Metric reasoning about λ-terms: The general case, in: Proc. of ESOP 2017, pp. 341–367.

Dal Lago, U., Gavazzo, F., Yoshimizu, A., 2019. Differential logical relations, part I: the simply-typed case, in: Proc. of ICALP 2019, pp. 111:1–111:14.

Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P., 2004. Metrics for labelled markov processes. Theoretical Computer Science 318, 323–354.

Du, W., Deng, Y., Gebler, D., 2016. Behavioural pseudometrics for nondeterministic probabilistic systems, in: Proc. of SETTA 2016, pp. 67–84.

Ehrhard, T., Regnier, L., 2003. The differential lambda-calculus. Theoretical Computer Science 309, 1–41.

Gavazzo, F., 2018. Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances, in: Proc. of LICS 2018, pp. 452–461.

Gebler, D., Larsen, K.G., Tini, S., 2016. Compositional bisimulation metric reasoning with probabilistic process calculi. Logical Methods in Computer Science 12.

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

Giarrusso, P.G., Régis-Gianas, Y., Schuster, P., 2019. Incremental lambda-calculus in cache-transfer style - static memoization by program transformation, in: Proc. of ESOP 2019, pp. 553–580.

Girard, J., 1987. Linear logic. Theoretical Computer Science 50, 1–102.

Girard, J., 1988. Normal functors, power series and λ-calculus. Annals of Pure and Applied Logic 37, 129–177.

Girard, J., Lafont, Y., Taylor, P., 1989. Proofs and Types. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press.

Lawvere, F.W., 1973. Metric spaces, generalized logic, and closed categories. Rendiconti del Seminario Matematico e Fisico di Milano 43, 135–166.

Misailovic, S., Roy, D.M., Rinard, M.C., 2011. Probabilistically accurate program transformations, in: In Proc. of SAS 2011, pp. 316–333.

Mittal, S., 2016. A survey of techniques for approximate computing. ACM Computing Surveys 48, 62:1–62:33.

Paige, R., Koenig, S., 1982. Finite differencing of computable expressions. ACM Trans. Program. Lang. Syst. 4, 402–454.

Ramalingam, G., Reps, T.W., 1993. A categorized bibliography on incremental computation, in: Proc. of POPL 1993, pp. 502–510.

Reed, J., Pierce, B.C., 2010. Distance makes the types grow stronger: a calculus for differential privacy, in: Proc. of ICFP 2010, pp. 157–168.

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

Shaikhha, A., Fitzgibbon, A., Vytiniotis, D., Peyton Jones, S., 2019. Efficient differentiable programming in a functional array-processing language. PACMPL 3, 97:1–97:30.

Spivak, M., 1971. Calculus On Manifolds: A Modern Approach To Classical Theorems Of Advanced Calculus. Avalon Publishing.

Steen, L.A., J. Arthur Seebach, J., 1995. Counterexamples in Topology. Dover books on mathematics, Dover Publications.

Van Breugel, F., Worrell, J., 2005. A behavioural pseudometric for probabilistic transition systems. Theoretical Computer Science 331, 115–142.

Westbrook, E.M., Chaudhuri, S., 2013. A semantics for approximate program transformations. CoRR abs/1304.5531. URL: `http://arxiv.org/abs/1304.5531`.